Politecnico di Torino

Master's Degree Course in

Computer Engineering

Thesis

A Gamified Learning Tool for Conceptual Modeling with UML Class Diagrams

Supervisors
Prof. Marco Torchiano
Prof. Riccardo Coppola
Doct. Giacomo Garaccione

Candidate
Christian Damiano Cagnazzo

Academic Year 2022-2023

Summary

Conceptual modeling is a crucial part of software engineering's information system design phase, allowing the abstraction of real-world concepts and the translation of complex requirements into a coherent system representation. The Unified Modeling Language (UML) is particularly relevant in this context, with the Class Diagram being a widely recognized and adopted UML diagram type. Teaching and learning conceptual modeling, especially with UML class diagrams, can be challenging due to theoretical and practical complexities. Teachers play a crucial role in conceptual modeling education, but challenges arise when dealing with a large number of students. Also, traditional teaching methods may lack the ability to engage students effectively. In recent years, gamification has emerged as a strategy in software engineering education: it consists of *'the use of game design elements in non-game contexts'* to increase motivation and engagement. The main goal of this thesis was to create a gamified educational tool for teaching conceptual modeling with UML class diagrams. First, an in-depth analysis was carried out of the main aspects of gamification and the benefits that its application in various contexts can bring. In particular, the Octalysys framework was analyzed and used as a tool for the application of gamification. After that, the main features and rules of conceptual modeling using UML class diagrams were explored. Based on this, the design and implementation of a gamified web tool for teaching conceptual modeling was carried out. The key feature of the tool is an automatic evaluation system for diagrams created by students. The system performs two different types of analysis: a syntactical one, based on verifying that the syntax rules about UML class diagram modeling are followed, and a semantical one based on a solution that aims to verify the completeness and correctness of the diagram modeled by the students. The gamification mechanics selected and implemented are the following: a system of gaining levels through the acquisition of experience obtainable by completing exercises correctly; an avatar that can be customized with items that can be unlocked by leveling up; a system of immediate feedback after the evaluation of an exercise through highlighting with different colors and description of errors made in a diagram. To evaluate the implemented tool two types of analyses were conducted. The first analysis focused on the automatic diagram evaluation system: a total of 30 diagrams created by students were analyzed by the tool, which generated a list of errors reviewed by a human evaluator to determine the accuracy of the tool's identification of errors and warnings. The analysis revealed that the system is able to correctly detect a very high percentage of violations, in a manner very similar to the evaluation of a human being. For the second analysis, a score expressing the completeness of the gamified environment was calculated using the method provided by the Octalysis Framework. The analysis showed that the gamified experience appears to have a balance between the different core drives defined in the framework. However, it becomes apparent that some core drives are not well-represented at present. For this reason, future development should expand the tool's features with other gamified mechanics that are suited for long-term usage, such as competition mechanisms like leaderboards and quest-line mechanics based on the exercises.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In software engineering, conceptual modeling plays an essential role in the design phase of an information system. A correct and accurate design model is crucial to abstract concepts from reality and to translate complex requirements into a clear and coherent representation of the system.

In this context, the use of the Unified Modeling Language (UML), a graphical representation language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems [1], emerges with particular relevance. Among the most recognized and widely adopted types of UML diagrams is the Class Diagram, which provides a clear and structured way to express the relationships and interactions between various class objects within a system.

Modeling concepts with UML class diagrams is a common topic covered in software engineering university courses, as it is a crucial method for the requirements formulation phase. Learning and teaching concepts modeling, especially using UML class diagrams, can be complex due to related theoretical and practical difficulties [2]. The conceptual modeling process requires a clear understanding of the methods involved to accurately capture the concepts of the system to be developed. In addition, using UML tools requires familiarity with representational conventions and the ability to translate abstract ideas into meaningful diagrams. Therefore, conceptual modeling requires practice and teaching that pays attention to the theoretical and practical challenges it presents. However, the benefits of mastering this field are significant, as it allows software engineers to translate complex ideas into innovative solutions effectively.

Teachers, in conceptual modeling education, play an important role in helping students understand the reasons for their mistakes and guiding them towards more correct modeling practices. However, it can be challenging for teachers when dealing with a large number of students, because it is difficult to provide individualized assistance to each student. In addition, the traditional teaching approach, which relies on textbooks and theoretical lectures, may be boring and insufficient

to stimulate students' interest. Overcoming these challenges is important to ensure that software engineers acquire solid skills in conceptual modeling with UML class diagrams, as this directly impacts the quality of the software developed and the understanding of requirements specifications.

In recent years, a new strategy in software engineering to increase student engagement has emerged: gamification of learning. Gamification is the practice of using elements, strategies, and mechanics that are typically found in games in non-recreational contexts [3]. The benefits of gamification include increased engagement in unattractive activities and interest in unappealing topics thanks to games' features like competitiveness and collaboration designed to capture players' curiosity and motivations.

Applying gamification as a teaching strategy can make conceptual modeling learning more interesting and effective. This allows students to experience theoretical concepts more practically, enhancing their ability to model complex software systems correctly. Additionally, utilizing dedicated education tools with detailed feedback mechanisms embedded in a playful environment may be more successful in capturing students' attention than traditional teacher feedback.

1.1 Goal

The main goal of this thesis is to develop a prototype gamified tool that can assist in teaching the principles of conceptual modeling using UML class diagrams in educational settings.

The intended final result is an application that includes an automatic evaluation system for diagrams created by students, along with gamification features that encourage greater engagement in learning the subject material. To achieve this, a set of syntactic rules about creating UML models must be defined, in addition to a semantic rule evaluation system for every exercise, capable of identifying any rule violations and assigning scores accordingly.

For these reasons, theoretical aspects of conceptual modeling and the UML representation language will be explored, while also examining the challenges that students face while learning this discipline.

Furthermore, the main principles of gamification and how it can offer various benefits in different settings if executed efficiently will be explored. Specifically, the Octalysis Framework's application for gamification will be examined in greater detail. This will enable the identification and selection of the most effective gamified approaches and strategies that are suitable for the development of the tool.

The prototype tool implementation will be explained in detail, and two initial evaluations will be conducted. The first evaluation will test the automatic diagram correction engine for accuracy, while the second evaluation will examine the

effectiveness and completeness of the gamified environment. These evaluations will help determine the potential impact of the tool and provide guidance for future development. Indeed, the plan is to use the tool in an academic environment because using gamification could make learning conceptual modeling easier by increasing students' interest.

The remainder of the thesis is structured in the following way: Chapter 2 introduces relevant background information and the state of the art about gamification and concept modeling with UML class diagrams, while Chapter 3 describes the process of implementation of the tool. In Chapter 4 two different types of evaluation of the tool are presented, and Chapter 5 details the conclusions, current limitations, and plans for future usage of the tool.

Chapter 2

Background and Related Work

In this section, an overview of the thesis context will be provided. The concepts and key features of the Unified Modeling Language, with a specific emphasis on its use in concept modeling, will be defined. Additionally, the gamification technique and a framework for its implementation will be described. Lastly, various scenarios in which gamification can be applied and some examples of its practical applications will be examined.

2.1 Unified Modeling Language

The increasing complexity of modern software solutions presents significant challenges in the analysis, design, and deployment of applications and information systems. In this context, conceptual modeling has become an essential element in achieving success in software projects.

Modeling helps developers and stakeholders to represent complex concepts more understandably through a visual representation that provides a simplified, abstract description of the system. It is simplified because it typically represents only a specific point of view about the system, and it is abstract because only describes a restricted set of elements considered relevant to the scope of the model.

Conceptual modeling offers a significant advantage in defining requirements more comprehensively and efficiently. It enables users' needs and key functionalities of the system to be identified more effectively, providing a strong foundation for design and development.

Moreover, the use of modeling promotes effective and coherent communication among project team members and stakeholders through visual representations understandable to all, which leads to improved collaboration and better teamwork.

In this context, the Unified Modeling Language (UML) has emerged as one of the most widely used tools for information system concept modeling. The main characteristic of UML is its well-defined graphical notation, which allows concepts and relationships within a system to be represented visually. It follows a set of syntax and semantics rules to ensure the model is legal and has meaning. It is independent of the scope of the project, the development process, and the programming language.

The term "unified" refers to the incorporation of various types of diagrams, each designed to analyze a specific perspective or aspect of the system. The most common are [4]:

- Class diagrams: they represent classes in the system with their attributes and relationships between each class;
- Use case diagrams: they give a graphic overview of the actors involved in a system and how they interact;
- Sequence diagrams: they describe how objects interact with each other and the order those interactions occur;
- Activity diagrams: they show workflows in a graphical way;
- Deployment diagrams: they define the hardware of the system and the software in that hardware.

2.1.1 UML for concepts modeling

In this thesis, we will focus on using UML Class Diagrams for concept modeling, as described in section 1.1. While there are other languages available for constructing concept information models, such as Entity-Relationship notation [5], UML was chosen due to its ease of use and comprehensiveness, as well as a requirement for the university course of study.

Creating a concept model is crucial during the initial phases of analyzing the requirements for an information system, which involves gathering data and information to establish a clear concept model. The goal is to create a concept model that captures and illustrates the essential abstract concepts (represented by *classes*) that define the domain of the addressed problem, their characteristics (represented by *attributes*), and the relationship among them (represented by *associations*).

Classes

The main concepts of the domain of the problem addressed are illustrated in a UML model with *classes*. A class is a component of the model that describes a

set of objects with shared characteristics. It must have a non-empty name which by convention is a singular noun and it must be unique in the model. A class is depicted as a rectangle divided into two parts with the name appearing in the first block, as illustrated in Figure 2.1.

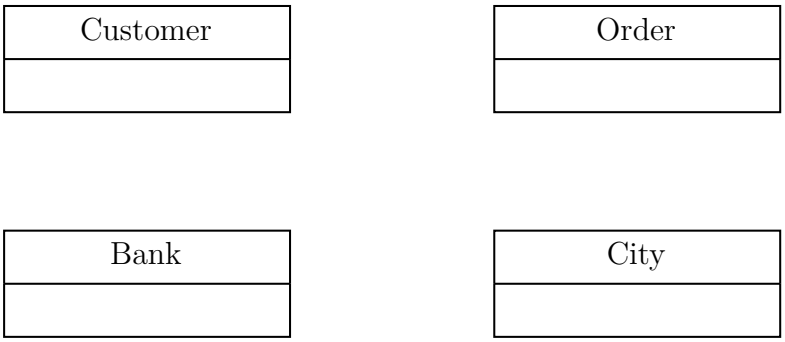

Figure 2.1. Visual representation of classes

Attributes

In a model, every class has its own set of characteristics describing each instance of it. These characteristics are called *attributes*. An attribute includes a name and a type representing the type of data it can hold. This is shown in Figure 2.2. Each instance of the classes will have its values for each attribute.

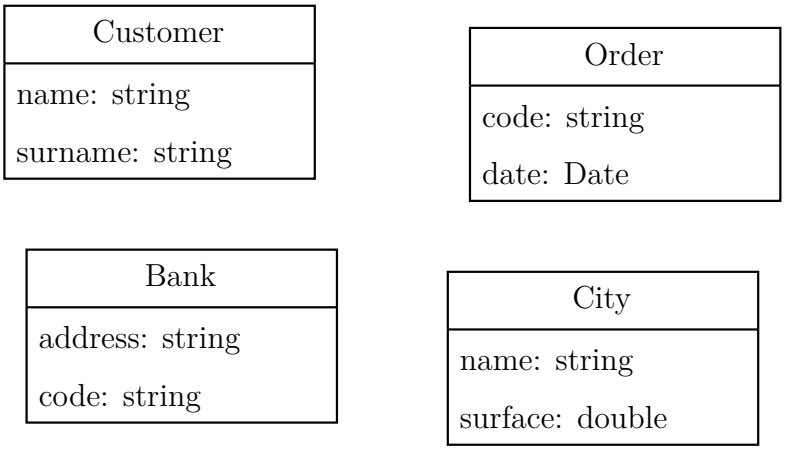

Figure 2.2. Visual representation of attributes

Associations

In a concept model classes alone do not provide enough information. That's why it is necessary to represent logical connections between them. Relationships between classes are illustrated through the concept of *associations*. The idea of association is based on the concept of a mathematical relationship, which is a subset of the Cartesian product between the sets of objects of the classes connected by the association. The graphic representation of an association consists of a line with a name that joins two classes (Figure 2.3).

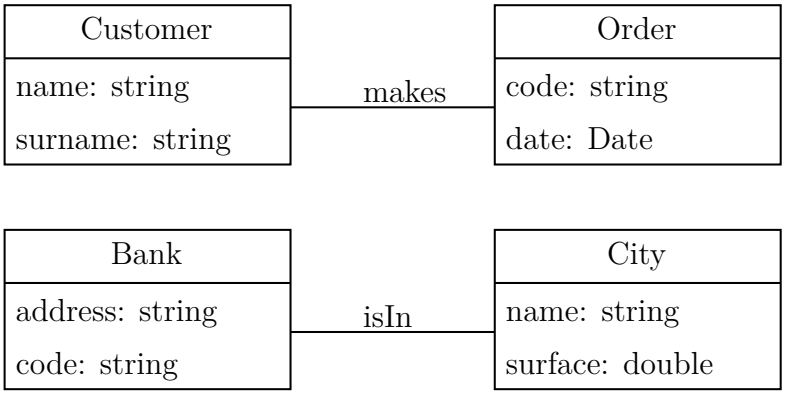

Figure 2.3. Visual representation of associations

It is possible to link a class with itself through an association, which is known as a *reflexive* association. To differentiate the ends of the association, a role name can be specified at each end as shown in Figure 2.4.

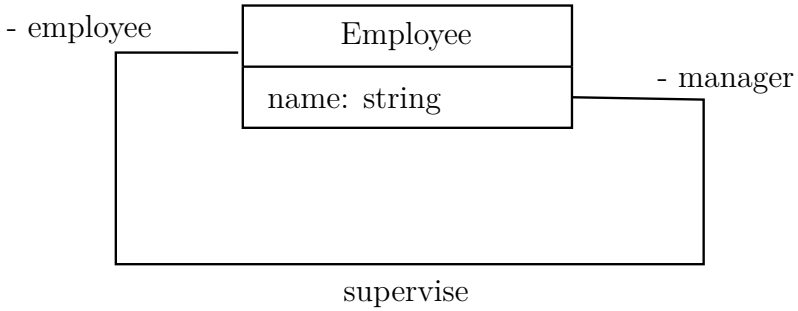

Figure 2.4. Visual representation of reflexive association

When defining an association it is necessary to include the *cardinality* (or cardinality) of it, which specifies the minimum and maximum number of links each

instance of a class can have for that particular relationship. Both sides of the association must have a valid cardinality specified at the end of the link as in Figure 2.5.

