UNIVERSITY
INSTITUTE
OF LISBON

Highlighting Model Elements to Improve OCL Comprehension

Maria Pedro dos Santos Sales

Master in
Computer Science and Business Management

Supervisor:
Doctor Fernando Brito e Abreu, Associate Professor,
Iscte
Co-supervisor:
Doctor Marina Alexandra Pedro Andrade, Assistant Professor,
Iscte

November, 2020

[This page has been intentionally left blank]

TECHNOLOGY
AND ARCHITECTURE

Highlighting Model Elements to Improve OCL Comprehension

Maria Pedro dos Santos Sales

Master in
Computer Science and Business Management

Supervisor:
Doctor Fernando Brito e Abreu, Associate Professor,
Iscte
Co-supervisor:
Doctor Marina Alexandra Pedro Andrade, Assistant Professor,
Iscte

November, 2020

[This page has been intentionally left blank]

Highlighting Model Elements
to Improve OCL Comprehension

[This page has been intentionally left blank]

To my family and friends, for their unconditional support and constant encouragement.

[This page has been intentionally left blank]

Acknowledgements

I want to express my sincere gratitude to my thesis supervisors Prof. Dr. Fernando Brito e Abreu and Prof. Dr. Marina Alexandra Pedro Andrade, for their continuous support, guidance, and encouragement. I would also like to thank all the involved professors and students who took part in the experiments and used the provided prototypes, allowing for the completion of this research.

[This page has been intentionally left blank]

Abstract

Models, metamodels, and model transformations play a central role in Model-Driven Development (MDD). Object Constraint Language (OCL) was initially proposed as part of the Unified Modeling Language (UML) standard to add the precision and validation capabilities lacking in its diagrams, and to express well-formedness rules in its metamodel. OCL has several other applications, such as defining design metrics, code-generation templates, or validation rules for model transformations, required in MDD.

Learning OCL as part of a UML course at the university would seem natural but is still the exception rather than the rule. We believe that this is mainly due to a widespread perception that OCL is hard to learn, as gleaned from claims made in the literature. Based on data gathered over the past school years from numerous undergraduate students of different Software Engineering courses, we analyzed how learning design by contract clauses with UML+OCL compares with several other Software Engineering Body Of Knowledge (SWEBOK) topics. The outcome of the learning process was collected in a rigorous setup, supported by an e-learning platform. We performed inferential statistics on that data to support our conclusions and identify the relevant explanatory variables for students' success/failure. The obtained findings lead us to extend an existing OCL tool with two novel features: one is aimed at OCL apprentices and goes straight to the heart of the matter by allowing to visualize how OCL expressions traverse UML class diagrams; the other is intended for researchers and allows to compute OCL complexity metrics, making it possible to replicate a research study like the one we are presenting.

Keywords: OCL; UML; highlighting model elements; OCL comprehension.

[This page has been intentionally left blank]

Resumo

Modelos, metamodelos e transformações de modelo desempenham um papel central em MDD. OCL foi inicialmente proposta como parte da UML para adicionar os recursos de precisão e validação que faltavam nestes diagramas, e também para expressar regras de boa formação no metamodelo. OCL possui outras aplicações, tais como definir métricas de desenho, modelos de geração de código ou regras de validação para transformações de modelo, exigidas em MDD.

Aprender OCL como parte de um curso de UML na universidade parecia portanto natural, não sendo no entanto o que se verifica. Acreditamos que isso se deva a uma percepção generalizada de que OCL é difícil de aprender, tendo em conta afirmações feitas na literatura. Com base em dados recolhidos em anos letivos anteriores de vários alunos de licenciatura de diferentes cursos de Engenharia de Software, analisámos como a aprendizagem por cláusulas contratuais de UML + OCL se compara a outros tópicos do SWEBOK. O resultado do processo de aprendizagem foi recolhido de forma rigorosa, apoiado por uma plataforma de e-learning. Realizámos estatísticas inferenciais sobre os dados para apoiar as nossas conclusões, de forma a identificar as variáveis explicativas relevantes para o sucesso / fracasso dos alunos. As conclusões obtidas levaram-nos a estender uma ferramenta OCL com duas novas funcionalidades: a primeira é voltada para os estudantes de OCL e permite visualizar como as expressões percorrem um diagrama de classes UML; a segunda é voltada para investigadores e permite calcular métricas de complexidade OCL, habilitando a réplica de um estudo semelhante ao apresentado.

Palavras-chave: OCL; UML; destaque de elementos do modelo; compreensão do OCL.

[This page has been intentionally left blank]

Contents

List of Figures

[This page has been intentionally left blank]

List of Tables

[This page has been intentionally left blank]

Listings

[This page has been intentionally left blank]

Acronyms

API	Application Programming Interface.
AST	Abstract Syntax Tree.
BPMN	Business Process Model and Notation.
MDD	Model-Driven Development.
NL	Natural Language.
OCL	Object Constraint Language.
OMG	Object Management Group.
SQL	Structured Query Language.
SWEBOK	Software Engineering Body Of Knowledge.
UML	Unified Modeling Language.
USE	UML-based Specification Environment.
XML	Extensible Markup Language.

[This page has been intentionally left blank]

Chapter 1

Introduction

Contents

This chapter describes the motivation and scope of this dissertation and presents the document's contributions and organization.

