AUTOMATED GRADING OF UML USE CASE 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 & Quantitative Methods M.Sc. Graduate Program

May 2023

ABSTRACT

Automated Grading of UML Use Case Diagrams

Mohsen Hosseinibaghdadabadi

This thesis presents an approach for automated grading of UML Use Case diagrams. Many software engineering courses require students to learn how to model the behavioural features of a problem domain or an object-oriented design in the form of a use case diagram. Because assessing UML assignments is a time-consuming and labor-intensive operation, there is a need for an automated grading strategy that may help instructors by speeding up the grading process while also maintaining uniformity and fairness in large classrooms.

The effectiveness of this automated grading approach was assessed by applying it to two real-world assignments. We demonstrate how the result is similar to manual grading, which was less than 7% on average; and when we applied some strategies, such as configuring settings and using multiple solutions, the average differences were even lower. Also, the grading methods and the tool are proposed and empirically validated.

Keywords: automated grading, use case diagram, model comparison

Acknowledgment

Foremost, I would like to express my sincere gratitude to my supervisor Dr. Omar Alam, and also Dr. Jorg Kienzle from McGill University for the continuous support of my master's study and research, for your patience, motivation, enthusiasm, and immense knowledge. Your guidance helped me in all the time of research and writing of this thesis. Also, thank you to my supervisor committee for your time and advice.

My heartfelt appreciation goes to my wife, Atifeh, who endured all the hardships of being away from her family, friends, and homeland, she was by my side throughout this whole process and gave me motivation and hope whenever I was disappointed so that I could achieve my lofty goals.

I am also eternally grateful to my parents and siblings. They preferred my success to being away from them and motivated me all along the way to continue with strength until the end.

I owe a great deal of gratitude to my University, Trent, and all its employees for providing me with such a valuable opportunity to study for my Master's in my favorite field.

Finally, I am grateful to my family and friends for their unwavering love and support, as well as for instilling in me the confidence and determination to attain my lofty ambitions. I also thank my dear friend Bahadur for all his help in this way. While I cannot expect to adequately thank everyone who has helped me with my MSc, I would like to offer my heartfelt gratitude to everyone for their help along this great journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Engineers use models to explain the shapes or actions of any structure they wish to build or develop. Typically, graphic symbols, relations, and explanations, will be designed to visualize the model[1]. As use case diagram is kind of a behavior diagram in the Unified Modeling Language[2], it is used to represent the dynamic behavior of a system and understand how the system should work [3].

The growth of new technology requires considerable adjustments in the educational system [4]. The advancement of computer-based information technology can facilitate learning, particularly in educational assessments or evaluating student learning outcomes [5, 6]. Due to the growing number of computer science students, one of the most important procedures in teaching and learning is assessment [7, 8]. When assessing each assignment, evaluating the student use case model might be challenging for the teacher [9]. A teacher must conduct an assessment method while teaching use case diagrams in this scenario. Based on the answer keys, a teacher will grade the use case diagrams created by students. This use case diagram can be very long, including many use cases, and since each use case contains a specification part comprising many texts, grading each assignment will take considerable energy and time. Furthermore, courses with a large number of participants cannot provide individual support in solving modelling tasks, as there are often multiple correct solutions. The use case diagram created by students may differ from the instructor's solution due to using different words when naming actors and use cases, or paraphrasing the use case Main Success Scenario steps. However, the student's answer may differ from the instructor's, but the whole process can be very similar, which causes difficulties for the instructor when grading a number of students' solutions manually. Moreover, individual feedback on the modelling solution is extremely crucial in this situation [9]. In addition to the increasing workload, instructors struggle to fairly mark assignments and tests, which is a difficult process, particularly, when matching texts and flattening steps become important in the use case specification part. Furthermore, teachers frequently alter their marking scheme after assessing multiple student papers. For example, instructors may decide to redistribute scores if they see that students struggled with a certain area of the model, indicating that the issue description was possibly confusing. To compensate, the instructor may decide to modify the grading weights for different parts of the model.

Besides, at a time when the internet has reached almost everywhere, automated grading is critical for e-learning and Massive Open Online Courses (MOOCs) [10, 11]. These online courses offer a very accessible means for learners to develop new and existing skills. Self-assessment methodologies have been integrated into popular online learning systems such as Linkedin [12], Coursera [13], EDX [14], and Skillshare [15]. Automated assessment can also be used to calibrate a learner's past knowledge [16], that is, to test the learner's prior knowledge of the subject before starting a new course. Furthermore, in the context of online courses, automated grading may be employed to offer learners fast feedback [17].

Motivated by the reasons stated above, we propose an automated grading approach for UML use case diagrams. We reused metamodels introduced by Bian. et al [18]. These grading metamodels were implemented in the TouchCORE tool [19], which visually displays the marks on the classes and prints feedback to the student.

1.2 Problem Summary

This thesis is an attempt to address the issue highlighted in the introduction. First, we want to see if automated grading is more effective for use case models than manual grading. As a result, by reusing metamodels introduced by Bian et al. [18], we propose an automated grading approach for UML use case diagrams . these grading metamodels were implemented in the TouchCORE tool [19], which visually displays the marks and prints feedback to the student. We offer an algorithm that uses syntactic, semantic, and structural matching techniques to create mappings between model components in the instructor's solutions and elements in the student solutions. We also demonstrate how flattening steps improves the grades of students and decreases instructor mistakes when grading a large use case assignment. In addition, our algorithm can compare those flattened steps, conceptually using some natural language processing techniques.

We ran this algorithm on real assignments for modeling a Gas Station for a class of 14 undergraduate students, and Elfenroads use case models for 9 undergraduate student groups. On average, our algorithm was able to automatically grade the assignments within a 7% difference from the instructor's grade.

1.3 Thesis Contributions

We offer an algorithm that uses syntactic, semantic, structural, and text matching techniques to create mappings between model components in the instructor's solutions and elements in the student solutions. Each precisely matched element earns the students a full mark. Partially right answers, such as a correct actor with incorrect multiplicity, receive a partial mark and earn no marks if the component is missed or completely incorrect. One significant advantage of our technique is that it can easily update the students' marks when the instructor changes the grading scheme. Additionally, we propose a flattening steps approach, which orders all steps in the instructor's and student's solution into a list, and since those steps play an important role in the use case model, our algorithm can compare them conceptually using some natural language processing techniques.

We evaluated our algorithm using modelling two real-world assignments; a Gas Station and Elfenroads game. At this point, there are 14 individual submissions for one case study and nine group submissions of seven students for another, which means 63 students contributed to this assignment. Although our automated grading results are close to manual grading and provide confident evidence, our sample size might not be large enough, and we need to examine more case studies in the future.

1.4 Research Questions

The following research questions can be answered by comparing the outcomes of automated grading with manual grading performed by instructors:

- **RQ1: How different are manual and automated grading?**

  There should not be a large difference between an instructor's grades and automated grades. Using two case studies, we will demonstrate that the average difference is less than 9% when using default settings.

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

  A configurable grading algorithm can generate results that are more similar to the instructor's manual grading scores. Overall, the difference in one case study decreased from 9.59% to 6.69%, while the difference in another case study is reduced from 8.27% to 7.61%.

- **RQ3: Does automated grading help ensure fairness?**

  Inconsistencies are common with manual grading, and we investigated the consistency aspect of fairness. Automated grading can detect discrepancies in manual grading. We will explain how the instructor could make mistakes when grading.

- **RQ4: Does the accuracy of automated grading improve when multiple solutions are matched against?**

  There could be more than one correct solution for a use case diagram modelling problem. So, the tool is capable of loading multiple correct solutions. When all three available solutions were used in one case study, the average differences decreased from 6.69% to 4.50%.

- **RQ5: How do flattening steps helps automated grading to achieve a better result?**

When there are multiple use cases in the diagram, they may be linked by a relationship. The best idea when grading is to first flatten all steps in all use cases based on their relationships. We proceed with this method from the root use case to the leaf use cases in the diagram, and at the end, we have a list of flattened steps. When manually grading large models, flattening can be a difficult task. When grading without flattening is compared to grading with flattening, it can be seen that the average mark is significantly improved in both case studies.

- **RQ6: Does using NLP improve the quality of the grading algorithm?**

  The most challenging part of automated grading in use case models is matching texts. The matching text algorithm is essential in the grading of use case models. The algorithm is more effective when we combine it with some natural language processing methods. The thesis examines and shows how to find the best threshold value that can be set in the algorithm when matching texts. In addition, different parts of speech could have different levels of importance. Hence, coefficient weights for two kinds of parts of speech (nouns and verbs) are examined, and as a result, we should consider weights for nouns and verbs 1.5 times more than other parts of speech.

1.5 Thesis Organization

This thesis is divided into seven chapters. In Chapter 2, we address related works on automated grading. In Chapter 3, we then present examples that motivate our matching strategies. Chapter 4 describes several methodologies for matching words and texts and also discusses the algorithms that compare the student's model to the instructor's model. The architecture, metamodels, and grading tools that facilitate automated grading are demonstrated in Chapter 5. In this chapter, we also review our configuration panel and elaborate on the configuration options for automated grading. The case study arrangement is described in detail in Chapter 6. We also evaluate our tool and answer research questions by comparing results in this chapter. Chapter 7 concludes this thesis and discusses directions for future work.

The contributions of the thesis are organized into five parts:

Part 1: Motivation (Chapter 3)

We motivate our approach using a use case diagram modelling the "Elfenroads" game. All the strategies we need to match and grade each component of the diagram are discussed in this part.

Part 2: Grading Algorithm (Chapter 4)

The thesis proposes and describes various matching techniques for grading the UML use case diagram. Motivating model examples are used to demonstrate the process. It also goes over the methodologies used in matching words and texts, as well as demonstrating the algorithm for automated grading of UML use case diagrams.

Part 3: Grading Architecture and Tool Support (Chapter 5)

The architecture of the grading strategy and two metamodels are introduced in the thesis to enable the automated grading algorithm. Furthermore, the thesis describes the grading tool that implements the automated grading algorithm.

Part 4: Case Study (Chapter 6)

The grading technique is used in the thesis for a case study of two modeling tasks.

Part 5: Research Questions (Chapter 6)

In this section, six research questions mentioned in Section 1.4 will be investigated in detail.

Chapter 2

Related Work

2.1 Automated Grading Tools

Automated grading approaches has been explored for other modeling languages. Alur et al. [20] proposed a method for automatically grading the standard computation-theory problem, which asks students to construct a deterministic finite automaton (DFA) from a description of its language. They focused on how to assign partial grades for incorrect students' answers using a hybrid of three techniques: syntactic, semantic, and capturing mistakes. Batmaz and Hinde [21] presented a system to assist human graders in evaluating conceptual database diagrams. It also helps students model their diagrams easily. They focused on semi-automatic diagram marking, aiming to reduce the number of sub-diagrams marked by the examiner. Simanjuntak [22] presented preliminary research and a proposed framework for an automated ER Diagram grading system. He presented two approaches: The Tree Edit Distance algorithm to measure ER Diagram similarity, and a machine learning technique to build a classifier that automatically grades ER Diagrams. Thomas et al. [23] provided a method for computer-aided diagram understanding and demonstrated how it may be effectively used for the automatic grading of student attempts at drawing entity-relationship (ER) diagrams. They concentrated on imprecise diagrams, in which the required characteristics are either malformed or missing, or extraneous features are included.

There are approaches that propose the automated assessment of other kinds of UML models. Bian et al. [18] presented an approach for automated grading of

UML class diagrams. They proposed a metamodel that establishes mappings between the instructor's solution and all the solutions for a class. Their approach employs a grading algorithm that uses syntactic, semantic, and structural matching to match a student's solutions with the template solution. They also investigated the effectiveness of an automated grading approach for UML class diagrams when applied to two classroom settings across different universities and compared the results of automated grading with manual grading. [24]. Striewe et al. [25] describe a method for grading UML behavioural diagrams based on trace information. Tsintisfas [26] created a generic student diagram editor that can be easily customised for the exercise. He also produced a Computer Based Assessment (CBA) that automatically marks diagram-based models, e.g., process and entity-relationship diagrams. Jayal and Shepperd [27] developed a label-matching approach for UML diagrams that employs multiple levels of decomposition as well as syntactical matching. Through text transformation processing, they empirically investigated the diversity of labels used by students in the activity diagram. Thomas et al. [28] presented the analysis and classification of errors in imprecise sequence diagrams as well as the performance of an automated grading system. The analysis informed the design of a syntax error-checking tool that detects and reports on syntax errors and also repairs the majority of error types using information gleaned from the error analysis. Tselonis et al. [29] proposed a graph-matching-based diagram marking approach for structural diagrams, e.g., UML class diagrams and electronic circuits. Sousa and Paulo [30] proposed a structural technique for graphs that develops mappings from an instructor's solution to elements in a student's solution in order to maximize a student's grade. The proposed algorithm is applicable to any type of document that can be parsed into its graph-inspired data model. This data model is able to accommodate diagram languages, such as UML or ER diagrams, for which this kind of assessment is typically used.

2.2 Automated Grading Tools for UML Use Case Diagrams

Limited work has been done in the area of automated evaluation of use case diagrams, or it is under process. Fauzan et al. [3] created a semantic use case diagram for an automatic assessment method that performs WordNet [31] searches using WuPalmer [32, 33]. They first translated the use case diagram's features (actor, use case, relationships) into XMI and matched the labels semantically using some natural language processing methods. Vachharajani et al. [34] proposed an architecture of automated assessment of use case diagrams. The essence of their architecture is to assess a large number of students' diagrammatic answers very easily in a short period of time. Their tool can also give quantitative feedback in terms of grades as well as qualitative feedback in terms of suggestions. Vachharajani et al. [35] also employed a hybrid technique for automatically assessing use case diagrams, in which labels were processed simultaneously in the syntactic and semantic matching phases. They considered a use case diagram as a mixed graph, with nodes representing different actors or use cases and edges indicating the various relationships among them. In another paper, Vachharajani et al. [36] introduced and developed a tool named "Use Case Extractor" towards achieving the research objective of automated evaluation of the Use Case diagram. Arifin et al. [37] proposed a method for measuring the similarity of use case diagrams using structural and semantic matching aspects. Their proposed method used the process of modelling the use case diagram as a graph and the graph similarity method to measure structural similarity, and WuPalmer [32] and Levenshtein [38] to measure semantic similarity. Kumar et al. [39] introduced the Static UML Model Generator from Analysis of Requirements (SUGAR) tool, which generates both use case and class models while emphasizing natural language requirements. SUGAR aims to bring together both the requirement analysis and design phases by identifying use cases, actors, classes, attributes, and methods, as well as proper class association. They discussed the tool that generates all static UML models in Java in conjunction with Rational Rose.

2.3 Conclusion

There are some significant differences between our method and the aforementioned approaches. The majority of them focused their research on grading the diagram, name (label), and relationship matching rather than the use case specification. They used syntactic and semantic matching to compare names and rarely used NLP methods for texts with more than one word. They also considered structural matching but could not match a use case with the wrong name based on its specification. We also couldn't find a work that matched the text (step) or took flattening steps into account.

On the other hand, our approach uses syntactic and semantic matching as well as NLP methods to match names and texts, e.g., use case multiplicity and steps. We used Stanford CoreNLP to tokenize, lemmatize, recognize names of entities, find parts of speech, and remove unimportant words before matching texts. We also consider the order of steps and match orders in addition to texts. Using use case specifications, we could figure out split use cases and determine which student used the wrong name based on the steps of that use case. So, the use case is correct, and the specification matches, but the student used the wrong name. The method that distinguishes our approach from others' works is flattening. When flattening all use case steps, all steps in the model will be compared, and students will receive points for any correct steps, regardless of whether they have used the correct name for the use case. Furthermore, more than one correct solution can be uploaded, and the student's solution compares with all possible solutions. The instructor is also able to assign points or redesign the marking scheme for each component and feature in the model and can manage the deduction percentage through the configuration settings. Finally, the tool provides the student and instructor with detailed feedback, indicating where the student lost points.

CHAPTER 3

Motivation

In this chapter, we will motivate our approach using a use case diagram modelling the "Elfenroads" game. The method proposed here will be applied in our automated grading system, which will be discussed in depth in Chapter 4.

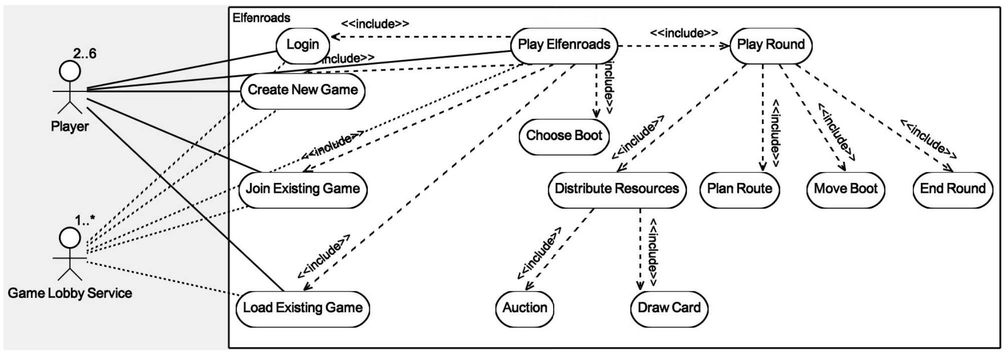


Figure 3.1: Instructor Solution for Elfenroads Use Case Model

The instructor's use case diagram solution, depicted in Fig. 3.1, contains 13 use cases. Play Elfenroads, Login, Create New Game, Join Existing Game, and Load Existing Game are all use cases that are linked to the "Player" as the primary actor and the "Game Lobby Service" as the secondary actor. One student's solution, shown in Fig. 3.2, uses "Aoction" which is the wrong spelling form for "Auction". This student also uses the word "Connect Game" as a name for the use case "Join Game", and uses "Create Game" instead of "Create New Game". Although the words that were used are not the same, we want our matching algorithm to determine that "Connect Game" is a synonym for "Join Game", which is called a semantic match. The use case "Aoction" should be matched with "Auction"

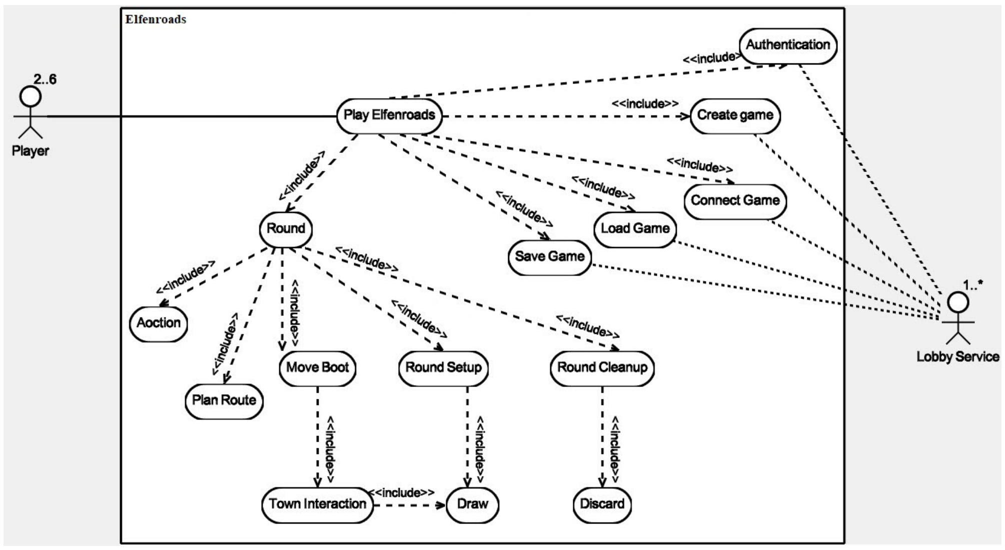
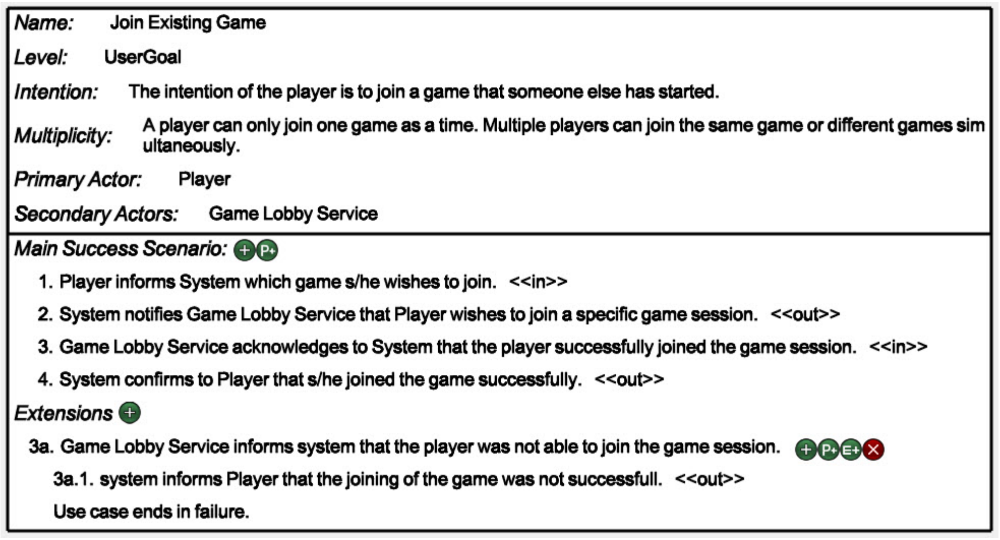


Figure 3.2: Student Solution for Elfenroads Use Case Model

syntactically, even though there is a spelling mistake. Similarly, the use case "Create Game" should be matched with "Create New Game".

In addition to names, each diagram includes some associations. It could be between use cases, e.g., include, which supports the reuse of functionality in a use-case model, between actors, e.g., generalization, and also between actors and use cases, e.g., primary actor. In Fig. 3.1, the instructor considered "Player" as a primary actor for five use cases, on the other hand, the student (Fig. 3.2) assumed "Player" as a primary actor just for the "Play Elfenroads" use case. Furthermore, the "Play Elfenroads" use case in the instructor's solution (Fig. 3.1) contains six include associations to other use cases, while in the student's solution, although there are six include associations, there is not any from the "Play Elfenroads" to the "Choose Boot" use case (the student missed the "Choose Boot" use case). Therefore, we need a strategy that recognise associations between each elements in the diagram. Besides the diagram, each use case has a specification containing all the details about that use case.

Fig. 3.3 shows the instructor's specification for the "Join Existing Game" use case, and Fig. 3.4 is the student's solution for the "Connect Game" use case specification. It is not difficult to compare levels, as there are just three kinds: User Goal, Subfunction, and Summary. On the other hand, Intention and Multiplicity both contain a sentence (text), and almost always, the texts that students write are dif-



Name: Join Existing Game

Level: UserGoal

Intention: The intention of the player is to join a game that someone else has started.

Multiplicity: A player can only join one game as a time. Multiple players can join the same game or different games sim ultaneously.

Primary Actor: Player

Secondary Actors: Game Lobby Service

Main Success Scenario:

1. Player informs System which game s/he wishes to join. <<in>>
2. System notifies Game Lobby Service that Player wishes to join a specific game session. <<out>>
3. Game Lobby Service acknowledges to System that the player successfully joined the game session. <<in>>
4. System confirms to Player that s/he joined the game successfully. <<out>>

Extensions

3a. Game Lobby Service informs system that the player was not able to join the game session.

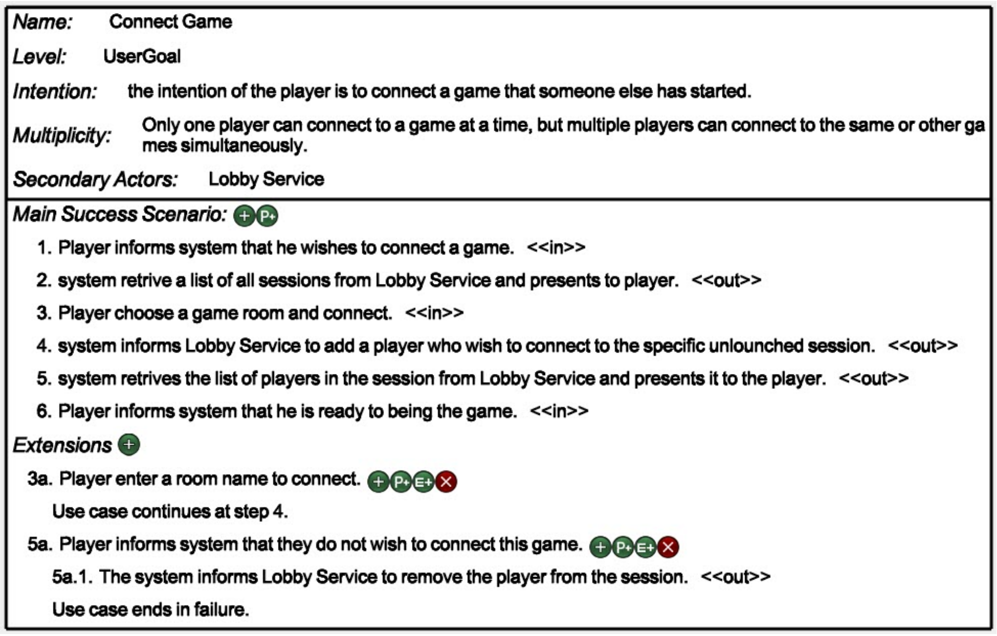
3a.1. system informs Player that the joining of the game was not successfull. <<out>>

Use case ends in failure.

Figure 3.3: Instructor Solution for "Join Existing Game" Use Case Specification

ferent from the instructor's, but they can convey the same meaning. So, matching them syntactically and semantically is not enough, and we need to propose a Natural Language Processing (NLP) approach to match those sentences conceptually. For example, the instructor's solution multiplicity for the "Join Existing Game" use case (Fig. 3.3) is *"A player can only join one game as a time. Multiple players can join the same game or different games simultaneously."*, but the student wrote in the corresponding use case specification (Fig. 3.4) *"Only one player can connect to a game at a time, but multiple players can connect to the same or other games simultaneously."* as a multiplicity. While the sentences are not identical, they share similar meanings and must be considered matched.

The most challenging aspect of comparing use case specifications is matching steps. Each step is a sentence (text) defining the interactions between the system and actors. In the Main Success Scenario, there are three kinds of steps: 1. **Communication** step, which has to be specified in directions: Input or Output. 2. **Context** step, including three kinds of different types: Internal, External, and Control Flow. 3. **Use Case Reference** step, which references another use case to continue from. In Fig. 3.3, there are four Communication steps in the Main Success Scenario, steps 1 and 3 in the "input" direction, which mean the actor has a responsibility toward the system, and steps 2 and 4 in the "output" direction, which show an interaction from the system to the actor. In addition to



Name: Connect Game

Level: UserGoal

Intention: the intention of the player is to connect a game that someone else has started.

Multiplicity: Only one player can connect to a game at a time, but multiple players can connect to the same or other ga mes simultaneously.

Secondary Actors: Lobby Service

Main Success Scenario:

1. Player informs system that he wishes to connect a game. <<in>>
2. system retrive a list of all sessions from Lobby Service and presents to player. <<out>>
3. Player choose a game room and connect. <<in>>
4. system informs Lobby Service to add a player who wish to connect to the specific unlounched session. <<out>>
5. system retrives the list of players in the session from Lobby Service and presents it to the player. <<out>>
6. Player informs system that he is ready to being the game. <<in>>

Extensions

3a. Player enter a room name to connect.

Use case continues at step 4.

5a. Player informs system that they do not wish to connect this game.