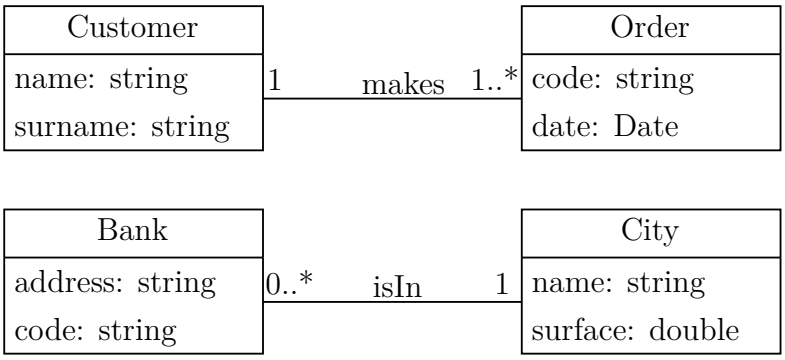

Figure 2.5. Visual representation of cardinalities

The following values can be used to indicate the minimum cardinality:

- **0**: Optional occurrence
- **1**: Mandatory occurrence

The following values can be used to indicate the maximum cardinality:

- **1**: At most an instance
- *****: Many instances

The concept of maximum cardinality enables the identification of three distinct types of associations.

1. *One to one*: an instance of a class is linked with exactly one instance of another class;
2. *One to many*: an instance of a class can be linked with more than one instance of another class;
3. *Many to many*: more instances of a class can be linked with more than one instance of another class.

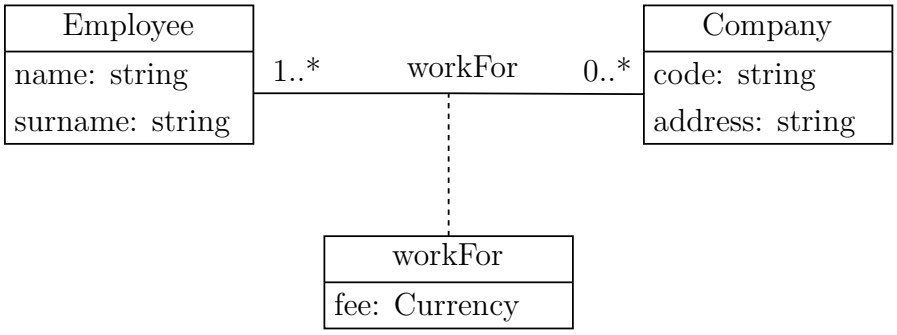

Figure 2.6. Visual representation of an association class

Association classes

In cases where many-to-many associations are defined, it is possible for some information to not belong to either of the two classes. The *association classes* enable the definition of attributes that are assigned to the association and it is illustrated as a normal class linked with a dotted line to the association, as in Figure 2.6.

Identifying an association class can be a complex task. Indeed, while teaching at Turin Polytechnic, professors observed that students often struggle with understanding how and when to use association classes, leading to frequent errors. To avoid this complexity, intermediate classes can be used instead of association classes as they serve as a feasible alternative.

An intermediate class is a standard class placed in between the two classes of the association. This replaces the original connection between the two classes with two separate associations without a name, as illustrated in Figure 2.7. It is important to take note of the new position of the cardinalities, which have now been swapped in comparison to the previous figure.

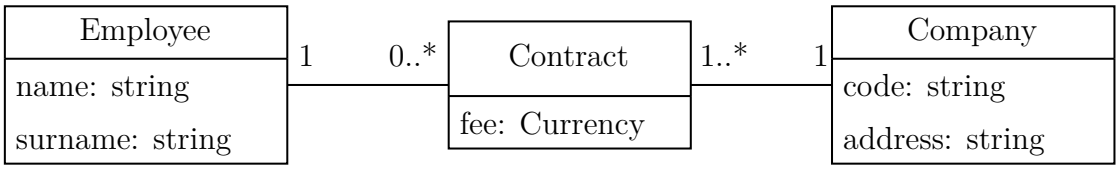

Figure 2.7. Visual representation of an intermediate class

The use of intermediate classes offers two benefits over association classes. Firstly, intermediate classes are standard classes that can be linked to other classes, which is not possible with association classes. Secondly, association classes can only represent relationships where instances of a first class can be linked to the

same instance of a second class once, whereas intermediate classes do not have this limitation.

Generalizations and Specializations

When we model classes that share many attributes and relationships, such as a class that represents a specific case of another class, we can use the concepts of *generalization* and *specialization* to prevent redundancy and maintain a clear and concise model. By using generalization, it is possible to define common attributes and associations once in a general class called *parent* and inherit them in the more specific classes called *children* that specialize the parent by adding additional attributes and associations.

The graphical representation (Figure 2.8) of generalization consists of an arrow with a large triangular tip, going from the most specific class to the most general class.

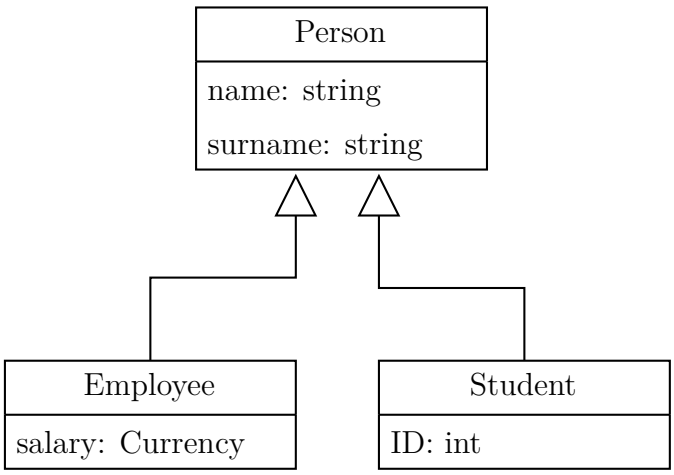

Figure 2.8. Visual representation of generalization and specialization

2.2 Gamification

Gamification is a concept that impacts various aspects of our lives, including education, marketing, business, and even personal wellness. As explained in the introduction section, gamification can be defined as the *use of game design elements in non-game contexts* [3, 6] to increase motivation and engagement. This thesis will explore how gamification can be implemented in different contexts, especially in the educational field, and how it has the potential to increase students'

motivation, stimulate their interest in the topics covered, and encourage them to develop deeper skills through the use of playful elements, such as prizes, badges, leaderboards, and challenges.

To successfully implement gamification, it's important to carefully design it and continuously evaluate the results. It's also crucial to maintain a balance between the fun elements of gamification and the seriousness of the training objectives to ensure that the learning is both meaningful and long-lasting.

2.2.1 The Octalysis Framework for Gamification

One of the most commonly used frameworks for assessing how well gamification is implemented in a gamified system is called Octalysis [7]. As explained by the author, the framework prioritizes a Human-Focused design over a Functional-Focused design, recognizing that individuals in a system have emotions, concerns, and motivations that impact their engagement. By optimizing these factors, gamification can improve user engagement and motivation.

The framework outlines eight Core Drives, which are represented in an octagon shape (Figure 2.9), and define various aspects of human behavior that are motivated by gamification. These core drives are summarized below.

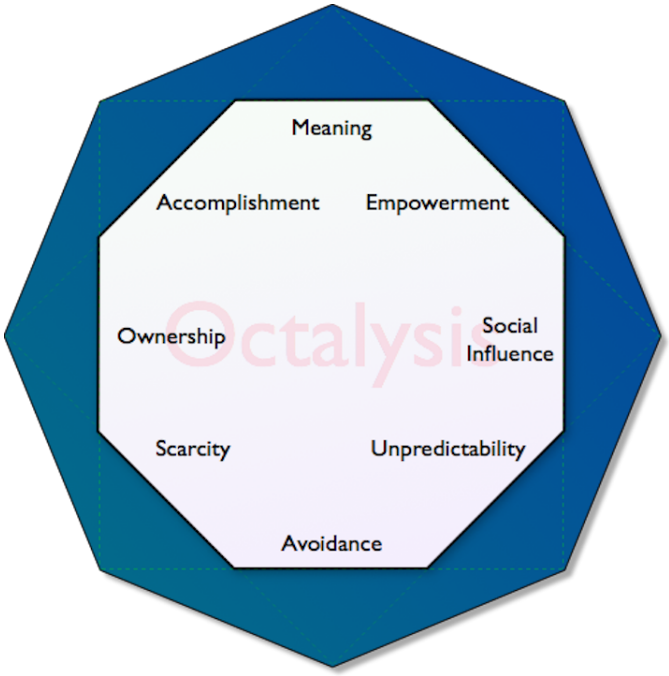

Figure 2.9. The Octalysis Framework

Epic Meaning & Calling This drive is about the need to feel part of something bigger, to have a sense of purpose and meaning in the action taken. Users are motivated when they feel involved in an activity that has an impact and leaves a meaningful legacy. Narrative, elitism, and humanity heroes can influence this drive.

Development & Accomplishment Users are motivated to achieve goals, develop skills, make progress, and overcome challenges. They are driven by the challenge and the sense of personal growth that comes with accomplishing tasks and overcoming obstacles. Elements such as rewards, badges, status points, leaderboard rankings, progress bars, and boss fights are all part of this core drive.

Empowerment of Creativity & Feedback People are more encouraged when they can see the fruits of their work and creativity and when they receive immediate feedback to respond in turn. Milestones unlock, dynamic feedback, chain combos, and boosters are examples of elements belonging to this drive.

Ownership & Possession The desire to own and improve something can be a powerful motivator. Users tend to be more motivated when they feel a sense of ownership. This is particularly true in gaming, where players who have customized their profiles or avatars are more likely to feel a stronger attachment to them and aim to improve them even further. Fall under this core, functionalities as exchangeable points, virtual goods, and avatars.

Social Influence & Relatedness This drive is based on motivation from social interactions and the need to belong to a community. Users are driven to participate in activities when they can share and compete with others, gain recognition, and feel part of a social network.

Scarcity & Impatience It is about the motivation generated by the feeling of wanting something you can't have or having to act quickly. Users are stimulated when they perceive the scarcity of resources or the time limit to obtain a reward or achieve a goal. Appointment dynamics and countdown timers can increase people's encouragement.

Unpredictability & Curiosity Novelty, unpredictability, and curiosity can generate incentives in users, which are stimulated when there are surprise elements and secrets to be discovered like easter eggs, glowing choices, and random rewards.

Loss & Avoidance Fear of missing something or not taking advantage of an opportunity encourages users to participate to avoid negative consequences like for example progress loss.

Octalysis categorizes its Core Drives into two groups. The right side in Figure 2.9 contains the Right Brain Core Drives, which are connected to creativity, self-expression, and social interaction. Meanwhile, the Left Brain Core Drives are situated on the left and are linked to logic, calculations, and ownership [7].

Left Brain Core Drives are *extrinsic* motivators. They are the things that motivate participants to achieve something, such as a goal or any object that they desire. On the other hand, Right Brain Core Drive are *intrinsic* motivators. These motivators do not require a specific goal or reward to be achieved. Activities such as being creative, spending time with friends, and experiencing unpredictability can be rewarding in and of themselves. It is important not to underestimate this aspect, as numerous studies have demonstrated that when extrinsic motivators are removed, user motivation can drop significantly lower than it was prior.

The Core Drives can be further classified into two distinct categories: *White Hat Drives*, located at the top of Figure 2.9 and *Black Hat Drives*, located at the bottom. White Hat Drives aim to make users feel powerful, in control of their actions, and satisfied with themselves. Meanwhile, Black Hat Drives exploit negative feelings in users, making them feel anxious, addicted, obsessed, or generally worried, to motivate them to participate in the gamified system.

The Octalysis Framework [7] also provides a way to assign a score to the gamified environment in order to express its completeness and quality: each of the eight Core Drives is assigned a number between 0 and 10 based on personal judgment, data, and experience flows. These numbers are then squared and added together to calculate the final Octalysis Score and represent it in an octagon.

Here are two examples of Octalisys Score calculations, one from the popular game Candy Crush (Figure 2.10) and the other from the social network Facebook (Figure 2.11) [7].

Looking at Candy Crush, we can see that all the motivational factors are present in a balanced way, leading to a high final score. However, the Meaning drive is assigned a lower score since the game is a simple puzzle game, making it difficult to incorporate elements like narrativity and heroes. Instead, mechanics like finding combos and boosters, passing levels, rewards, and sharing scores with friends are more prevalent, increasing the scores assigned to the Empowerment, Accomplishment, and Social Influence drives. The Scarcity, Avoidance, and Unpredictability drives are also highly rated due to game mechanics like countdowns, random score combos, and a limited number of moves.

It is often easy to associate specific elements and scores with pre-existing games, but it's important to note that gamification through the Octalysis Framework can

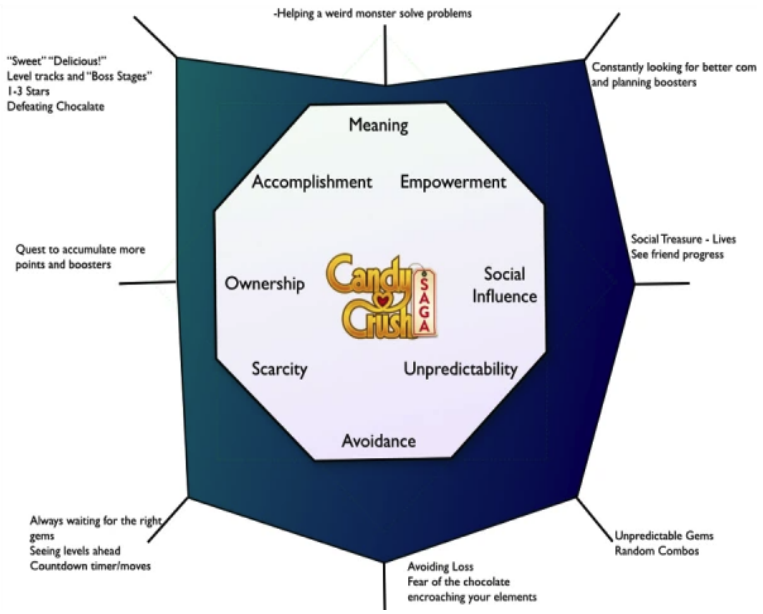

Figure 2.10. Candy Crush Octalysis Score - 432

be applied to a wide range of contexts. Even in analyzing Facebook's Octalysis Score, we can see that many game mechanics are present, resulting in a higher score than that of Candy Crush. While analyzing the octagon, we observe that the mechanics related to the Meaning, Accomplishment, and Scarcity drives were fewer in number, whereas the remaining drives had very high scores. For instance, the mechanics of showing information to selected people, posting photos and memories, etc. create a sense of belonging that aligns with the Ownership drive. Since the platform is a social network, the Social Influence drive had the highest score as users can interact with friends, share ideas, and create groups or communities. Additionally, features such as the automatic refresh of content and immediate feedback through likes and comments also contribute to high scores for the Empowerment and Unpredictability drives.