[This page has been intentionally left blank]

UML[1] was created by the Object Management Group (OMG)[2] and had its first specification draft proposed in January 1997. It is currently the standard language used in software development for specifying, visualizing, assembling, and documenting artifacts of software systems. However, UML graphical modeling constructs are not precise enough to express all relevant aspects of a specification. To mitigate this problem, a semi-formal language named OCL[3] was included in the UML standard and has been employed in a beneficial way to provide precision to models [4]. OCL is based on first-order logic (*predicate calculus*) and set theory, provides multiple collection operations, and can be used in several contexts, such as expressing constraints (class invariants, pre-and post-conditions, ...) and derivation rules for attributes or associations in a class diagram. It also allows the specification of well-formedness rules and metrics at the metamodel level, querying objects in an object diagram (equivalent to Structured Query Language (SQL) queries in relational databases), and specifying guards on state transitions in state diagrams.

Formal Methods are a long-discussed topic in Software Engineering, and several studies have been conducted to assess the benefits of using OCL alongside UML models [2, 3]. Although there are still disputes about under what circumstances formal methods and languages should be used, there appears to be a consensus that OCL is advantageous to modelers once they overcome the initial learning curve, which was proved to be a difficult task [22].

1.1 Motivation and research problem

Several support tools were developed to assist in MDD, including the analysis and design phases where modelers need to interpret and write OCL expressions. These tools have their specific characteristics and provide a variety of useful functionalities, including syntactic analysis, connection with the UML model, and debugging [19]. To the best of our knowledge, none of these tools provides highlighting of class diagrams' elements for manually introduced OCL expressions, which we believe could soften the learning curve for this language by reducing the mental burden when reading, analyzing, and writing expressions. Our hypothesis is based on the multiple studies available that investigate the impact of visual aspects (color, layout, font, ...) on program comprehension, and they all reveal positive outcomes on subject's comprehensibility when enhancing programs with visual features [11, 12, 16, 21].

1.2 Objectives

As a first objective, we aim to shed some light on which factors influence OCL's learning process. To achieve this, we study the results of OCL-related questionnaires taken by undergraduate students across different school years and how distinctive variables affect their results, including questions' complexity (given by readability formulas) and answers' complexity (provided by OCL complexity metrics).

[1] UML - Available: http://www.uml.org/what-is-uml.htm Accessed: 2020-11-28
[2] OMG - Available: http://www.omg.org/ Accessed: 2020-11-28
[3] OCL - Available: https://www.omg.org/spec/OCL/2.4/PDF Accessed: 2020-11-28

As a second objective, we seek a solution to soften OCL's learning curve by reducing the cognitive effort needed to produce a clause from a specification in Natural Language (NL) correctly, in the context of a UML class diagram. To accomplish this goal, we propose the OCL Highlight Plugin (presented in the next section).

1.3 Contributions

This dissertation's contributions are the design, implementation, prototype, and experimental validation of two plugins for the UML-based Specification Environment (USE) tool. The first plugin, named OCL Highlight Plugin, provides highlighting of elements for manually introduced OCL expressions in the context of a UML class diagram. This plugin analyses the syntax tree of an OCL expression and highlights the diagram components referred in that specific expression, including classes, properties, and navigations. The second plugin, named OCL Complexity Plugin, analyses the same syntax tree and evaluates OCL expressions' complexity using a set of metrics defined by Reynoso et al. [13, 15]. The idea of including the calculation of these metrics in a plugin resulted from the studies performed to fulfill this dissertation's first objective.

1.4 Dissertation organization

This dissertation is structured in 5 chapters, where the first is the introduction. The remainder of this document is organized as follows. Chapter 2 presents the topic's background and related work, consisting of relevant research contributions. Chapter 3 describes the implemented prototypes, from a broader perspective to a more detailed one. Chapter 4 discusses the conducted experiments and relevant results. Chapter 5 contains the concluding remarks and addresses open issues and future research work directions.

Chapter 2

Related Work

Contents

In this chapter, we explore the concepts and state of the art concerning the topics that relate to our intended work, allowing us to better understand the problem under study. This chapter is organized in 5 sections, structured as follows. Section 2.1 starts by presenting the main concepts of Model-Driven Development, including the definition of UML and OCL. Section 2.2 covers existing studies regarding the comprehension of UML class diagrams and the impact of OCL on UML-based development. Section 2.3 discusses OCL expressions' complexity and presents existing metrics proposed by experts to calculate it. Section 2.4 examines relevant papers concerning the impact of syntax highlighting on program comprehension. Finally, Section 2.5 presents some OCL support tools and their functionalities.

[This page has been intentionally left blank]

To identify relevant studies for our topic, we started by defining a search strategy, which states that the title, abstract, and keywords of the articles, books, and conference proceedings will be searched in the appropriate electronic databases (IEEE Xplore, ACM Digital Library, Elsevier, Springer, ...) according to the following search *string*:

(((”uml” AND ”class diagram”) OR ”ocl” OR ”program”) AND (”comprehension” OR ”complexity” OR ”highlight”)) OR (”ocl” AND ”tool”)

To only select articles that are pertinent to our topic, the following inclusion/exclusion criteria were applied: if the abstract of the article is not related to the topic under study, or if a top publisher did not publish it (e.g., IEEE, ACM, Wiley, Springer, Elsevier), then the article was not considered. Additionally, we used forward snowballing whenever the references of the selected articles were relevant in the context of this dissertation.

2.1 Concepts of model-driven development

Model-Driven Development is presented as a software development methodology with models as the primary focus, rather than computer programs. Models are artifacts used to specify the system's required functionality and architecture as well as to guide the development of the final piece of software. Since these models are closer to the problem domain, abstracted from the underlying technologies, they are a high-level specification, easier to specify, understand, and maintain [1, 17].