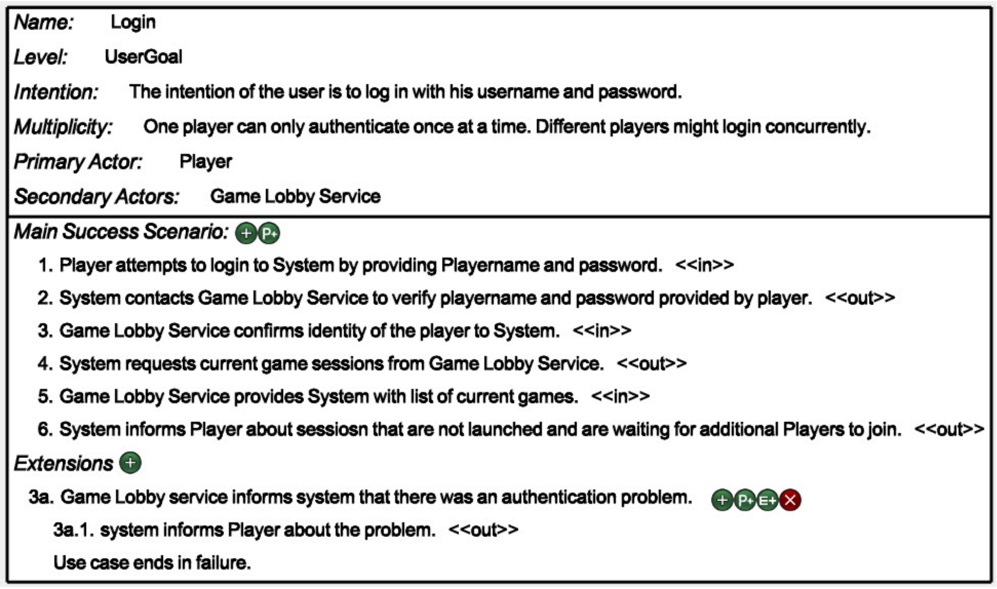
5a.1. The system informs Lobby Service to remove the player from the session. <<out>>

Use case ends in failure.

Figure 3.4: Student Solution for "Connect Game" Use Case Specification

matching steps' texts, directions of steps, and types, the order of those steps is also important, and writing steps in a different order than the instructor' solution is not acceptable to most of the instructors. in Fig. 3.3, the text of the first step is *"Player informs system which game s/he wishes to join."* which is matched with the first step in Fig. 3.4 *"Player informs system that he wishes to connect a game."* Both directions are "input" and they are in the same order. So, we can conclude that the first step in the student's solution for the "Connect Game" use case is completely matched with the first step in the corresponding use case in the instructor's solution.

The "Extensions" part describes what should happen when something goes wrong or if something succeeds but in a different way. Each Extension can include three parts: 1. **Precondition**, which refers to one step in the Main Success Scenario and explains a different way of doing that step. 2. **Alternate Flows**, which are exactly the same as Main Success Scenario steps, describe contracting the system with an actor in a different condition explained in Precondition. 3. **Conclusion Type**, which specifies what will happen after this Precondition and Alternate Flows, could be Failure, Successful, or continuing from one step in the Main Success Scenario. In Fig. 3.3, there is only one precondition, which refers to step 3 in the Main Success Scenario, and one alternate output communication flow. The conclusion type also explains that the use case ends in failure if this condition happens. The corresponding Extension in the student's solution in Fig. 3.4 refers to step 5 in the Main Success Scenario. While the number of steps is different (3 in the instructor's solution and 5 in the student's), we want our matching strategy to identify if the reference steps are matched (in this scenario, step 3 in the instructor's solution and step 5 in the student's solution). The alternate flows and conclusion types also have to be compared if the preconditions are matched.



Name: Login

Level: UserGoal

Intention: The intention of the user is to log in with his username and password.

Multiplicity: One player can only authenticate once at a time. Different players might login concurrently.

Primary Actor: Player

Secondary Actors: Game Lobby Service

Main Success Scenario:

1. Player attempts to login to System by providing Playername and password. <<in>>
2. System contacts Game Lobby Service to verify playername and password provided by player. <<out>>
3. Game Lobby Service confirms identity of the player to System. <<in>>
4. System requests current game sessions from Game Lobby Service. <<out>>
5. Game Lobby Service provides System with list of current games. <<in>>
6. System informs Player about sessiosn that are not launched and are waiting for additional Players to join. <<out>>

Extensions

3a. Game Lobby service informs system that there was an authentication problem.

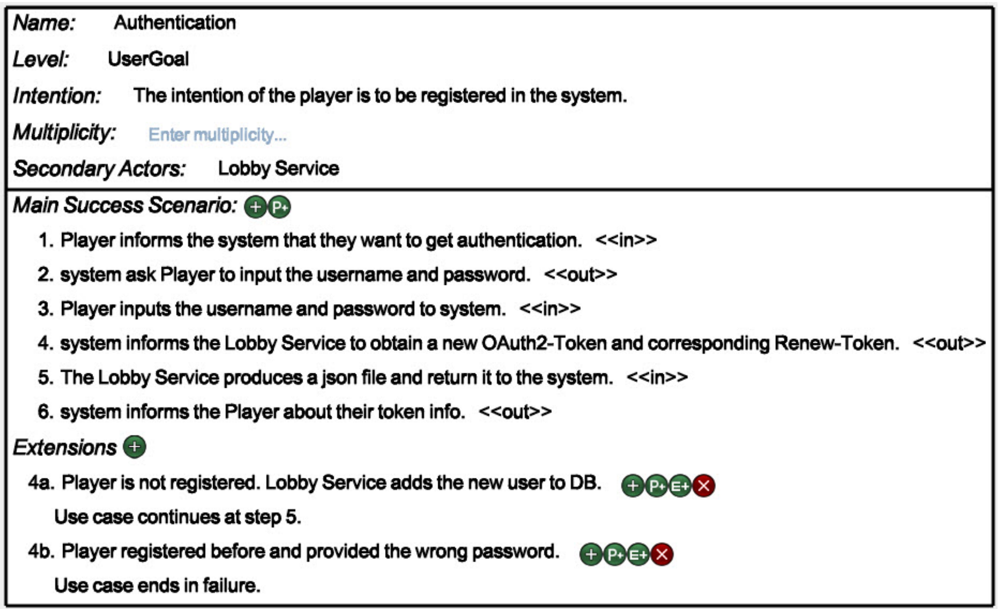
3a.1. system informs Player about the problem. <<out>>

Use case ends in failure.

Figure 3.5: Instructor Solution for "Login" Use Case Specification

In practice, it is possible that the names of two use cases are not matched, neither syntactically nor semantically, but some steps in their use case specification are matched. In the Elfenroads example, there are no use cases in the instructor's solution (Fig. 3.1) that match the "Authentication" use case in the student's solution (Fig. 3.2). Fig. 3.5 demonstrates the instructor's solution for the "Login" use case specification, and Fig. 3.6 shows the student's solution for the "Authentication" use case specification. From the steps the student wrote, we can say some steps like inputting the user name and password and confirming the identity are similar, so these two use cases could be matched, but with the wrong name.

Additionally, it is possible that the number of use cases in the instructor's solution is different than the number of use cases in the student's solution. In this situation,


Name: Authentication
Level: UserGoal
Intention: The intention of the player is to be registered in the system.
Multiplicity: Enter multiplicity...
Secondary Actors: Lobby Service
Main Success Scenario:
1. Player informs the system that they want to get authentication. <<in>>
2. system ask Player to input the username and password. <<out>>
3. Player inputs the username and password to system. <<in>>
4. system informs the Lobby Service to obtain a new OAuth2-Token and corresponding Renew-Token. <<out>>
5. The Lobby Service produces a json file and return it to the system. <<in>>
6. system informs the Player about their token info. <<out>>

Extensions
4a. Player is not registered. Lobby Service adds the new user to DB.
Use case continues at step 5.
4b. Player registered before and provided the wrong password.
Use case ends in failure.

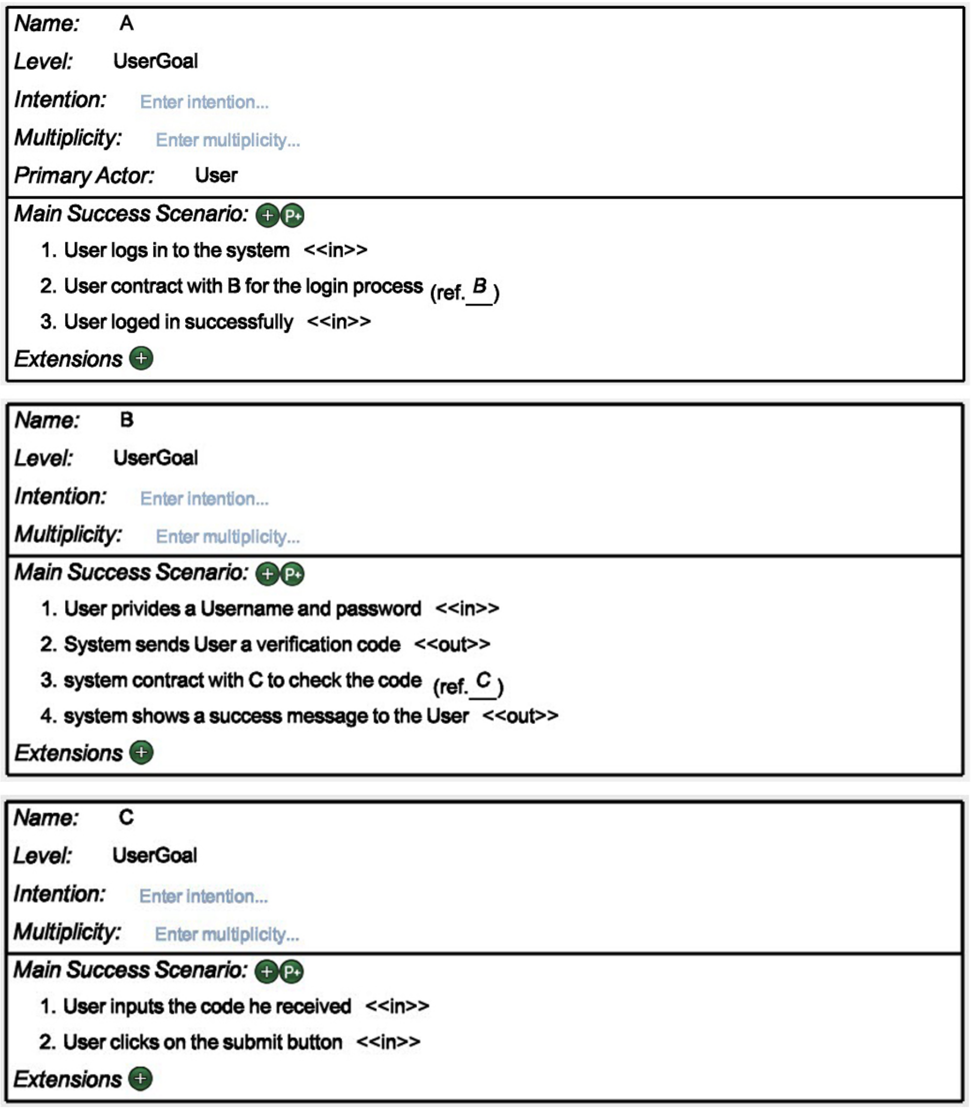
Figure 3.6: Student Solution for "Authentication" Use Case Specification

if we just match use cases one by one, in the end, there would be some use cases for which there are no use cases to match with. One reason could be that the student splits one use case into two or more or uses unnecessary use cases. In the Elfenroads example, in the instructor's solution (Fig. 3.1) there are 13 use cases, while the student uses 15 use cases (Fig. 3.2). In such a case, therefore, there is a need to flatten all steps in the Main Success Scenario and also in the Extensions part. Flattening is based on use case reference steps that can be used under the Main Success Scenario or Extensions. After flattening, each step in the flattened instructor's solution compares to each step in the flattened student's solution one by one and in order.

For example, Fig. 3.7 shows three sample use cases (A, B, and C) with some communication steps. There is one "Include" association from use case A to B and one from B to C. So, in use case A, there would be one use case reference step that refers from A to B and one that refers from B to C in use case B. The flattened Main Success Scenario steps of those use cases are as follows (same process for flattening Extensions):

1=[input, User logs in to the system],

2=[input, User provides a Username and password],


Name: A
Level: UserGoal
Intention: Enter intention...
Multiplicity: Enter multiplicity...
Primary Actor: User
Main Success Scenario:
1. User logs in to the system <<in>>
2. User contract with B for the login process (ref. B)
3. User loged in successfully <<in>>
Extensions

Name: B
Level: UserGoal
Intention: Enter intention...
Multiplicity: Enter multiplicity...
Main Success Scenario:
1. User privides a Username and password <<in>>
2. System sends User a verification code <<out>>
3. system contract with C to check the code (ref. C)
4. system shows a success message to the User <<out>>
Extensions

Name: C
Level: UserGoal
Intention: Enter intention...
Multiplicity: Enter multiplicity...
Main Success Scenario:
1. User inputs the code he received <<in>>
2. User clicks on the submit button <<in>>
Extensions

Figure 3.7: Example of Flattening Steps

3=[output, system sends User a verification code],

4=[input, User inputs the code he received],

5=[input, User clicks on the submit button],

6=[output, system shows a success message to the User],

7=[input, User logged in successfully]

Therefore, our matching strategy should be able to first flatten all steps in all use cases in both instructor and student solutions, and then compare just flattened steps.

3.1 Conclusion

From all the examples above, we identified several matching techniques that should be taken into account in our strategy. First, strict name matching is not sufficient when comparing actors and use cases names. It is essential to combine syntactic matching (eliminating spelling mistakes) and semantic matching (considering synonyms and words with related meanings) for names in our strategy. Also, one Natural Language Processing (NLP) method must be employed to match steps and sentences. Stanford CoreNLP [40] which is one of the best NLP techniques, can be used for this purpose. Second, structural matching strategies should be incorporated, e.g., comparing the content of a use case, and finding similarities based on the steps in the use case specification. Third, the order of the steps is important and should be considered. Fourth, the strategy should handle Extensions steps properly by first checking the precondition's reference step. Fifth, when multiple use cases are associated with each other, we should first flatten all steps of the associated use cases before matching them.

Chapter 4

Grading Algorithm

In this chapter, we discuss the methodologies and algorithms we employed to match instructor and student use case models. The overall algorithm is divided into four principal categories: matching names, matching texts, matching use case features, and flattening steps. In the following, all four categories are explained in detail.

4.1 Matching name

Two alternative methodologies are employed to compare the names of actors and use cases. In the first method, the Levenshtein distance [38] is used to perform syntactic matching to quantify the similarity between the two names. The Levenshtein distance computes the smallest number of single-character modifications necessary to transform one word into another. When the Levenshtein distance between two names is less than 40% of the longest name string length, they are matched. The second strategy involves semantic matching. We employed three WS4J (WordNet Similarity for Java) [31] algorithms that construct a similarity metric between two words based on the WordNet database: Hirst and St-Onge Measure (HSO) [41], Wu and Palmer (WUP) [32], and LIN [42]. The utilisation of three measures in combination outperforms the use of a single measure. If the calculated score is acceptable, the match is recorded [18].

- HSO: This measure compares two concepts based on the path distances be-

Algorithm 1 Compare Actors

```
1: procedure COMPAREACTOR(InstructorModel, StudentModel)
2:     instList← InstructorModel.getActor()
3:     studList← StudentModel.getActor()
4:     for all Actor $A_i$ in instList, $A_s$ in studList do
5:         if syntacticMatch($A_s$.name, $A_i$.name) or semanticMatch($A_s$.name, $A_i$.name) then
6:             return similarityRatio
7:             find among the matches of $A_i$ the $A_s$ that obtains the highest similarityRatio
8:             actorMatchMap($A_i$, $A_s$)
9:     for all Actor $A_i$ in missActorList do
10:        for all Actor $A_s$ in studList do
11:            if no match exists for $A_s$ then
12:                if no match exists for $A_i$ then
13:                    missActorList.add($A_i$)
14:        if actorMatchMap.get($A_i$).equals(actorMatchMap.get($A_s$)) then
15:            $U_i$← $A_i$.getUpperBound() and $U_s$← $A_s$.getUpperBound()
16:            $L_i$← $A_i$.getLowerBound() and $L_s$← $A_s$.getLowerBound()
17:            if ($U_i$ == $U_s$) then
18:                return Actor upper bound is matched
19:            if ($L_i$ == $L_s$) then
20:                return Actor lower bound is matched
21:    return actorMatchMap, missActorList
```

tween them in the WordNet database. It measures the similarity by the number of directions changed, which should be needed to connect one concept 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 [43] and uses the information content (IC) of two concepts to calculate their semantic similarity. The IC of a term is calculated by measuring its frequency in a collection of documents.

The combination of those algorithms returns a similarity ratio value. The actor's or use case's name from the instructor solution is matched with the actor's or use case's name in the student solution, which has the highest ratio.

Algorithms 1 and 2 illustrate the process of matching actors and use case names in detail. The algorithm takes as input the instructor model, InstructorModel, and the student model, StudentModel.

Semantic and syntactic methods are used to compare actors and use cases' names

Algorithm 2 Compare Use Cases

```
 1: procedure COMPAREUSECASE(InstructorModel, StudentModel)
 2:     instList← InstructorModel.getUseCase()
 3:     studList← StudentModel.getUseCase()
 4:     for all UseCase $U_i$ in instList, $U_s$ in studList do
 5:         if syntacticMatch($U_s$.name, $U_i$.name) or semanticMatch($U_s$.name, $U_i$.name) then
 6:             return similarityRatio
 7:             find among the matches of $U_i$ the $U_s$ that obtains the highest similarityRatio
 8:             useCaseMatchMap($U_i$, $U_s$)
 9:     for all UseCase $U_i$ in missUseCaseList do
10:         for all UseCase $U_s$ in studList do
11:             if no match exists for $U_s$ then
12:                 ListI← $U_i$.getCommunicationStep()
13:                 ListS← $U_s$.getCommunicationStep()
14:                 for all CommunicationStep $CS_s$ in ListI,  $CS_i$ in ListS do
15:                     if COMPARETEXT($CS_i$.name, $CS_s$.name) then
16:                         counter++
17:                         if (counter/$CS_i$.size) > 0.33) then
18:                             useCaseMatchMap($U_i$, $U_s$)
19:                     if no match exists for $U_i$ then
20:                         missUseCaseList.add($U_i$)
21:     return useCaseMatchMap, missUseCaseList
```

(line 5). Since it is possible that one actor's or use case's name in the instructor's solution is matched with more than one actor's or use case's name in the student's solution, the algorithm matches the actors or use cases that have the highest similarity ratio (line 7).

After finding matched actors in algorithm 1, if there are any actors A_i in the instructor's solution that could not be matched with actors A_s in the student's solution, those actors are stored as "Missed Actors" (lines 11 to 13). Furthermore, the lower bound (L) and upper bound (U) for those Actors A_i that are matched with A_s are compared (lines 15 to 20).

Since algorithm 2 matched use case names and stored missed or wrong use cases in the missUseCaseList, the communication steps (CS) are checked for all U_i in the missUseCaseList and U_s that could not be matched in the studList(lines 12 to 16). If there is more than 33% (at least a third) similarity between the use cases' communication steps (line 17), we can conclude that the student most likely named the use case incorrectly. We examined different threshold values when comparing use case names, considering their steps rather than the exact name, and we found that the threshold of 33% was the best value. When we examined threshold values greater than 33%, the automated grading was lower, and when examined lower than 33%, the automated grading was higher than manual grading in most cases. Therefore, the U_i from the missUseCaseList is considered matched with the U_s and stored in the useCaseMatchMap (line 18). MissUseCaseList (line 20) stores use cases that could not be matched, either in names or communication steps.

4.2 Matching Text

Based on our approach, each word in one text, e.g., multiplicity, in the instructor's solution must be compared to each word in the corresponding text in the student's solution to calculate the similarity between the two texts. Therefore, we used Natural Language Processing (NLP) methods to split the text and find keywords, as we did not need to match negligible words. While there are other outstanding natural language analysis toolkits, Stanford CoreNLP [40] is one of the most widely used. It simplifies and expedites text data analysis and can extract all kinds of text properties, such as named-entity recognition or part of speech tagging. The linguistic annotation of naturally occurring text may be viewed as a series of changes to the original text, with each stage removing surface distinctions.

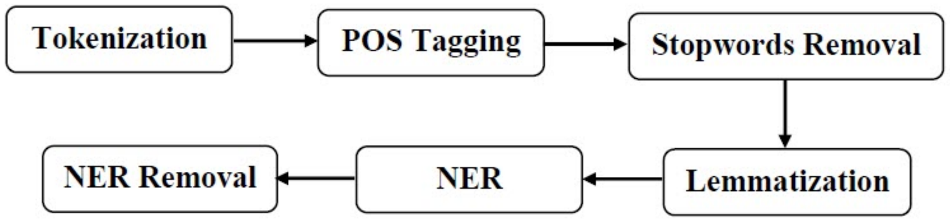


Figure 4.1: Matching Text Flowchart

Fig. 4.1 depicts how we employ text matching in our strategy. Tokenization is one of the first phases of the natural language processing transition [44]. This involves identifying tokens, or those basic units that do not need to be decomposed in subsequent processing. The entity word is one kind of token for NLP, the most basic one [45]. Generally, "Tokenization" is the process of breaking down a phrase, sentence, paragraph, or entire text document into smaller parts, such as individual words or phrases. Tokens are the names given to these smaller units. The tokens could be words, numbers, or punctuation marks. When comparing text, the most important part is the words, followed by numbers. Therefore, first, we split the entire text and find the part of speech of each word. "Part Of Speech" (POS) [46] is a category to which a word is assigned in accordance with its syntactic functions. For example, *address* in the verb form will not be compared to *address* in the noun form. In the English language, there are eight essential parts of speech, including nouns, pronouns, verbs, adjectives, adverbs, prepositions, conjunctions, and interjections. Comparing nouns, verbs, adjectives, adverbs, and also numbers, and eliminating other parts of speech called "Stopwords", e.g., prepositions, leads to better results.

"Lemmatization" is an important data preparation step in many Natural Language Processing (NLP) tasks [47]. It is a text normalization method that converts any type of word to its basic root mode [48]. For instance, the suffixes of the words *studying*, *studies*, and *studied* would change to get the normalized form *study* standing for the infinitive: study. This technique helps us to lemmatize all words in a part of speech and just compare the base format of each word. This is beneficial because we can process syntactic and semantic matching easier, faster, and with fewer mistakes.

"Named Entity Recognition" (NER) [49] is another natural language processing technique that automatically detects and categorizes named entities in text. Entities can be people's names, organizations' names, locations, times, quantities, monetary values, and percentages. We can use named entity recognition to extract key information to comprehend what a text is about. In our strategy, we eliminated those words with inessential NERs when comparing texts, like person, location, and date.

For a sample step, *"The system presents the list of games that have less than the maximum number of players"* if we apply all of the methods mentioned in Fig. 4.1, the matching text algorithm (3) produces the results shown below.

Nouns: [system, list, game, number, player]

Verbs: [present, have]

Adjectives: [less, maximum]

Algorithm 3 Compare Texts

```
 1: procedure COMPARETEXT(instText, studText)
 2:     wordList← Tokenizer(instText or studText.splitBySpace())
 3:     if wordlist contains "and" or "then" then
 4:         return marge step
 5:     for all word W in wordList do
 6:         W.partOfSpeech
 7:         if W.stopwords = true  then
 8:             W.remove
 9:         W.lemmatize
10:         instMap← map<POS , listInstWords>
11:         studMap← map<POS , listStudWords>
12:         for all POS $P_i$ in instMap,  $P_s$ in  studMap do
13:             for all Word $W_i$ in wordList[$P_i$],  $W_s$ in  wordList[$P_s$] do
14:                 if syntacticMatch($W_s$, $W_i$) or semanticMatch($W_s$, $W_i$) then
15:                     if $P_i$ = $P_s$ = (noun or verb) then
16:                         increase counter * 1.5
17:                     else
18:                     increase counter * 1
19:         if counter / listWords [$P_i$].size > 0.55 then
20:             return true
```

As a result, the algorithm simply compares nouns from the instructor's solution to nouns from the student's solution, verbs to verbs, adjectives to adjectives, and numbers to numbers, and finally, a threshold value is generated.

In algorithm 3, the tokenizer method is used to separate the entire text into a list of words (line 2), and then the parts of speech of each word are determined using Stanford CoreNLP (line 6), and those parts of speech that are not necessary to compare (stopword), e.g., prepositions, are eliminated (lines 7 and 8). Finally, the lemmatization procedure converts all words to their basic format (line 9). Lines 10 and 11 maps all words to their parts of speech, and then lines 12 to 14 compare all words W_i in the instructor wordList syntactically and semantically to all words W_s in the student wordListare with the same part of speech. Because nouns and verbs are more significant than other parts of speech, for each matched verb and noun, the counter increases 1.5 times more than for other parts of speech (lines 15 and 16). As mentioned in line 19, two texts are considered matched if more than 55% of all words are matched. In chapter 6, we will discuss how we determined 1.5 times more weight for nouns and verbs, and that 55% is the best value for setting the matching. Recognizing the Merge steps is hardcoded in this algorithm. As line 3 shows, if a step text contains "and" or "then" words, we can say the student potentially merged two or more steps into one.

Algorithm 4 Compare Use Case Features

```
procedure COMPAREFEATURE(useCaseMatchMap)
    if useCaseMatchMap.get($U_i$).equals(useCaseMatchMap .get($U_s$) then
        instLevel← $U_i$.getUseCaseLevel()
        studLevel← $U_s$.getUseCaseLevel()
        for all UseCaseLevel $L_i$ in instLevel, $L_s$ in studLevel do
            if $L_i = L_s$ then
                return Use Case Level is Matched
        instIntention← $U_i$.getUseCaseIntention()
        studIntention← $U_s$.getUseCaseIntention()
        return COMPARETEXTS (instIntention, studIntention)
        instMultiplicity← $U_i$.getUseCaseMultiplicity()
        studMultiplicity← $U_s$.getUseCaseMultiplicity()
        return COMPARETEXTS (instMultiplicity, studMultiplicity)
        instPrimaryActor← $U_i$.getPrimaryActor()
        studPrimaryActor← $U_s$.getPrimaryActor()
        if syntacticMatch (instPrimaryActor, studPrimaryActor) or semanticMatch (instPrimaryActor, studPrimaryActor) then
            return Use Case Primary Actor is Matched
        instSecondaryActor← $U_i$.getSecondaryActor()
        studSecondaryActor← $U_s$.getSecondaryActor()
        for all actor $A_i$ in instSecondaryActor, $A_s$ in studSecondaryActor do
            if syntacticMatch($A_i$, $A_s$) or semanticMatch($A_i$, $A_s$) then
                return Use Case Secondary Actors are Matched
        instInclusion← $U_i$.getInclusion()
        studInclusion← $U_s$.getInclusion()
        for all use case $U_i$ in instInclusion, $U_s$ in studInclusion do
            if syntacticMatch($U_i$, $U_s$) or semanticMatch($U_i$, $U_s$) then
                return Use Case Inclusions are Matched
```

4.3 Matching Use Case Feature

Algorithm 4 matches use case features: Level, Intention, Multiplicity, Inclusion, Primary and Secondary Actors. For matched use cases, first, check the level (lines 3 to 7). Since there are just three kinds of levels, we simply check if they are matched. Levels are matched if the level in U_i is identical to the level in U_s. Intention (lines 8 to 10) and multiplicity (lines 11 to 13) each contain only one sentence (text). So, using algorithm 3, check if studIntention and studMultiplicity are matched with instIntention and instMultiplicity respectively.

Each use case can be connected to just one primary actor, so the primary actor in U_i compares syntactically and semantically with the primary actor in U_s (lines 14 to 17). The strategy for secondary actors is the same as for primary actors. The only difference is that there could be more than one secondary actor per use case and all those actors are compared (lines 18 to 22).

Algorithm 5 Compare Use Case Main Success Scenario

```
 1: procedure COMPAREMAINSUCCESSS(useCaseMatchMap)
 2:     if useCaseMatchMap.get($U_i$).equals(useCaseMatchMap .get($U_s$) then
 3:         instCommStepList← $U_i$.getCommunicationSteps()
 4:         studCommStepList← $U_s$.getCommunicationSteps()
 5:         for all Step $S_i$ in instCommStepList, $S_s$ in studCommStepList do
 6:             if ($S_i$.direction=$S_s$.direction) then
 7:                 if COMPARETEXT($S_i$, $S_s$)and ($S_i$.order=$S_s$.order) then
 8:                     point += $S_i$.point
 9:         instContStepList← $U_i$.getContextSteps()
10:         studContStepList← $U_s$.getContextStep()
11:         for all Step $CS_i$ in instContStepList, $CS_s$ in studContStepList do
12:             if ($CS_i$.type = $CS_s$.type = Internal) or ($CS_i$.type = $CS_s$.type = External) then
13:                 if COMPARETEXT($CS_i$, $CS_s$) then
14:                     point += $CS_i$.point
15:             if ($CS_i$.type = $CS_s$.type=ControlFlow) then
16:                 Tokenize($CF_i$ , $CF_s$)
17:                 instNumbers← Tokenize($CF_i$.Numbers)
18:                 studNumbers← Tokenize($CF_s$.Numbers)
19:                 for all flowNum $FN_i$ in instNumbers, $FN_s$ in studNumbers do
20:                     if all $FN_i$=$FN_s$ and COMPARETEXT($CF_i$ , $CF_s$) then
21:                         point += $CF_i$.point
22:         instListUC← $U_i$.getUseCaseReference()
23:         studListUC← $U_s$.getUseCaseReference()
24:         for all UseCaseReference $R_i$ in instListUC, $R_s$ in studListUC do
25:             if syntacticMatch($R_i$, $R_s$) or semanticMatch($R_i$, $R_s$) then
26:                 point += $R_i$.point
27:     return point
```

Each use case could be associated with one or more other use cases by include association. As a result, all the included use case names in U_i and U_s are compared syntactically and semantically (lines 23 to 27).