2.2.2 Gamification in Education

In recent years, gamification has been increasingly used as an effective strategy for increasing student motivation and participation in educational contexts [8]. Typical elements found in gamified systems in this context include rewarding players for their actions, using unknown and unexpected events to increase motivation, competition with other participants, and features that make the experience more

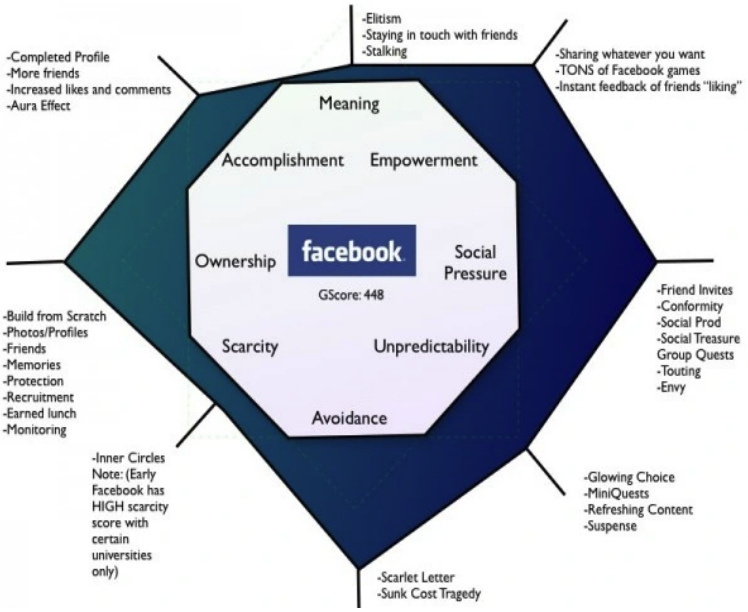

Figure 2.11. Facebook Octalysis Score - 448

enjoyable. It has proven to be a successful technique for simplifying complex subjects and motivating students to complete tasks in the classroom.

Gamification has been most successful in the area of software engineering. This is because the topics covered in software engineering can often be complex or may require monotonous and repetitive tasks and gamifying the educational material can improve comprehension of difficult topics and help to better understand these areas [9].

An example of gamification being used in an educational context, in particular in the field of software engineering, is given by the CleanGame platform implemented by dos Santos et al. [10] and used to assist with the process of refactoring, which is a common practice during software development. Refactoring involves changing the internal structure of code to make it easier to understand and maintain, without altering its functionalities. During this practice, developers need to identify and correct code smells, which are instances of poor design and implementation choices that can affect code maintainability and understandability. The CleanGame platform implements a quiz that helps students identify code smells and presents examples of code that contains code smells to fix. According to the authors, the results of the study were mostly positive, with a 50% improvement in correcting code smells. However, the study was limited and conducted with a small sample of participants, so further experimentation is necessary to conclude that gamification can benefit the refactoring process.

In the context of software development, more precisely about software testing, another example of the application of gamification is provided by the CodeDefenders platform [11]. This is an online game where attackers and defenders can compete. Attackers use a code editor to introduce artificial faults. Every "attack" creates a mutant, which is a version of the Java class being tested that may contain one or more faults. Defenders use a code editor to write JUnit tests. These tests are executed on the mutants created by the attackers. If a test passes on the class under test but fails on a mutant, that mutant is considered detected or "killed" and is removed from the game [12]. A preliminary assessment on gamifying a software testing course with the Code Defenders game [13], showed that its application brought very positive results: students enjoyed the experience and were well engaged, mainly due to the leaderboard and the scoring system to which they paid much attention.

Although gamification can provide benefits like improved student engagement and motivation, creating a gamified environment that also produces educational outcomes is not an easy task. It's essential to recognize that gamification alone cannot solve all educational problems and that other aspects of teaching cannot be ignored. Depending entirely on gamification can result in drawbacks, such as superficial learning or oversimplification.

2.2.3 Gamification in Conceptual Modeling learning

As described in the previous section, gamification has been applied to different fields within computer engineering, such as mutation testing [11], refactoring [10], and requirements definition [14]. In this case, as specified in Section 1.1, the goal of this thesis is to implement a gamified tool for teaching conceptual modeling using UML.

In the literature, there are some examples of gamification applied to concept modeling. One of these is a plugin called Papygame [15], created for Papyrus [1], a tool that allows the creation of UML class diagrams and other modeling languages based on UML. With this plugin, students can complete assignments made up of different exercises, each with its own difficulty level. These exercises can either be played as a game of Hangman, where a new piece of the hanged man is drawn for every mistake made, or without any game elements at all. If you complete an exercise without drawing the full hangman picture, you'll unlock new exercises and rewards. If you fail, however, you won't unlock any new exercises and the full hangman picture will be displayed. The plugin's usability and user experience were tested with students, and the results were promising. The authors plan to

[1] https://www.eclipse.org/papyrus/

improve the plugin and use it more in the future.

Another example of usage is LearnER [16]. It features a gamified editor that allows users to create UML class diagrams and Entity-Relationship diagrams. The tool uses common elements such as points awarded for correctly solved exercises, leaderboards, progress indicators, and hints toward the correct solution expected for the model. According to the authors, the tool has been in continuous use since 2017, and gamification and feedback have been identified as effective and beneficial for the learning process.

One additional related work is by Cosentino et al. [17] where authors defined a gamified system for teaching Unified Modeling Language that includes a game model composed of levels with increasing difficulty. Their work is noteworthy because it focuses on aspects that are often overlooked in-game tools, such as cheating prevention (by encrypting the game status) and user privacy (by allowing users to decide whether or not their in-game progress can be collected by developers).

Jurgelaitis et al. [18] conducted a study that supports the benefits of using gamification in the context of concept modeling. The study involved undergraduate students who were taking a UML modeling course. The gamification mechanics that were applied included levels, gradual unlocking of course content, points, coins, items, badges, and leaderboard.

The study found that the gamification application had a positive effect on the student's grades [19]. The student's average grade for the gamified course was higher by 0.3 points in comparison to the previous year's non-gamified course. Moreover, a student questionnaire confirmed the positive impact of gamification on student motivation.

The previously described use cases demonstrated encouraging results and served as inspiration for the development of the tool discussed in this thesis. The subsequent sections will elaborate on its detailed description.

Chapter 3

Tool Implementation

In this chapter, the key elements and processes involved in creating the application subject of this thesis will be discussed. The aim is to develop a gamified prototype tool for teaching conceptual modeling with UML Class Diagrams in a university environment that incorporates an automatic evaluation system of the students' diagrams and various gamification features to make learning more enjoyable for students. The first two sections will provide a detailed explanation of these two components, while the third section will give a general overview of the tool, including the technologies employed and its various functionalities.

3.1 Diagram Evaluation

In order to implement a tool that can effectively assist students in learning correct modeling practices in a classroom environment, a way to assess the accuracy of their diagrams is needed. Therefore, the initial stage was to establish a correctness check within the tool, which examines the students' diagrams and generates a list of all violations present.

Following commonly used specifications regarding conceptual modeling [20], the possible violations were divided into three different categories:

- *Syntax Errors*: violations of the syntax rules defined for UML class diagrams;
- *Semantics Errors*: violations that are specific to the exercise for which the student is modeling a class diagram;
- *Pragmatic Quality Warnings*: violations that are not considered errors, but still represent something that can be modeled in a more correct way.

3.1.1 Syntax Checking

When creating a conceptual model, the initial step is to ensure that the language rules are followed. This guarantees that the resulting diagram is valid and proposes the intended meaning. The list of syntax rules to be observed implemented in the tool is demonstrated below.

1. *Each class must have a non-empty name and it must be unique in the model.*
2. *Each class must have at least one valid attribute.*
3. *Each class must have at least one valid association.*
4. *Each attribute must have a non-empty name and a valid type, separated by a colon, and it must be unique in the class.*

   The types of attributes used in this context correspond to the principal variable types defined in the Java language; these attributes are presented in Table 3.1.
5. *Each association must have a non-empty name.* [1]
6. *Each association must have a non-empty valid cardinality on both sides.*

   The cardinalities allowed are listed in Table 3.2.
7. *In the case of recursive associations, a role name must be specified on both sides.*
8. *An intermediate class can be only between two classes.*

   The implemented check is to verify that a class indicated as intermediate is associated with only two other classes through two unnamed associations.
9. *Foreign key as attributes are not allowed.*

   The implementation of the check ensures that none of the attributes contains the word "id" along with the name of any associated classes.

Any violation of the rules previously described will correspond to a syntax error. In case the syntax evaluation finds missing relevant information (e.g. a class without attributes, an association without a name, an attribute without a type) the semantic check is performed with placeholder values, with the affected elements not being considered in the subsequent evaluation.

[1] *Unless its source or destination is an intermediate class*

Type	Description
int	integer number
float	real number (with decimal part) in single precision
double	real number (with decimal part) in double precision
String	character string, text
boolean	logical value, true/false, yes/no
Date	date in terms of year, month, day
Time	time (hours, minutes, seconds, ...)
Currency	denaro (value and currency)
enum	data type that can only take a list of values
LatLong	geographical coordinates

Table 3.1. List of allowed attributes types

0
1
*
0..1
0..*
1..1
1..*

Table 3.2. List of cardinalities allowed

3.1.2 Semantics Checking

Semantic errors are specific to each exercise and depend on its solution since they are related to how well a diagram models the specific context of the exercise, both in terms of completeness and correctness.

The solution of an exercise includes the elements described below.

- A list of *classes* with:
  - A name;
  - A list of synonyms for the name;
  - A weight that can be *STRONG* or *WEAK*;
  - An optional message to be displayed in case of errors related to the class;
  - A list of attributes;
  - A list of attribute names not allowed for the class.

- Each *attribute* is represented by:
  - A name;
  - A list of synonyms for the name;
  - A type;
  - A boolean indicating if the attribute can be modeled as a class or not.
- A list of *associations* with:
  - The source class and the destination class
  - a list of allowed cardinalities for the source side and the destination side;
  - A name;
  - A list of synonyms for the name;
  - An optional message to be displayed in case of errors related to the association.
- A list of intermediate classes associations including:
  - The two classes that are part of the association;
  - The intermediate class between the two classes of the association.
- A list of generalizations and specializations represented by:
  - The parent class;
  - The list of the children classes.
- Other options are:
  - A list of names that are not allowed for naming classes;
  - A list of names that are not allowed for naming associations;
  - A list of pairs of classes that are not allowed to be associated with each other;
  - A maximum percentage of the errors made compared to the possible errors not to be exceeded to consider the exercise passed.

Every exercise also includes a *title*, a *description*, the maximum *experience* that can be earned by completing it, and the *level* that it is appropriate for. Here is an example of an exercise (Table 3.3) with its solution (Table 3.4)[2]. The diagram corresponding to the solution described in the table is presented in Figure 3.1.

Title	Ethical Purchasing Group
Description	A km-zero store plans to create a management system for an Ethical Purchasing Group. The system includes a product catalog with photos, prices, and order quantities. Farmers can add their available products with prices. Customers, as representatives of buying groups, can place orders via the web. Minimum quantities are checked, and orders are confirmed once sufficient quantities are available. If not, orders are modified or canceled based on precedence rules. Customers receive a summary email once their order is completed
Level	1
Experience	500

Table 3.3. Example of an exercise description

Classes	***Name***	***Weight***	***Synonyms***	***Not allowed attributes***	
	Customer	STRONG	Client	id	
	Order	STRONG			
	Availability	STRONG			
	Week	WEAK			
	Product	STRONG		quantity	
	OrderElement	STRONG			
Attributes	***Name***	***Type***	***Synonyms***	***Class***	***Can be a class***
	name	string		Customer	false
	surname	string		Customer	false
	wallet	int	score	Customer	true
	state	string		Order	false
	price	currency		Availability	false
	date	Date		Week	false
	description	string		Product	false
	quantity	int		OrderElement	false
Associations	***Name***	***Src - Card***	***Dst - Card***	***Synonyms***	
	make	Customer - 1	Order - *, 0..*	generate,produce	
	for	Week - 1	Availability - *, 0..*		
	type	Product - 1	Availability - 0..*		
	OrderElement	Order - 0..*	Availability - 1..*		
N.A. Associations	***Src Class***	***Dst Class***			
	Product	Order			
Intermediate Classes	***Class***	***Src***	***Dst***		
	OrderElement	Order	Availability		
Generalization Classes	***Parent Class***	***Children classes***			
N. A. class names	store				
N. A. association names	see,show				
Error Threshold	15				

Table 3.4. Example of an exercise solution (table)

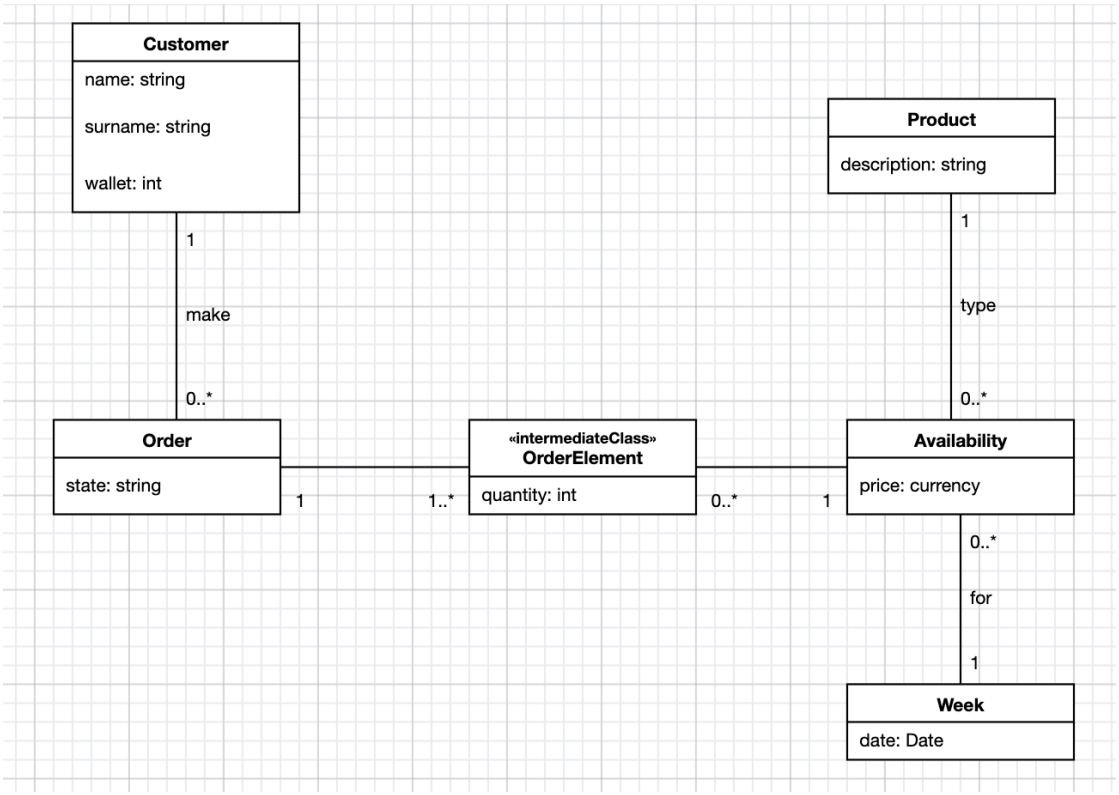

Figure 3.1. Example of an exercise solution (diagram)

Whenever students desire, they can have their diagrams evaluated. The diagram is thoroughly checked for any possible violation. For every violation discovered in the diagram evaluation, there will be either a semantic error (referred to as **SE**) or a pragmatic quality warning (referred to as **PQW**).