To express the necessary information of a system, models need to be represented in a way that can be comprehended by modelers, developers, stakeholders, and other supporting systems. A modeling language is either a graphical or textual computer language that allows the creation of models, following a specific set of structures, terms, notations, syntaxes, semantics, and integrity rules[1]. There are many well-known modeling languages, including Business Process Model and Notation (BPMN), SQL Schema, Extensible Markup Language (XML) Schema, and UML. UML is currently accepted as the standard graphical notation for software development, but its diagrams cannot express all constraints and essential aspects of a system precisely. To address the existing limitations in UML diagrams, IBM proposed Object Constraint Language, which is now part of the UML standard. OCL is a ”semi-formal” language as it lacks inference mechanisms (second-order logic) and allows specification of invariants, pre-and post-conditions, as well as the description of guards and constraints on operations.

2.2 Comprehension of UML class diagrams

Over the years, several studies have been conducted to assess how software engineers build, understand, and design programs. In the topic of program comprehension, we are particularly interested in addressing how humans process class diagrams to acquire information about a system. These are a type of UML diagram useful for both modeling and understanding of a program. They describe a program's structure and global behavior by showing its classes, attributes, operations, and relationships.

[1] MDA Guide rev. 2.0 - Available: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01 Accessed: 2020-11-28

Guehéneuc [8] and Yusuf [21] used eye-tracking technology to obtain data on how subjects process UML class diagrams, namely how they explore, visualize, and examine data. Sight is essential in this process, making it relevant to study the human eye's behavior, where three types of information are fundamental: fixations, saccades, and scanpaths. Fixations are defined by a period where the subject maintains the visual gaze on a single location, represented by a pair of coordinates; saccades involve a quick movement between two fixation points; finally, scanpaths are defined as directed paths formed by saccades between fixations. Results of eye-tracking studies can be represented as follows: in Figure 2.1 we observe a *gaze plot* of a UML class diagrams with fixations, saccades, and scanpaths, whereas in Figure 2.2 we perceive a *heatmap* of fixations. By analyzing these two figures, it is possible to discern the path taken by the subject's eyes through the diagram and which were the main focus points. This information is essential when trying to understand how subjects visually process class diagrams. Relevant conclusions show that relationships mainly contain fixations on their ends and saccades on the line, indicating that lines are mainly used for navigation purposes. A large number of fixations on a *stimulus*, an object that is viewed by a subject, indicate a greater effort to understand the object in question due to its complexity or the poor layout arrangement. The use of color information was found useful for experts when trying to explore class diagrams more efficiently.

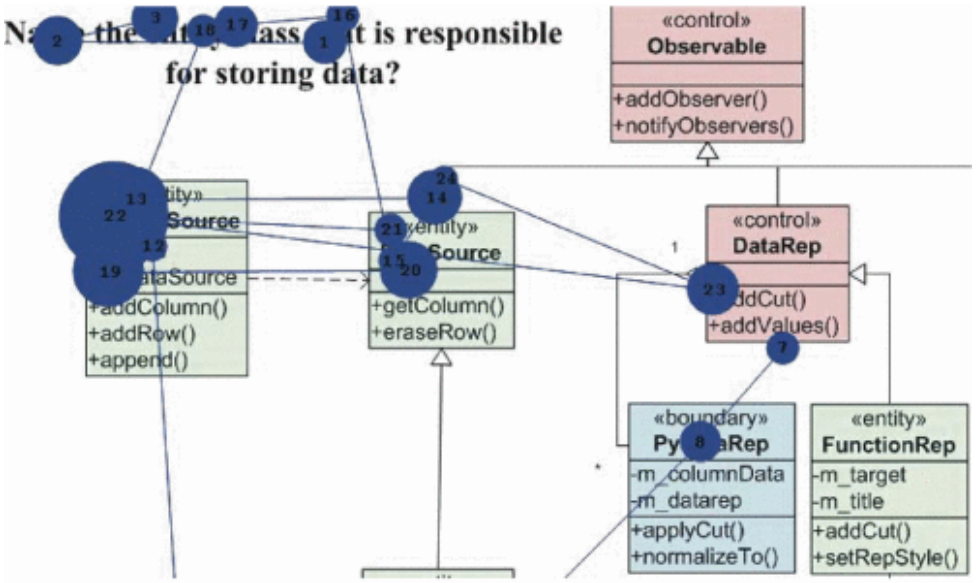

Figure 2.1: *Gaze plot* of the exploration of a question and respective UML class diagram [21]

Comprehension of UML class diagrams is fundamental when designing and maintaining OCL expressions for the reason that they are always tied to the context of this type of diagram. These expressions can be used to specify objects' invariants, pre-and post-conditions on operations, as well as to query the system data to return information to the user. The impact of OCL in UML-based development has been discussed throughout several investigations over the past years, focusing on OCL's influence on comprehension and maintainability of UML models, and considering if the additional effort and formality justify the benefit. Briand et al. [2, 3] investigated the impact of OCL during the development, understanding, and modification of UML analysis documents, which are three typical software engineering activities. The authors compared subjects' (4^{th} year software/computer engineering students) performance

Figure 2.2: *Heatmap* of a UML class diagram exploration [21]

on comprehension and maintenance tasks when using OCL to document invariants, pre-and post-conditions, which are typically documented in NL. Results showed that OCL has the potential to promote comprehension and modification of UML diagrams but requires a high level of experience and training.