Algorithm 5 demonstrates how to match the use case's Main Success Scenario. The Main Success Scenario contains 3 kinds of steps: communication, context, and use case reference. Communication includes input and output directions, and context comprises internal, external, and control flow types. When trying to match communication and context steps, the algorithm first checks if the directions and types of those steps are matched (lines 6, 12, and 15), then matches the steps. As a result, if two steps with the same direction or type are matched using the matching text algorithm (algorithm 3) and if the step in the U_s is in the same order as in the U_i (line 7), we can conclude that those steps are matched. If the context is of the "Control Flow" type (line 15), the step numbers and the number of repetitions of those steps are significant to compare. We find those numbers

Algorithm 6 Compare Use Case Extension

```
1: procedure COMPAREEXTENSION(useCaseMatchMap)
2:     if useCaseMatchMap.get($U_i$).equals(useCaseMatchMap .get($U_s$) then
3:         instListPreCondition← $U_i$.getPreCondition()
4:         studListPreCondition← $U_s$.getPreCondition()
5:         for all preCondition $P_i$ in instListPreCondition,
6:         $P_s$ in studListPreCondition do
7:             if COMPARETEXT($P_i$, $P_s$) and COMPARETEXT ($P_i$.getReferenceStep, $P_s$.getReferenceStep) then
8:                 COMPAREMAINSUCCESSS($P_i$.AlternateFlow, $P_s$.AlternateFlow)
9:         instConclusionType← $U_i$.getConclusionType()
10:        studConclusionType← $U_s$.getConclusionType()
11:        if instConclusionType = studConclusionType = Failure or instConclusionType = studConclusionType = Success then
12:            point += instConclusionType.point
13:        if instConclusionType = studConclusionType = step then
14:            if COMPARETEXT(instConclusionType.getReferenceStep(), studConclusionType.getReferenceStep()) then
15:                point += instConclusionType.point
16:    return point
```

using the Tokenizer method (line 16). Line 20 checks if all numbers in the control flow step in the instructor's solution (FN_i) are matched with the numbers in the control flow step in the student's solution (FN_s), and also checks if the text of the control flow step in CF_i is matched with CF_s. For the use case reference step, the use case referenced in the U_i is syntactically and semantically compared to the use case referenced in the U_s (lines 24 and 25).

Algorithm 6 illustrates the matching of use case Extensions. Each Extension includes a precondition, which refers to a step in the Main Success Scenario. The precondition could have one or more alternative flows. Those alternative flows could be communication, context, and use case reference steps (exactly the same as in the Main Success Scenario). The last part of an Extension specifies in which condition the use case ends: fail, success, or continue from another step in the Main Success Scenario. Therefore, to compare Extensions, first, the precondition must be matched.

To match the precondition, we must determine whether the reference step in U_i is the same as the reference step in U_s. If the reference steps and the text of those preconditions are matched (line 7), the algorithm continues to check alternative flows (line 8). Matching alternate flows is done using algorithm 5, so we refer to that algorithm to match all steps. If both conclusion types in P_i and P_s are Fail or Success (line 11), the algorithm returns true, which means the conclusion type is correct. If the conclusion type in P_i references one step in the Main Success Scenario (line 13) of the use case U_i, P_s also has to be referenced to the same step in the Main Success Scenario in the use case U_s (line 14), otherwise, the conclusion type is incorrect.

4.4 Flattening Step

Most use case diagrams contain more than one use case, and those use cases can be connected using *include* associations. In this case, the Main Success Scenario would also include the use case reference step(s). As there is not just one solution for a particular issue, one student may choose a name for a use case that is not matched with the instructor's use case name (like "Login" in Fig. 3.1 and "Authentication" in Fig. 3.2), or a student may split one use case into two or more use cases (in Fig. 3.2, the student split the "Distribute Resources" use case into "Round Setup" and "Town Interaction" use cases), but it would not be a completely wrong answer. Therefore, if we check steps only in matched use cases, there could be some steps in unmatched use cases that could be the correct answer, and as a result, some points for those steps will be unfairly deducted from the student's mark. To avoid these problems, the algorithm first flattens all the steps in the Main Success Scenario and the Extensions part of the use case specification in all use cases. After flattening, there will be one ordered list of steps for the instructor and one for the student's solution, and the algorithm will simply compare these two flattened lists from the instructor to the student's solution.

We show how to flatten steps in algorithm 7. For this purpose, first, we need to determine the root use case. A root use case is not included in other use cases, but other use cases, directly or indirectly, are included in the root use case. Line 4 checks all use cases UC and stores all that included other use cases in the inclusionList (line 5). As a result, there would be just one use case that does not exist in inclusionList which means it is not included by the other use cases, and that would be the root use case (line 7). The algorithm checks the Main Success Scenario step M_i for root use case UCI (line 8). If it is an instance of a

Algorithm 7 Flattening Steps

```
 1: procedure FLATSTEP(InstructorModel)
 2:     instList← InstructorModel.getUseCase()
 3:     for all UseCase UC in instList do
 4:         if UC.getInclusion = true then
 5:             inclusionList.add(UC.getIncludedUseCase)
 6:             for all UseCase UCI in instList do
 7:                 if UCI does not exist in inclusionList then
 8:                     instStepList← UCI.getMainSuccessScenario()
 9:     for all mainSuccessScenario $M_i$ in instStepList do
10:         if $M_i$ instanceof CommunicationStep then
11:             stepFlat.put(communicationStep.stepText, Direction)
12:         else if $M_i$ instanceof ContextStep then
13:             stepFlat.put(communicationStep.stepText, Type)
14:         else if $M_i$ instanceof UseCaseReferenceStep then
15:             useCase U = UseCaseReferenceStep
16:             instStepList← U.getMainSuccessScenario()
17:             Continue at line 9
18:             if step s is the last step in use case U then
19:                 Go to the previous use case
20:                 Continue at line 10
21:     return stepFlat
```

communication step (line 10), step text and direction are added to the step flat (line 11). If M_i is an instance of a context step (12), the step text and type are added to the stepFlat (line 13). Finally, if M_i is an instance of a use case reference step (line 14), the Main Success Scenario for referenced use case U is acquired (line 16), and the algorithm gets back and continues from line 9 for use case U, and all the Main Success Scenario steps in use case U are added to the stepFlat. If the Main Success Scenario in one use case, i.e., the last use case, does not contain the use case reference step, and all steps of that use case are added to the stepFlat, the algorithm goes back to the last checked use case and continues from the step after the reference step (line 20). This process continues until the algorithm goes back to the root use case and all steps are added to the stepFlat.

Algorithm 8 flattens use case Extensions. The root use case is determined in the same way as in algorithm 7 (line 3). If the extension step E_i of the root use case UCI is an instance of precondition, the precondition step text and the referenced step text are added to the extensionFlat (lines 7 and 8). three conditions exist if E_i is an instance of conclusion type: failure, success, and step. For "failure" and "success", the exact type will be added to the extensionFlat (lines 10 and 11), but if the conclusion type refers to one step from the Main Success Scenario to

Algorithm 8 Flattening Extensions

```
 1: procedure FLATEXTENSION(InstructorModel)
 2:     instList← InstructorModel.getUseCase()
 3:     "Find root use case UCI in instList"
 4:     instExtList← UCI.getExtension()
 5:     instMainList← UCI.getMainSuccess()
 6:     for all extensionStep $E_i$ in instExtList do
 7:         if $E_i$ instanceof PreCondition then
 8:             extensionFlat.put($E_i$.referenceStep + $E_i$.stepText)
 9:         else if $E_i$ instanceof ConclusionType then
10:             if $E_i$ = Failure then extensionFlat.put("Failure")
11:             else if $E_i$ = Success then extensionFlat.put("Success")
12:             else if $E_i$ = Step then  extensionFlat.put(Referenced step)
13:         else if $E_i$ instanceof AlternateFlow then
14:             for all AlternateFlow AF in $E_i$ do
15:                 if AF instanceof CommunicationStep then
16:                     extensionFlat.put(commStep.stepText, Direction)
17:                 else if AF instanceof ContextStep then
18:                     extensionFlat.put(ContextStep.stepText, Type)
19:                 else if AF instanceof UseCaseReferenceStep then
20:                     useCase U = UseCaseRefStep
21:                     instList← U.getExtension()
22:                     Continue at line 4
23:         for all mainSuccess MF in instMainList do
24:             if $M_i$ instanceof useCaseReferenceStep then
25:                 UseCase U = Mi.getUseCaseRef
26:                 Continue at line 4
27:         if step s is the last step in use case U then
28:             Go to the previous use case
29:             Continue at line 7
30:     return extensionFlat
```

continue, that step will be added to the extensionFlat (line 12).

If E_i is an alternate flow AF and AF is an instance of a communication step (line 15), the step text and direction are added to the extensionFlat (line 16). If AF is an instance of a context step (line 17), the step text and type are added to the extensionFlat (line 18). On the other hand, if AF is an instance of a use case reference step (line 19), the algorithm continues from line 4 for the use case that is referenced here (line 22). In addition to the Extensions, the Main Success Scenario must be reviewed to determine if any use case reference steps exist (lines 23 and 24). If there is any referenced use case U, the algorithm continues for the use case U from line 4 (line 26), and all extensions for those use cases are added to extensionFlat. In the end, after adding the last extension step in the last use case to the extensionFlat, the algorithm gets back to the previous use case and finds extensions after the use case reference step (lines 27 to 29). This process continues

Algorithm 9 Compare Flatted Steps

```
1: procedure COMPAREFLATTEDSTEP(InstStepFlat, StudStepFlat)
2:     for all Step $S_i$ in InstStepFlat $S_s$ in StudStepFlat do
3:         if $S_i$.direction = $S_s$.direction then
4:             if COMPARETEXT($S_i$, $S_s$) and $S_i$.order = $S_s$.order then
5:                 point += $S_i$.point
6:                 $S_i$.remove
7:                 $S_s$.remove
8:     return point
```

Algorithm 10 Compare Flatted Extension

```
1: procedure COMPAREFLATTEDEXTENSION(instExtFlat, studExtFlat)
2:     for all Step $S_i$ in  instExtFlat, $S_s$ in studExtFlat do
3:         if $S_i$ and $S_s$ instance of PreCondition then
4:             if COMPARETEXT($S_i$.getRefStep(),$S_s$.getRefStep()) then
5:                 if COMPARETEXT($S_i$, $S_s$) and ($S_i$.order=$S_s$.order) then
6:                     point += $S_i$.point
7:         else if $S_i$ and $S_s$ instance of ConclusionType then
8:             if $S_i$.ConclusionType = $S_s$.ConclusionType = success
9:              and $S_i$.order = $S_s$.order then
10:                point += $S_i$.point
11:            else if $S_i$.ConclusionType = $S_s$.ConclusionType = Failure
12:             and $S_i$.order = $S_s$.order then
13:                point += $S_i$.point
14:            else if $S_i$.ConclusionType = $S_s$.ConclusionType = Step  and
15:             COMPARETEXT($S_i$.referenceStep, $S_s$.referenceStep) and
16:             $S_i$.order = $S_s$.order then
17:                point += $S_i$.point
18:        else if $S_i$ and $S_s$ instance of AlternateFlow and
19:             $S_i$.direction = $S_s$.direction then
20:            if COMPARETEXT($S_i$, $S_s$) and $S_i$.order = $S_s$.order then
21:                point += $S_i$.point
22:            $S_i$.remove
23:            $S_s$.remove
24:    return point
```

until all extensions are reviewed and added to the extensionFlat in order.

After executing algorithm 7 for both the student and instructor's solutions, there would be a list of Main Success Scenario steps in InstStepFlat and StudStepFlat. Algorithm 9 matches all flatted steps S_i in InstStepFlat to flatted steps S_s in StudStepFlat. We just need to compare the pairs of step texts that have the same directions (line 3). If the texts are matched and the steps are in the same order (line 4), we consider the steps to be matched. Finally, the matched steps are removed from the InstStepFlat and StudStepFlat to avoid being reviewed more than once (lines 6 and 7).

Now, when instExtFlat and studExtFlat exist after the implementation of algorithm 8, we can compare those flattened Extensions using algorithm 10. If the steps S_i and S_s are instances of precondition (line 3), using the matching text algorithm (line 4) checks if the referenced step's text of S_i and S_s are matched. If matched, check if the texts and orders of S_i and S_s are also matched (line 5). If S_i and S_s are instances of the same conclusion type for the same precondition, and both types are success (line 8) or failure (line 11), or if they refer to the same step in the Main Success Scenario to continue from (line 14), we can say those conclusion types are matched. If S_i and S_s are instances of alternate flow (line 18), we check whether the directions are matched. If directions are the same (line 19) and texts of steps S_i and S_s are matched using algorithm 3 (line 20), and they are in the same order, we can conclude that S_i and S_s are matched. Finally, those matched steps will be eliminated from instExtFlat and studExtFlat in order to avoid getting assessed again (lines 22 and 23).

4.5 Conclusion

When comparing use case diagram models, we consider different kinds of variations. This chapter outlines the grading algorithm, which is divided into four parts. When comparing use cases, we considered structural matching, like recognizing split use cases, or the correct use case for which the student has given the wrong name. We demonstrated how matching names using syntactic and semantic matching, as well as using NLP when comparing texts, plays an important role. We also offered to flatten the Main Success Scenario and Extensions steps in instructor and student solutions before matching them. This process causes students to lose fewer marks if they split one use case.

Chapter 5

Grading Architecture and Tool Support

In this chapter, we discuss how to redesign the architecture of the class diagram grading tool so that it can be used for developing grading tools for other modeling languages, e.g., use case diagrams.

The metamodels and grading tools that support the automated grading approach are also discussed in this chapter. The TouchCORE tool [19] incorporates the grading method and metamodels. We cover the automatic grading process in our tool and several essential features in further depth by presenting the graphical user interface (GUI) of the TouchCORE tool. Then we go through the configuration panel, which allows users to customize different grading procedures for different instructors.

5.1 Automated Grading Architecture

This part goes through the grading tool that we used to assist with the automated grading technique. We extended a tool called TouchCORE [50] to employ the previously proposed grading metamodels and implement the matching method. TouchCORE, is a multi-touch enabled software design modelling tool aimed at developing scalable and reusable software design models following the concern driven software development paradigm [19]. TouchCORE is built on the Eclipse Modeling

Framework (EMF)[51] and the Multi-Touch Library for Java (MT4J)[52].

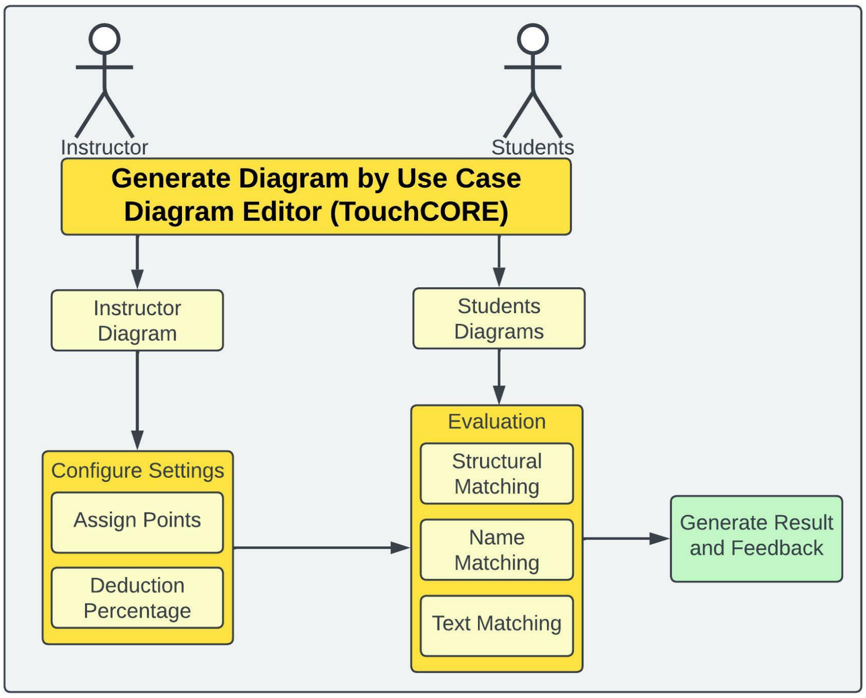


Figure 5.1: TouchCORE Architecture for Automated Grading of Use Case Models

Fig. 5.1 shows TouchCORE grading architecture. In the first step, the instructor and students generate their diagrams using the tool. Then, the instructor is able to assign points to each feature in the diagram, e.g., use case, actor, and their associations, and all use case specifications, e.g., steps. The instructor is also able to modify deduction percentages for each feature through the configuration settings panel. After setting these configurations, the tool compares the instructor's model and the students' model by matching structure, name, and text. In the end, the tool generates a result and provides feedback to the students and instructor.

We proposed some generic packages for those methods that could be reused in other modelling languages. For example, matching names and text and some structural matching from the evaluation part can be reused in the activity diagram. The configuration setting panel, which is also used in the automated grading of all modelling languages, can be modified and reused easily in our approach.

5.2 Grading Metamodels

This chapter goes over the metamodels that have been created by Bian et al. [18, 24] to work for the automatic grading of Class Diagrams. This grading metamodel can be applied to any modelling language with a metamodel expressed in ECore. They defined separate metamodels rather than augmenting the class diagram metamodel to support the definition of grades and matchings for model elements [24]. This was less invasive, as it left the class diagram metamodel unchanged, and hence all existing modelling tools could continue to work. So, we reused their metamodels that were used to store a model's grade as well as mappings between the student's model elements and the instructor's model elements to aid in the automatic grading of use case diagrams.

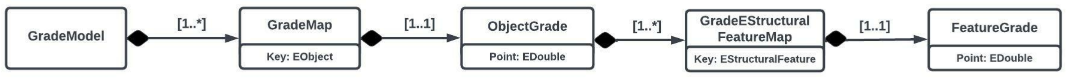


Figure 5.2: Grade Metamodel

Fig. 5.2 displays the grade metamodel. This metamodel augments any model expressed in ECORE with grades. The *GradeModel* maps Grade, which contains a *key* attribute (*EObject*), to *ObjectGrade* including a *point* attribute. In this way, points can be assigned to any modelling element in a language that is modelled using a metaclass. *ObjectGrade* maps *StructuralFeature* which contains a *key* attribute (*EStructuralFeature*) to *FeatureGrades* which contain a *point* attribute. So, points could also be assigned to any structural features in order to award points for the properties of modelling elements.

For example, the instructor may want to consider 2 points for the *Player* actor in Fig. 3.1 and an additional point if the multiplicity of the actor is *2..6*. In this case, one would create an *ObjectGrade* and insert it into the *GradeMap* using as a key *Actor*, and assign the points value 2. Additionally, one would create a *FeatureGrades*, and insert it into the *GradeStructuralFeatureMap* using a key *ActorMultiplicity* of *Player* actor.

The classroom metamodel is illustrated in Fig. 5.3. The *ClassRoomModel* maps

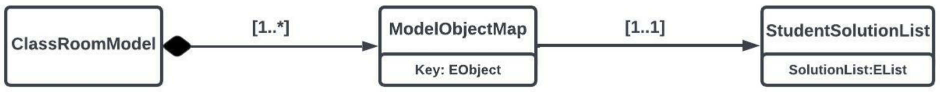


Figure 5.3: Classroom Metamodel

ModelObject which contains a *key* attribute (*EObject*) to *StudentSolutionList* containing the *SolutionList* attribute. This is used to store the mappings identified after running the automatic grading algorithm. The classroom metamodel associates each model element in the instructor solution (*EObject key*) to a list of possible matched model elements in the student solution (*Elist SolutionList*) by the matching algorithm. The instructor is also able to update the grades of the students in case he or she decides to modify the point weights in his or her solution.

5.3 Grading Tool Support

In this chapter, we discuss the graphical user interface of the grading tool that we modified to support the automated grading of the use case diagram. So, we reused the grading metamodel described in the previous section and extend a tool called TouchCORE [19]. We also describe how to assign points, configure deduction, and grade a use case model using the tool in detail.

The main graphical user interface (GUI) of the TouchCORE is seen in Fig. 5.4. By clicking (tapping) on the *New Concern* (the button likes +) button, a user can choose a folder to create a model. The user may also load existing models created earlier from a file browser by choosing the *Load Concern* button. The *Create Homework* button directs the user to a view in which instructor and student models can be uploaded and prepared for grading. By choosing the *Load Homwork* button, a previously saved Homework can be browsed and opened.

Fig. 5.5 shows one new Homework created by a user. By clicking on the *Import Student* button, the instructor can upload one student's model, or by clicking on the *Import All Students* button, he or she can import all students' models at once. The instructor is able to create a solution by clicking on the *Create Teacher Solu-*

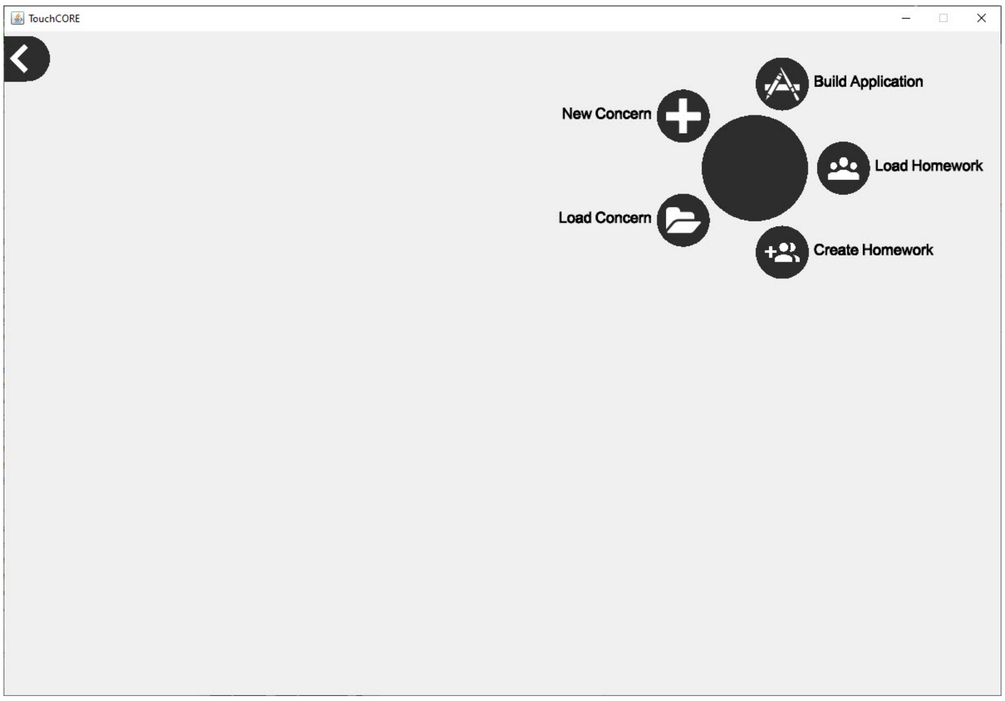


Figure 5.4: TouchCore Main Graphical User Interface

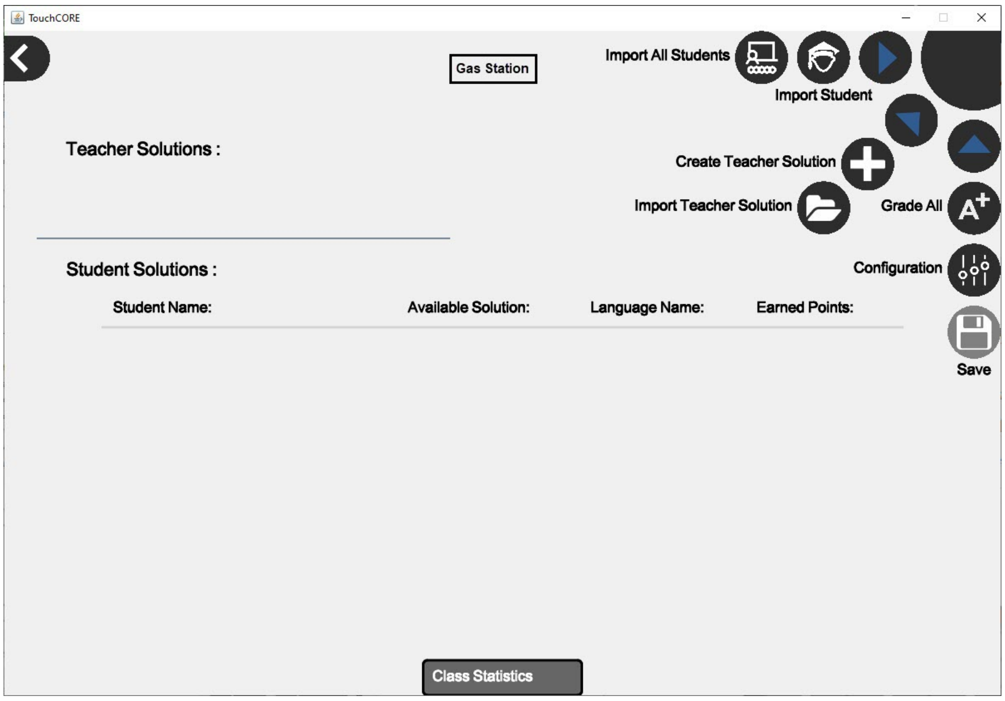


Figure 5.5: TouchCore New Homework View

tion or upload a previously created solution by choosing *Import Teacher Solution* button.