The process of evaluating the exercise is explained below.

Classes and Attributes The initial stage of the evaluation involves verifying the presence of all required classes. For each non-intermediate class in the solution, the closest corresponding class in the student diagram is identified and mapped to the desired class. If no match is found, two potential violations may occur.

1. *A class with a strong weight is absent (SE).*
2. *A class with a weak weight is absent (PQW).*

[2]A custom error message can be indicated for each element in the solution, which is not shown in the table due to limited space

3. *A class is probably misnamed (PQW).*

Listing 3.1. Class Search Pseudocode

```
for (solutionClass in solutionClasses){
    if (solutionClass is not intermediate){
        foundedClass = findTheClosestClass(solutionClass, diagramClasses)

        if (foundedClass){
            solutionClass -> foundedClass
            add foundedClass in presentClasses
        } else {
            if (solutionClass weight is STRONG)
                add solutionClass in strongAbsent
            else
                add solutionClass in weakAbsent
        }
    }
}
```

Spelling errors are very common and sometimes there may not be an exact match between a word predicted by the solution and one thought up by a student. Therefore, to avoid considering absent concepts that are actually represented due to a spelling error, the Levenshtein distance between each class in the diagram and the name (including synonyms) of the desired class is calculated to find the closest matching class.

Levenshtein distance is a measure introduced by Russian scientist Vladimir Levenshtein that determines how similar two strings are. It is often applied to perform spell-checks or to do similarity research between texts and represents the minimum number of elementary changes to transform the first word into the second, where elementary change means the deletion of a character, the replacement of one character by another, or the insertion of a character.

In this context, the distance calculation is also normalized to the length of the strings, and the class with the highest match value that is equal to or greater than 75% is selected in order to detect possible spell checks such as "costumer" instead of "customer".

The next step is to examine the attributes of the classes. Given every class in the solution that has a corresponding class in the diagram the existence of its attributes is assessed by comparing their names and synonyms with those of the class in the diagram, checking that these are not part of the list of forbidden attributes. If a match is discovered and a type for the attribute is specified in the solution, it is checked that the type also corresponds.

When an attribute is not identified, an additional check is performed to ensure that it's truly absent. The evaluation algorithm checks if the attribute is absent

because it was modeled as a class. In this case, it is checked whether this type of modeling was allowed or not, otherwise, the attribute is marked as missing.

As a result, the following violations may be identified through attribute analysis:

4. *An attribute name is not allowed (SE).*
5. *An attribute of an existing class is absent (SE).*
6. *An attribute founded has the wrong type (SE).*
7. *An attribute modeled as a class can be an attribute of the related class (PQW).*
8. *An attribute modeled as a class must be an attribute of the related class (SE).*

Intermediate classes The intermediate class check consists of an iteration performed to verify that all intermediate classes provided in the solution have been modeled. For each intermediate class, the two classes between which it is to be modeled are also given by the solution and the check that is described below is performed only if these two classes are present and have been modeled in the diagram by the student.

For each intermediate class predicted by the solution, it is checked that there exists in the model a class designated as "intermediate" that has a relationship to the two associated classes with the intermediate. No checks are made on the class name, only on the presence and correctness of the predicted attributes.

If no matching class is found among those modeled as intermediate, a new iteration is performed by searching among the normal classes in the diagram that have not been mapped to any class predicted by the solution and are therefore apparently unnecessary. Among these, a search is made if there is a class that has a relationship with the two classes associated with the intermediate class. If it is found, a further check is made on the attributes of this class to check if they match those predicted by the initial intermediate class. If the attributes match, it is highly probable that the intermediate class initially searched for, was modeled as a normal class by the student and consequently is marked as found, while reporting the inaccuracy as a pragmatic warning.

Violations that may be found as a result of these checks are as follows:

9. *An intermediate class between two existing classes in the model is absent (SE).*
10. *A class should be indicated as 'intermediate' class (PQW).*

For each class left in the diagram that has not been mapped to a class or an intermediate class in the solution, it is checked that its name is not among the forbidden names for classes. If the name is not forbidden, the class is considered unnecessary. Violations in this case can be:

11. *A class name is not allowed (SE).*
12. *A class is not necessary (PQW).*
13. *An intermediate class is not necessary (SE).*

Generalization and Specialization The last check for classes is to verify that generalization and specialization relationships have been correctly represented. The check is done only if the classes provided by the solution have been represented in the diagram; in case of incorrect modeling between parent class and children classes, the following violation is found:

14. *A class should be a child class of another existing parent class (PQW).*

Associations Once all the class checks are completed, it's necessary to ensure that the associations between them are existent and correct. Only if the source and destination classes are modeled in the diagram, it is checked that the association between them is represented correctly. For each association provided by the solution, if a relation between the source and destination class of the searched association is found in the diagram, additional checks are performed. Firstly, the name of the association is verified similar to how the names of the classes were checked (Levenshtein normalized distance). Then, if the name passes the check, it is checked that the source and destination cardinalities of the association fall within the correct cardinality list predicted by the solution.

Before considering them unnecessary, all associations in the diagram that have not been mapped to an expected association in the solution in the solution undergo further checks. These checks include checking that the association name is not forbidden and that the association does not fall between those forbidden between two specific classes.

Violations that may be found during the association analysis are as follows:

15. *An association between two existing strong classes is absent (SE).*
16. *An association between one existing strong class and one existing weak class is absent (PQW).*
17. *An association between two existing classes is not necessary (PQW).*
18. *An association name is not allowed (SE).*
19. *An association between two existing classes is not allowed (PQW).*
20. *An association found has the wrong cardinalities (PQW).*

3.1.3 Progress and Error Score

Each time a student's diagram is analyzed, a progress percentage based on the number of correct elements present in comparison to the total number of elements in the solution (classes, attributes, and associations) is calculated. This provides the student with an idea of the completeness of their diagram.

To determine whether an exercise is passed or not, an error score is calculated taking into account the different types of violations as described earlier. For syntax violations, a fixed penalty is assigned, while each semantic violation previously described is assigned a weight that is used to calculate the total error score. If the percentage of the accomplished error score compared to the total minimum possible error score is lower than the threshold provided in the solution, then the exercise is considered passed.

Listing 3.2. Error Score Computation Pseudocode

```
fun computeErrorScore() {
        let tot = 0

        tot += strongAbsentClasses.length *
                PENALTY.STRONG_CLASS_ABSENT
        tot += weakAbsentClasses.length *
                PENALTY.WEAK_CLASS_ASBENT
        tot += notAllowedClasses.length *
                PENALTY.NOT_ALLOWED_CLASS
        tot += absentAttributes.length *
                PENALTY.ABSENT_ATTRIBUTE
        tot += notAllowedAttributes.length *
                PENALTY.NOT_ALLOWED_ATTRIBUTE
        tot += absentStrongAssociations.length *
                PENALTY.ABSENT_STRONG_ASSOCIATION
        tot += absentWeakAssociations.length *
                PENALTY.ABSENT_WEAK_ASSOCIATION
        tot += notAllowedAssociationsNames.length *
                PENALTY.NOT_ALLOWED_NAME_ASSOCIATION
        tot += notAllowedAssociations.length *
                PENALTY.NOT_ALLOWED_ASSOCIATION
        tot += wrongCardinalityAssociations.length *
                PENALTY.WRONG_CARDINALITY
        tot += unnecessaryClasses.length *
                PENALTY.UNNECESSARY_CLASS
        tot += unnecessaryAssociations.length *
                PENALTY.UNNECESSARY_ASSOCIATION
        tot += attributesAsClassWrong.length *
                PENALTY.ATTRIBUTES_AS_CLASS_WRONG
        tot += attributesAsClassOk.length *
                PENALTY.ATTRIBUTES_AS_CLASS_OK
        tot += attributeErrorType.length *
                PENALTY.ATTRIBUTES_ERROR_TYPE
        tot += absentInheritance.length *
                PENALTY.ABSENT_GENERALIZATION
        tot += absentAssociationClasses.length *
                PENALTY.ABSENT_ASSOCIATION_CLASS

        return tot
}
```

3.2 Gamified Mechanics

Designing a gamified environment is not a trivial task and it requires a thorough analysis of which mechanics would be most effective in a particular context to fully utilize the benefits of gamification.

Several gamification mechanics such as levels, points and rewards, badges, leaderboards, quests, and stories, are commonly used to capture attention and incentivize user engagement. These elements can effectively engage users and encourage them to interact with the gamified environment.

To evaluate their impact in an academic setting with the goal of expanding and applying new ones, some of these mechanics were selected for the prototype tool being developed in this thesis. These mechanics are described below.

3.2.1 Progress, Levels, and Experience

Levels are a popular feature in gamified environments; in this context, they provide a numerical and direct indication of a student's conceptual modeling skills. Students start at level 1 and can raise it by earning experience points by completing exercises of increasing difficulty (Figure 3.2).

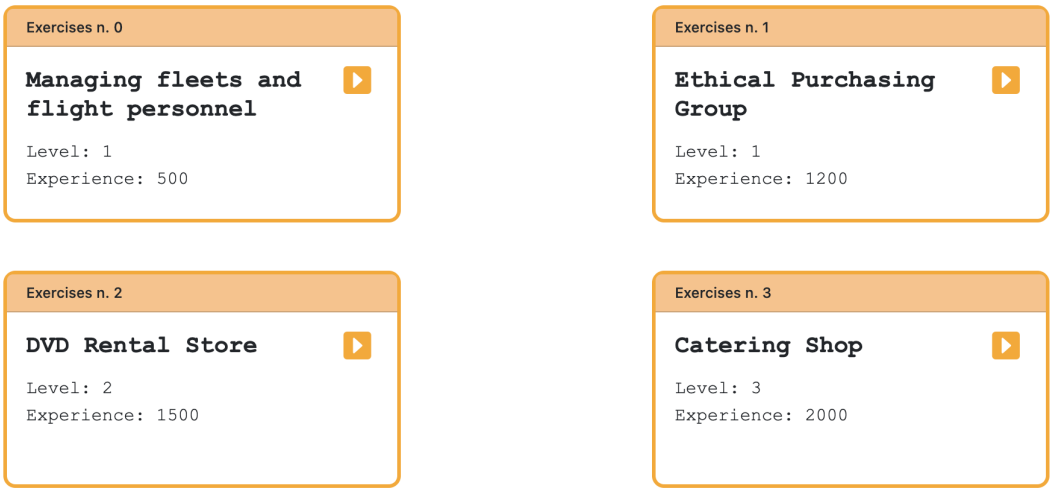

Figure 3.2. Exercises List

The amount of experience gainable decreases every time a student performs a check of his diagram, according to the number of syntax errors and the percentage of semantic error score calculated as shown in the section 3.1.3. However, each exercise has a minimum experience value that cannot be decreased by penalties.

Once the exercise is completed (that is, as previously described, the percentage of errors committed is lower than the threshold established), the final experience is assigned after having applied some multipliers to it. These multipliers are:

- *Level Multiplier*: There is no specific order in which students must complete exercises. This means that they are free to try solving more advanced exercises first for greater rewards. If the exercise is at a higher level than the student's current level, they will receive a multiplied amount of experience. However, this comes at the cost of facing a more difficult challenge;
- *Checks Multiplier*: Every time students perform a check on their diagram, they receive feedback on mistakes that progressively guide them toward the correct solution. If they complete the exercise with only a few checks, it means that they have successfully arrived at the solution independently, and are rewarded by increasing their gained experience. However, if they use a higher number of checks, the experience gained will be slightly reduced;
- *Progress Multiplier*: After each check of a diagram, a progress percentage is calculated to show how close the diagram is to the optimal solution. When a student completes the exercise for the first time, the higher the progress percentage, the greater the multiplier applied.

After completing the exercise, students can continue to work on their diagrams without losing experience points until they achieve 100% progress. Figure 3.3 shows an example of a completed exercise message with a summary of the calculation of the experience obtained.

According to the Core Drives outlined in the Octalysis framework (section 2.2.1), mechanics such as progress, levels, and experience are categorized under the *Development and Accomplishment* drive. They serve as typical indicators of a user's advancement within a gamified system, and unlocking new levels with corresponding rewards can provide a fulfilling experience.

Furthermore, the potential reduction of experience points for repeated failures aligns with the *Loss and Avoidance* drive.

3.2.2 Avatars

Avatars offer a way for students to customize their experience and express their own individuality inside the tool.

Initially, students have access to a limited set of avatar props, but they can unlock new ones by completing exercises and advancing to higher levels.