2.3 Metrics for OCL expressions

The cognitive effort needed to comprehend, design, or maintain systems is influenced by the complexity levels that characterize OCL expressions. There has been an ongoing effort to define OCL complexity metrics [14] and study how they can capture different levels of comprehensibility and maintainability of OCL expressions. Reynoso et al. [13, 15] sought insight into the process of understanding OCL expressions by applying the techniques of chunking (which involves recognizing groups of declarations and the extraction of information that is remembered as a chunk) and tracing (which involves scanning through a program in different directions in search of relevant chunks). While there are many distinct metrics that capture the different aspects that characterize expressions' complexity, these authors decided to focus on the impact of tracing-based metrics since this technique was proven as an essential activity in program comprehension [14]. They found statistical significance on the impact of tracing-related metrics on the understandability and, consequently, modifiability of OCL expressions. The process is shown in Figure 2.3: the structural properties of an OCL expression affect the cognitive complexity needed to comprehend that expression, which consequently influences understandability and maintainability. Table 2.1 shows the metrics in question and their classifications, depending on their focus on the structural properties of OCL expressions: coupling (degree of interdependence between objects), size, and length.

The definition of these metrics is described below, based on [14]:

Figure 2.3: Relationship between structural properties of an OCL expression, cognitive complexity related to tracing, understandability and maintainability (based on [14])

Table 2.1: Tracing-related OCL expression metrics (based on [14])

Metric Acronym	**Metric Description**	**Metric Classification**
NNR	Number of Navigated Relationships	C
NAN	Number of Attributes referred through Navigations	C
WNO	Weighted Number of referred Operations through Navigations	C
NNC	Number of Navigated Classes	C
WNM	Weighted Number of Messages	C, S
NPT	Number of Parameters whose Types are classes defined in a class diagram	C
NUCA	Number of Utility Classes Attributes Used	C
NUCO	Number of Utility Classes Operations Used	C
DN	Depth of Navigations	L
WNN	Weighted Number of Navigations	S
WCO	Weighted Number of Collection Operations	S

Note: In the table above, C stands for coupling, S stands for size, and L stands for length.

Definition 1. *NNR Metric (Number of Navigated Relationships): Measures the total number of navigated relationships in an expression. Relationships are only counted once, even if they are navigated multiple times.*

Definition 2. *NAN Metric (Number of Attributes referred through Navigations): Counts the number of attributes referred through navigations.*

Definition 3. *WNO Metric (Weighted Number of referred Operations through navigations): Measures the sum of weighted operations (where the weight is defined by the number of in/out parameters, including the return type) mentioned through navigations.*

Definition 4. *NNC Metric (Number of Navigated Classes): Measures the number of classes, association classes or interfaces referred through navigations.*

Definition 5. *WNM Metric (Weighted Number of Messages): This metric is defined as the sum of weighted messages (where the number of parameters defines the weight) present in an expression.*

Definition 6. *NPT Metric (Number of Parameters whose Types are classes defined in the class diagram): This metric is mainly used in pre-and post-conditions, and it counts the number of different classes and interfaces used as in/out parameters or result.*

Definition 7. *NUCA Metric (Number of Utility Class Attributes used): Measures the number of utility classes' attributes used in an expression. Attributes that belong to the same utility class are only counted once, even if they are referred multiple times.*

Definition 8. *NUCO Metric (Number of Utility Class Operations used): This metric definition is similar to the NUCA metric, but instead of attributes, it considers operations.*

Definition 9. *DN Metric (Depth of Navigations): Measures the maximum depth of a navigation tree, where the root node represents the class name of the contextual instance (self). For each navigation that starts from self, we create a branch that connects to a new node, which represents the navigated class. A new tree is created if a navigation contains a collection operation defined in terms of new navigation(s), and it is connected to the original tree through a "definition connection".*

Definition 10. *WNN Metric (Weighted Number of Navigations): This metrics is defined as the sum of weighted navigations presented in an expression, where the weight is defined by the level on which the operation is used in an expression.*

Definition 11. *WCO Metric (Weighted Number of Collection Operations): Measures the sum of weighted collection operations, where the weight is defined by the level on which the operation is used in an expression.*

Later in this document (subsection 4.2.1), we present the results of our investigation on the influence of these metrics on undergraduate students' success/failure when answering OCL questionnaires. As a contribution of this dissertation, we decided to include a plugin for the USE tool[2] that allows modelers to evaluate OCL expressions and get the calculated complexities given by these metrics. The implementation details for the OCL Complexity Plugin are described in section 3.2.

2.4 Impact of syntax highlighting on program comprehension

In the context of this dissertation, it is not only crucial to understand what affects the comprehensibility of UML class diagrams and OCL expressions, but also to investigate the effort that has been made to improve program comprehension. The tasks of understanding and maintaining software systems are greatly influenced by the levels of readability and comprehensibility of programs. These can be affected by many factors, including naming scheme, indentation, commenting, and documentation. This section focuses on relevant research findings regarding the impact of syntax highlighting in program readability and comprehension. Several text editors include this feature, which allows text to be displayed in multiple colors and fonts, depending on how it is categorized. This enables the distinction between different types of text without affecting the behavior of the code. Figure 2.4 shows the same code snippet with

[2]USE - Available: https://sourceforge.net/p/useocl/wiki/ Accessed: 2020-11-28

and without syntax highlighting. On the right side, the essential keywords, e.g., *while* or *print*, and the comment are easily detected, compared to the rest of the code as they are displayed in different colors.

```
# What is the output of fa(17)?

def fa(x):
    i = 2
    while x/2 > i:
        if x%i == 0: print "No"
        i = i+1
    print "Yes"
```

```
# What is the output of fa(17)?

def fa(x):
    i = 2
    while x/2 > i:
        if x%i == 0: print "No"
        i = i+1
    print "Yes"
```