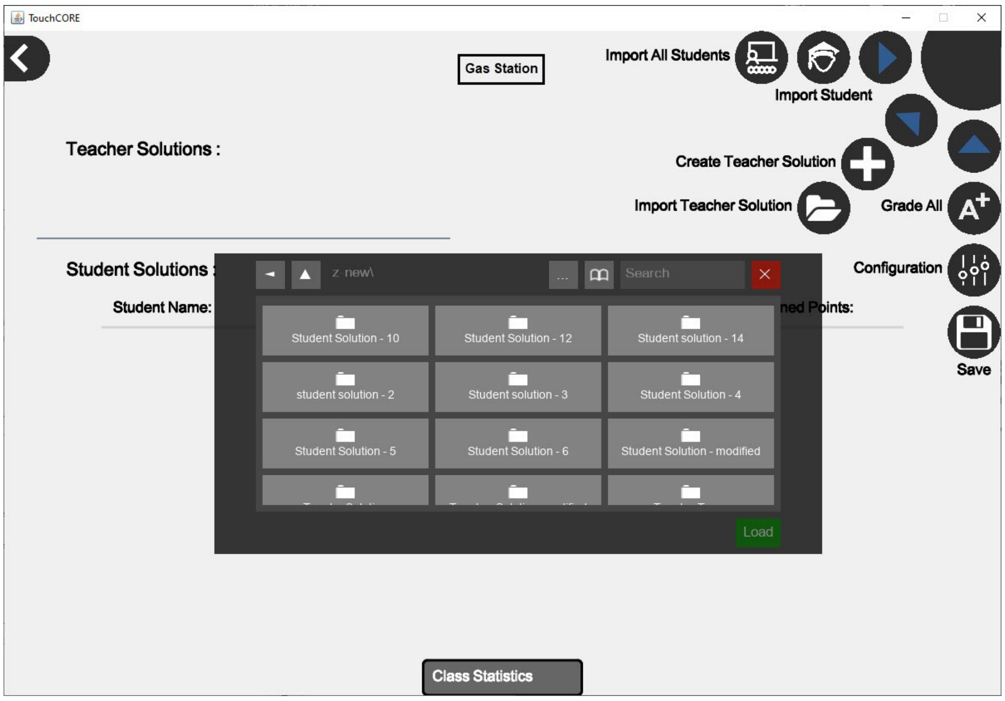
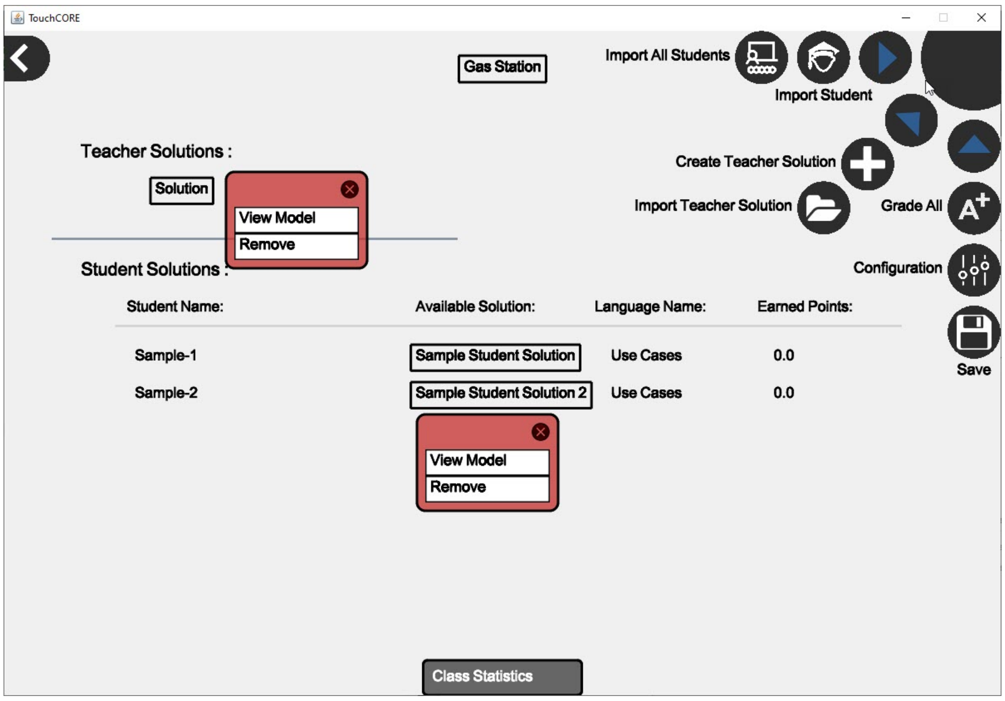


Figure 5.6: Student Models in TouchCORE File Browser

Fig. 5.6 depicts various student models after choosing the *Import Student* button. There is one folder per student, and the instructor can open students' models from this view easily. The same browsing file appears by clicking on the *Import Teacher Solution*. The user can upload all available student solutions, and as there could be more than one solution for a particular problem, the user can also upload all possible solutions under *Teacher Solution*.

After choosing a folder and clicking on the file that contains the model for both instructor and student solutions, those models upload to the Homework as shown in Fig. 5.7. In this case, one instructor solution and two student solutions are uploaded. The user can press and hold on to each model to see a new menu. By selecting *Remove*, the selected model is removed, and by selecting *View Model* a use case similar to that shown in Fig. 5.8 appears. This figure shows one student's use case model for the Gas Station assignment.

Fig. 5.9 depicts the Gas Station solution designed by the instructor. It can be



Student Name:	Available Solution:	Language Name:	Earned Points:
Sample-1	Sample Student Solution	Use Cases	0.0
Sample-2	Sample Student Solution 2	Use Cases	0.0

Figure 5.7: Instructor and Student Models Displayed in Homework

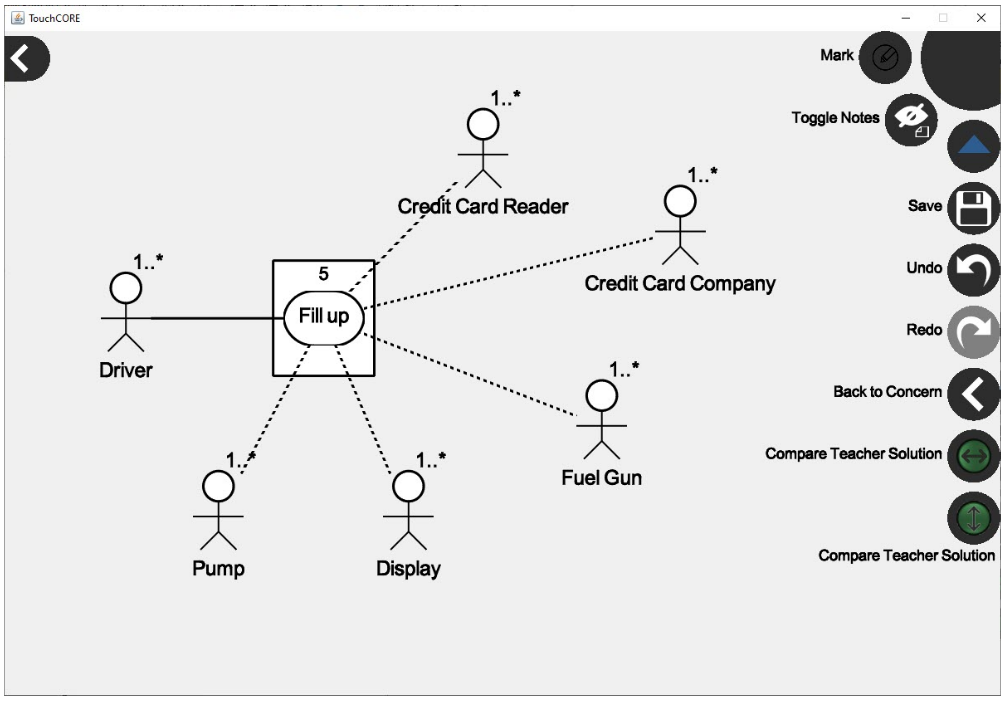


Figure 5.8: Sample Student Model Displayed in TouchCore

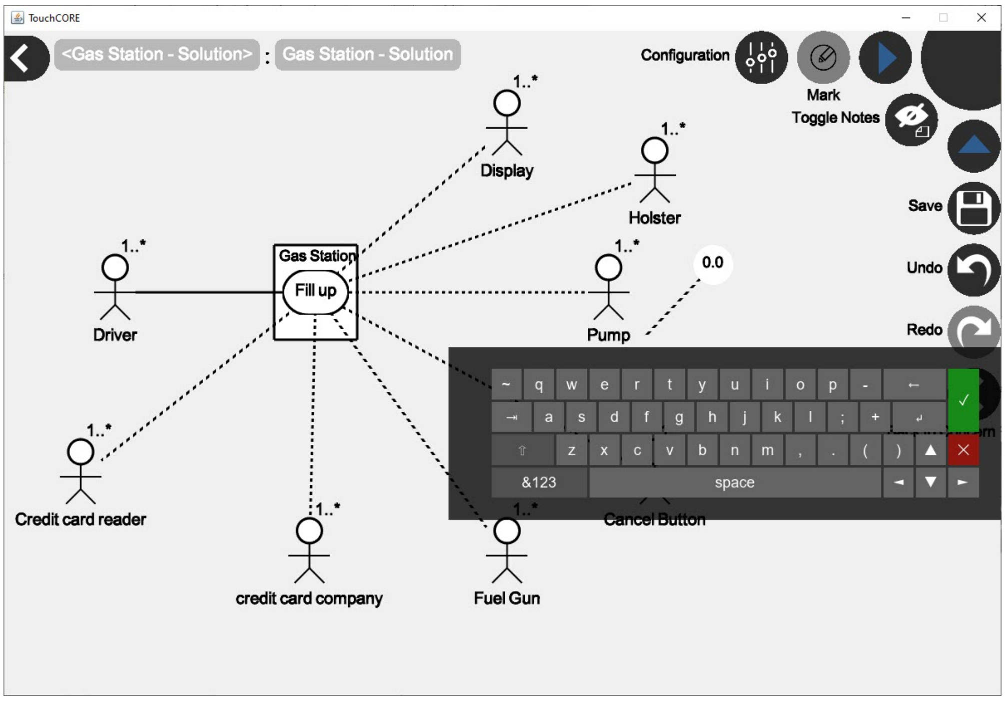
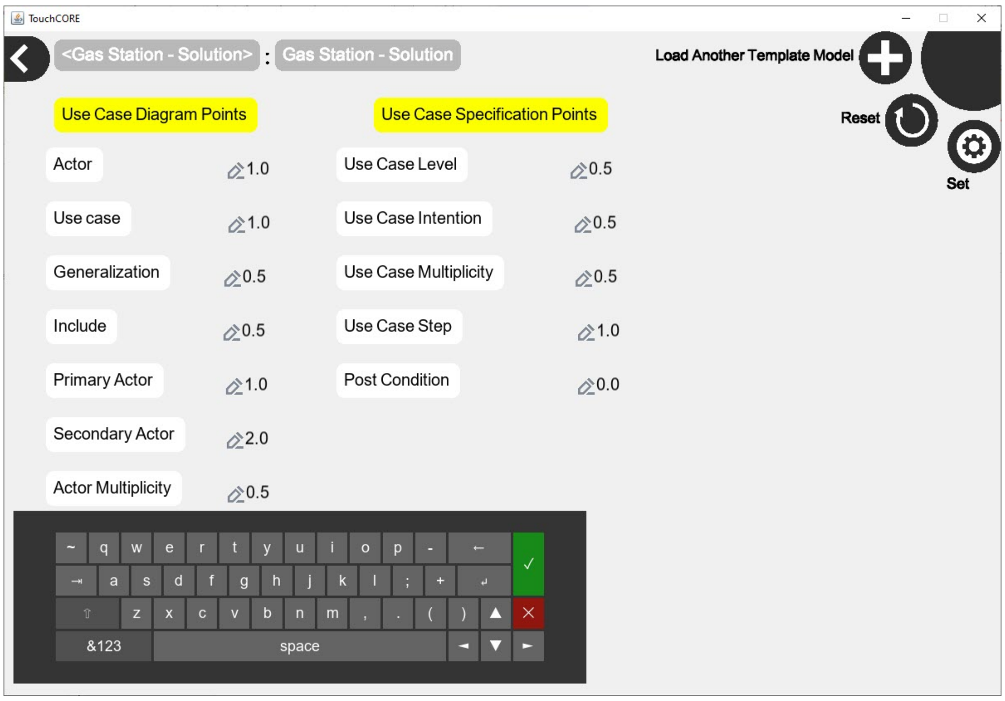


Figure 5.9: Instructor Solution Model Displayed in TouchCore

opened by clicking on the *Import Teacher Solution* or create it by choosing the *Create Teacher Solution* button. The instructor can assign points to each element from this view. One technique for assigning points to model elements is shown in this figure. The user can enter the marking mode by clicking the *Mark* button. In the marking mode, the user can press and hold on any model element, such as the actor "Pump" in Fig. 5.9, for a few seconds to correctly assign the grade to the model element. The grade is presented in a circle next to the model element, with a dotted line connecting the model element to the grade. The instructor can also drag the grade in the tool to move it around. Each grade has a default value of zero, but the instructor can modify the grade value by double-clicking the grade to open the keyboard and changing the grade from zero to, for example, one. After assigning points, the user can tap on the *Mark* button again, and the button's colour will change to black, indicating that the user has turned off the mark mode.

There is also another technique that is faster and easier to assign points to all model elements. By selecting the *Configuration* button in the instructor's view (Fig. 5.9), a panel (Fig. 5.10) appears in which the use case diagram and use case


TouchCORE
<Gas Station - Solution> : Gas Station - Solution
Load Another Template Model
Reset
Set
space
&123

Use Case Diagram Points		Use Case Specification Points	
Actor	1.0	Use Case Level	0.5
Use case	1.0	Use Case Intention	0.5
Generalization	0.5	Use Case Multiplicity	0.5
Include	0.5	Use Case Step	1.0
Primary Actor	1.0	Post Condition	0.0
Secondary Actor	2.0		
Actor Multiplicity	0.5		

Figure 5.10: Configuration Panel to Set Initial Points

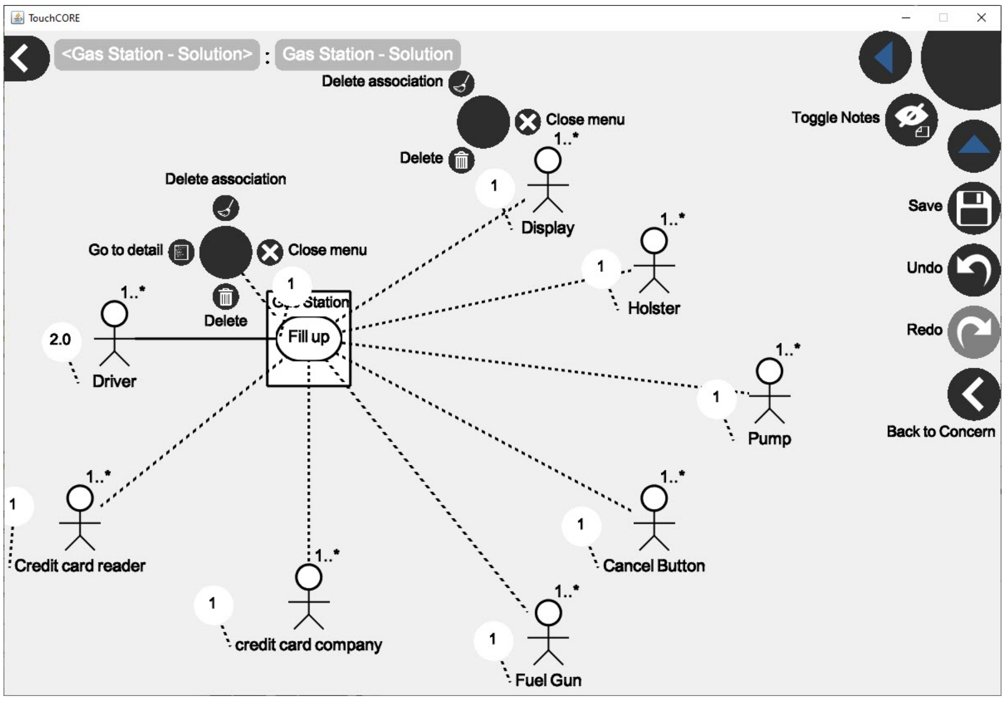


Figure 5.11: Assigned and Modification Points in the Diagram

specification initial points can be modified. In this way, the instructor can assign points to any model element in the use case diagram and specification. When the instructor assigns points to all model elements, he or she can select the *Set* button, and those points will be applied to all features in the instructor's model. Fig. 5.11 demonstrates the assigned points to the diagram. All points in circles can also be modified by double-clicking on the points. For example, the instructor decided to modify *Driver* actor's points from 1 to 2.

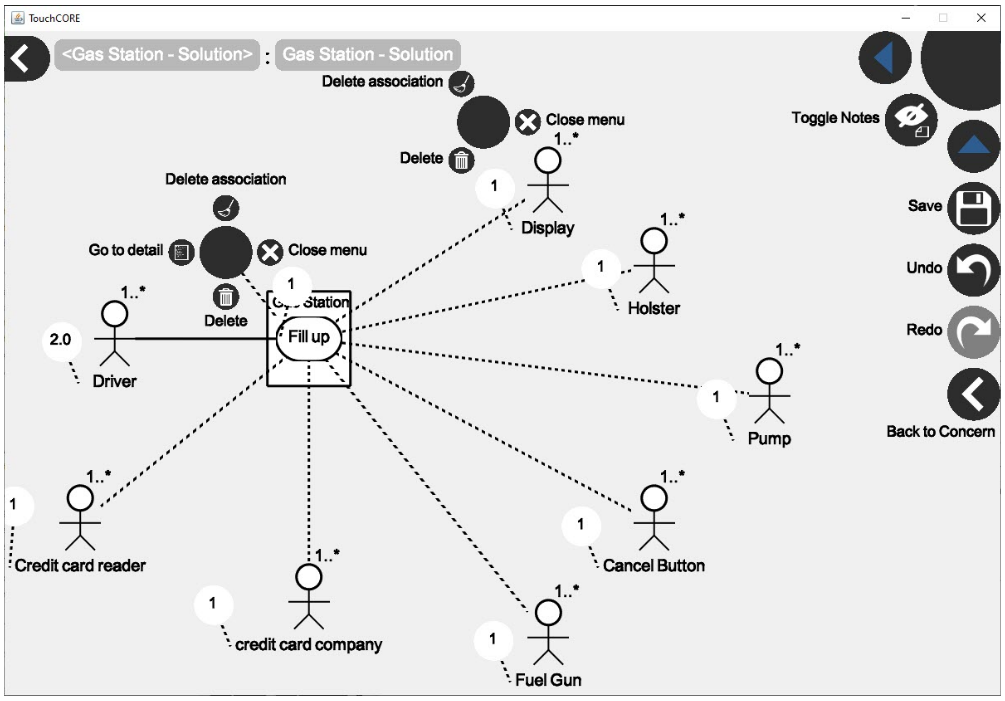
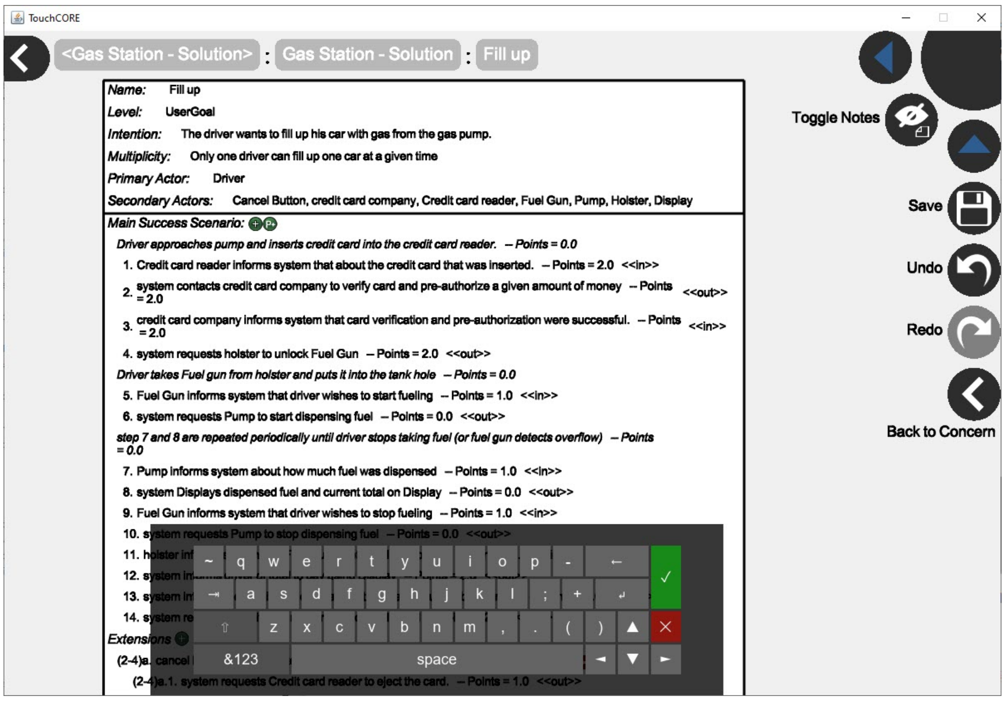


Figure 5.12: Use Case Model Modification

If the user presses and holds on each element in the diagram, some new options are displayed, as shown in Fig. 5.12. There are three options for actors, *Delete* button deletes the actor and all its associations with other elements in the diagram. By clicking on the *Delete association*, the user can choose one association from all connected associations to that actor and delete it. The *Close menu* button closes this menu. There is one more option for use cases (*Go to detail* button), which directs the user to the selected use case specification.

Fig. 5.13 shows the *Fill Up* use case specification. It includes Name, Level, Intention, Multiplicity, and Primary and Secondary actors of the selected use case. The most important parts of the use case specification are steps in the



Name: Fill up

Level: UserGoal

Intention: The driver wants to fill up his car with gas from the gas pump.

Multiplicity: Only one driver can fill up one car at a given time

Primary Actor: Driver

Secondary Actors: Cancel Button, credit card company, Credit card reader, Fuel Gun, Pump, Holster, Display

Main Success Scenario:

Driver approaches pump and inserts credit card into the credit card reader. -- Points = 0.0

1. Credit card reader informs system that about the credit card that was inserted. -- Points = 2.0 <<in>>
2. system contacts credit card company to verify card and pre-authorize a given amount of money -- Points = 2.0 <<out>>
3. credit card company informs system that card verification and pre-authorization were successful. -- Points = 2.0 <<in>>
4. system requests holster to unlock Fuel Gun -- Points = 2.0 <<out>>

Driver takes Fuel gun from holster and puts it into the tank hole -- Points = 0.0

5. Fuel Gun informs system that driver wishes to start fueling -- Points = 1.0 <<in>>
6. system requests Pump to start dispensing fuel -- Points = 0.0 <<out>>

step 7 and 8 are repeated periodically until driver stops taking fuel (or fuel gun detects overflow) -- Points = 0.0

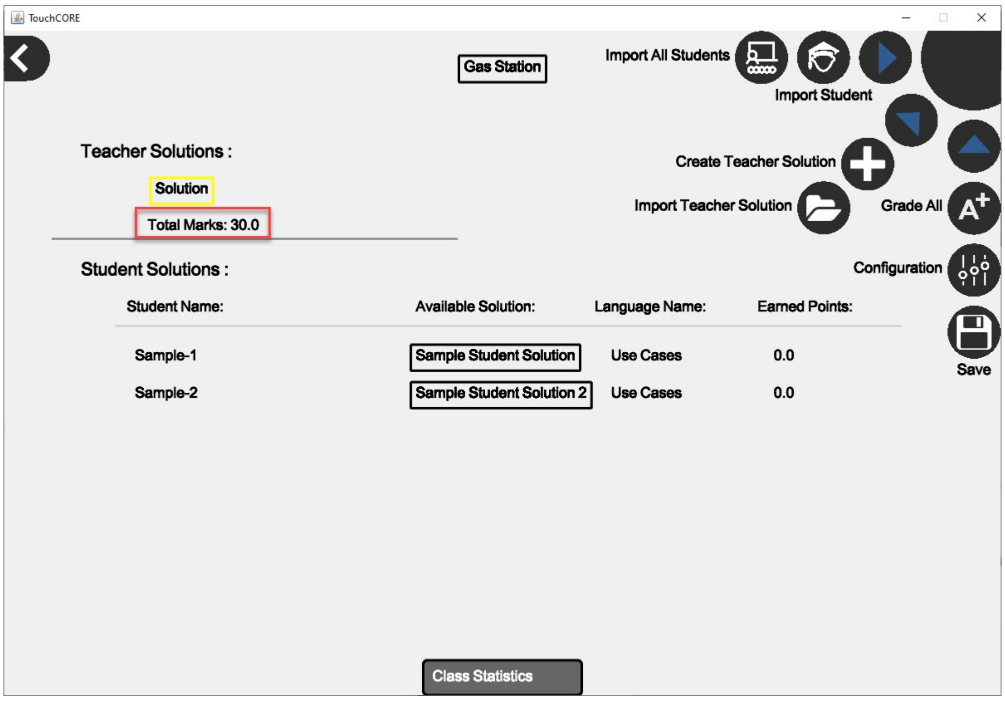
7. Pump informs system about how much fuel was dispensed -- Points = 1.0 <<in>>
8. system Displays dispensed fuel and current total on Display -- Points = 0.0 <<out>>
9. Fuel Gun informs system that driver wishes to stop fueling -- Points = 1.0 <<in>>
10. system requests Pump to stop dispensing fuel -- Points = 0.0 <<out>>
11. holster inf
12. system in
13. system in
14. system re

Extensions

(2-4)a. cancel

(2-4)a.1. system requests Credit card reader to eject the card. -- Points = 1.0 <<out>>

Figure 5.13: Steps' Points Modification in Use Case Specification



Teacher Solutions :

Solution

Total Marks: 30.0

Student Solutions :

Student Name:	Available Solution:	Language Name:	Earned Points:
Sample-1	Sample Student Solution	Use Cases	0.0
Sample-2	Sample Student Solution 2	Use Cases	0.0

Figure 5.14: Show Total Marks in Homework

Main Success Scenario and Extensions. After assigning step points through the configuration panel shown in Fig. 5.10, all steps will be assigned the same points. These points are shown at the end of each step in the use case specification (Fig 5.13). The instructor can modify any points assigned to steps in the use case specification panel. As can be seen, the instructor decided to assign 2 and 1 points for some steps, and no points for others based on their importance. Finally, the instructor can click on the *Save* button to save all assigned points for all steps, and using *Back to Concern* button, go back to the Homework view (Fig. 5.14). When coming back to the Homework, the user can see the total assigned marks by double-clicking on the instructor's model under *Teacher Solutions*. In this case, the *Total Marks* is 30.

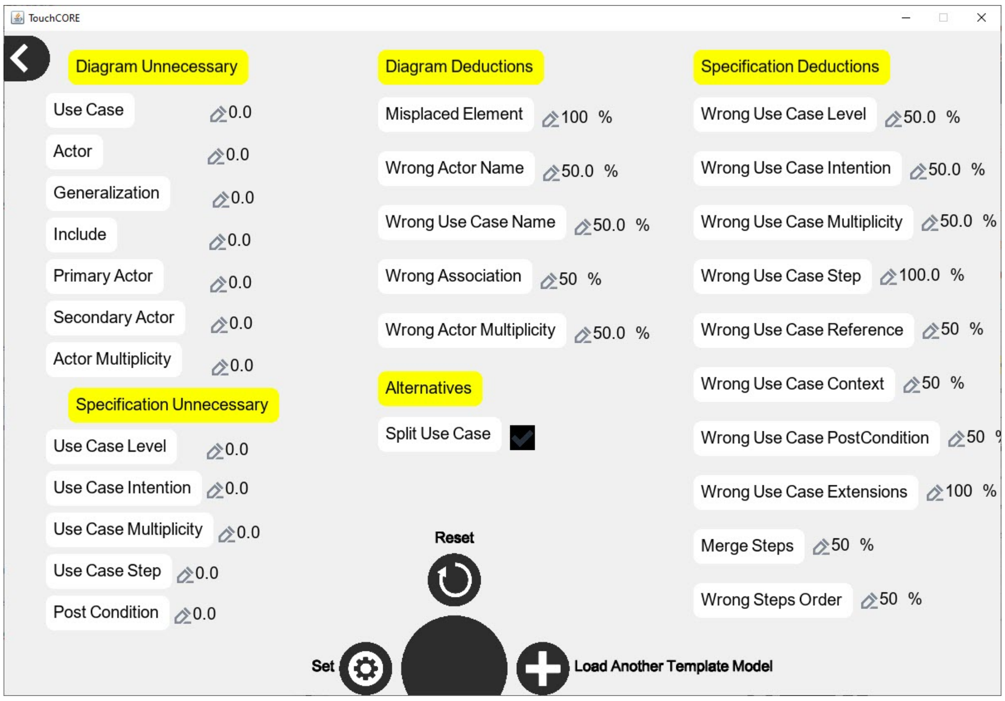


Diagram Unnecessary
Use Case 0.0
Actor 0.0
Generalization 0.0
Include 0.0
Primary Actor 0.0
Secondary Actor 0.0
Actor Multiplicity 0.0

Specification Unnecessary
Use Case Level 0.0
Use Case Intention 0.0
Use Case Multiplicity 0.0
Use Case Step 0.0
Post Condition 0.0

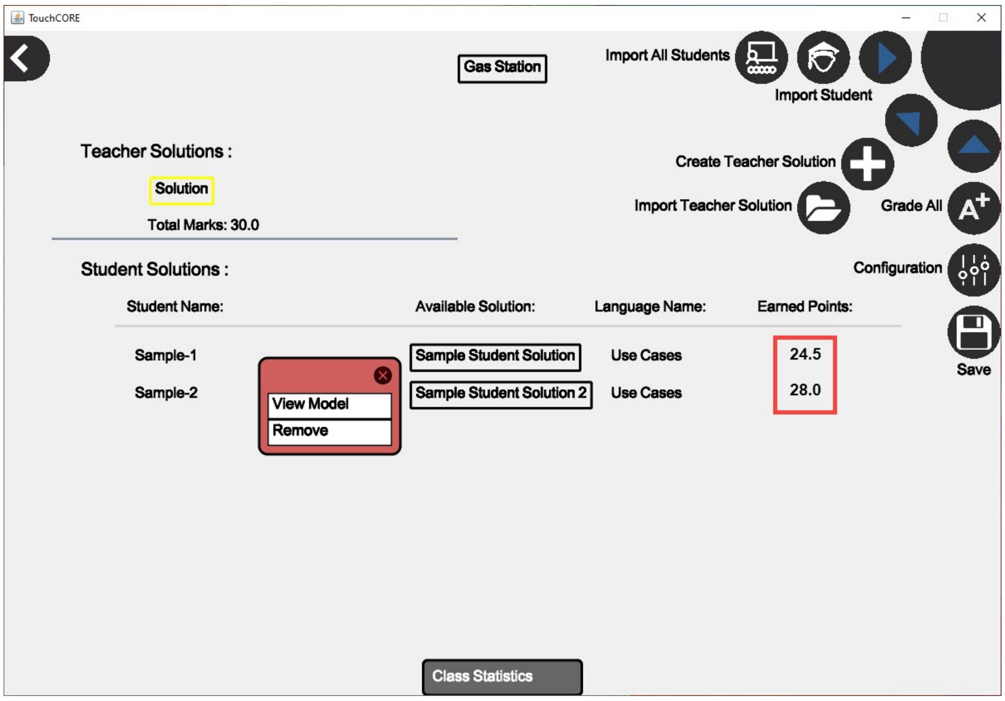
Diagram Deductions
Misplaced Element 100 %
Wrong Actor Name 50.0 %
Wrong Use Case Name 50.0 %
Wrong Association 50 %
Wrong Actor Multiplicity 50.0 %

Alternatives
Split Use Case ✓

Specification Deductions
Wrong Use Case Level 50.0 %
Wrong Use Case Intention 50.0 %
Wrong Use Case Multiplicity 50.0 %
Wrong Use Case Step 100.0 %
Wrong Use Case Reference 50 %
Wrong Use Case Context 50 %
Wrong Use Case PostCondition 50
Wrong Use Case Extensions 100 %
Merge Steps 50 %
Wrong Steps Order 50 %

Figure 5.15: Configuration Settings Panel

By pressing the *Configuration* button from the Homework (Fig. 5.14), the user can access the configuration setting panel (Fig. 5.15) and set the deduction percentage for each incorrect element in the student solution. The configuration settings panel's contents will be discussed later in this chapter. After modifying the configuration options, the user may save this configuration by pressing the *Set* button. If more than one correct solution is uploaded by the instructor, the user can set different configuration settings for other solutions by clicking the *Load Another Template Model* button.