A dedicated tool section offers the chance to customize the avatar by changing every single prop in detail; the section also shows the props that are yet to be

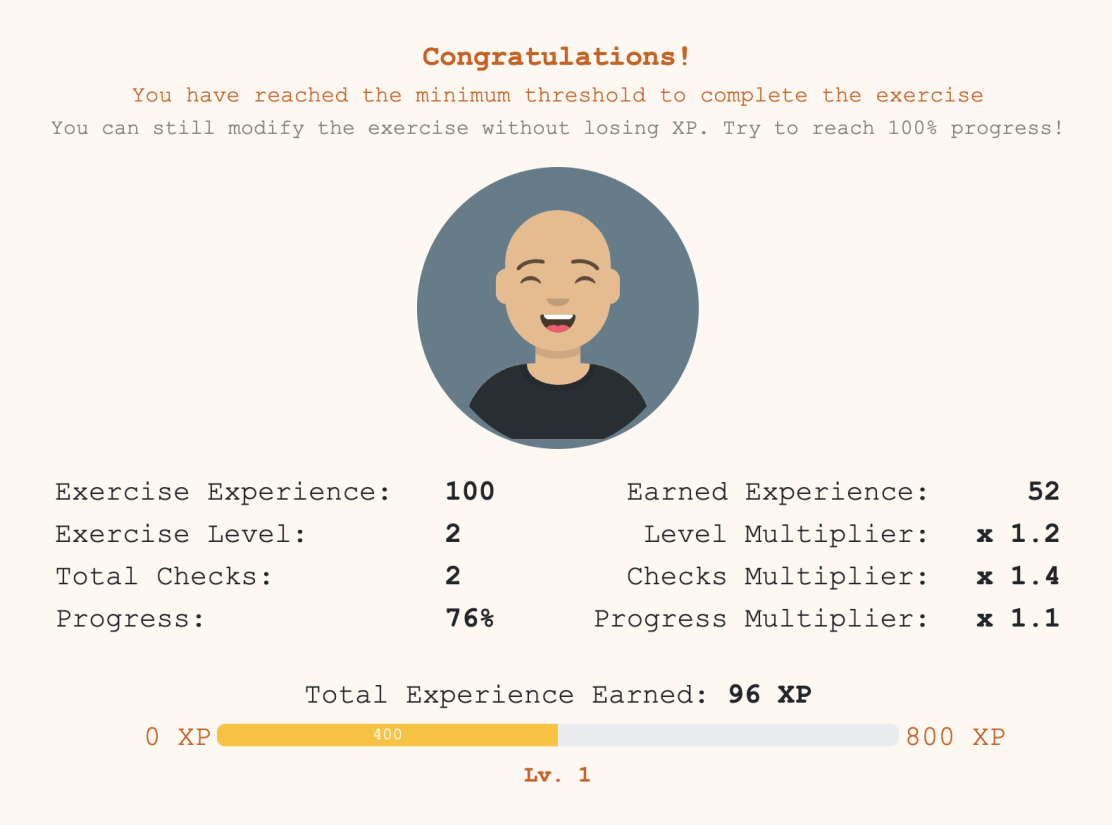

Figure 3.3. Summary of the completed exercise

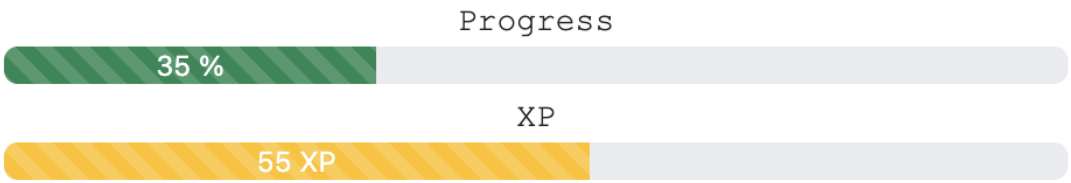

Figure 3.4. Progress and XP

unlocked, together with the needed level. An example of the avatar customization page is shown in Figure 3.5.

Avatars are implemented with a web-based implementation of the Avataaars Sketch library [3]: the library allows for the customization of a human avatar by

[3] http://avataaars.com

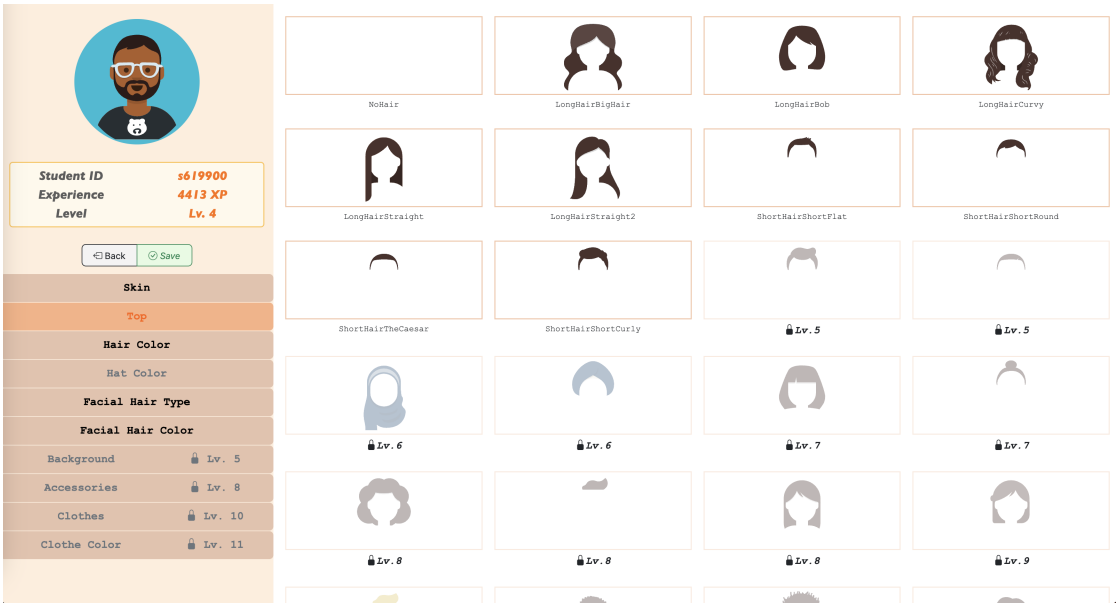

Figure 3.5. Avatar Customization Section

changing different components such as its clothes, its hairstyle and hair color, presence of facial hair, and accessories.

Using customizable avatars refers to Octalysis's concept of *Ownership*: students are more motivated in using the tool if they feel a close connection with something personal that lets them express their individuality.

3.2.3 Feedback

To improve student learning, feedback is given after each correctness check. Any errors found in a diagram are highlighted by coloring the part that contains the error. This helps students identify where they need to focus to improve their model. The color used reflects the type of error: orange for syntax errors, red for semantic errors, and blue for pragmatic quality warnings (Figure 3.6).

In addition, a list of all violations found is available for students to consult and correct as shown in Figure 3.7.

An additional feedback mechanism, directly tied to the student's *Avatar*, consists of a visual change of the avatar itself after too many experience points are lost: the student's avatar will change its facial expression, changing from a happy emotion to a progressively sadder one the more mistakes the student makes (Figure 3.8). This kind of negative feedback is used to motivate the student to think carefully and avoid making mistakes.

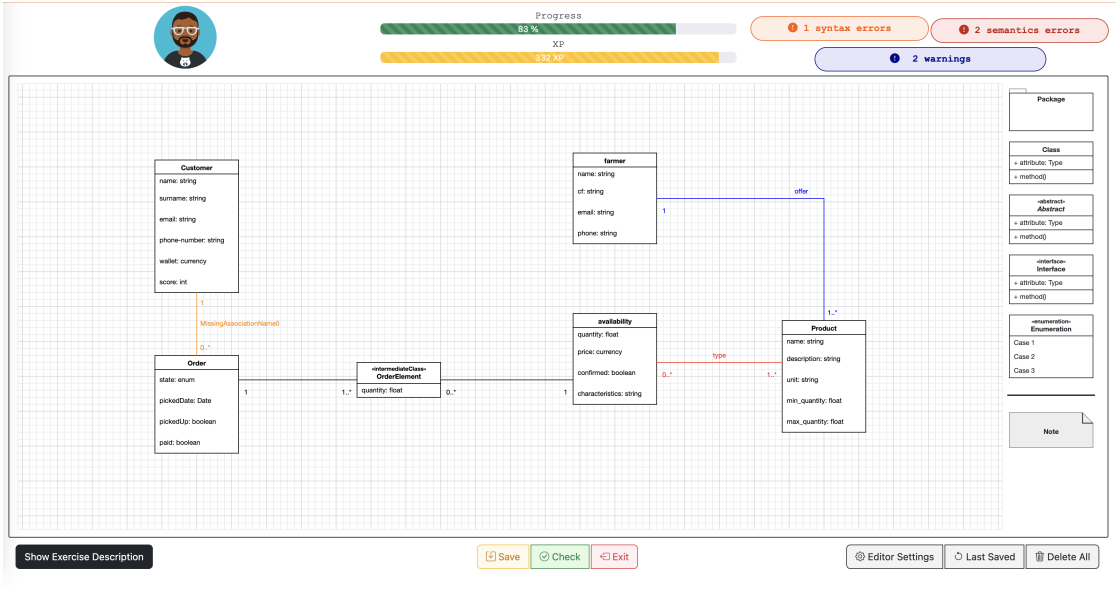

Figure 3.6. Diagram Evaluation Feedback

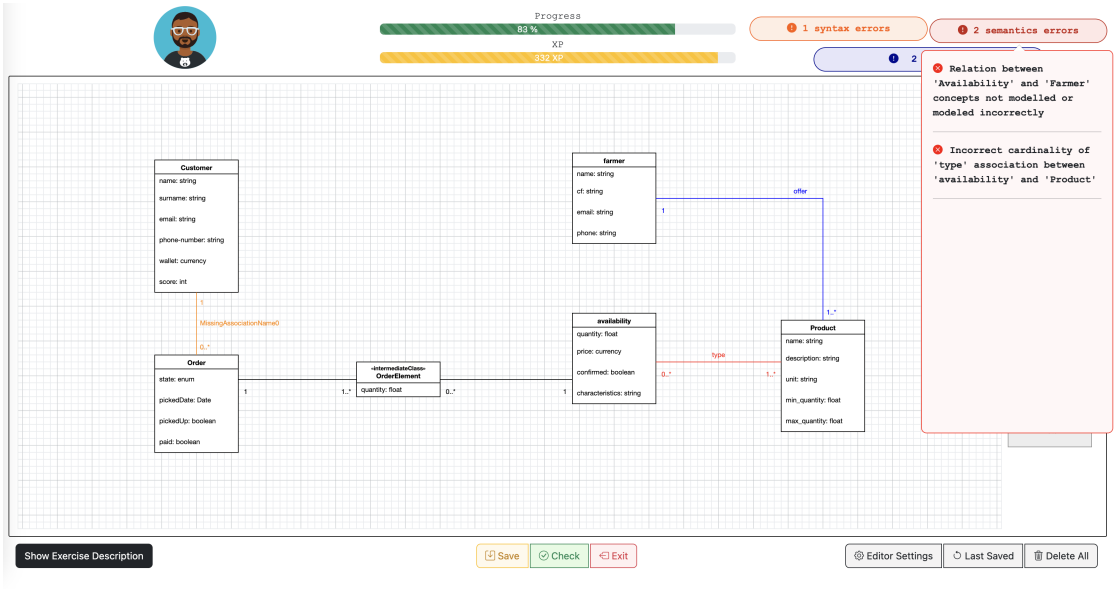

Figure 3.7. Diagram Errors Description

The feedback in the form of error messages and coloring parts of the diagram can be connected to Octalysis's *Empowerment of Creativity and Feedback* Core Drive: students can easily identify the incorrect parts of their diagrams and understand

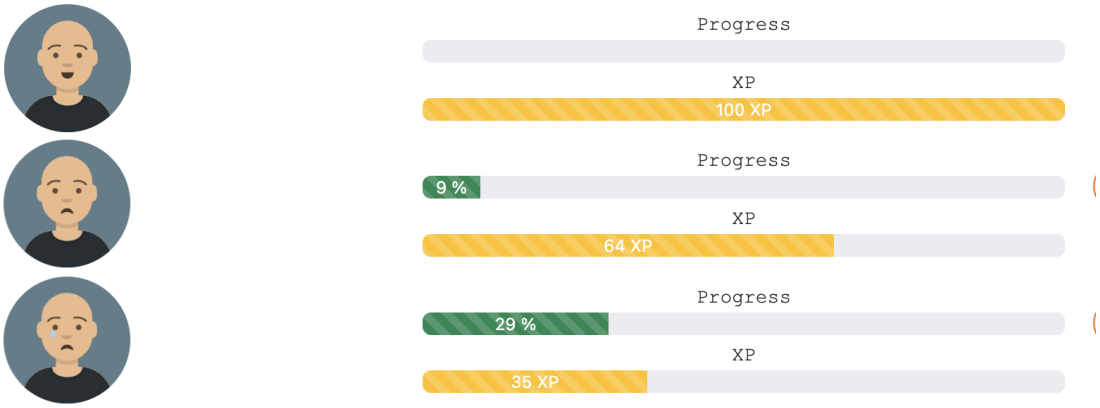

Figure 3.8. Avatar Expressions

the reasons behind the errors. This motivates them to take corrective actions, such as changing the model and checking again, to overcome the errors.

Additionally, the visual effects on the student's avatar can be thought of as an example of *Loss and Aversion* drive: students who see the change in the avatar's status will be more motivated in future modeling tasks, in order to spare the avatar further "suffering".

3.3 Web Application Design

Once the core functionalities of the tool, including an automatic exercise evaluation system and gamification techniques, were defined, it was necessary to design the final application accessible to the users. The use case diagram in Figure 3.9 shows the two user roles planned for the use of the tool and the main functionalities that will be implemented.

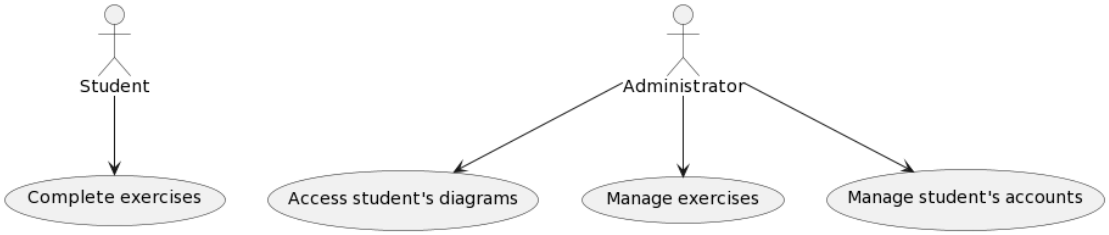

Figure 3.9. Use Case Diagram

3.3.1 Diagram Editor

The basic necessary functionality of a tool for teaching conceptual modeling is precisely to provide an editor where diagrams can be implemented. After a search for some possible libraries that offered this possibility, the choice fell to *Apollon* [4], a UML modeling editor written in React and TypeScript.

The editor provided by Apollon is user-friendly and equipped with flexible layout features. It also offers convenient shortcuts for copying, pasting, deleting, and moving elements around the canvas. Additionally, the infinite canvas ensures that students will never run out of space, while the grid provides a helpful reference for placing elements precisely (Figure 3.10).

This library was chosen cause of its completeness and ease of use for users. However, a few adjustments to accommodate certain features for the specific environment where the tool will be used were needed.

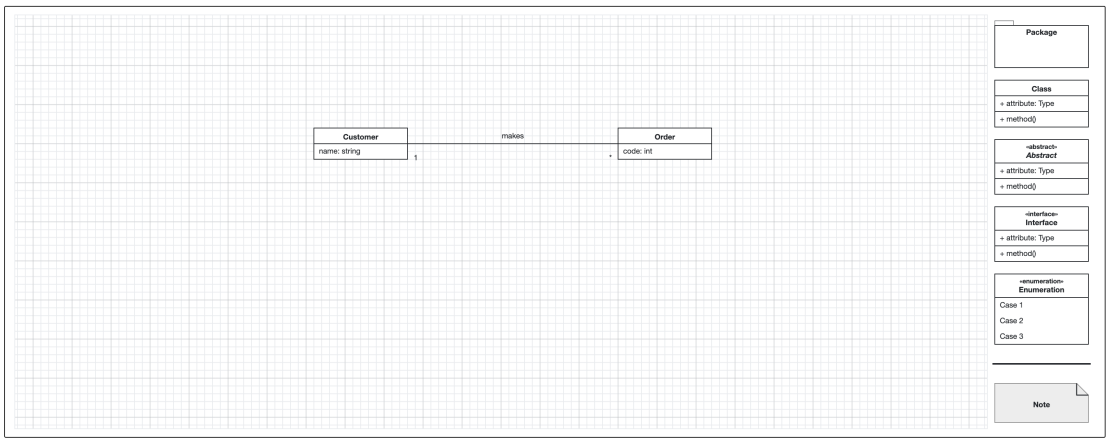

Figure 3.10. Apollon Editor

Additional features have also been added to enhance the experience of using the editor: they include the possibility for users to export the diagram in JSON, SVG, or PDF format, as well as import a previously exported JSON file.