Figure 2.4: Same code snippet with (right) and without (left) syntax highlighting [16]

Initial studies were mainly concerned with the effect of colors on program comprehension to facilitate the learning process and increase developers' productivity, as most tools were only available in black and white. Rambally [12] studied the effect of two color schemes on program comprehension when compared to a version of the same program presented in black and white: Colour-scheme-A used different colors for code blocks, e.g., loops and conditionals; Colour-scheme-B used different colors to identify statements and functions, e.g., I/O (Input/Output), decision and declaration statements, and variable binding. From a sample of 44 intermediate-level (novice) and 35 senior-level (expert) programming students, results showed that, in general, both groups could better understand programs when using Colour-scheme-B. An exciting note mentioned in this experiment is that the participants subjectively favored Colour-scheme-A as the easiest to understand, even though results supported Colour-scheme-B as the most effective for completing the given comprehension tasks.

Several years after this research, Sarkar [16] evaluated the impact of syntax coloring on program comprehension time, using ten graduate computer science students. Even though the small number of participants threatens the experiment's validity, it showed that assignments were solved remarkably faster when using syntax highlighting. The author collected the participants' sessions with an eye-tracking device and discerned that syntax highlighting directed participants' focus on smaller code regions (see Figure 2.5). In some cases, even some keywords were ignored completely, as seen in Figure 2.6 (the numbers represent the order in which the fixations occurred). Another interesting remark taken from this investigation is that the benefits of syntax highlighting in novice programmers are significantly more relevant when compared to experienced programmers, which might indicate that it is a useful feature when learning a certain language, but less important for using it.

An important aspect when using syntax highlighting is to decide which features to include and how to code them. It is essential to determine which is the most efficient way to display certain information, e.g., choosing to use color X over Y for a specific category of information, or selecting a background color instead of text color. Mehta and Zhu [11] focus on the effect of colors (red and blue) on cognitive task performance. They start by describing the theory defended by color experts, which states that people create specific associations to colors depending on the repeated situations they experience with that color: red is an intensive color, generally associated with errors or dangers, while blue is mostly associated with calmness, peace, and tranquillity. This research shows that distinct colors offer different benefits, and choosing the

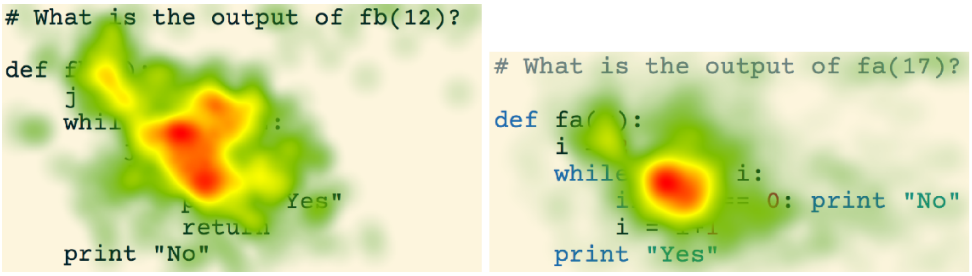

Figure 2.5: Fixation *heatmap* for the same code snippet with (right) and without (left) syntax highlighting [16]

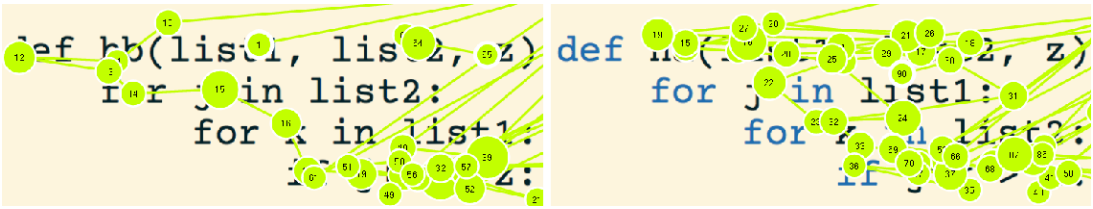

Figure 2.6: *Gaze plot* for the same code snippet with (right) and without (left) syntax highlighting [16].

right one depends on the nature of the task. As red primarily activates an avoidance motivation, it is best suited to enhance performance on detail-oriented tasks. On the other hand, blue activates an approach motivation, which is more beneficial for creative tasks.

In summary, previous research shows that visual coding hints through syntax highlighting can improve program comprehension and consequently reduce the time needed to complete a given task, especially for novice developers. As many distinct features can be used (e.g., color and font size), it is essential to understand the different effects they provoke on people to be able to choose the appropriate one depending on the desired outcome. As one of the main goals of this dissertation is to soften OCL's learning curve, we developed a second plugin for USE, this time providing syntax highlight to the elements of a UML class diagram referenced by an OCL expression. The implementation details for the OCL Highlight Plugin are presented in section 3.1, and the experiments of its success are described in subsection 4.3.

2.5 OCL tools

There are currently several OCL tools available, either freely or for commercial use, that assist modelers in developing, analyzing, and maintaining systems. It is necessary to investigate their main functionalities to understand what this dissertation can offer to the current body of knowledge. Toval et al. [19] defined a set of features found desirable when working with OCL and presented a comparison of the main characteristics (shown in Table 2.2) of the existing tools. In the meantime, some of them received updates (for example, to support newer versions of OCL), some had their development canceled or were integrated into other systems, and other tools emerged. For our topic, we are mainly interested in the available features of the two most common OCL tools, which are USE (UML-based Specification Environment), and Eclipse OCL[3].