Teacher Solutions :

Solution

Total Marks: 30.0

Student Solutions :

Student Name:	Available Solution:	Language Name:	Earned Points:
Sample-1	Sample Student Solution	Use Cases	24.5
Sample-2	Sample Student Solution 2	Use Cases	28.0

Figure 5.16: Graded Student Models

After setting the configurations, now the user can grade all uploaded student models by clicking on the *Grade All* button (the button likes *A+*) and a few seconds later, the grade for each student model shows behind the *Earned Points* as it is displayed in 5.16. It can be seen that both students are graded, and their total marks are displayed. Again, by pressing and holding on to one student model and selecting the *View Model* option, that student model can be opened. But this time, the grade received by the student for each element in the diagram can be seen in addition to the model.

Fig. 5.17 displays the grading result for one student. The student's original model is shown in Fig. 5.8. When the student makes any mistakes, the grade for the model element will be highlighted in yellow. So, for this example, the student received full points for all actors but missed some points for the use case. Based on the instructor model (Fig. 5.9), the student also lost points for two missed actors (Cancel Button and Holster). There are also two new buttons labelled *Compare Teacher Solution* in this view (green buttons). They present instructor

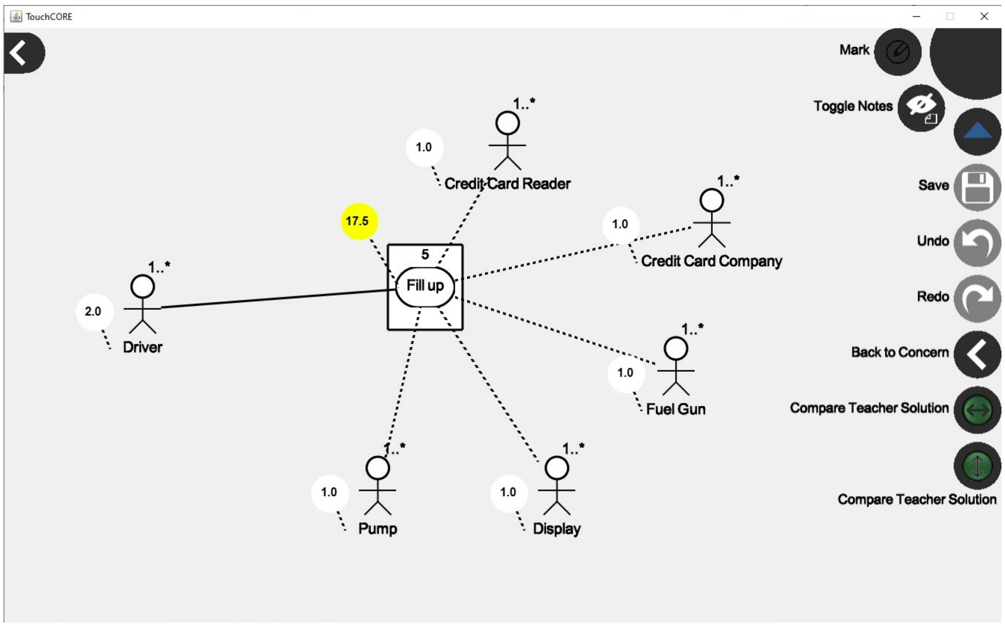


Figure 5.17: Student Model Grading Result

and student models side by side, whether horizontally (upper button) or vertically (lower button).

Fig. 5.18 shows the teacher's solution on the right side and the student's solution on the left side, horizontally, so the user can compare them easily.

The tool also generates feedback for students. It contains the points each student earned for each component in the model. table. 5.1 shows example feedback for one student solution. It shows the student received marks for 6 actors and missed 2 actors. The student also lost points for one input and two output communication steps as well as one Extension.

Finally, our tool implementation allows the instructor to change the grading scheme through the instructor's solution model, such as the model in Fig. 5.9 and Fig. 5.10. Since we keep a map of 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.

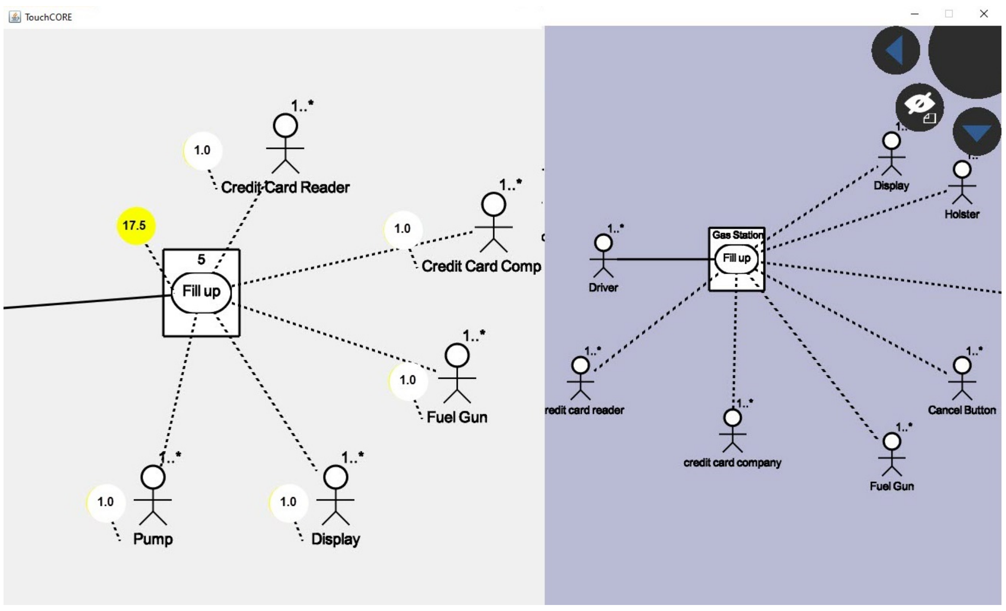


Figure 5.18: Compare Models Horizontally

5.4 Configuration Settings in Automated Grading

As we mentioned in the previous section, the grading tool could be configured using the configuration setting panel shown in Fig. 5.15. instructors can modify the algorithm using this panel based on their grading strategies. The configuration settings panel is divided into five sections: Diagram Unnecessary, Specification Unnecessary, Diagram Deductions, Specification Deductions, and Alternatives.

5.4.1 Unnecessary

Students frequently include extraneous components in their models. These elements may not cause the model to be wrong, but they may add unnecessary features. In a use case model, for example, some students may attempt to optimize their results by including everything they can think of in their model in order to enhance the likelihood of not omitting what the instructor is looking for. To address this scenario, the instructor may choose to deduct points for the presence of unnecessary elements in the model.

The Unnecessary list in the configuration settings allows instructors to specify how much they want to deduct if students have an unnecessary use case, actor,

Table 5.1: Feedback for One Student Model for Gas Station Case Study

Final Grade: 24.5/30
Pump: 1.0|(out of 1.0) Actor Matched with Pump! actor lower bound matched! actor upper bound matched!
Credit card: 1.0|(out of 1.0) Actor Matched with Credit card reader! actor lower bound matched! actor upper bound matched!
Fuel Gun: 1.0|(out of 1.0) Actor Matched with Fuel Gun! | actor lower bound matched! actor upper bound matched!
Driver: 2.0|(out of 2.0) Actor Matched with Driver! actor lower bound matched! actor upper bound matched!
Credit Card Company: 1.0|(out of 1.0) Actor Matched with credit card company! actor lower bound matched! actor upper bound matched!
Display: 1.0|(out of 1.0) Actor Matched with Display! actor lower bound matched! actor upper bound matched!
Actor 'Cancel Button' is missed! Actor 'Holster' is missed!
Fill Up: 17.5.0|(out of 21.0) Use Case Matched with Fill Up! 5 input communication steps matched out of 6! 8 output communication steps matched out of 10! 1 extension matched out of 2!

generalization, include, primary actor, and secondary actor in the diagram, and level, intention, multiplicity, step, and post condition in the use case specification panel.

5.4.2 Deductions

The second group of configuration settings is Deductions, which lists deduction strategies. The algorithms initially hard-coded the default deduction approach for almost all types of mistakes to 50%. However, we allow instructors to specify the deduction for each kind of mistake.

There are 15 kinds of mistakes currently handled by the tool in the diagram:

- **Misplaced Element**

  A student might forget to add one use case or actor to the diagram.

- **Wrong Actor Name/ Wrong Use Case Name**

  A student might use a different actor or use case name than the one chosen by the instructor. If the name is different, the percentage will be deducted by the tool.

- **Wrong Actor Multiplicity**

  A student may use an incorrect multiplicity for the correct actor.

- **Wrong Association**

  A student may miss an association like *Include*, or use a *Secondary Actor* instead of a *Primary Actor*. So, the percentage deduction is applied for these kinds of mistakes.

- **Wrong Use Case Level**

  The student can select *Level* from three available options. One student might choose an incorrect *Level*, which is different than the instructor's solution, for a particular use case.

- **Wrong Use Case Intention / Multiplicity**

  A student might use a different text for the intention or multiplicity of the use case than the one chosen by the instructor.

- **Wrong Use Case Step**

  The deduction is applied if the algorithm cannot find a matching communication step between the instructor's solution and the student's solution. The student may have used a totally different text as a communication step or not written that step at all.

- **Wrong Use Case Reference**

  A student may refer to an incorrect use case in the Main Success Scenario or in the alternative flows in the Extensions section.

- **Wrong Use Case Context**

  If the algorithm is unable to identify a context step in the student's solution that matches the instructor's solution, the deduction is applied. The student might not have added a context step at all or may have chosen a completely different text.

- **Wrong Use Case PostCondition**

  The percentage points are deducted when the Postcondition text in the student solution is not matched with the corresponding Postcondition text in the instructor solution.

- **Wrong Use Case Extensions**

  A use case Extension includes the precondition and conclusion types. A student may refer to a different communication step from the instructor's solution in the precondition or select the incorrect conclusion type.

- **Merge Steps**

  Merge steps occur when a student merges two (or more) steps into one step. The algorithm does not deduct points for merged steps by default, but the user can change the deduction percentage if a student merges steps.

- **Wrong Step Order**

  The order of steps is important, and the algorithm can find where the student wrote a correct step but in a different order compared to the instructor's solution. So, the user can determine the deduction percentage for any incorrect order of steps.

5.4.3 Alternatives

There is just one alternative option in the configuration settings panel: *Split Use Case*. This option lets instructors decide if the student is allowed to split one use case into two or more use cases. When a student writes some steps from one use case in the instructor's solution into one use case, and some other steps into another use case in his or her solution, this is referred to as a split use case. For example, use case A in the instructor's solution contains steps a, b, c, d, and e. The student used two use cases, A and B, in his or her model and wrote steps a and b in use case A and steps c and d in use case B. So, we can say that this student split use case A into two use cases, A and B.

5.5 Conclusion

In this chapter, we explained the architecture of the TouchCORE tool when utilizing it for automated grading of use case diagrams. Then two metamodels to assist with automated grading were investigated. The grade metamodel stores grades for model elements as well as grades for the structural features of the element. The mappings between student model elements and instructor model elements are stored in the classroom metamodel. We also demonstrated our tool's graphical user interface and thoroughly explained all options. We also showed how the instructor is allowed to simply upload models, adjust the grading scheme, and automatically grade all students. We also presented a configuration settings panel that enables instructors to alter different grading criteria. The instructor can decide how many points students lose when they make mistakes by modifying the deduction percentage. Finally, we demonstrated feedback information to the students by displaying the TouchCORE tool's GUI.

Chapter 6

Case Study

In this chapter, we apply our approach to two real-world assignments. The first assignment is about a Gas Station problem that has only one use case. The second assignment, called Elfenroads, has 13 use cases. In the Gas Station assignment, the main focus of the instructor was on evaluating the use case specification, while Elfenroads aimed to evaluate both the diagram and the use case specification.

Following the explanation of both case studies, we compare the automated grading and manual grading for two case studies and answer the first Research Question. Then, to address the second Research Question, we modify the automated grading algorithm to match the instructors' grading practises and re-grade all student models using the configuration options. The findings demonstrate that utilizing the configuration settings brings the automated grading closer to the instructor's manual grading. We also analyze each component of the grading tool and present instances of how the automated grading supplied works effectively for use case diagrams. We show that automated grading may increase efficiency and assure fairness in the grading process by addressing Research Questions 3. We also discuss the impact of uploading multiple solutions on the automated grading of use case models in Research Question 4. In Research Question 5, we demonstrate how we improved the matching steps procedure by first flattening the steps and then comparing them. In the last Research Question, we examine the quality of the automated grading using NLP methods.

These assignments were given a year before we developed our tool. Therefore, the students and the instructors did not use the tool that is used in these case studies. In addition, they did not make the assumption that the assignments would be automatically graded by a tool.

Students were given assignment handouts detailing the assignment question and requirements. The Gas Station Use Case Model was given to students in Fall 2016 during an exam, and they had around 30 minutes to do it. These students are either last year undergraduate students or graduate students. Elfenroads was the course project in the last academic year, i.e., Fall 2021–Winter 2022. These students are typically undergraduate students in their last year of study. The Elfenroads Use Case Model was part of a group assignment for which the students did not have any time pressure.

6.1 Gas Station

The first assignment (Case Study 1) was performed in the context of a beginner software design and modelling course. The problem was described as follows:

A gas station is to be set up for fully automated operation. Payment is done by credit card only. The interaction with the pump is as follows:

"Drivers insert their credit card into a reader connected to the pump, the card is verified by communication with a credit card company computer, and a credit limit is granted (sufficiently high to fill up any car). If the validation succeeds, the fuel gun is unlocked, and the driver may then take fuel. When fuel delivery is complete and the fuel-dispensing gun is returned to its holster, the driver's credit card account is debited with the cost of the fuel taken. After debiting, the credit card is returned. If the card is invalid, it is returned by the pump, and the fuel gun remains locked in the holster. The driver can also cancel his interaction with the pump, but not after he started filling his tank with actual fuel.

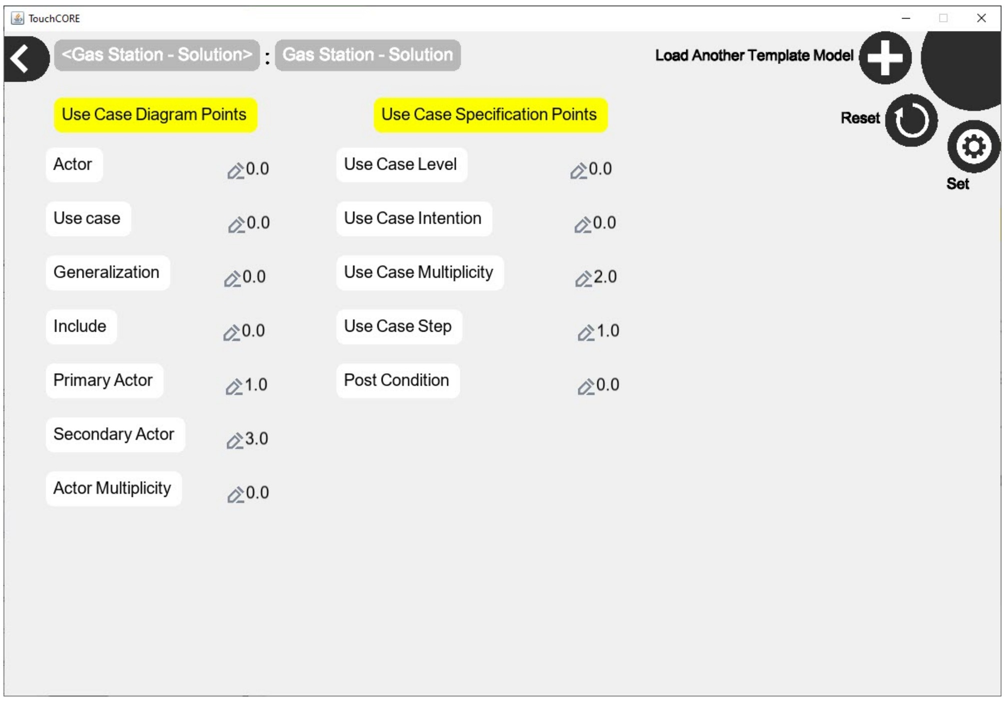
Elaborate on the FillUp use case, which describes all the interaction steps between the system and the environment that occur when a driver

uses the pump to fill up his tank.

You are asked to write the FillUp use case at a low-level of abstraction, clearly stating which hardware devices the software is interacting with. You might have to "discover" additional actors or hardware devices that are not mentioned in the description to actually ensure that the Driver gets fuel in his tank."

Fourteen students submitted their assignments, and we used their submitted solutions to run our experiment. Fig. 5.11 shows the instructor's solution diagram.

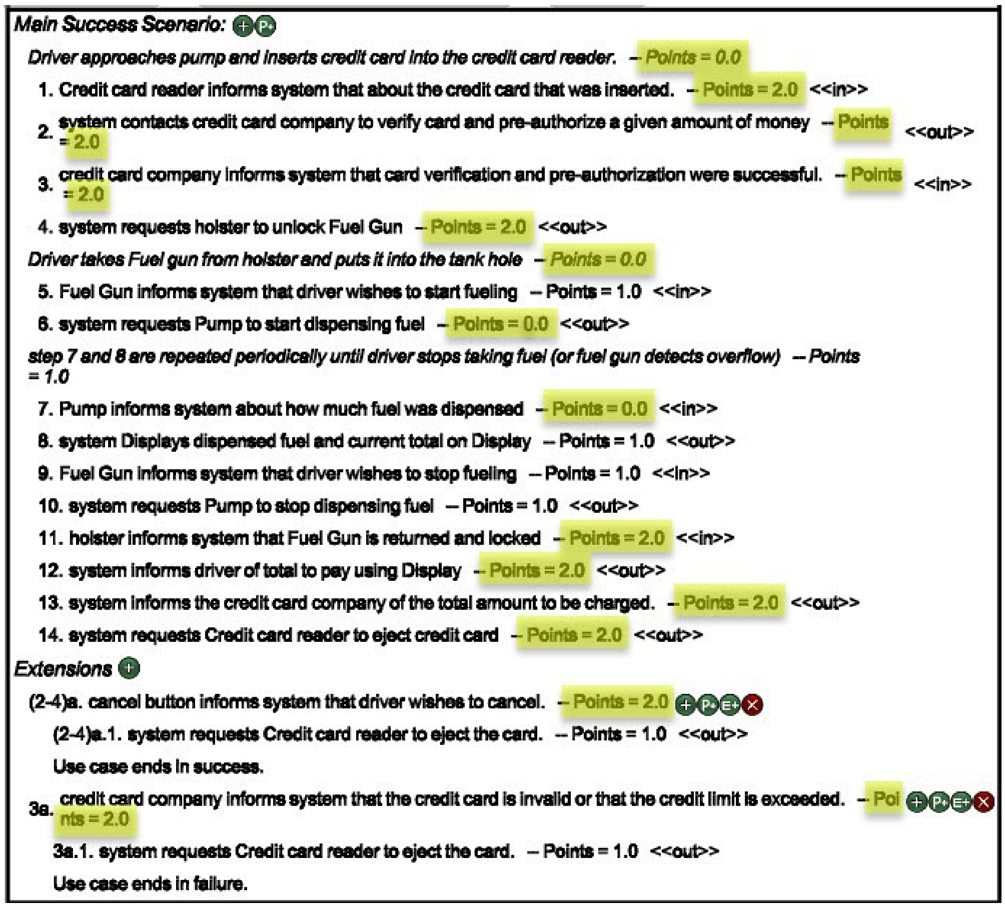
In this model, there is one use case (Fill Up) and 7 secondary actors (Display, Holster, Pump, Cancel Button, Fuel Gun, Credit Card Company, and Credit Card Reader), and one primary actor (Driver).


TouchCORE
<Gas Station - Solution> : Gas Station - Solution
Load Another Template Model
Reset
Set

Use Case Diagram Points		Use Case Specification Points	
Actor	0.0	Use Case Level	0.0
Use case	0.0	Use Case Intention	0.0
Generalization	0.0	Use Case Multiplicity	2.0
Include	0.0	Use Case Step	1.0
Primary Actor	1.0	Post Condition	0.0
Secondary Actor	3.0		
Actor Multiplicity	0.0		

Figure 6.1: Instructor Grading Scheme for Gas Station Use Case Model

The grading scheme shown in Fig. 6.1 is filled by looking at the instructor's scheme, which can be modified through the configuration setting panel in the tool. As Fig. 6.1 shows, the instructor's focus is on primary and secondary actors, use case multiplicity, and steps. After clicking on the set button, all marks will be assigned to each element in the diagram and in the use case specification, which can be modified later on.



Main Success Scenario:

Driver approaches pump and inserts credit card into the credit card reader. -- Points = 0.0

1. Credit card reader informs system that about the credit card that was inserted. -- Points = 2.0 <<in>>
2. system contacts credit card company to verify card and pre-authorize a given amount of money -- Points = 2.0 <<out>>
3. credit card company informs system that card verification and pre-authorization were successful. -- Points = 2.0 <<in>>
4. system requests holster to unlock Fuel Gun -- Points = 2.0 <<out>>

Driver takes Fuel gun from holster and puts it into the tank hole -- *Points = 0.0*

5. Fuel Gun informs system that driver wishes to start fueling -- Points = 1.0 <<in>>
6. system requests Pump to start dispensing fuel -- Points = 0.0 <<out>>

step 7 and 8 are repeated periodically until driver stops taking fuel (or fuel gun detects overflow) -- Points = 1.0

7. Pump informs system about how much fuel was dispensed -- Points = 0.0 <<in>>
8. system Displays dispensed fuel and current total on Display -- Points = 1.0 <<out>>
9. Fuel Gun informs system that driver wishes to stop fueling -- Points = 1.0 <<in>>
10. system requests Pump to stop dispensing fuel -- Points = 1.0 <<out>>
11. holster informs system that Fuel Gun is returned and locked -- Points = 2.0 <<in>>
12. system informs driver of total to pay using Display -- Points = 2.0 <<out>>
13. system informs the credit card company of the total amount to be charged. -- Points = 2.0 <<out>>
14. system requests Credit card reader to eject credit card -- Points = 2.0 <<out>>

Extensions

(2-4)a. cancel button informs system that driver wishes to cancel. -- Points = 2.0

(2-4)a.1. system requests Credit card reader to eject the card. -- Points = 1.0 <<out>>

Use case ends in success.

3a. credit card company informs system that the credit card is invalid or that the credit limit is exceeded. -- Poi nts = 2.0

3a.1. system requests Credit card reader to eject the card. -- Points = 1.0 <<out>>

Use case ends in failure.

Figure 6.2: Instructor's Use Case Specification for Gas Station Case Study

Fig. 6.2 shows the grading scheme for the "Fill Up" use case specification. After assigning points through the grading scheme by the instructor, step points are specified in the use case specification at the end of each step, which can be modified by the instructor. For instance, if the instructor sets the "Use Case Step" point in Fig. 6.1 equal to 1, all individual steps in the use case specification shown in Fig. 6.2 will be set to *points = 1*. The instructor then changed some step points from 1 to 2 or zero, which is highlighted in Fig. 6.2.

Based on the grading schemes in the configuration panel and the step points assigned in the use case specification, the maximum grade that one student can achieve is 30.

6.2 Elfenroads

The second experiment (Case Study 2) was performed on advanced undergraduate students that are enrolled in a software engineering project course. In McGill University's Software Engineering Project class for 2021-2022, students are tasked with designing and implementing a networked version of Elfenroads [53]:

"Elfenroads [54] is a board game designed by Alan R. Moon and published by Rio Grande Games. The Elfenroads box set provides a version of the original base game, Elfenland, as well as the expansion, Elfengold. The basic objective of Elfenland is to traverse the board using the set of predefined roads to visit as many towns as possible within the allotted turns. Roads can be made traversable through the use of transportation markers played by any of the players, and on a player's moving phase, they can traverse any adjacent marked road using travel cards in their hand. Elfengold adds the concept of gold to the game. Gold is awarded when visiting a town for the first time or by drawing a gold transportation card, and it can be used to purchase transportation markers in an auction at the start of each round. Elfengold also introduces new spells, which can be used to obstruct a marked road, circumvent obstacles currently placed on a road, exchange any two previously played transportation markers, or immediately transport a player to a

particular town."

The game must be playable by multiple players over the Internet and must support the selection of either the base Elfenland rules or the expanded Elfengold rules when creating a game. Players should be allowed to create their own game lobby, which can then be joined by other players before beginning to play the selected game. To aid in implementation, all students have access to a generic game lobby service [55]. This lobby service provides functionality for user authentication and game session management, including the creation and joining of a session, starting and ending a session, and saving and loading sessions. Student implementations of Elfenroads are expected to make use of the lobby service to handle these aspects of the implementation [53].

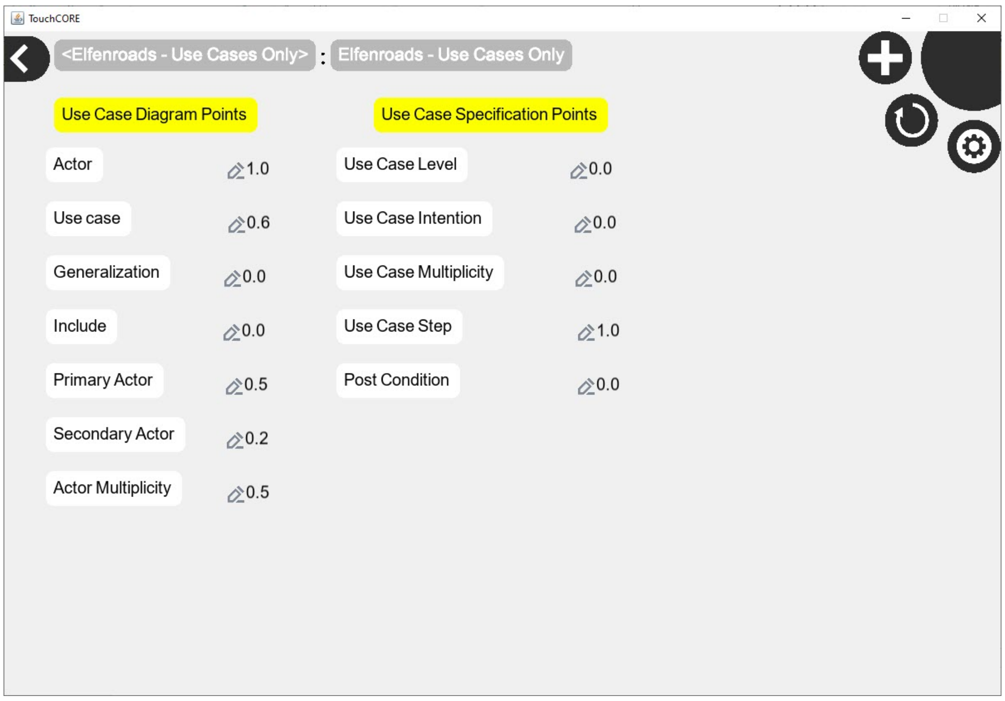
Since that was a large assignment, the instructor asked students to do it in different groups. The assignment was provided by nine groups, and we used their solutions to execute our experiment. Fig. 3.1 represents the instructor's use case diagram solution.