Students using the editor can save the progress of their diagram at any time without the need for an evaluation check. They can also retrieve the last saved diagram or clear the entire canvas using the toolbar at the bottom of the page. The exercise description can also be accessed at any time through a pop-up menu that is designed to optimize the page space for editor use, as shown in Figure 3.11.

Furthermore, there is a dedicated section that provides a free editor for students

[4]https://github.com/ls1intum/Apollon

to practice and become familiar with the tool. They can apply only syntactic checks without consequence on experience and levels; this should help in preventing trivial mistakes while working on real exercises.

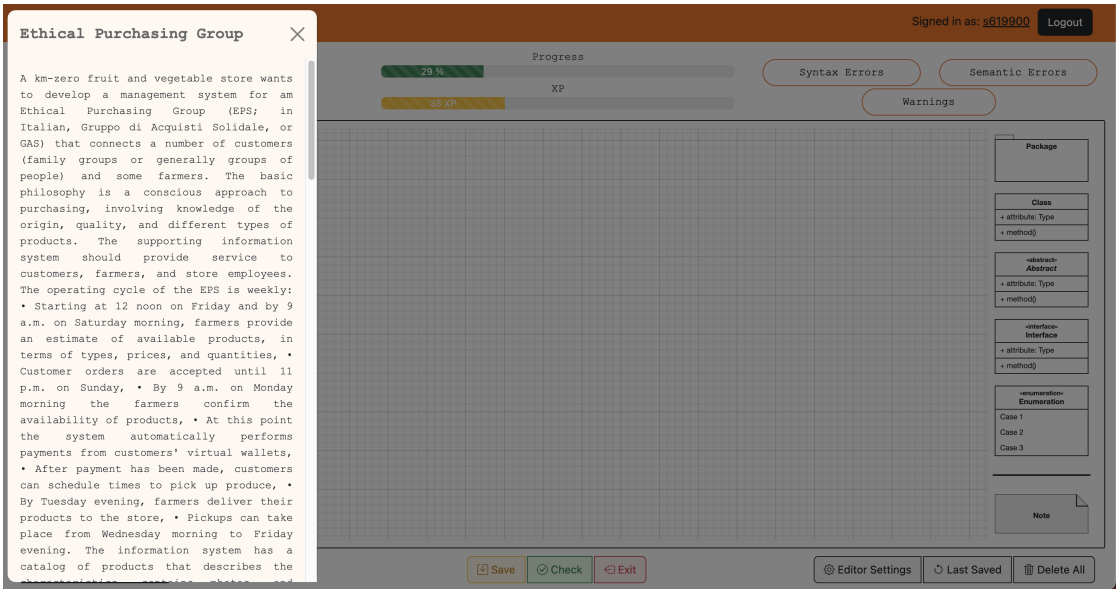

Figure 3.11. Exercise Description in the editor

3.3.2 Administrator Functionalities

In addition to the student features mentioned previously, such as an editor, avatar customization, diagram evaluation, etc., the tool also implements a specific section accessible only to administrators that offer three main functionalities.

One of these features is the ability to access the most recent diagram saved by searching by the student or by exercise, and the option to export it at any time (Figure 3.12).

The administrator panel provides also the feature to manage student accounts. The administrator can add new accounts or import multiple accounts from a CSV file, remove an account or reset a student's password to its original state as shown in Figure 3.13. Upon adding a student, they are given a username and password that corresponds to their academic ID. The system prompts students to customize their password when they do the first login, as shown in Figure 3.14. Password reset functionality is utilized when students forget their password and need to create a new one.

Finally, administrators have the ability to add, edit, or delete exercises (Figure 3.15) as well as manage their corresponding solutions.

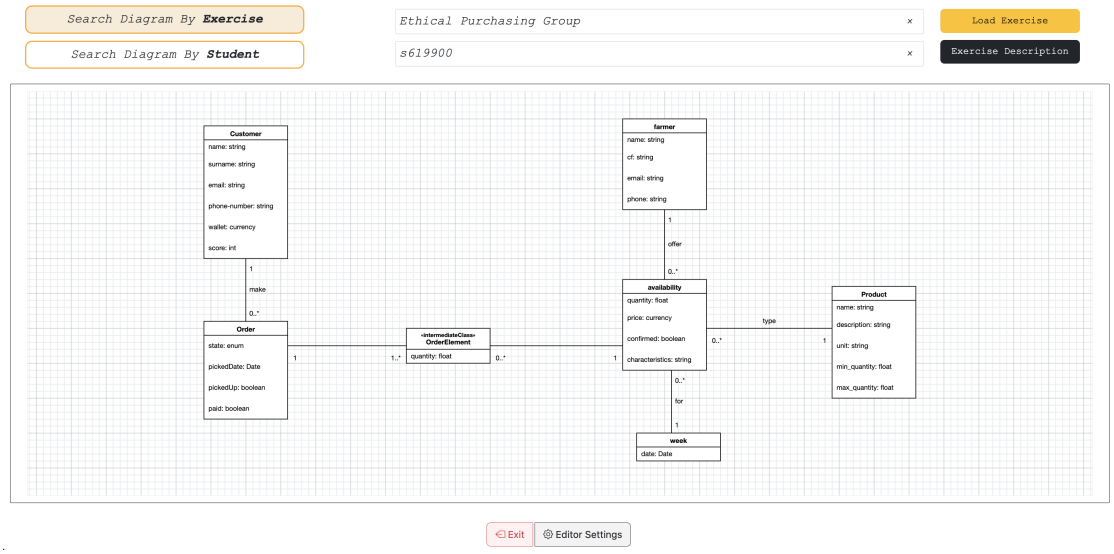

Figure 3.12. Diagrams Admin Panel

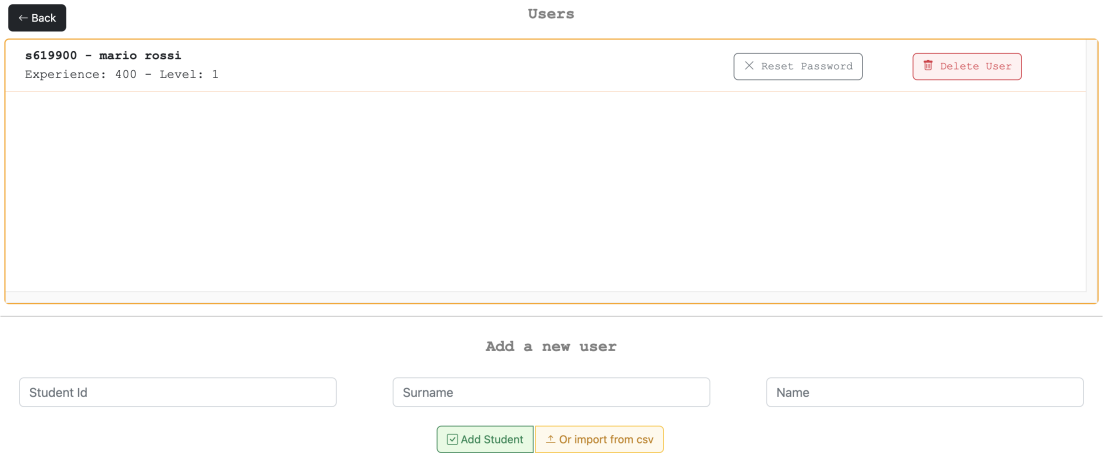

Figure 3.13. Students Admin Panel

All the elements that make up a solution, which are described in Section 3.1.2, can be added one by one using the panel shown in Figure 3.16, or the administrator can draw a diagram and all the elements will be automatically imported.

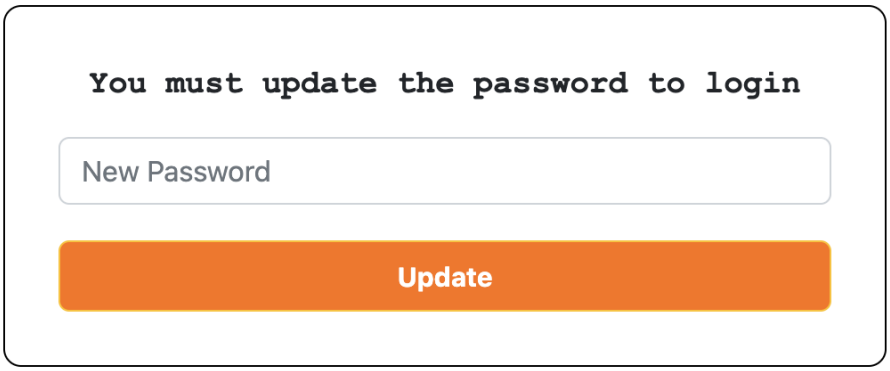

Figure 3.14. Passord Updating Panel

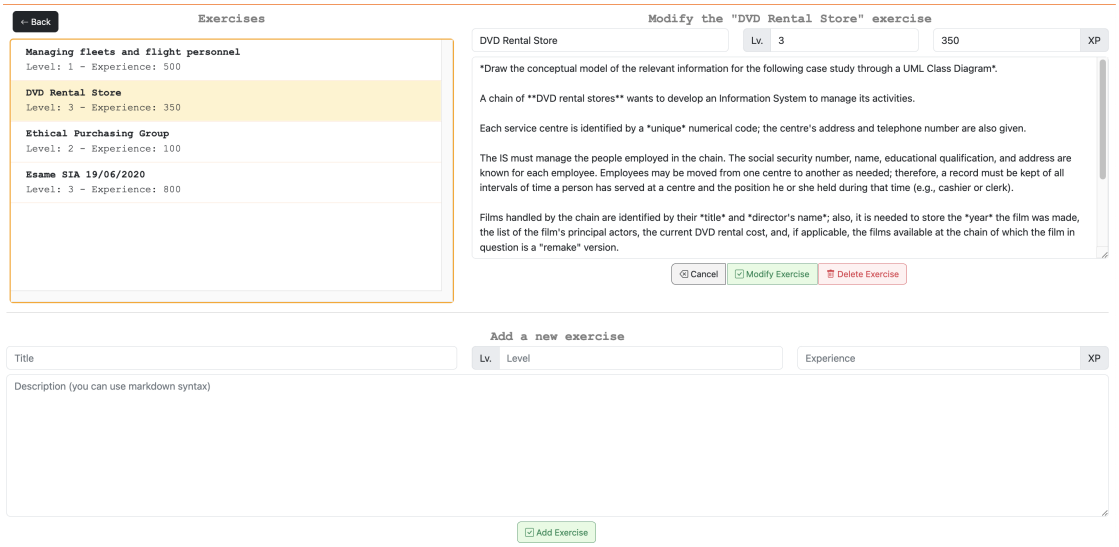

Figure 3.15. Exercises Admin Panel

3.3.3 Software Architecture

The tool was implemented as a web application to enable multiple users to access the same version of the app from various platforms, including desktops, laptops, and mobile devices, without requiring installation.

The software architecture of the web application consists of 2 tiers, the client and the server, and it is illustrated in Figure 3.17.

Typescript is the programming language used for both the client and server.

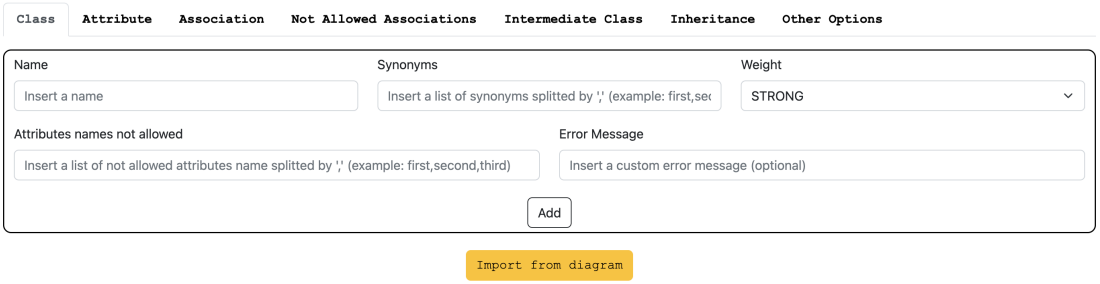

Figure 3.16. Solution Elements Tabs

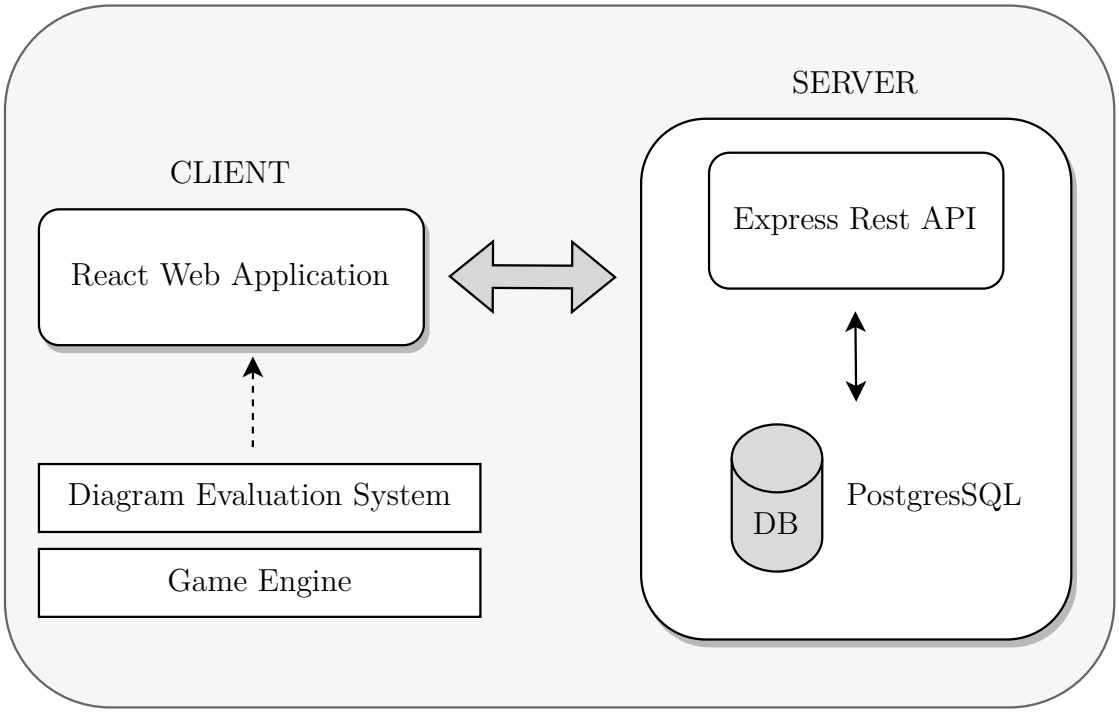

Figure 3.17. Software Architecture

The front end was developed by using the React [5] library, an open-source JavaScript framework and library developed for building interactive user interfaces and web applications. React was used together with React-bootstrap [6], a library taking the CSS framework of Bootstrap and replacing any existing JavaScript with

[5]https://react.dev

[6]https://react-bootstrap.github.io

strictly React components.