[3]Eclipse OCL - Available: https://projects.eclipse.org/projects/modeling.mdt.ocl Accessed: 2020-11-28

Table 2.2: Main OCL tools characteristics (based on [19])

Analysis	Communication	Features	Dynamic validation
Syntatic analysis	Model-independent	Guided support for constraint development	Invariant validation
Type checking	With connection to model	Code generation from OCL specifications	Consistent checking of contraints
		Insertion of OCL expressions*	Pre- and post-conditions validation

* Includes three possibilities: imported from a UML model, imported from an independent file or manually introduced.

USE is one of the pioneering OCL tools. It was initially developed in Java by Mark Richters as a dissertation project at the University of Bremen [5]. It is freely distributed under GPLv2 (GNU General Public License version 2.0), and the latest version, USE 6.0.0, was released in September 2020. Main functionalities include syntactic analysis, type checking, dynamic validation of invariants and pre-and post-conditions, and consistency checking. USE's core feature is to validate specifications that consist of UML class diagrams and OCL expressions (invariants and pre-and post-conditions), which can be effective when helping modelers and developers in early development stages. It is also important to refer that USE allows the connection with UML models, made through a specific file (*.use*). Additional functionalities include an evaluation browser, which allows users to introduce OCL expressions manually (see Figure 2.7) and debug them (see Figure 2.8), as it generates an Abstract Syntax Tree (AST) from the input text that is evaluated against the model and returns either a syntax error or the result of the given expression; and highlighting of the coverage of the expressions in a class diagram (see Figure 2.9). As a last remark, this tool allows third parties to contribute with additional functionalities through plugins. These three features are the most significant in this dissertation's context, as they provide the basis for the development of the OCL Highlight Plugin, which highlights class diagram elements when evaluating OCL expressions in the evaluation browser.

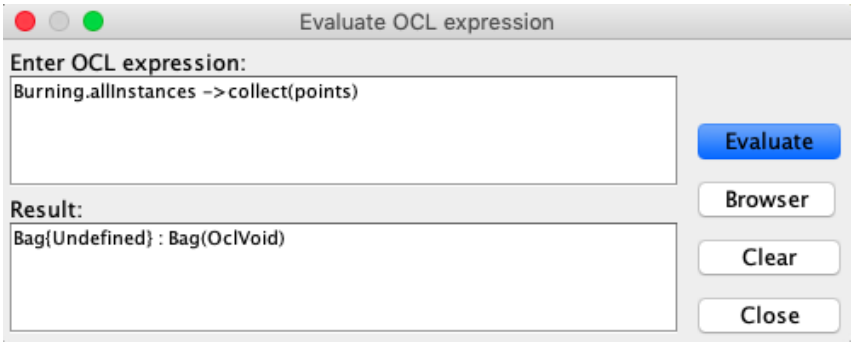

Figure 2.7: USE: Expression evaluation (*Royal and Loyal* example from [20])

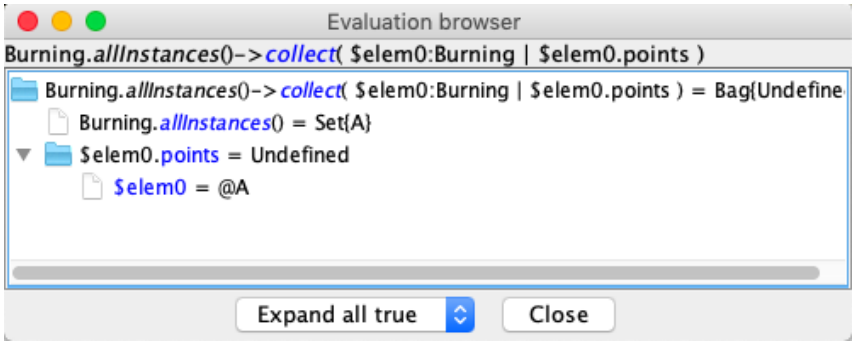

Figure 2.8: USE: Evaluation browser (*Royal and Loyal* example from [20])