In this use case model, there are two actors (Player as a primary and Game Lobby Service as a secondary) and 13 use cases that are connected to each other using «include» relationship.

Fig. 6.3 depicts the grading scheme for the Elfenroads use case model.

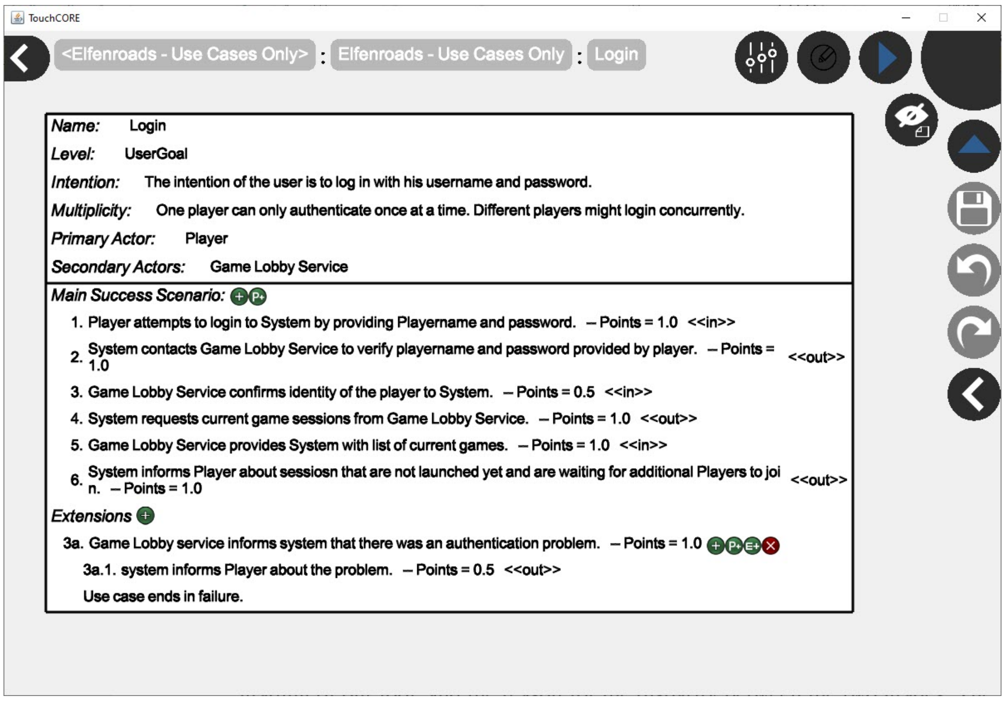
The instructor decided to assign 1 and 0.6 points for each actor and use case, respectively. Each primary actor association is considered 0.5 points, and since there are 5 primary actor associations, the total points for that would be 2.5, and secondary actors contain 1 mark (5 use cases include Game Lobby Service as a secondary actor, and each contains 0.2 points). The initial step's point considered 1 point which the instructor modified in each use case specification based on their importance. For instance, Fig. 6.4 shows the "Login" use case specification panel and points the instructor assigned to each step.

According to these schemes, the highest possible grade is 100.


TouchCORE

Use Case Diagram Points		Use Case Specification Points	
Actor	1.0	Use Case Level	0.0
Use case	0.6	Use Case Intention	0.0
Generalization	0.0	Use Case Multiplicity	0.0
Include	0.0	Use Case Step	1.0
Primary Actor	0.5	Post Condition	0.0
Secondary Actor	0.2		
Actor Multiplicity	0.5		

Figure 6.3: Instructor Grading Scheme for Elfenroads Case Study


TouchCORE

<Elfenroads - Use Cases Only> : Elfenroads - Use Cases Only : Login

Name: Login
Level: UserGoal
Intention: The intention of the user is to log in with his username and password.
Multiplicity: One player can only authenticate once at a time. Different players might login concurrently.
Primary Actor: Player
Secondary Actors: Game Lobby Service
Main Success Scenario:
1. Player attempts to login to System by providing Playername and password. – Points = 1.0 <<in>>
2. System contacts Game Lobby Service to verify playername and password provided by player. – Points = 1.0 <<out>>
3. Game Lobby Service confirms identity of the player to System. – Points = 0.5 <<in>>
4. System requests current game sessions from Game Lobby Service. – Points = 1.0 <<out>>
5. Game Lobby Service provides System with list of current games. – Points = 1.0 <<in>>
6. System informs Player about sessiosn that are not launched yet and are waiting for additional Players to joi n. – Points = 1.0 <<out>>

Extensions
3a. Game Lobby service informs system that there was an authentication problem. – Points = 1.0
3a.1. system informs Player about the problem. – Points = 0.5 <<out>>
Use case ends in failure.

Figure 6.4: Steps' Points in "Login" Use Case Specification Panel

6.3 RQ1: How different are manual and automated grading?

Table 6.1: Grades for Gas Station Use Case Model

No.	Instructor	Algorithm	Reason for Difference
1	18	18	
2	23	23	
3	20	20	
4	24	25	Main Success Scenario (I:16, A:17)
5	22	21	Main Success Scenario (I:15, A:14)
6	21	20	Main Success Scenario (I:14, A:13)
7	29	27	Main Success Scenario (I:17, A:15)
8	30	27	Main Success Scenario (I:18, A:16), Extensions (I:6, A:5)
9	26	23	Main Success Scenario (I:16, A:15), Extensions (I:6, A:4)
10	24	21	Main Success Scenario (I:12, A:11), Extensions (I:6, A:4)
11	30	25	Main Success Scenario (I:18, A:13)
12	28	23	Main Success Scenario (I:15, A:10)
13	21	17	Main Success Scenario (I:15, A:14), Extensions (I:4, A:1)
14	19	14	Main Success Scenario (I:13, A:10), Extensions (I:4, A:2)

In this section, we examined the efficiency of the automated grading algorithm in these case studies by comparing the total score of each student acquired by the automated grading algorithm with the score given manually by the instructor. Table 6.1 lists the grades that each student received from the gas station case study. It shows the instructor's grading, our tool's grading, and the reason for the difference between the two gradings. For example, in the automated grading, student 10 received 1 point in the Main Success Scenario and 2 points in the Extensions part less than the instructor's grading scheme. The classroom average based on the instructor's grading was 23.92, compared to the 21.71 automatically achieved by our tool. The average difference between the instructor's grade and our tool's grade was 2.21, i.e., our tool was able to automatically grade the students with less than 10% difference from manual grading.

The reason for the difference between automated and manual grading for some students, for example, students 11 and 14, is that those students used very short sentences as use case steps in the use case specifications, and the matching text algorithm (algorithm 3) could not match most words from the instructor's solution to the student's solution. For example, student 14 wrote *"Fuel Gun locks"* as one step, whereas the corresponding instructor's step is *"Holster informs system that Fuel Gun is returned and locked"*.

Table 6.2: Grades for Elfenroads Use Case Model

No.	Instructor	Algorithm	Reason for Difference
1	82	82.5	Diagram(I:15, A:14), Specification(I:67, A:68.5)
2	76.5	75.5	Diagram(I:12.5, A:11.5)
3	58	60	Diagram(I:10.5, A:8), Specification(I:47.5, A:52)
4	72	68	Diagram(I:10.5, A:11), Specification(I:61.5, A:57)
5	74	69.5	Diagram(I:13, A:12.5), Specification(I:61, A: 57)
6	84.5	90	Diagram(I:15, A:14), Specification(I:69.5, A:76)
7	78	84	Specification(I:66.5, A:72.5)
8	85.5	78	Diagram(I:14.5, A:14), Specification(I:71, A:64)
9	56.5	76	Diagram(I:10.5, A:11.5), Specification(I:46, A:64.5)

Table 6.2 shows the grades assigned to each student group for the Elfenroad case study. It displays the instructor's grading, the grading of our tool, and the reason for the disparity between the two grades. For example, for group 4, the instructor's grade for the diagram is 10.5 and the specification grade is 61.5, whereas the automated grade is 11 and 57, respectively. The classroom average based on the instructor's grade was 74.11, but our tool automatically reached 72.83. The average difference between the instructor's grade and our tool's grade was 1.28, indicating that our tool could automatically evaluate the students within 8.27% of the instructor's manual grading score.

One reason for the difference in manual and automated grading in Group 9 is that students forgot to add the "Game Lobby Service" actor, but mentioned and interacted with it in use cases, which the instructor may have missed and deducted points for. In other words, if the instructor was able to first flatten all steps manually, the group would get a better mark.

Finding 1. Based on the default configuration settings, the automated grading is close to the manual grading, with an average difference of 9.59% in the Gas Station case study and 8.27% in the Elfenroads case study. The difference between the automated and manual grading was large in some cases because the algorithm could not match a long sentence with a few words. It is also due to the inability to manually flatten steps correctly by the instructor.

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

To answer this question, we consulted the instructor to configure the algorithm to match their grading strategies and then applied the customized algorithm to regrade all student models.

For the first case study (Gas Station), because this exercise was specifically designed to test students' knowledge about the use case specification, the instructor suggested that the deduction for the wrong multiplicity should be set to 100%. In addition, the wrong use case communication and context steps, and also Extensions were set to 100%. The instructor also did not allow students to merge steps by setting the "Merge Step" option to 100%. Students must also write steps in the same order as the instructor's solution, for example, first *"Fuel Gun informs the system that Driver wishes to stop fueling"* followed by *"system requests Pump to stop dispensing fuel"* (Fig. 6.2). The opposite order is not acceptable and 100% of that step's point deducts. Because there are eight actors, and as a result, eight primary and secondary associations from the use case "Fill Up" to those actors, the instructor suggests setting the deduction for wrong associations to 70% as it is probable that students will use the different names for some of those actors. In this case, since the instructor set the secondary actor points to 3 (Fig. 6.1), and there are 7 secondary actors, the point for each of them is 3/7 = 0.42. For example, if one student writes 5 out of 7 secondary actors correctly and writes 2 of them with the wrong name (or missed them), he or she will get 2.35 ((5 * 0.42) + (2 * 0.42 * 0.30)) out of 3 points.

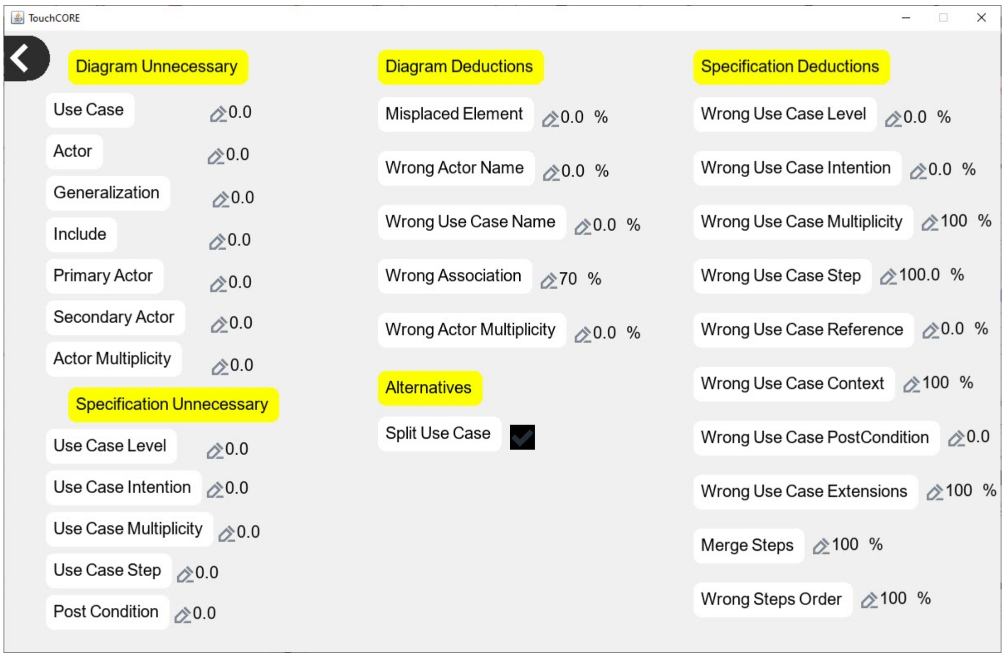

TouchCORE

Diagram Unnecessary

Use Case	0.0
Actor	0.0
Generalization	0.0
Include	0.0
Primary Actor	0.0
Secondary Actor	0.0
Actor Multiplicity	0.0

Specification Unnecessary

Use Case Level	0.0
Use Case Intention	0.0
Use Case Multiplicity	0.0
Use Case Step	0.0
Post Condition	0.0

Diagram Deductions

Misplaced Element	0.0 %
Wrong Actor Name	0.0 %
Wrong Use Case Name	0.0 %
Wrong Association	70 %
Wrong Actor Multiplicity	0.0 %

Alternatives

Split Use Case	✓

Specification Deductions

Wrong Use Case Level	0.0 %
Wrong Use Case Intention	0.0 %
Wrong Use Case Multiplicity	100 %
Wrong Use Case Step	100.0 %
Wrong Use Case Reference	0.0 %
Wrong Use Case Context	100 %
Wrong Use Case PostCondition	0.0
Wrong Use Case Extensions	100 %
Merge Steps	100 %
Wrong Steps Order	100 %

Figure 6.5: Configuration Settings for Gas Station Case Study

The configuration settings on which the instructor set all of these deduction percentages are shown in Fig. 6.5.

Fig. 6.6 depicts the distinction between the instructor's manual and automated grading for case study 1. The difference for the default settings is shown in blue, while the difference after modifying the configuration settings is shown in orange. It can be seen how the customized settings slightly outperform the default. The number of correctly graded models increased from 3 to 5 as a result of the configuration settings. With this configuration, 10 students received marks within 10% difference from manual grading, and no model differed by more than 18%. The manual grade average was 23.92 out of a possible 30 points, while the customized settings grade average was 22.42 and the default setting grade average was 21.71. As a result, the configuration dropped the average point difference from 2.21 to 1.50 (or from 9.59% to 6.69%).

In the second case study, the instructor considered a 50% deduction if the student chose the wrong name for actors or use cases, (like "Authentication" instead of "Login") as well as the wrong associations (include, secondary and primary actors). Students also receive no points for wrong actor multiplicity, for example, 1..* instead of 2..6 for the actor "Player" (Fig. 3.1). The instructor did not set

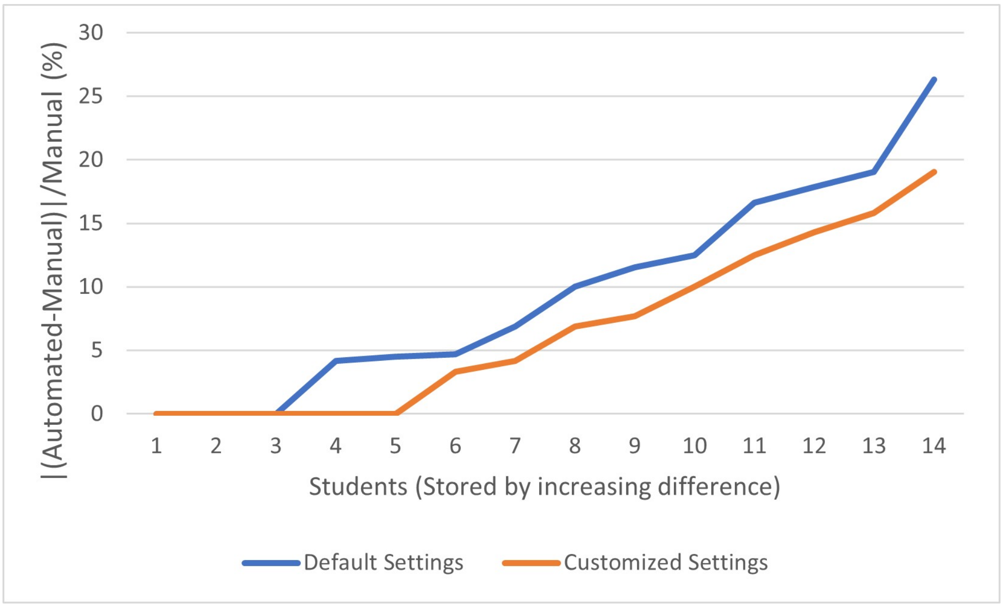


Figure 6.6: Default vs. Customized Configuration Settings for Gas Station Case Study

any deduction for use case level, intention, and multiplicity, but, set 100% for communication, context, and reference steps, as well as for the wrong Extensions part. If students merge two or more steps into one, 50% of those steps' points are deducted. For example, if the student merges step 1, which has 1 point, with step 2, which has 2 points, the student will get 1.5 out of 3 possible points for these 2 steps.

Fig. 6.7 shows the configuration settings on which the instructor set all these deduction percentages.

Fig. 6.8 shows the difference between the default settings and those acquired after customizing deduction percentages by the instructor. The blue line depicts the default configuration settings, whereas the orange line displays the scores after customizing the configuration settings according to the instructor's preferences. There are no models with exactly the same score as the instructor, but 4 models' automated grading marks were quite close to the instructor's grading. There are now eight models with grades that differ by less than 7%, and only one automated grade that differs by more than 7% from manual grading. For the whole class of 9 groups, the average difference between the manual and automated grading using the adjusted configuration was 7.61%, while the default configuration was 8.27%.

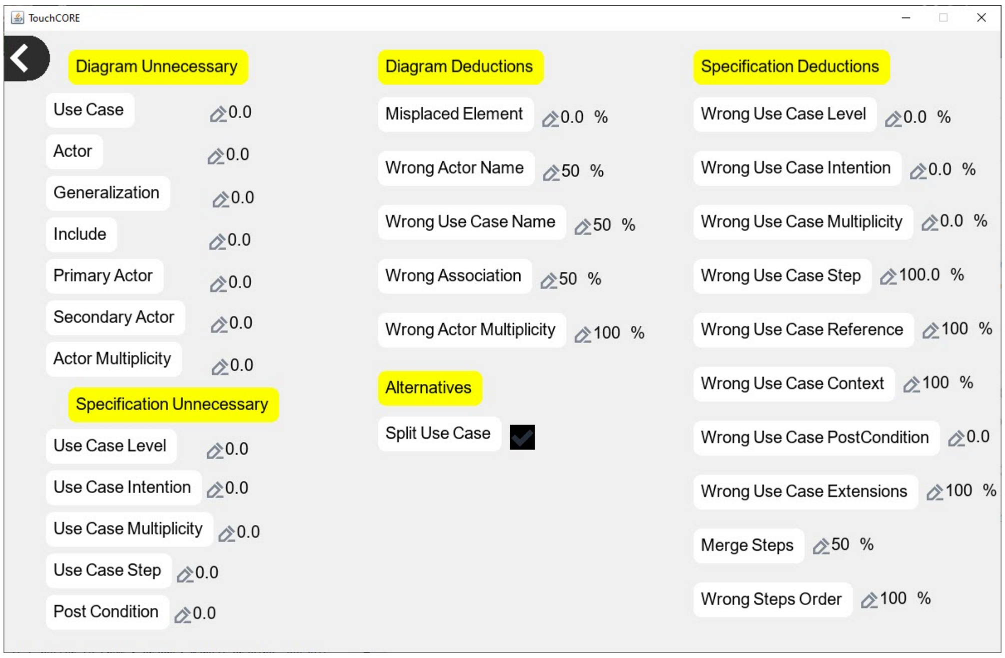


Diagram Unnecessary

Use Case	0.0
Actor	0.0
Generalization	0.0
Include	0.0
Primary Actor	0.0
Secondary Actor	0.0
Actor Multiplicity	0.0

Specification Unnecessary

Use Case Level	0.0
Use Case Intention	0.0
Use Case Multiplicity	0.0
Use Case Step	0.0
Post Condition	0.0

Diagram Deductions

Misplaced Element	0.0 %
Wrong Actor Name	50 %
Wrong Use Case Name	50 %
Wrong Association	50 %
Wrong Actor Multiplicity	100 %

Alternatives

Split Use Case	✓

Specification Deductions

Wrong Use Case Level	0.0 %
Wrong Use Case Intention	0.0 %
Wrong Use Case Multiplicity	0.0 %
Wrong Use Case Step	100.0 %
Wrong Use Case Reference	100 %
Wrong Use Case Context	100 %
Wrong Use Case PostCondition	0.0
Wrong Use Case Extensions	100 %
Merge Steps	50 %
Wrong Steps Order	100 %

Figure 6.7: Configuration Settings for Elfenroads Case Study

Finding 2. When the instructor customized the configuration settings, the scores that are automatically achieved by the tool are closer to the instructor's manual grading. The average difference between manual and automated grading improved from 9.59 to 6.69 and from 8.27 to 7.61 in the Gas Station and Elfenroads case studies, respectively. We can generalise this finding to all possible cases because grading with instructor criteria is always more precise than grading with the default values, which are hard-coded in the tool.

6.5 RQ3: Does automated grading help ensure fairness?

When instructors evaluate students, they consider fairness and consistency [56, 57, 58]. Furthermore, the majority of student comments about grading concern fairness [59, 60]. Since use case assignments can often have more than one correct solution, guaranteeing fairness can be difficult for instructors, especially when grading a large number of students.

When we re-examined the Gas Station case study for some of the student models where the grade difference between the manual and automated grades was high, we then discovered that the instructor was not fair to some students. Fig. 6.2 shows

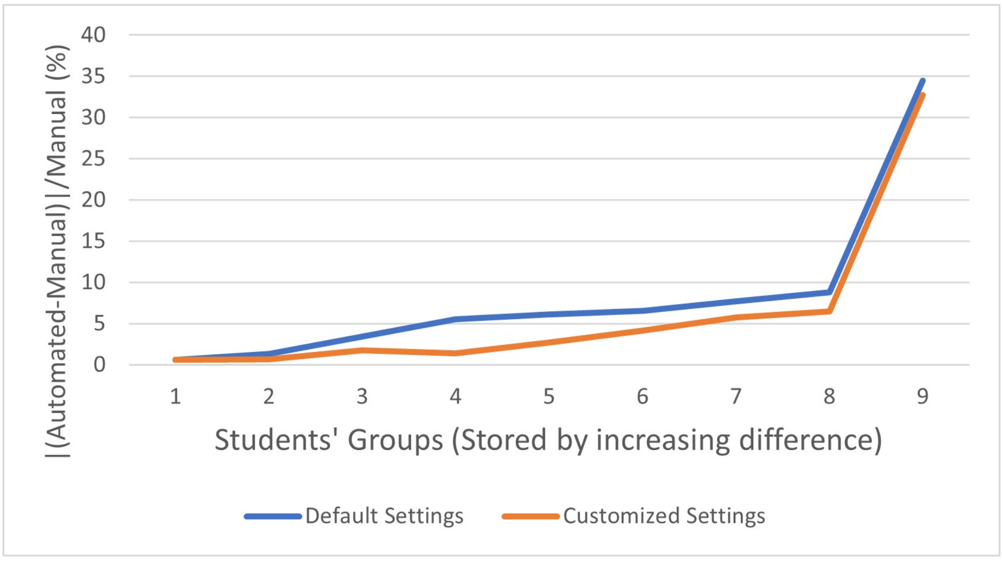


Figure 6.8: Default vs. Customized Configuration Settings for Elfenroads Case Study

the instructor's Main Success Scenario and Extensions for the "Fill Up" use case. Step 10, which is *"System requests Pump to stop dispensing fuel."* contains one point, and student 5 wrote *"System instructs Fuel Gun to stop dispenses fuel."* as step 11 (Fig. 6.9). While the student mentioned the step, the instructor probably missed adding this one point to the student's grade.

Fig. 6.10 displays the Main Success Scenario and Extensions for student 6. However, step 5 which is *"Driver release Fuel Gun handle."* is not mentioned in the instructor's solution (Fig. 6.2), which means it should not receive any points, but the instructor awarded 1 point by mistake for this step. In addition, the first Extension (2a) in the student's solution is matched with the first Extension in the instructor's solution and received points correctly, but the other two Extensions (2b and 3a) are different from the instructor's second Extension (3a) and should not receive any points, but the instructor added 1 point.

Fig. 6.11 shows the part of student number 12's Main Success Scenario. Step 7 is matched with step 9 in the instructor's solution (Fig. 6.2), which contains just one point, but the student received 2 points for this step.

Fig. 6.12 depicts student 13's Main Success Scenario. However, step 6 in this student's solution is matched with step 6 in the instructor's solution, but this step is considered no point, and the instructor added one point to the student's grade

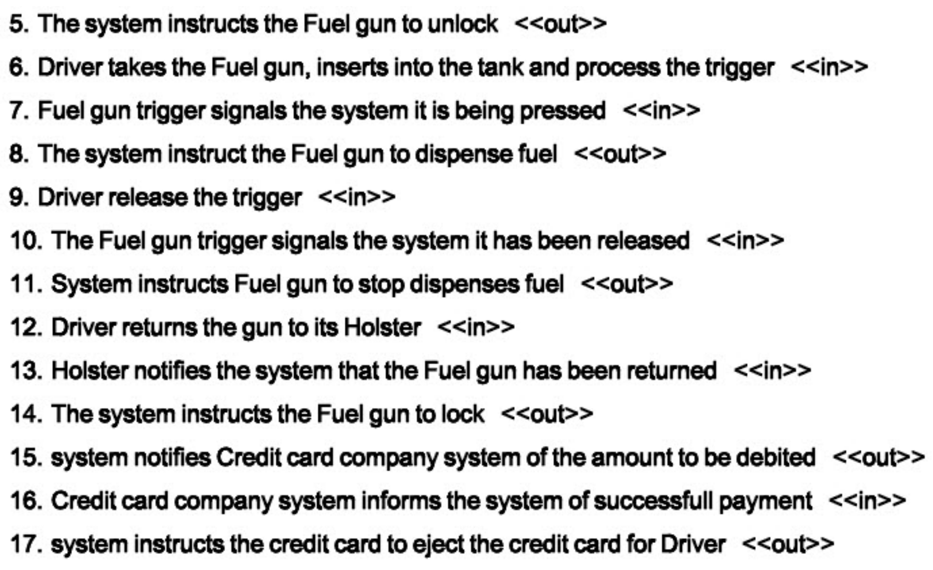

5. The system instructs the Fuel gun to unlock <<out>>
6. Driver takes the Fuel gun, inserts into the tank and process the trigger <<in>>
7. Fuel gun trigger signals the system it is being pressed <<in>>
8. The system instruct the Fuel gun to dispense fuel <<out>>
9. Driver release the trigger <<in>>
10. The Fuel gun trigger signals the system it has been released <<in>>
11. System instructs Fuel gun to stop dispenses fuel <<out>>
12. Driver returns the gun to its Holster <<in>>
13. Holster notifies the system that the Fuel gun has been returned <<in>>
14. The system instructs the Fuel gun to lock <<out>>
15. system notifies Credit card company system of the amount to be debited <<out>>
16. Credit card company system informs the system of successfull payment <<in>>
17. system instructs the credit card to eject the credit card for Driver <<out>>

Figure 6.9: Part of Student 5 Main Success Scenario for "Fill Up" Use Case

because of this step. Furthermore, step 12 in the instructor's solution was not mentioned in the student's solution, which includes 2 points, but the instructor added these 2 points to the student's grade.