Within the client tier, there are two extra modules. The first is the Diagram Evaluation System, which can parse diagrams and assess them based on a given solution. The second is the Game Engine, responsible for managing the experience, levels, scores, and other gamification features.

To communicate with the server, the client can use a Rest API built with Express [7], a Node.js framework that offers advanced features like HTTP utility methods and middleware. This makes it simple and fast to create an API.

The list of available APIs is shown in Tables 3.5, 3.6, 3.7, 3.8, and 3.9.

The data is stored in a SQL database that is managed by Postgres [8], a powerful and open-source object-relational database system.

URI (/api/exercises)	**Method**	**Description**
/	POST	Insert a new exercise
/	GET	Get all exercises
/{exerciseId}	GET	Get exercise by id
/{exerciseId}	PUT	Modify exercise by id
/{exerciseId}	DELETE	Delete exercise by id
/{exerciseId}/solution	GET	Get exercise solution by exercise id
/{exerciseId}/solution	PUT	Modify exercise solution by exercise id
/{exerciseId}/solution	DELETE	Delete exercise solution by exercise id

Table 3.5. List of available APIs - Exercises

[7] https://expressjs.com

[8] https://www.postgresql.org

URI (/api/diagrams)	Method	Description
/{exerciseId}/students	POST	Insert a new status log for the logged student diagram with the actual progress and experience
/{exerciseId}/students	PUT	Update the last logged student diagram saved
{exerciseId}/students	GET	Get all logs or only the last log for the logged student diagram
/{exerciseId}/students	DELETE	Delete all logs for the logged student diagram
/{exerciseId}/students/completed	POST	Insert the exercise completed summary for the logged student diagram
/{exerciseId}/students/completed	GET	Get the exercise completed summary for the logged student diagram
{exerciseId}/admin/{studentId}	GET	Get all logs or only the last log for the student diagram by student id (only for admin)

Table 3.6. List of available APIs - Diagrams

URI (/api/avatars)	Method	Description
/	PUT	Modify avatar props of the logged student
/	GET	Get avatar props of the logged student

Table 3.7. List of available APIs - Avatars

URI (/api/users)	**Method**	**Description**
/	GET	Get all users (optionally with a specific role)
/{userId}	GET	Get user by id
/students/xp	PUT	Update the experience of the logged student
/students	POST	Insert a new student
/students/{username}	DELETE	Delete student by username
/students{username}/reset	PUT	Reset a student password to the initial value
/students/updatePassword	PUT	Update the logged student password

Table 3.8. List of available APIs - Users

URI (/api/sessions)	**Method**	**Description**
/	POST	Perform login
/current	GET	Get current logged user
/current	DELETE	Perform logout

Table 3.9. List of available APIs - Authentication

Chapter 4

Tool Evaluation

Two analyses were conducted to evaluate the implemented tool. The first analysis focused on the automatic diagram evaluation system, while the second analysis focused on the gamification mechanics that were introduced.

4.1 Diagram Analyzer Evaluation

The main goal of the tool is to assist students in learning correct modeling practices by providing detailed feedback directly on the diagrams produced by the students themselves. To achieve this, the tool should evaluate the diagrams as accurately as possible. To verify the correctness of the automatic diagram correction system, the following analysis was conducted.

Procedure An exercise proposed to students from the previous edition of the course was selected. The text of the exercise required to model a management system for an Ethical Purchasing Group (EPS). This EPS connects customers with farmers, emphasizing a conscious approach to purchasing, including knowledge of product origin, quality, and types. The EPS operates on a weekly cycle, involving product estimates from farmers, customer orders, confirmation of product availability, automated payments, and product pickups. The information system includes a product catalog with descriptions, photos, pricing, and units of measurement. Customers can place orders directly through the web or with store employees, with the group representative responsible for entering group orders. Orders move through different states (entered, confirmed, modified) based on product availability. The system handles payments and allows customers to schedule pickups. It also tracks fair play scores for customers, which influence precedence in case of limited product availability. Unpicked orders are marked as "forfeited," and the fair play score is updated accordingly.

The solution of the exercise, shown in Figure 4.1, has been inserted within the tool and included the following classes considered mandatory, thus weighted STRONG: *Customer*, *Order*, *Availability*, *Farmer* and *Product*. The class *Week*, on the other hand, was considered a WEAK class. For each class name or attribute name of each class, the possible synonyms have also been entered. The solution also provided an association class between the classes *Order* and *Availability*. In addition to the information represented in the figure, a list of forbidden class names or attributes and a list of forbidden associations between certain classes were also included. An alternative way to model the system provided by the solution was to represent the association class as an intermediate class and the representation of the attribute *fairPlayScore* of the class Customer, as a class by itself.

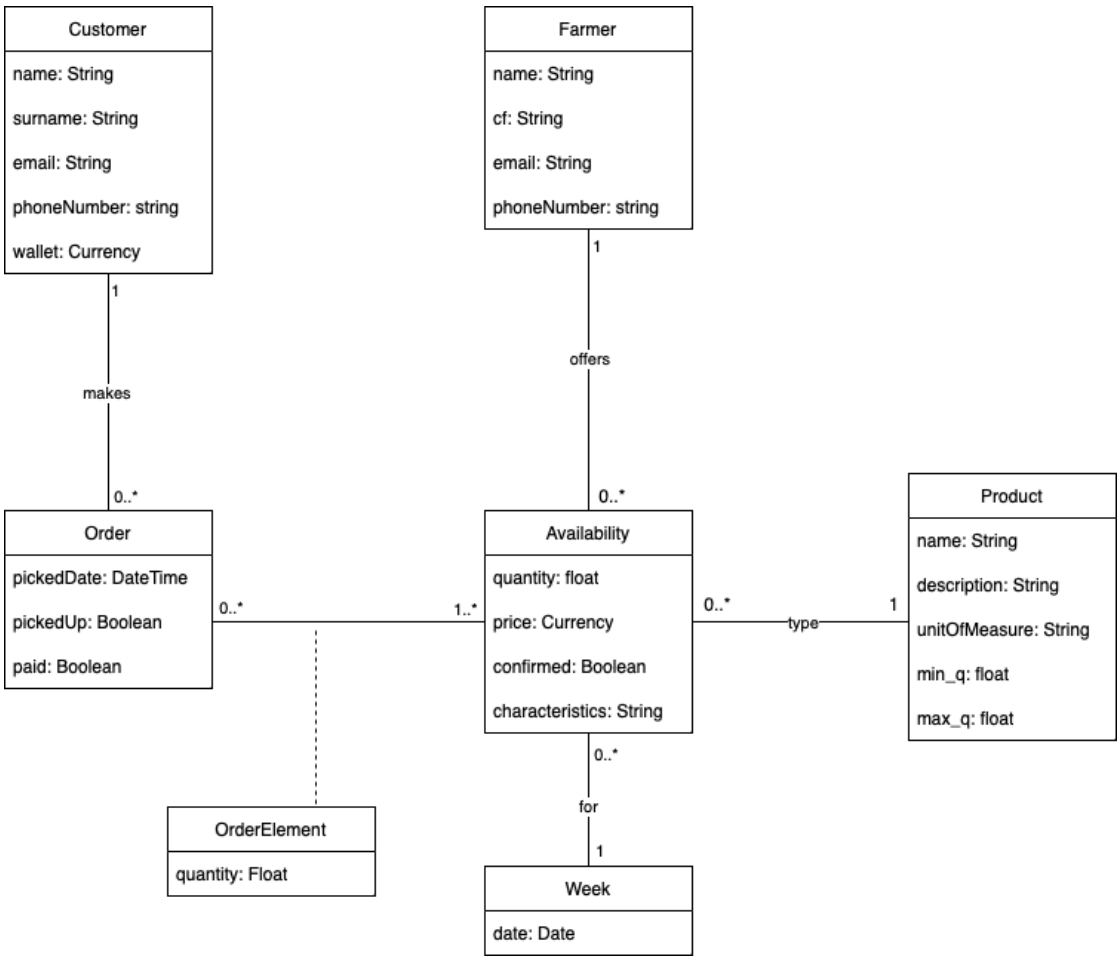

Figure 4.1. Exercises Solution

Once entered into the tool the exercise with its description and its solution,

30 diagrams created by students of the course for the exercise were anonymously selected. Starting from their PDF representation, they were modeled one by one in the tool's editor taking an average of 10 minutes per exercise. This made it possible to run the automatic evaluation done by the evaluation engine and retrieve the list of violations found by the tool.

The list of violations of each exercise was reviewed by a human evaluator to determine the accuracy of the tool's identification of errors and warnings. Each violation found was analyzed and classified by the evaluator into one of the following categories:

- *Correctly identified syntax error*;
- *Correctly identified semantic error*;
- *Correctly identified pragmatic quality warnings*;
- *Should be error*: violations reported as warnings that a human evaluator would more likely have reported as errors;
- *Should be warning*: violations reported as errors that a human evaluator would more likely have reported as warnings;
- *Debatable error*: errors that could be ignored by a human evaluator (such as a multiplicity entered as * where a 0..* was expected);
- *Debatable warning*: warnings that could be ignored by a human evaluator (such as a class that is not included in the solution and therefore flagged as unnecessary, but which might be fine with a human evaluator);
- *Wrongly identified error*: violations found that are totally incorrect.

Results The chart in Figure 4.2 shows the overall distribution of accurately identified errors and warnings.

However, as described above, the evaluator not only defined whether an error or warning was correct or incorrect but also identified several categories of error in the violation found by the tool. This allowed a more in-depth analysis described below to be conducted.

Two main causes were identified for the erroneous syntax violations that were found. The first is spelling errors made by students, such as using 'strig' instead of 'string' as an attribute type. The second cause involves attribute types that are not provided by the tool but are still acceptable to a human evaluator, such as using 'integer' instead of 'int'. In the first case, there is little room for improvement other than calculating the distance between the expected and entered attribute types to avoid reporting errors (assuming the spelling error is minor). In the second case,

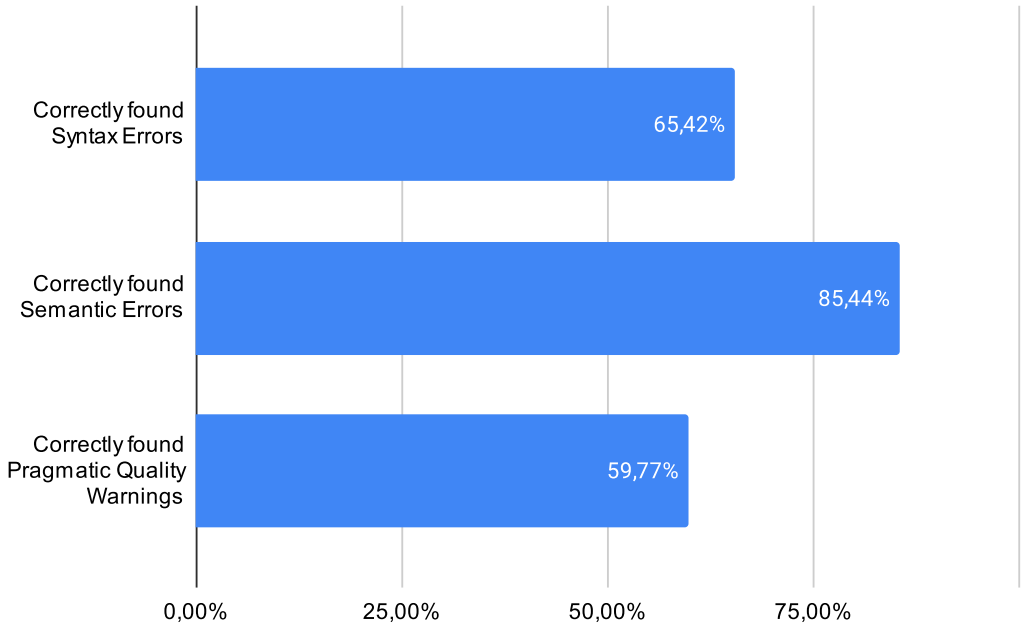

Figure 4.2. Percentages of syntax errors, semantic errors, and pragmatic quality warnings correctly identified by the tool

expanding the list of attribute types provided by the tool would prevent these types of errors from being reported. By analyzing these causes, percentages of incorrect syntactic error violations were adjusted as shown in Figure 4.3. As a result, the percentage of completely incorrect syntactic violations was reduced to 0.93%.

Analyzing warnings was easier as most of the incorrect ones were due to a violation being treated as a semantic error rather than a warning. This issue can be resolved by making a minor fix to the source code, making certain errors that were previously classified as warnings as semantic errors, such as using an unnecessary intermediate class. As a result, the percentage of warnings that a human evaluator would not have detected decreases to 2.30%, as illustrated in Figure 4.4.

In the first round of evaluation, a high percentage of semantic errors were correctly identified. However, there were two categories of semantic violations that were found incorrectly. The first category included semantic errors that a human evaluator would have considered as warnings. The second category included errors reported as semantic, but they would be more accurately considered as syntax. Errors in both of these categories can be corrected through quick and easy code

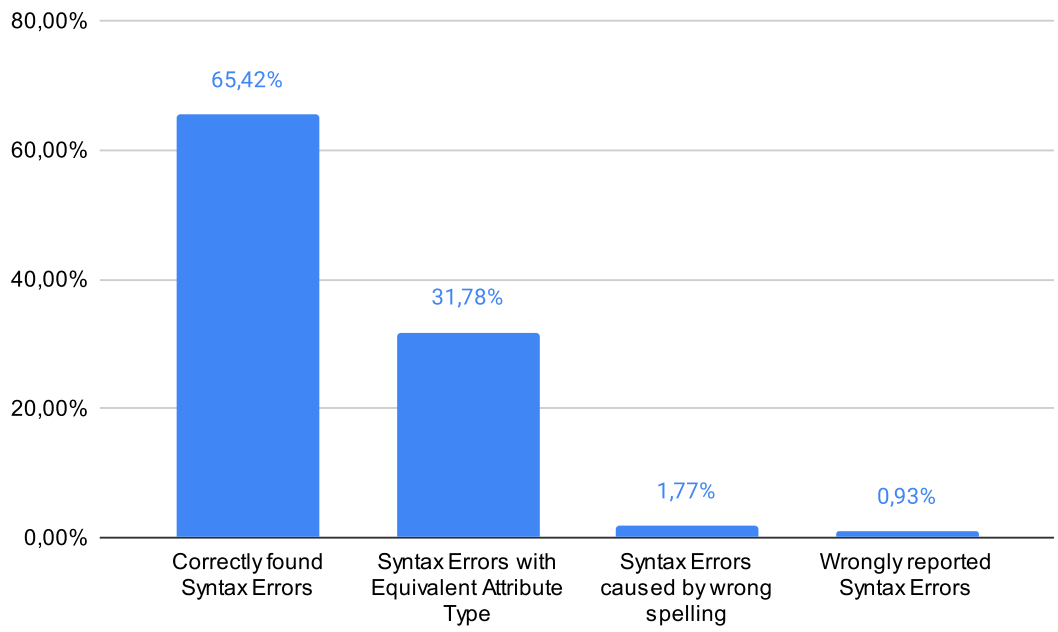

Figure 4.3. Distribution of syntax error classification

fixes. As a result, considering a small percentage of violations found not wrong but that a human would have allowed, the percentage of semantic errors found to be completely wrong was 0,77%, as illustrated in Figure 4.5.