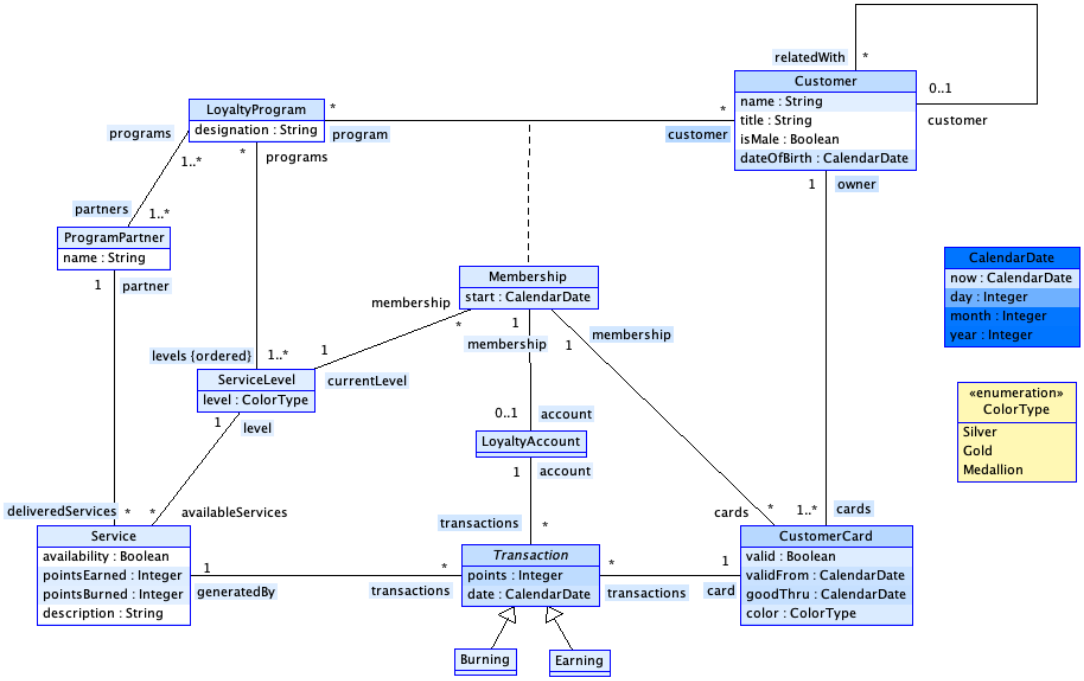

Figure 2.9: Class diagram view with coverage (*Royal and Loyal* example from [20])

As mentioned above, Eclipse OCL is also a relevant tool to analyze, as it is becoming the most popular OCL tool [19]. It is part of the Eclipse Modeling Project, which focuses on supporting model-driven development by presenting a consolidated set of modeling frameworks, tools, and standards implementations. It enables editing, refactoring, code generation, execution, and interactive debugging of OCL constraints, in the context of a class model. The dynamic input and evaluation of OCL expressions can not only be made on the OCL console (see Figure 2.10) but also using the Java Application Programming Interface (API). It also provides a debugger tool (see Figure 2.11), which includes syntax highlighting with error indications, quick fixes, and content assist. Unfortunately, the available documentation does not mention highlighting on the UML class diagram, which is the most critical topic of this dissertation.

Both tools offer modelers a wide variety of useful functionalities. Still, none provides dynamic highlighting of UML class diagram elements present in OCL expressions, which we believe can soften this language's learning curve. For our research topic, USE is the most relevant tool as it provides the basis for highlighting on the UML class diagram.