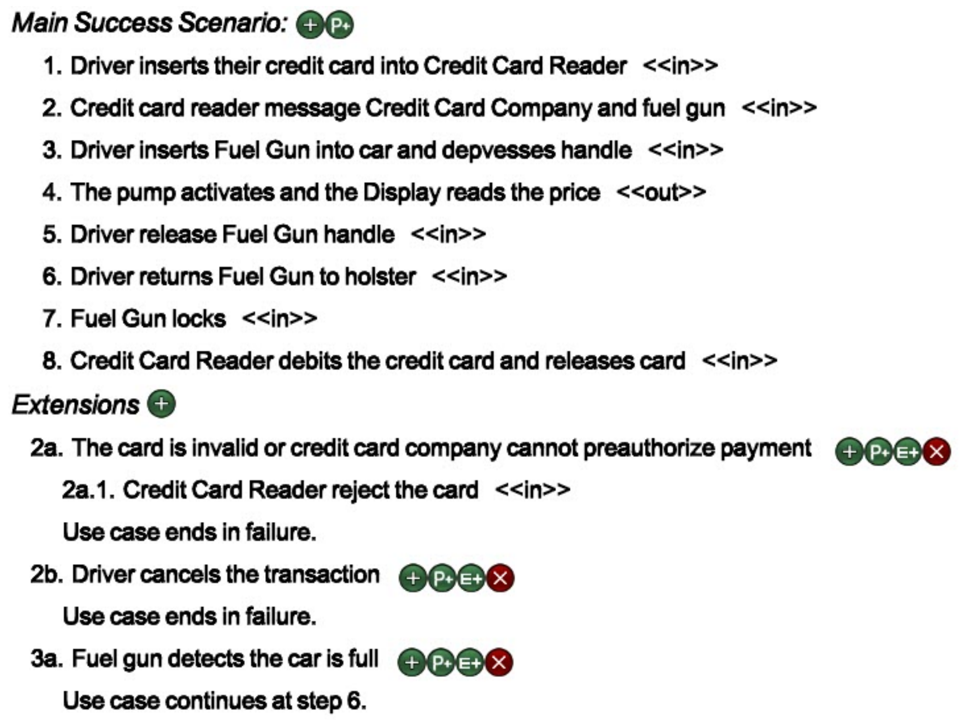
In the Elfenroads case study, there are plenty of mistakes when grading manually. For example, group 4 used "Draw Cards" use case in their diagram. They mentioned that the player is allowed to choose gold cards or travel cards from a deck. These steps contain 3 points, but the instructor did not consider any points for those steps.

Fig. 6.13 proves that group 4 used the required steps but did not earn points for them. This group also received 8 points out of 15 for interacting with the "Game Lobby Service" with the system, even though they did not use any "Game Lobby Service" actor in their diagram and just mentioned "Lobby Service" in the "Play Elfenroads" use case. So, they should receive 4 out of 15.

There is a large difference between manual and automated grading in Group 9. They interacted "Lobby" in "Quit Game" use case, but did not earn any points for that.

This group's interactions with Lobby are depicted in Fig. 6.14.

Finally, as it is shown in Fig. 6.15, This group also prompted "Player" for which transportation counter they must have, but did not earn 2 points for the mentioned



Main Success Scenario: + P+

1. Driver inserts their credit card into Credit Card Reader <<in>>
2. Credit card reader message Credit Card Company and fuel gun <<in>>
3. Driver inserts Fuel Gun into car and depvesses handle <<in>>
4. The pump activates and the Display reads the price <<out>>
5. Driver release Fuel Gun handle <<in>>
6. Driver returns Fuel Gun to holster <<in>>
7. Fuel Gun locks <<in>>
8. Credit Card Reader debits the credit card and releases card <<in>>

Extensions +

2a. The card is invalid or credit card company cannot preauthorize payment + P+ E+ ✗

2a.1. Credit Card Reader reject the card <<in>>

Use case ends in failure.

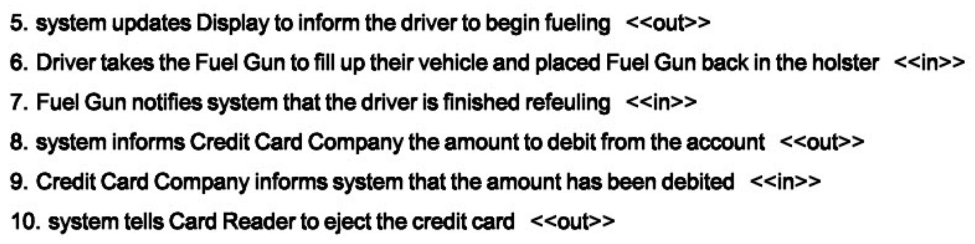
2b. Driver cancels the transaction + P+ E+ ✗

Use case ends in failure.

3a. Fuel gun detects the car is full + P+ E+ ✗

Use case continues at step 6.

Figure 6.10: Student 6 Main Success Scenario and Extension for "Fill Up" Use Case

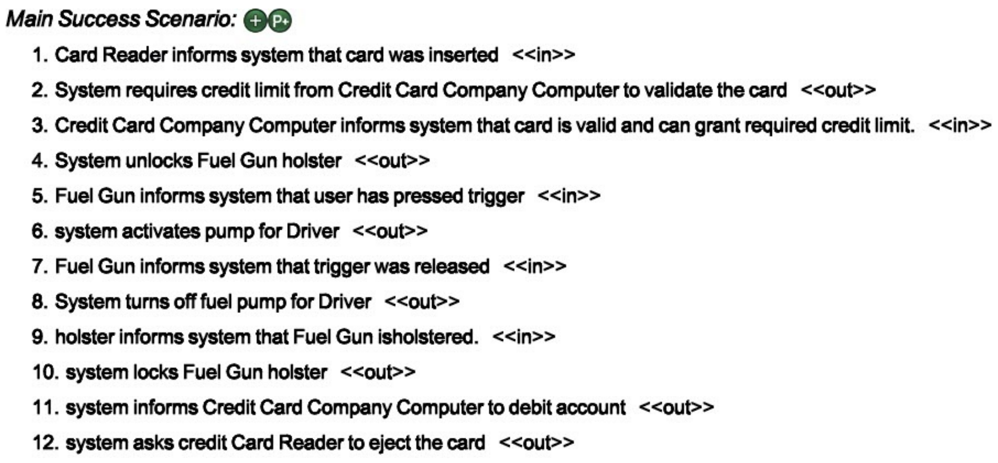


5. system updates Display to inform the driver to begin fueling <<out>>
6. Driver takes the Fuel Gun to fill up their vehicle and placed Fuel Gun back in the holster <<in>>
7. Fuel Gun notifies system that the driver is finished refeuling <<in>>
8. system informs Credit Card Company the amount to debit from the account <<out>>
9. Credit Card Company informs system that the amount has been debited <<in>>
10. system tells Card Reader to eject the credit card <<out>>

Figure 6.11: Student 12 Main Success Scenario and Extension for "Fill Up" Use Case

step.

Finding 3. In these two case studies, we just showed a few mistakes that the instructor made when grading manually. Because use cases often contain many texts and it is not easy to compare them conceptually with the solution, and also because flattening steps manually is a problematic process, the grader probably makes some mistakes when grading. As a result, all those grades that are determined automatically by the tool are fairer and more consistent because all students are graded according to the same grading scheme. The number of mistakes made by the instructor was not that high to change the average manual grading marks significantly, so we are still trying to achieve a result that is close to the manual grading.


Main Success Scenario:

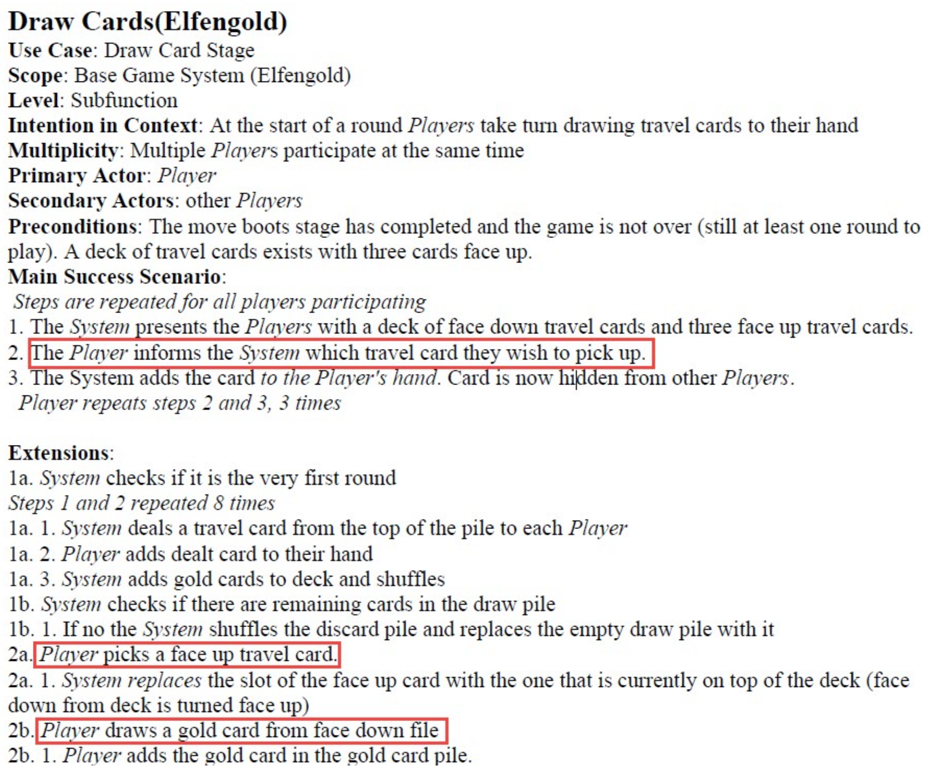
1. Card Reader informs system that card was inserted <<in>>
2. System requires credit limit from Credit Card Company Computer to validate the card <<out>>
3. Credit Card Company Computer informs system that card is valid and can grant required credit limit. <<in>>
4. System unlocks Fuel Gun holster <<out>>
5. Fuel Gun informs system that user has pressed trigger <<in>>
6. system activates pump for Driver <<out>>
7. Fuel Gun informs system that trigger was released <<in>>
8. System turns off fuel pump for Driver <<out>>
9. holster informs system that Fuel Gun isholstered. <<in>>
10. system locks Fuel Gun holster <<out>>
11. system informs Credit Card Company Computer to debit account <<out>>
12. system asks credit Card Reader to eject the card <<out>>

Figure 6.12: Student 13 Main Success Scenario for "Fill Up" Use Case

6.6 RQ4: Does the accuracy of automated grading improve when multiple solutions are matched against?

In many cases, in use case models, it is possible that there is more than one solution for a particular problem. Hence, the tool is capable of loading more than one instructor's model. The algorithm compares the student's model to each instructor's model that is currently loaded and the highest mark is used to determine the student's final grade. In the Gas Station case study, the average for manual grading was 23.92. When grading automatically with just one solution, i.e., the solution shown in Fig. 5.9, the average grade for automated grading using the customized algorithm was 22.42. When using an additional solution model (e.g., using different texts as multiplicity and steps), the average increased to 22.89. We identified 9 out of 14 students who obtained higher grades with this additional case. When we used all three available solution models, the average became even closer to that of manual grading: 23.82. We obtained 12 out of 14 models with grades closer to the manual grade when using all three solutions.

Fig. 6.16 shows the percentage difference between automated and manual grading. The blue line depicts this difference when using one solution, the orange line when using two solutions, and the grey line when using all three solutions. The percentage difference diminished when we used two solutions, and it decreased even more when we used three solutions. When grading with one solution, four students receive marks exactly the same as in the manual grading, and nine students have



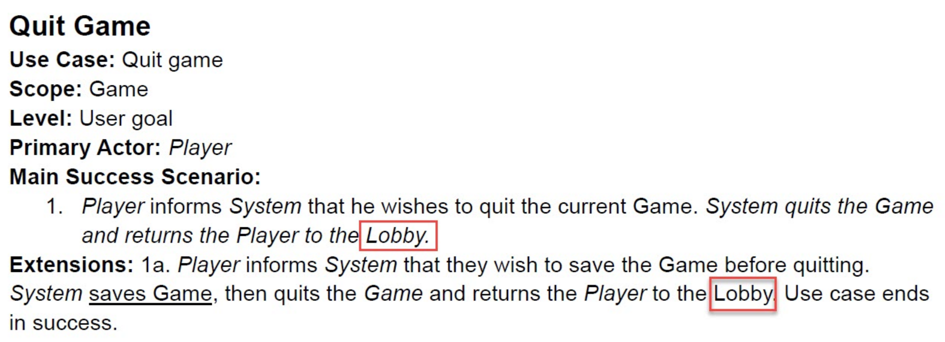
Draw Cards(Elfengold)
Use Case: Draw Card Stage
Scope: Base Game System (Elfengold)
Level: Subfunction
Intention in Context: At the start of a round *Players* take turn drawing travel cards to their hand
Multiplicity: Multiple *Players* participate at the same time
Primary Actor: *Player*
Secondary Actors: other *Players*
Preconditions: The move boots stage has completed and the game is not over (still at least one round to play). A deck of travel cards exists with three cards face up.
Main Success Scenario:
Steps are repeated for all players participating
1. The *System* presents the *Players* with a deck of face down travel cards and three face up travel cards.
2. The *Player* informs the *System* which travel card they wish to pick up.
3. The System adds the card *to the Player's hand*. Card is now hidden from other *Players*.
Player repeats steps 2 and 3, 3 times

Extensions:
1a. *System* checks if it is the very first round
Steps 1 and 2 repeated 8 times
1a. 1. *System* deals a travel card from the top of the pile to each *Player*
1a. 2. *Player* adds dealt card to their hand
1a. 3. *System* adds gold cards to deck and shuffles
1b. *System* checks if there are remaining cards in the draw pile
1b. 1. If no the *System* shuffles the discard pile and replaces the empty draw pile with it
2a. *Player* picks a face up travel card.
2a. 1. *System replaces* the slot of the face up card with the one that is currently on top of the deck (face down from deck is turned face up)
2b. *Player* draws a gold card from face down file
2b. 1. *Player* adds the gold card in the gold card pile.

Figure 6.13: Group 4 Main Success Scenario and Extension for "Draw Cards" Use Case

a grade difference that is less than 10%. The exact same marks increased from 4 to 5 with two solutions, and 11 models received marks that differed by less than 10% from the manual grading. When considering three solutions, the exact marks increased to six, and just two models had a grade difference of more than 10%. Finally, the average grade difference dropped from 6.69% for one solution to 4.50% when using three solutions.

Finding 4. Running the automated grading algorithm several times and selecting the highest grade as the final grade brings the average automated grading scores closer to the manual grade in cases where multiple solutions are available. In one case study, the average difference improved from 6.69% for one solution to 4.50% when using three solutions.

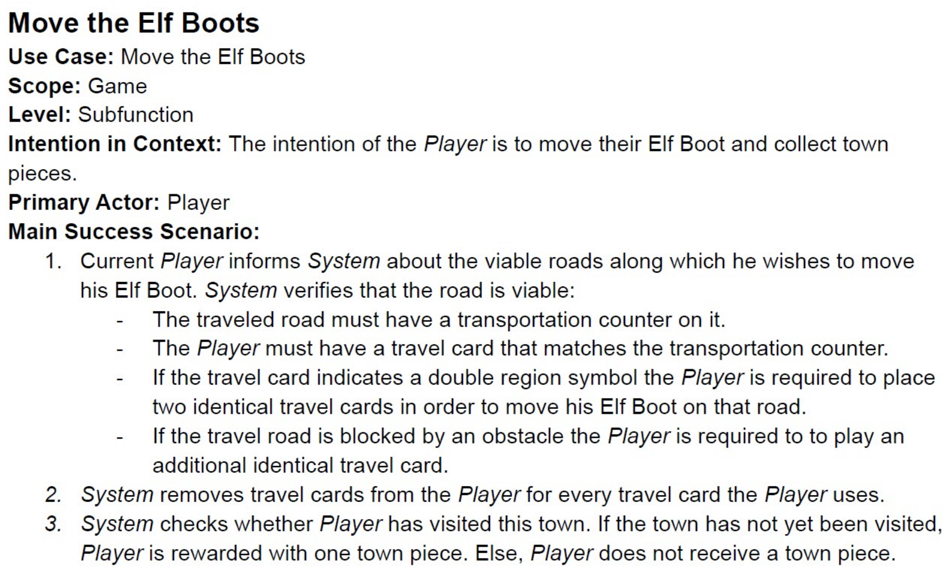


Quit Game
Use Case: Quit game
Scope: Game
Level: User goal
Primary Actor: *Player*
Main Success Scenario:

1. *Player* informs *System* that he wishes to quit the current Game. *System quits the Game and returns the Player to the Lobby.*

Extensions: 1a. *Player* informs *System* that they wish to save the Game before quitting. *System* saves Game, then quits the *Game* and returns the *Player* to the Lobby. Use case ends in success.

Figure 6.14: Group 9 Main Success Scenario and Extension for "Quit Game" Use Case



Move the Elf Boots
Use Case: Move the Elf Boots
Scope: Game
Level: Subfunction
Intention in Context: The intention of the *Player* is to move their Elf Boot and collect town pieces.
Primary Actor: Player
Main Success Scenario:

1. Current *Player* informs *System* about the viable roads along which he wishes to move his Elf Boot. *System* verifies that the road is viable:
   - The traveled road must have a transportation counter on it.
   - The *Player* must have a travel card that matches the transportation counter.
   - If the travel card indicates a double region symbol the *Player* is required to place two identical travel cards in order to move his Elf Boot on that road.
   - If the travel road is blocked by an obstacle the *Player* is required to to play an additional identical travel card.
2. *System* removes travel cards from the *Player* for every travel card the *Player* uses.
3. *System* checks whether *Player* has visited this town. If the town has not yet been visited, *Player* is rewarded with one town piece. Else, *Player* does not receive a town piece.

Figure 6.15: Group 9 Main Success Scenario for "Move the Elf Boots" Use Case

6.7 RQ5: How do flattening steps helps automated grading to achieve a better result?

When there are multiple use cases in the diagram, they may be linked by a relationship, such as *include*. An "include" relationship denotes the inclusion of a use case as a sub-process (the inclusion use case) of another use case (the base use case) [61]. So, we can combine all the steps from the base use case with the steps in the inclusion use case based on the use case reference steps. We proceed with this method from the root use case to the leaf use cases in the diagram, and at the end, we have a list of flattened steps. When manually grading large models, flattening can be a difficult task.

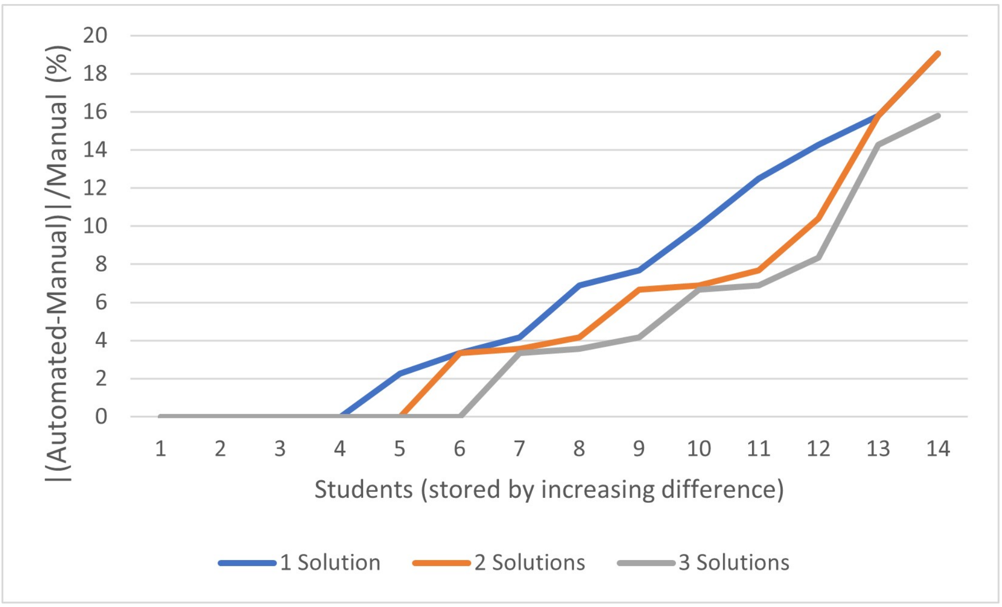


Figure 6.16: Multiple Solutions for Gas Station Case Study

Fig. 3.1 shows the instructor's use case diagram solution for the Elfenroads case study. It contains 13 use cases, and all use cases are connected to each other using include relationships. Fig. 6.17 demonstrates one group submission for this assignment which includes 17 use cases. As a result, if we simply match the use cases by name and compare the specifications of each matched use case, we can match 13 use cases in the best-case scenario. In the end, there would be 4 use cases in the student's solution that are not matched. Finally, their steps are not considered when grading, but they can include some correct steps. In addition, it is possible that one student writes the correct steps in an irrelevant use case instead of the right one or splits one use case into two use cases. In these cases, however, we cannot match use cases by name, but the student should receive some points for the correct steps in unmatched use cases. To deal with this issue, we just compare flattened steps. So, after flattening all steps in the Main Success Scenario and Extensions in both the student's and instructor's solution, we just compare them one by one and in order.

The blue line in Fig. 6.18 plots the instructor's grades. The orange line shows the automated algorithm when flattening was considered, and the grey line depicts automated grading without flattening. It is obvious that students' marks without flattening are always less than the instructor's marks because steps in those un-

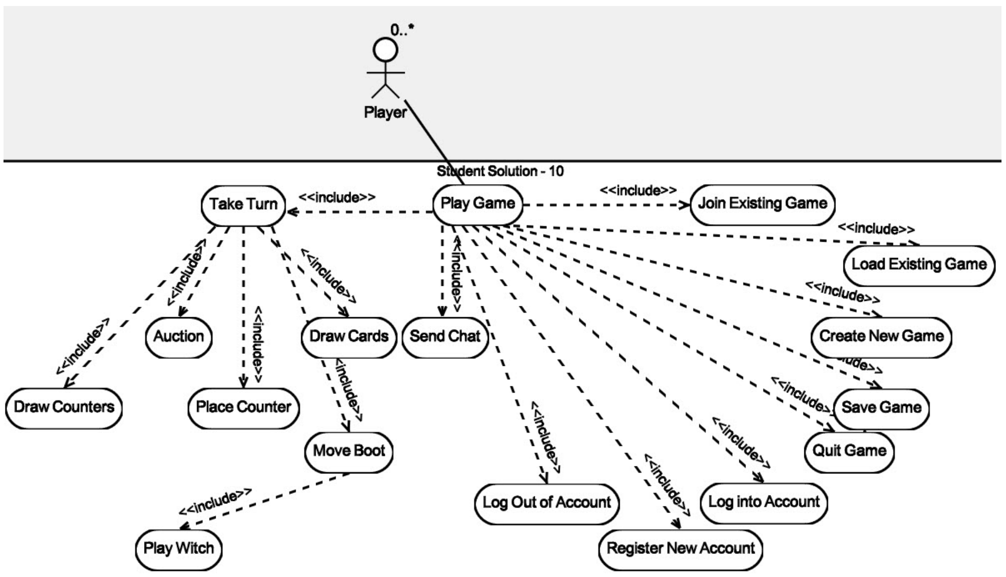


Figure 6.17: One Student Group Use Case Diagram for Elfenroads Case Study

matched use cases are not compared. Group 7, which has the highest difference from the instructor's mark (without flattening), uses 27 use cases in their diagram compared to 13 in the instructor's diagram (Fig. 3.1), but the difference is very small when considering flattening. There is also a larger difference between manual grading (blue line) and automated grading when considering flattening (orange line) for group 8. The reason is that the instructor may have missed some correct steps since he could not flat all steps manually. As a result, the manual grading average was 74.11, and when we graded without flattening, the average was 65.33, but when we considered flattening for steps in all use cases in the Elfenroads case study, the average was closer to the manual grading average: 75.94.

Finding 5. Flattening tries to match all possible use case steps in the student's solution to the instructor's solution and therefore improves the final marks. In one case study, the average difference decreased from 14.48 when we did not consider flattening steps to 7.61 when we considered flattening steps.

6.8 RQ6: Does using NLP improve the quality of the grading algorithm?

The most challenging part of automated grading in use case models is matching texts. Steps play an important role in use case models, and as we investigated in

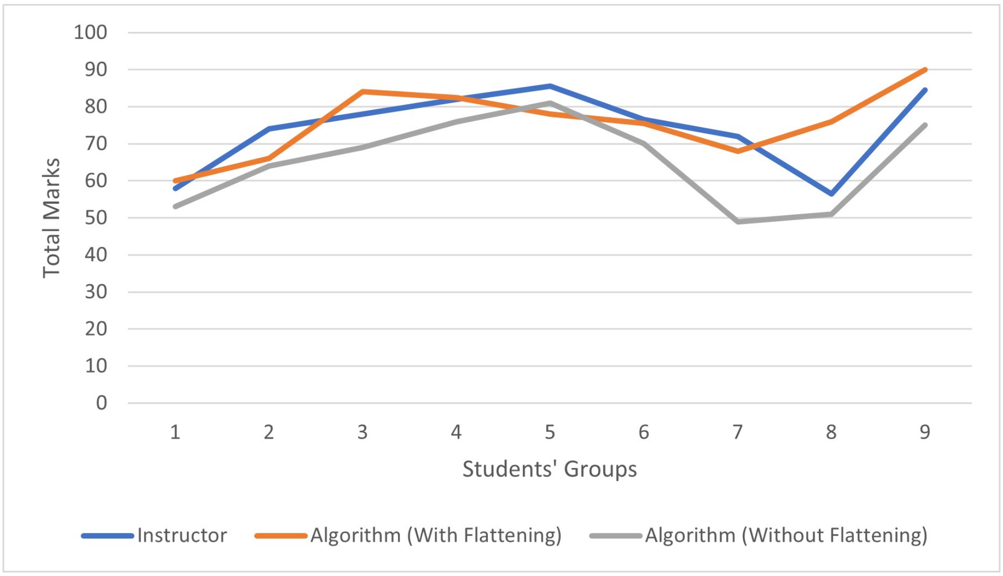


Figure 6.18: Auto-grading with Flattening vs. Without Flattening

the Elfenroads case study, the instructor considered 85% of the whole grade for the use case specification part and only 15% for the diagram. We discussed the matching text algorithm, which incorporated natural language processing methods, in chapter 4 (algorithm 3). In this question, we want to show how this algorithm improves the efficiency of automated grading of use case models using NLP methods.

As there is a group of words in each text (e.g., step), it is not possible to match them syntactically and semantically like use cases and actor names. Hence, we need an algorithm to check the entire text conceptually because all students use their own words when writing a sentence.

At first, we provided an algorithm that split the entire text into a list of words without any natural language processing methods. It compared each word in the text from the instructor's solution with each word in the corresponding text in the student's solution syntactically and semantically and returned a threshold value based on similarities. We considered whether the threshold was greater than a certain amount of value (for example, greater than 50%), so those texts are matched. However, the output was good but not admissible.

When using NLP, we realized that students get higher grades in general, and the average marks improved from 72.83 to 75.94 in the Elfenroads case study, while the manual grading average was 74.11. In the Gas Station case study, the average marks increased from 20.42 to 22.42, while the manual grading average was 23.92. The average difference also decreased, from 14.48% to 8.27% in the Elfenroads case study and from 15.75% to 9.59% in the Gas Station case study when considering default configuration settings.