Based on this analysis and the necessary improvements to enhance the automatic evaluation system, it can be concluded that the tool has the ability to detect errors in a way that is comparable to a human evaluator, as illustrated in Figure 4.6. Moreover, it is important to highlight that the accuracy of the automatic evaluation increases as the solution the administrators provide becomes more comprehensive with synonyms, messages, and lists of elements that compose it.

4.2 Octalysis score

A successful gamified system doesn't necessarily require all the Core Drives outlined in Section 2.2.1, but it must excel in the ones it does incorporate. To evaluate the tool from this point of view the Ocatlysys Score was calculated as explained by the framework. The evaluation results can be viewed in Figure 4.7.

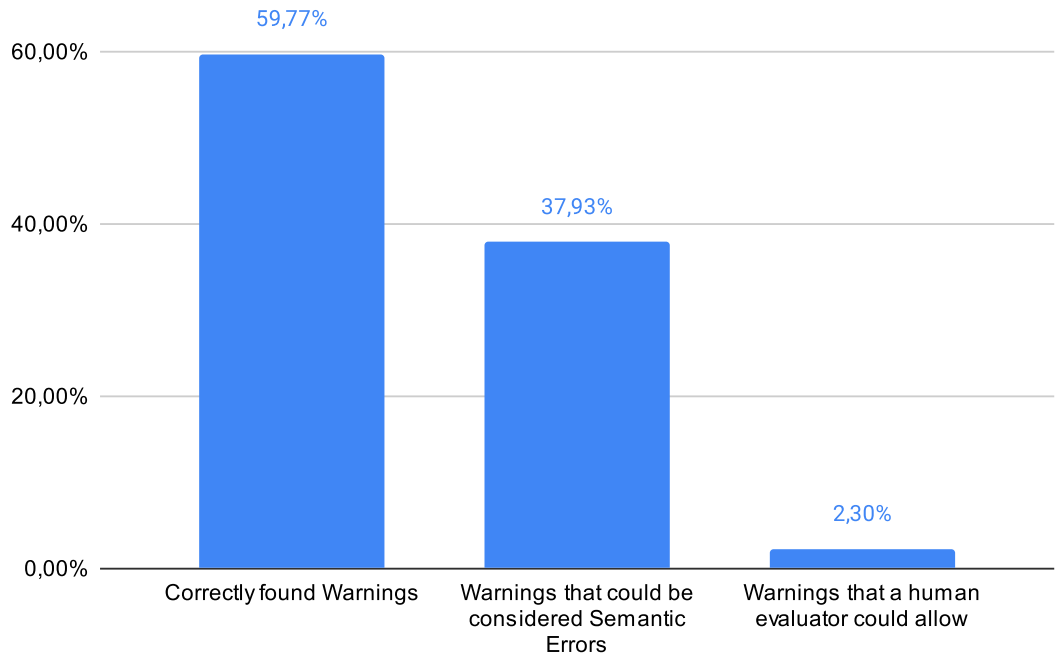

Figure 4.4. Distribution of pragmatic quality warning classification

Based on the analysis, the gamified experience appears to have a balance between White Hat and Black Hat drives, as well as left-brain and right-brain drives, even if there was a slightly higher score given to the Ownership drive. However, upon visually examining the figure, it becomes apparent that some core drives are not well-represented at present. This outcome is not surprising since the tool is still in its prototypical stage. Despite this, the balance on both vertical and horizontal sides is a promising initial achievement.

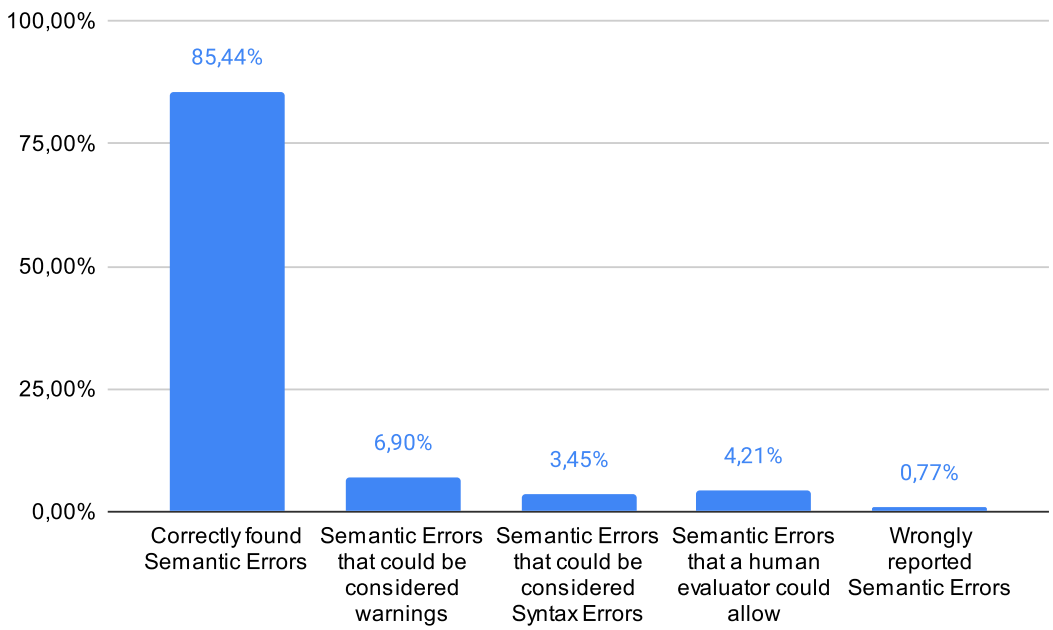

Figure 4.5. Distribution of semantics error classification

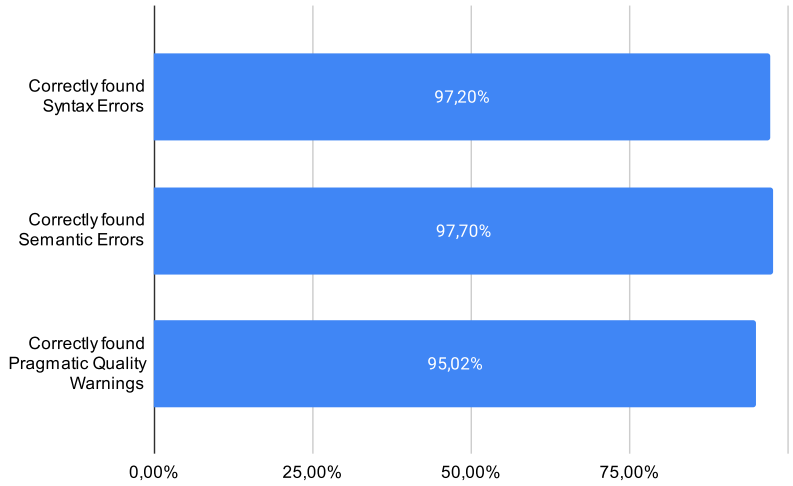

Figure 4.6. Percentages of syntax errors, semantic errors, and pragmatic quality warnings correctly identified by the tool after the tool evaluation and improvement

Figure 4.7. Tool Ocatlysis Score - 113

Chapter 5

Conclusion and Future Work

In the field of software engineering, conceptual modeling plays a key role as it serves as the foundation for designing and creating advanced software systems that are of high quality. However, it demands a detailed understanding of the domain and the skill to transform abstract concepts into structured representations, which can be an ability difficult to teach or learn. By incorporating gamification, which is the implementation of game elements and principles in a non-gaming environment, software engineering education can become more engaging, motivating, and effective because provides students with a stimulating and interactive learning environment.

The goal of this thesis was to implement a gamified prototype tool to assist students in learning correct modeling practices by providing detailed feedback directly on the diagrams produced by the students themselves.

Firstly, a thorough exploration of the key concepts of conceptual modeling with UML Class Diagrams was conducted. Next, an analysis was carried out to determine the advantages and disadvantages of applying gamification. Specifically, the Octalysis Framework for Gamification was examined in detail to identify the required elements for a correct application of gamification.

Once the thesis context was defined and explored in depth, the tool has been designed and implemented. It includes an evaluation engine that assesses the accuracy of the diagrams produced by the students. The primary gamification mechanics used are:

- Levels and experience points obtainable through completing exercises
- Immediate feedback received on students' diagrams, which helps them understand their mistakes and correct them

- Avatars customizable through features unlockable by passing levels

The tool's graphical user interface, the software architecture, and other features were then explained.

Finally, two separate analyses were carried out to evaluate the goodness of the implemented tool. The first analysis aimed to evaluate the accuracy and efficiency of the students' diagrams correction engine. It showed that the diagram correction system, with certain modifications, can be compared to a human evaluator in terms of accurately reporting violations. However, there is still room for improvement. The second analysis aimed to evaluate the completeness of the gamification elements implemented through the calculation of the Octalysis Score. The analysis showed a balance in the gamification elements applied and at the same time the absence of other mechanics that could bring greater benefits. For this reason, an important future development could be to expand the tool's features with other gamified mechanics that are suited for long-term usage, such as competition mechanisms like leaderboards and quest-line mechanics based on the exercises.

A minor limitation of the tool is that the library for implementing the editor to draw diagrams did not provide the possibility to support the graphical representation of association classes as described in Section 2.1.1. The limitation can be partially overcome by using intermediate classes as an alternative to association classes, though they are not entirely equivalent.

The tool implemented is planned to encourage students to solve modeling exercises independently. The presence of in-game rewards for solving exercises in a correct way and other gamified mechanisms is something that can act as an effective motivator for students. Future plans include the usage of the tool in an academic environment with a longitudinal experiment, where students' progress can be tracked during the course.

Bibliography

[1] Object Management Group. *OMG, Unified Modeling Language (UML) 2.5.1 Superstructure Specification*. 2017.

[2] John Erickson and Keng Siau. "Theoretical and practical complexity of modeling methods". In: *Communications of the ACM* 50.8 (2007), pp. 46–51.

[3] S. Deterding et al. "From game design elements to gamefulness: defining" gamification"". In: *Proceedings of the 15th international academic MindTrek conference: Envisioning future media environments*. 2011, pp. 9–15.

[4] *UML Diagram Types Guide: Learn About All Types of UML Diagrams with Examples*. https://creately.com/blog/diagrams/uml-diagram-types-examples/. Accessed: 2023-09-21.

[5] Peter Pin-Shan Chen. "The entity-relationship model—toward a unified view of data". In: *ACM transactions on database systems (TODS)* 1.1 (1976), pp. 9–36.

[6] Navid Memar et al. "Investigating information system testing gamification with time restrictions on testers' performance". In: *Australasian Journal of Information Systems* 24 (2020).

[7] Y.k. Chou. *Actionable Gamification: Beyond Points, Badges, and Leaderboards*. Createspace Independent Publishing Platform, 2015. ISBN: 9781511744041. URL: https://books.google.it/books?id=jFWQrgEACAAJ.

[8] Michael Sailer et al. "How gamification motivates: An experimental study of the effects of specific game design elements on psychological need satisfaction". In: *Computers in human behavior* 69 (2017), pp. 371–380.

[9] Mantas Jurgelaitis, L Ceponiene, and Vaidotas Drungilas. "Using Gamification for Teaching UML in Information System Design Course". In: *CEUR-WS* 2145 (2018), pp. 88–94.

[10] H. M. dos Santos et al. "CleanGame: Gamifying the Identification of Code Smells". In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. SBES 2019. Salvador, Brazil: Association for Computing Machinery, 2019, pp. 437–446. ISBN: 9781450376518. DOI: 10.1145/3350768.3352490. URL: https://doi.org/10.1145/3350768.3352490.

[11] José Miguel Rojas et al. "Code Defenders: Crowdsourcing Effective Tests and Subtle Mutants with a Mutation Testing Game". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017, pp. 677–688. DOI: 10.1109/ICSE.2017.68.

[12] Gordon Fraser, Alessio Gambi, and José Miguel Rojas. "A preliminary report on gamifying a software testing course with the code defenders testing game". In: *Proceedings of the 3rd European Conference of Software Engineering Education*. 2018, pp. 50–54.

[13] G. Fraser et al. "Gamifying a Software Testing Course with Code Defenders". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Minneapolis, MN, USA: Association for Computing Machinery, 2019, pp. 571–577. ISBN: 9781450358903. DOI: 10.1145/3287324.3287471. URL: https://doi.org/10.1145/3287324.3287471.

[14] W. Prasetya et al. "Having fun in learning formal specifications". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE. 2019, pp. 192–196.

[15] A. Bucchiarone et al. "Gamifying model-based engineering: The PapyGame tool". In: *Science of Computer Programming* 230 (2023), p. 102974. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2023.102974.

[16] Olav O. Dæhli et al. "Exploring Feedback and Gamification in a Data Modeling Learning Tool." In: *Electronic Journal of e-Learning* 19.6 (2021), pp. 559–574.

[17] Valerio Cosentino, Sébastien Gérard, and Jordi Cabot. "A Model-based Approach to Gamify the Learning of Modeling". In: Nov. 2017.

[18] Mantas Jurgelaitis et al. "Implementing gamification in a university-level UML modeling course: A case study". In: *Computer Applications in Engineering Education* 27.2 (2019), pp. 332–343. DOI: https://doi.org/10.1002/cae.22077. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.22077. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cae.22077.

[19] Vacius Jusas, Dominykas Barisas, and Mindaugas Jančiukas. "Game elements towards more sustainable learning in object-oriented programming course". In: *Sustainability* 14.4 (2022), p. 2325.

[20] Narasimha Bolloju and Felix SK Leung. "Assisting novice analysts in developing quality conceptual models with UML". In: *Communications of the ACM* 49.7 (2006), pp. 108–112.