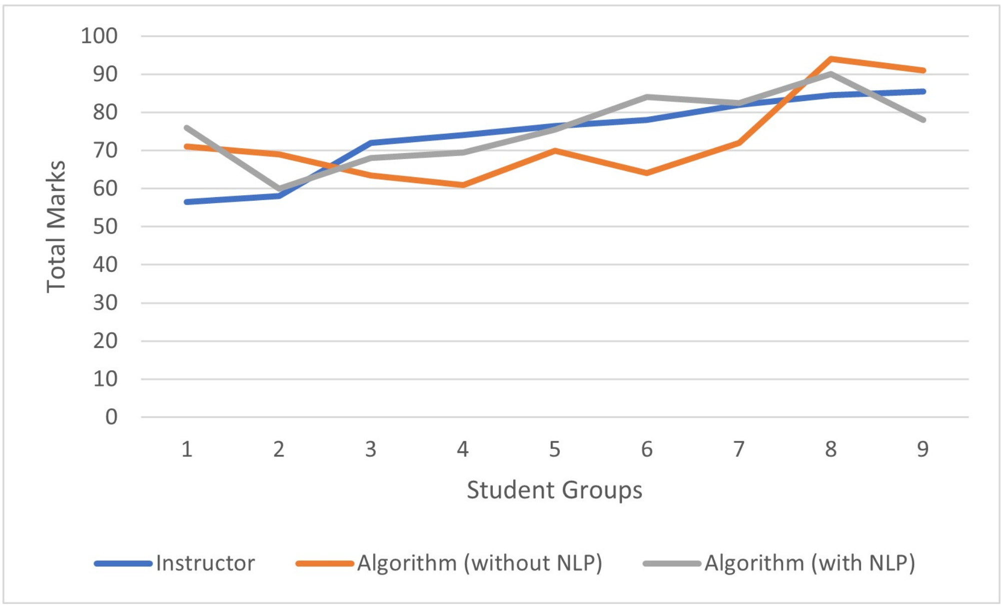


Figure 6.19: Matching Texts Algorithm With vs. Without NLP in Elfenroads

The blue line in Fig. 6.19 and 6.20 represents the instructor's total marks for the Elfenroads and Gas Station case studies, respectively. while the orange line represents the total marks without the use of natural language processing methods. Although the outcome is acceptable for some groups, we decided to improve the algorithm in order to give closer marks to the instructor. Therefore, we decided to use the Stanford CoreNLP when matching texts in a more conceptual way. As we discussed in chapter 4, we first split the entire text into a list of words and then found parts of speech for each word. We removed some of the parts of speech afterward, e.g., prepositions, and transferred all words to their base forms using the lemmatization method. Finally, we compare the same parts of speech from the instructor to the student's solutions in terms of syntactic and semantic similarities. Eventually, a threshold value is returned based on the similarities between each group of words, and if the threshold value is greater than 0.55, we can say those texts are matched. The grey line in Fig. 6.19 and Fig. 6.20 shows the total marks

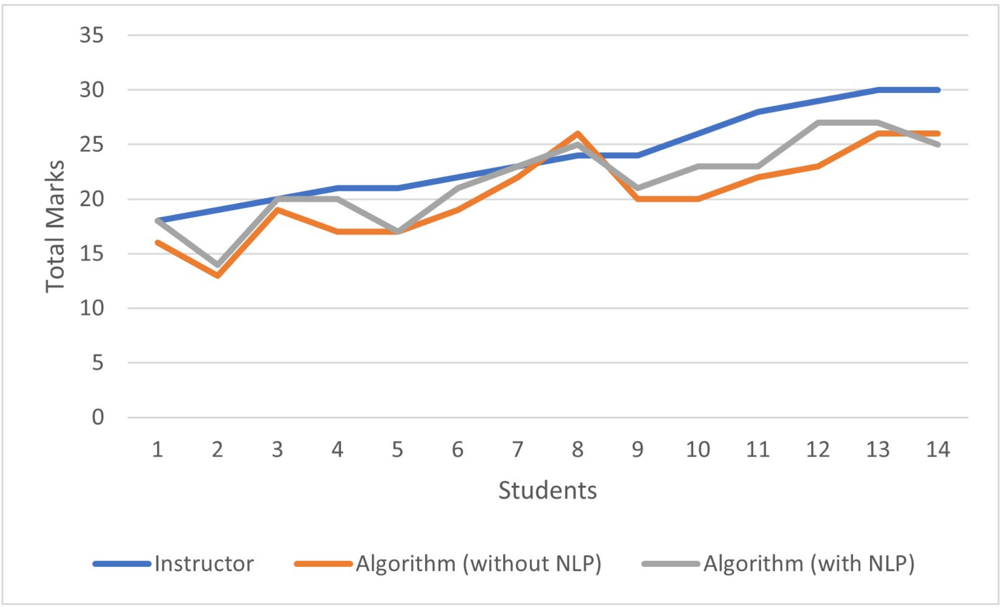


Figure 6.20: Matching Texts Algorithm With vs. Without NLP in Gas Station

when using natural language processing methods. It is clear that the student's marks are closer to the manual grading. The reason for some large differences between grades with and without natural language processing, e.g., group 6 in Fig. 6.19 and student 12 in Fig. 6.20, is that they sometimes used long sentences as steps that the algorithm without NLP could not match them. Also, students used many prepositions in some texts, and as a result, the threshold value was low, and matching them failed. The students rarely received higher marks when using the algorithm without NLP, e.g., groups 2 and 9 in Fig. 6.19 and students 8 and 14 in Fig. 6.20. The reason is that some sentences were matched based on some unimportant parts of speech, i.e., stopwords.

To find the exact threshold, we examined different values when grading both case studies. We found that 0.55 gave the best result on average. Table. 6.3 shows the different marks for each group when setting the threshold (T) equal to 0.45, 0.50, 0.55, and 0.65. For most cases, the grades in column 4, where T = 0.55, are closer to the manual grading.

Fig. 6.21 also plots the differences between the four alternative thresholds. The dark blue box represents the instructor's grading, and when we set a lower threshold (orange box), the total marks got higher, and as the threshold went up, the grades got lower (light blue box).

Table 6.3: Different Grades for Different Threshold Values

Manual Grading	(T = 0.45)	(T = 0.50)	(T = 0.55)	(T = 0.65)
56.5	80	78	76	68
58	69	61	60	49
72	76	73	68	61
74	77	69	66	56
76.5	83.5	79.5	75.5	70.5
78	90	86	84	71
82	87.5	84.5	82.5	70.5
84.5	96	92	90	82
85.5	91	84	78	73

In summary, the average of total marks when we considered the threshold equal to 0.45 was 83.3, for 0.50 it was 78.55, for 0.55 it was 75.94, and for 0.65 it was 66.77. Since the average instructor's total mark is 74.11, we can conclude that the threshold of 0.55 (yellow box) is the best value when matching texts.

As it is mentioned in chapter 4, we consider different weights when comparing parts of speech (algorithm 3). Most parts of speech that are used in the English language are nouns, verbs, adverbs, and adjectives, respectively [62]. The most important parts of speech when comparing steps are verbs, followed by nouns. Therefore, it is more important that the student use the same nouns and verbs or their synonyms as the instructor's solution. Hence, we examined multiple weights for nouns and verbs, and the best result we achieved was when we set verbs and nouns to be 1.5 times heavier than other parts of speech. For instance, in the Gas Station case study, when we set the threshold to 1, the average grade was 20.14, considering the threshold of 2, the average was 21.06, and with the threshold of 1.5, the average was 22.42, while the average of manual grading was 23.92. Also, for the Elfenroads case study, the average grade was 71.83, 75.94, and 77.94 for threshold equals 1, 1.5, and 2, respectively, while the average manual grading was 74.11.

Fig. 6.22 depicts the different students' marks in the Gas Station case study, while

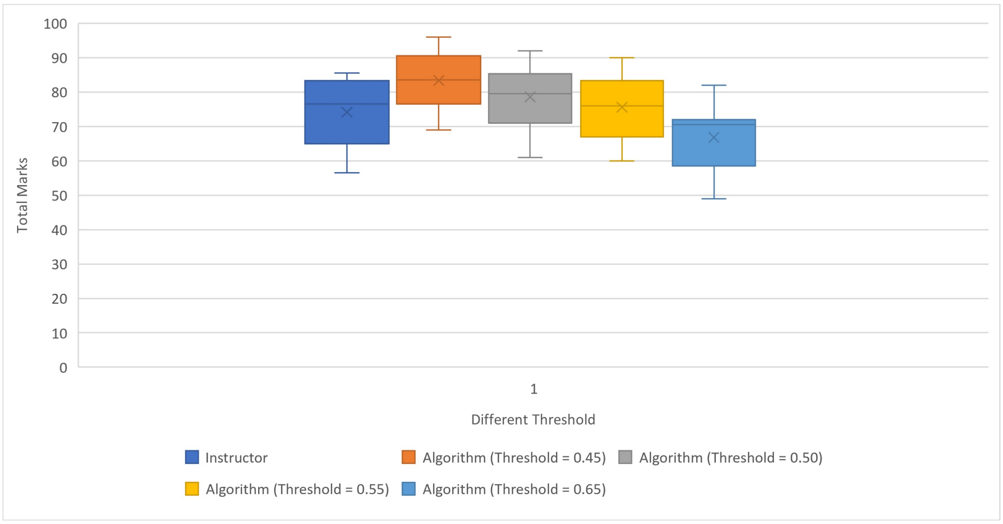


Figure 6.21: Different Matching Texts Threshold Values

Fig. 6.23 depicts the different marks in the Elfenroads case study, taking into account three different coefficient weights when comparing nouns and verbs in the matching text algorithm. The blue bar demonstrates the manual grading, while the orange, grey, and yellow bars show automated grading marks with coefficients equal to 1, 1.5, and 2, respectively. In most cases, the grey bar is closer to the instructor's mark compared to the other two bars.

Finding 6. However, the algorithm worked fine without NLP for some groups, but Stanford CoreNLP improved the quality of the grading algorithm. When considering NLP, in the Elfenroads case study, the average marks improved from 71.83 to 75.94, while the manual grading average was 74.11 (from 14.48% to 7.61%). In the Gas Station case study, the average marks increased from 20.42 to 21.71, while the manual grading average was 23.92 (from 15.75% to 6.69%).

6.9 Conclusion

This chapter describes the case studies to assess the efficacy of the automated grading algorithm presented in Chapter 4. The grading method is applied to two case studies. In case study 1, there are 14 student models, and there are 9 groups of student models in case study 2.

When comparing the automated grading algorithm result to the instructor's manual grading result, the average difference for case study 1 was less than 10%,

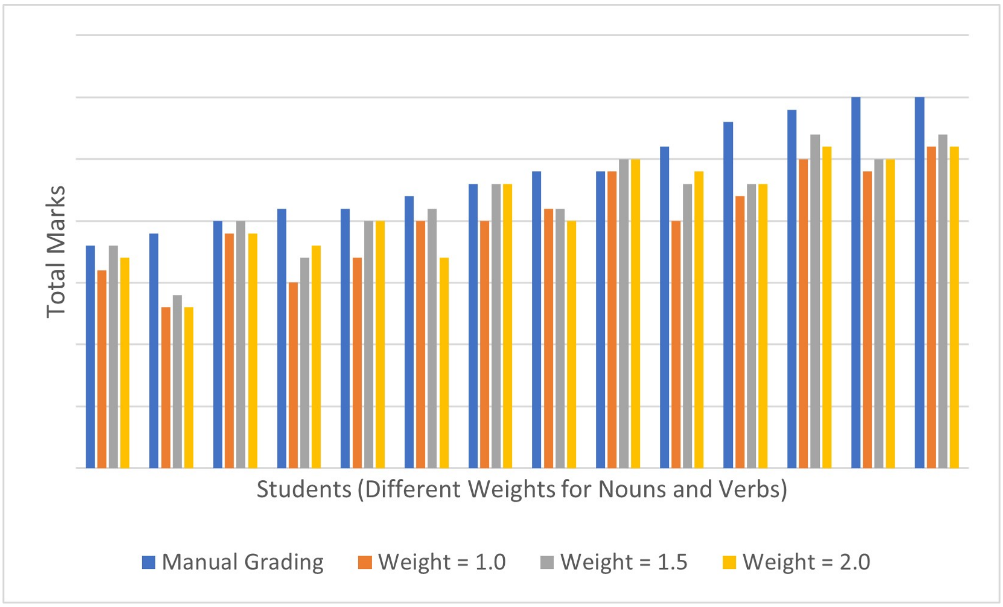


Figure 6.22: Different Weights for Nouns and Verbs in the Gas Station Case Study

whereas the average difference for case study 2 was around 7.5%. We applied some methods to achieve a closer result to manual grading by answering six research questions that discuss the effectiveness of configuration settings, using multiple solutions, flattening steps, and using NLP methods to improve the accuracy of automated grading. They also indicate how automated grading helps ensure fairness for students. Finally, our algorithm was able to automatically grade the first case study by 6.69% and case study 2 by 7.61% difference compared to manual grading.

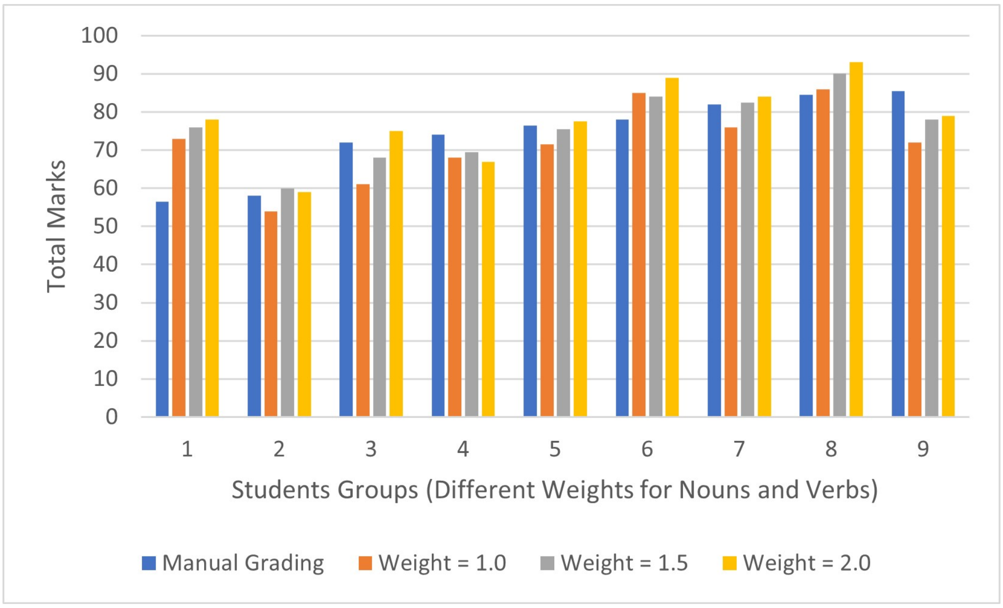


Figure 6.23: Different Weights for Nouns and Verbs in the Elfenroads Case Study

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Many computer science courses require students to do assignments or answer test questions involving use case diagrams. Manual grading of these diagrams is commonly done by instructors by comparing each student's solution to the template solution that they provided for the assignment. This might be a challenging task, especially when there are many use cases with complicated relationships and steps or if there are a high number of student papers to grade. Furthermore, in use case diagrams, a specific problem may have several design solutions, and it is sometimes impossible to flatten all use case steps manually for all students' solutions.

This thesis presents an automated grading method for use case diagrams. It is going to be very useful for instructors by assisting in the assessment of their students. They can grade the use case models in all aspects, i.e., structural, name, and text matching. In order to automate grading, we reused two metamodels that were presented for the automated grading of class diagrams. One metamodel establishes mappings between an instructor's solution and student solutions, and the other assigns and stores grades to model elements. The presented methods and algorithms will help us achieve our goal of automatically grading use case diagrams and providing students with instant feedback on both quantitative and qualitative measures.

We evaluated the effectiveness of our automated approach for grading use case diagrams in practice. Particularly, we compare manual grading against automated grading in two case studies. We developed a configuration settings panel with 28 configurable options that allow the grading tool to be customized to a specific instructor's grading style. We discovered that customizing the algorithm to an instructor's grading approach brings automated grading scores closer to manual grading scores. After customizing the grading configuration settings to match the instructor's style in a case study with 14 students, the average difference between the manual grading and our tool's grading was less than 7%. In another case study which involves 9 student groups from an advanced modeling course the average difference was less than 8%. Automated grading has been found to be more consistent and capable of ensuring fairness in the grading process when compared to manual grading. We also presented how automated grading improves when multiple solutions match against. In one case study the average differences dropped from 6.69% when using one solution to 4.5% when using all three available solutions. Finally, We offered flattening steps before matching them in our algorithm. In addition to flattening, we combined our algorithms with Stanford CoreNLP methods to make the automated grading more effective. In one case study, the average differences when using NLP was 7.61% while it was more than 14% without using any NLP methods.

7.2 Threat to Validity

There are some threats to validity in our tool when grading use case models automatically. The first threat is related to biases in grading the assignments. We relieved this threat by examining two case studies that had been manually graded by the instructors. So, instructors had not made any assumptions that their graded assignments would be graded automatically by the tool, and they did not know the result of the tool before grading manually.

The number of samples examined is the second threat. We evaluated our work using two real-world assignments. The quality and accuracy of the study are impacted by small sample sizes, and the majority of statisticians suggested that a sample size of 100 is necessary to obtain any form of significant results [63, 64].

At this point, there were 14 individual submissions for one case study and nine group submissions of seven students, which means altogether 77 students and 21 different models contributed to these two assignments. Although our automated grading results are close to manual grading, but our sample size might not be large enough, and we need to examine more case studies in the future to evaluate our work more confidently. In addition, although we cannot generalise our result based on this limited sample size, we repeated research questions 2, 3, and 4, which were also answered in the automated grading of the UML Class Diagram [24] and we can conclude that the finding generalises to more than one modelling language.

The third threat is multiple aspects of fairness. We just investigated the consistency of the instructors when grading assignments manually. We need to consider other aspects of fairness, such as the situation when the instructor's solution is not good enough to compare with the students' solutions. In this case, some students can have a more precise and correct answer than the instructor.

The number of instructors' grading styles is the fourth threat. In our case studies, we just examined our algorithm using one instructor's style in each case. We need to consider different instructor grading styles for each assignment.

Finally, our current text matching algorithm cannot match very short sentences with long sentences using Stanford CoreNLP. There were some examples of students' solutions in which they used very short sentences as steps, so, we cannot guarantee that this algorithm always works fine.

7.3 Future Work

As it is mentioned, in some case studies, students wrote just a few words as a step instead of a complete sentence. In such cases, the matching text algorithm is not able to match those steps if they have the same meaning. To deal with this issue, we can either examine some other NLP methods or modify the presented algorithms to be able to do this kind of matching. Since those students who write short sentences often just mention keywords, we can also find all keywords in the teacher's solution (rather than comparing all parts of speech) and then just compare keywords.

We plan to extend our approach to include grading additional UML models, such as sequence and activity diagrams. When presenting more UML models in our tools, we should develop generic packages that can be reused for other modelling languages. In this way, presenting the automated grading of the next modelling language would be easier and take less time.

In addition, we want our algorithm to be able to provide more detailed feedback, such as in exactly which step the student lost points. Finally, it may provide data such as the average for the entire class and the accuracy rate of each element.

Bibliography

[1] N Haji Ali, Zarina Shukur, and Sufian Idris. “Assessment system for UML class diagram using notations extraction”. In: *International Journal on Computer Science Network Security* 7 (2007), pp. 181–187.

[2] Ivar Jacobson. *The unified software development process.* Pearson Education India, 1999.

[3] Reza Fauzan et al. “A different approach on automated use case diagram semantic assessment”. In: *International Journal of Intelligent Engineering and Systems* 14.1 (2021), pp. 496–505.

[4] Jakub Głowacki, Yelizaveta Kriukova, and Nataliya Avshenyuk. “Gamification in higher education: Experience of Poland and Ukraine”. In: (2018).

[5] I-Hui Hwang et al. “Mediating effects of computer self-efficacy between learning motivation and learning achievement”. In: *Education Management, Education Theory and Education Application.* Springer, 2011, pp. 67–73.

[6] Haytske Zijlstra et al. “Prevention of reading difficulties in children with and without familial risk: Short-and long-term effects of an early intervention.” In: *Journal of Educational Psychology* 113.2 (2021), p. 248.

[7] Felipe Restrepo-Calle, Jhon J Ramırez Echeverry, and Fabio A González. “Continuous assessment in a computer programming course supported by a software tool”. In: *Computer Applications in Engineering Education* 27.1 (2019), pp. 80–89.

[8] Jacqueline P Leighton. “Students’ interpretation of formative assessment feedback: Three claims for why we know so little about something so important”. In: *Journal of Educational Measurement* 56.4 (2019), pp. 793–814.

[9] Michael Striewe et al. “Towards an automated assessment of graphical (business process) modelling competences”. In: *INFORMATIK 2020* (2021).

[10] Thanasis Daradoumis et al. "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.

[11] Helder Pina Correia, José Paulo Leal, and José Carlos Paiva. "Enhancing feedback to students in automated diagram assessment". In: (2017).

[12] Megan Robin Kennedy. "LinkedIn learning product review". In: *Journal of the Canadian Health Libraries Association/Journal de l'Association des bibliothèques de la santé du Canada* 40.3 (2019), pp. 142–143.

[13] Arnis Silvia. "COURSERA ONLINE COURSE: A PLATFORM FOR ENGLISH TEACHERS'MEANINGFUL AND VIBRANT PROFESSIONAL DEVELOPMENT." In: *TEFLIN Journal: A Publication on the Teaching & Learning of English* 26.2 (2015).

[14] Steve Kolowich. "How EdX plans to earn, and share, revenue from its free online courses". In: *The Chronicle of Higher Education* 21 (2013), pp. 1–5.

[15] Education Australia. Department of Employment and Training. *SkillShare, It's Working: A Summary of Findings from the First Year Review of SkillShare*. Australian Government Pub. Service, 1990.

[16] René F Kizilcec, Chris Piech, and Emily 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.

[17] Mark Kassop. "Ten ways online education matches, or surpasses, face-to-face learning". In: *The Technology Source* 3 (2003).

[18] Weiyi Bian, Omar Alam, and Jörg Kienzle. "Automated grading of class diagrams". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE. 2019, pp. 700–709.

[19] Matthias Schöttle et al. "Feature modelling and traceability for concern-driven software development with TouchCORE". In: *Companion Proceedings of the 14th International Conference on Modularity*. 2015, pp. 11–14.

[20] Rajeev Alur et al. “Automated grading of DFA constructions”. In: *IJCAI’13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. 2013, pp. 1976–1982.

[21] Firat Batmaz and Chris J Hinde. “A diagram drawing tool for semi–automatic assessment of conceptual database diagrams”. In: (2006).

[22] Humasak 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.

[23] Pete Thomas, Kevin Waugh, and Neil 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.

[24] Weiyi Bian, Omar Alam, and Jörg Kienzle. “Is automated grading of models effective? assessing automated grading of class diagrams”. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 2020, pp. 365–376.

[25] Michael Striewe and Michael Goedicke. “Automated assessment of UML activity diagrams”. In: *Proceedings of the 2014 conference on Innovation & technology in computer science education*. 2014, pp. 336–336.

[26] Athanasios Tsintsifas. “A framework for the computer based assessment of diagram based coursework”. PhD thesis. University of Nottingham, 2002.

[27] Ambikesh Jayal and Martin Shepperd. “The problem of labels in e-assessment of diagrams”. In: *Journal on Educational Resources in Computing (JERIC)* 8.4 (2009), pp. 1–13.

[28] Pete Thomas, Neil Smith, and Kevin Waugh. “Automatic assessment of sequence diagrams”. In: (2008).

[29] Christos Tselonis, John Sargeant, and Mary McGee Wood. “Diagram matching for human-computer collaborative assessment”. In: (2005).

[30] Rúben Sousa and José Paulo Leal. “A structural approach to assess graph-based exercises”. In: *International Symposium on Languages, Applications and Technologies*. Springer. 2015, pp. 182–193.

[31] Hideki Shima. “Wordnet similarity for java (ws4j)”. In: *HYPERLINK" https: ll-code. google. com/p/ws4jl" https: llcode. google. comlp/ws4j* (2016).

[32] Zhibiao Wu and Martha Palmer. “Verb semantics and lexical selection”. In: *arXiv preprint cmp-lg/9406033* (1994).

[33] Sébastien Harispe et al. “Semantic similarity from natural language and ontology analysis”. In: *Synthesis Lectures on Human Language Technologies* 8.1 (2015), pp. 1–254.

[34] Vinay Vachharajani and Jyoti Pareek. “Framework to approximate label matching for automatic assessment of use-case diagram”. In: *International Journal of Distance Education Technologies (IJDET)* 17.3 (2019), pp. 75–95.

[35] Vinay Vachharajani and Jyoti Pareek. “A proposed architecture for automated assessment of use case diagrams”. In: *International Journal of Computer Applications* 108.4 (2014), pp. 35–40.

[36] Vinay Vachharajani and Jyoti Pareek. “Use case extractor: XML parser for automated extraction and storage of use-Case diagram”. In: *2012 IEEE International Conference on Engineering Education: Innovative Practices and Future Trends (AICERA)*. IEEE. 2012, pp. 1–5.

[37] Mohammad Nazir Arifin and Daniel Siahaan. “Structural and Semantic Similarity Measurement of UML Use Case Diagram”. In: *Lontar Komputer: Jurnal Ilmiah Teknologi Informasi* 11.2 (2020), p. 88.

[38] Vladimir I Levenshtein et al. “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet physics doklady*. Vol. 10. 8. Soviet Union. 1966, pp. 707–710.

[39] Deeptimahanti Deva Kumar and Ratna Sanyal. “Static UML model generator from analysis of requirements (SUGAR)”. In: *2008 Advanced Software Engineering and Its Applications*. IEEE. 2008, pp. 77–84.

[40] Christopher D Manning et al. “The Stanford CoreNLP natural language processing toolkit”. In: *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 2014, pp. 55–60.

[41] Graeme Hirst, David St-Onge, et al. "Lexical chains as representations of context for the detection and correction of malapropisms". In: *WordNet: An electronic lexical database* 305 (1998), pp. 305–332.

[42] Dekang Lin et al. "An information-theoretic definition of similarity." In: *Icml*. Vol. 98. 1998. 1998, pp. 296–304.

[43] Philip Resnik. "Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language". In: *Journal of artificial intelligence research* 11 (1999), pp. 95–130.

[44] Gregory Grefenstette. "Tokenization". In: *Syntactic Wordclass Tagging*. Springer, 1999, pp. 117–133.

[45] Jonathan J Webster and Chunyu Kit. "Tokenization as the initial phase in NLP". In: *COLING 1992 volume 4: The 14th international conference on computational linguistics*. 1992.

[46] John R Searle. "What is a speech act". In: *Perspectives in the philosophy of language: a concise anthology* 2000 (1965), pp. 253–268.

[47] Iskander Akhmetov et al. "Highly language-independent word lemmatization using a machine-learning classifier". In: *Computación y Sistemas* 24.3 (2020), pp. 1353–1364.

[48] Joël Plisson, Nada Lavrac, Dunja Mladenic, et al. "A rule based approach to word lemmatization". In: *Proceedings of IS*. Vol. 3. 2004, pp. 83–86.

[49] Behrang Mohit. "Named entity recognition". In: *Natural language processing of semitic languages*. Springer, 2014, pp. 221–245.

[50] Maximilian Schiedermeier et al. "Multi-Language Support in TouchCORE". In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE. 2021, pp. 625–629.

[51] Dave Steinberg et al. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[52] Uwe Laufs, Christopher Ruff, and Jan Zibuschka. "Mt4j-a cross-platform multi-touch development framework". In: *arXiv preprint arXiv:1012.0467* (2010).

[53] Ryan Languay. "Concern-Oriented Use Case Modelling". In: *McGill University* (2022).

[54] "Rio Grande Games. Elfenroads". In: (2010).

[55] Ryan Languay. “Concern-Oriented Use Case Modelling”. In: (2022).

[56] James W Howatt. “On criteria for grading student programs”. In: *ACM SIGCSE Bulletin* 26.3 (1994), pp. 3–7.

[57] Robin D Tierney. “Fairness in classroom assessment”. In: *SAGE handbook of research on classroom assessment* (2012), pp. 125–145.

[58] Robin D Tierney, Marielle Simon, and Julie Charland. “Being fair: Teachers’ interpretations of principles for standards-based grading”. In: *The Educational Forum*. Vol. 75. 3. Taylor & Francis. 2011, pp. 210–227.

[59] Kay Sambell, Liz McDowell, and Sally Brown. ““But is it fair?”: an exploratory study of student perceptions of the consequential validity of assessment”. In: *Studies in educational evaluation* 23.4 (1997), pp. 349–371.

[60] Paul L Nesbit and Suzan Burton. “Student justice perceptions following assignment feedback”. In: *Assessment & Evaluation in Higher Education* 31.6 (2006), pp. 655–670.

[61] Stéphane S Somé. “Supporting use case based requirements engineering”. In: *Information and Software Technology* 48.1 (2006), pp. 43–58.

[62] Douglas Biber et al. *Longman grammar of spoken and written English*. Vol. 2. Longman London, 1999.

[63] Ranatunga RVSPK, HMS Priyanath, and RGN Megama. “Methods and rules-of-thumb in the determination of minimum sample size when applying structural equation modelling: a review”. In: *J Soc Sci Res* 15.2 (2020), pp. 102–9.

[64] Shelby J Haberman and Sandip Sinharay. “Sample-size requirements for automated essay scoring”. In: *ETS Research Report Series* 2008.1 (2008), pp. i–43.