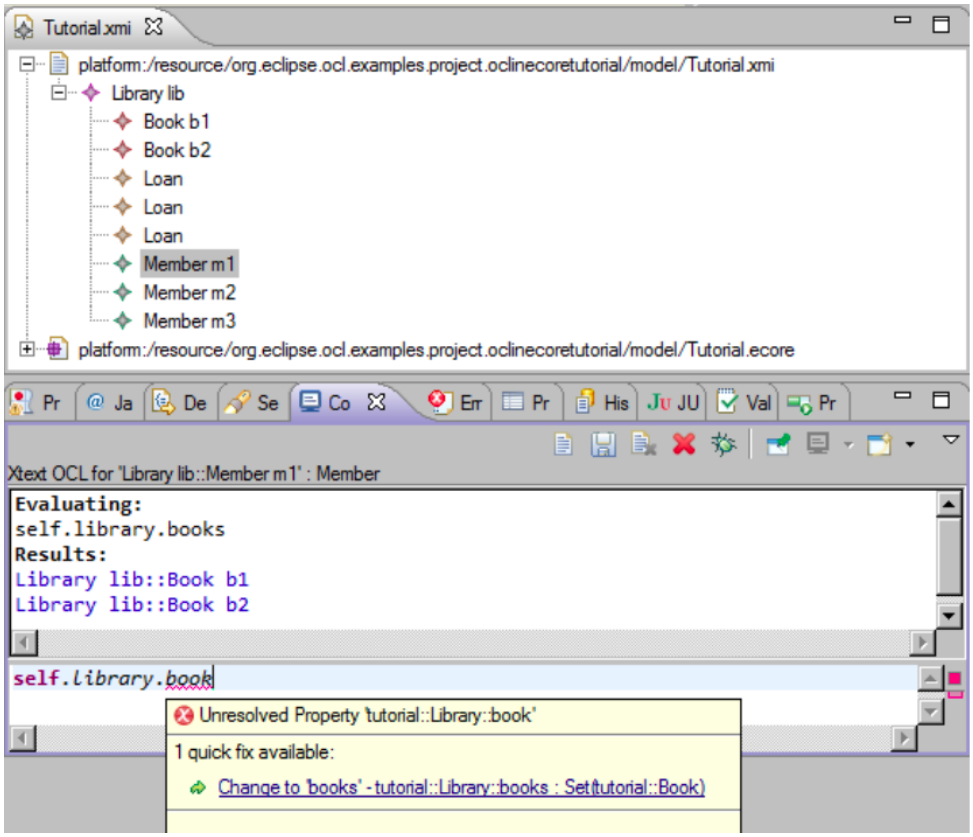

Figure 2.10: EclipseOCL: OCL console

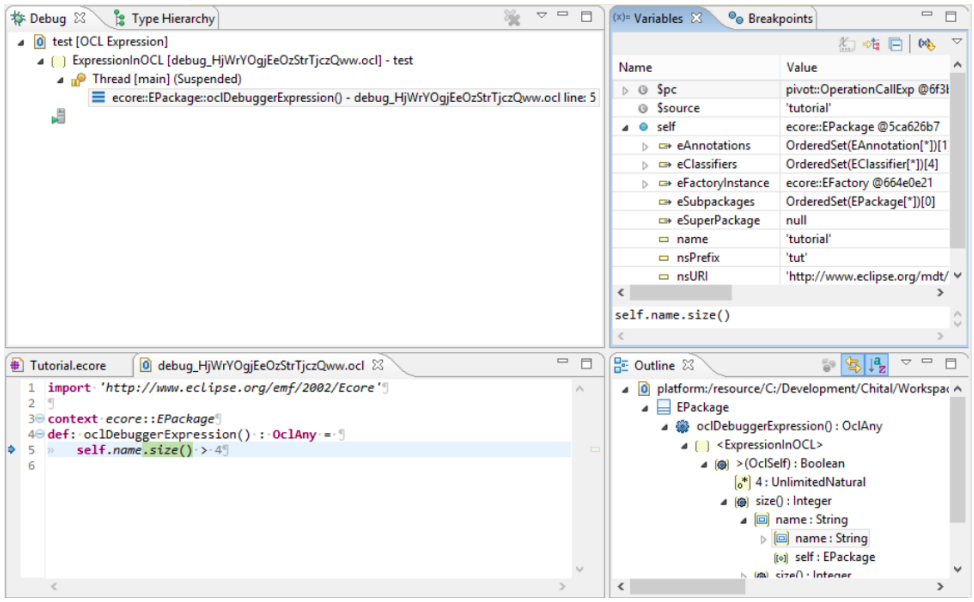

Figure 2.11: EclipseOCL: OCL debugger

Chapter 3.

Prototypes

Contents

This chapter presents the requirements, design, and implementation of the two proposed prototypes, from a broader perspective of the component diagram to a detailed description of the implementation details. Section 3.1 concerns the OCL Highlight Plugin, followed by Section 3.2, where the OCL Complexity Plugin is described.

[This page has been intentionally left blank]

As mentioned in previous sections, in this dissertation we propose a tool-based learning feature, dubbed "OCL Highlight Plugin", and an investigation feature, entitled "OCL Complexity Plugin", reified as plugins to the USE tool. Both plugins use the same components provided by USE, as presented in the component diagram in Figure 3.1. The concrete implementation of each plugin is described in detail in the following sections.

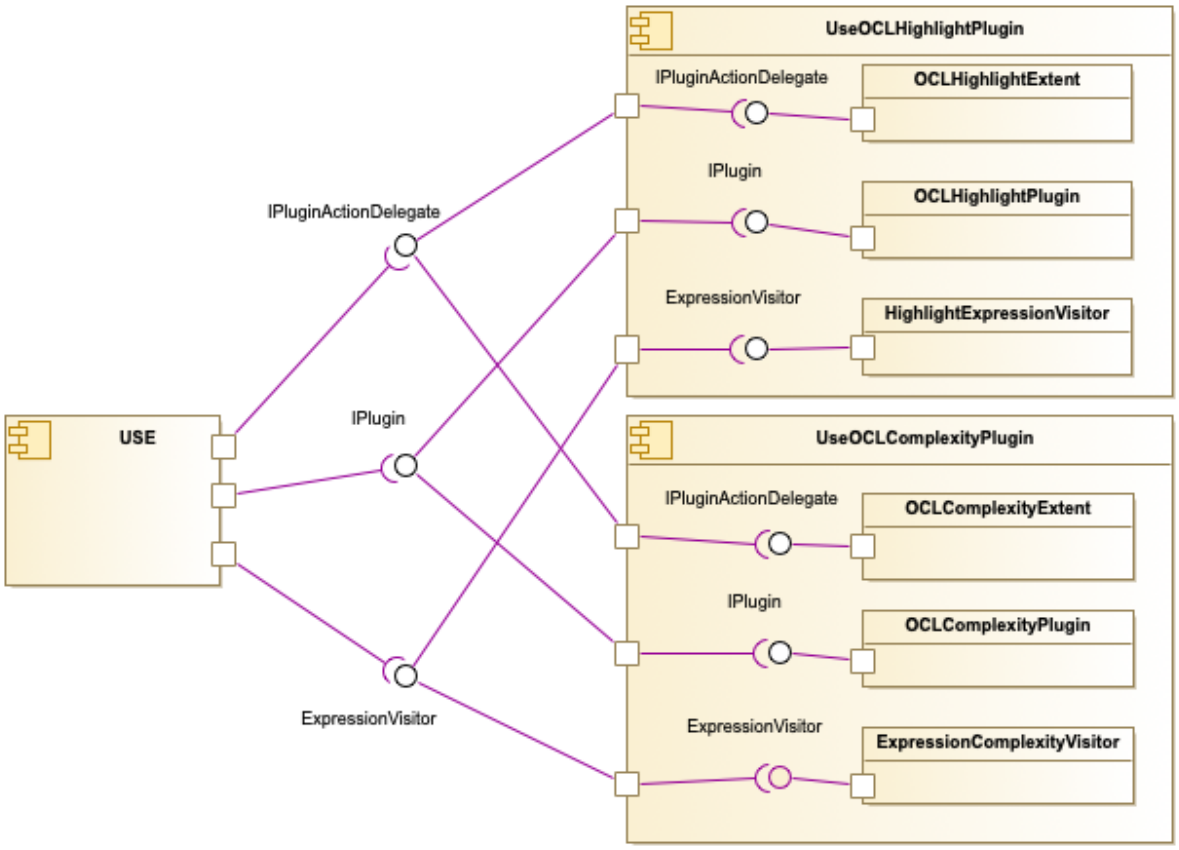

Figure 3.1: OCL Highlight Plugin and OCL Complexity Plugin: component diagram

To make use of these plugins, users should download the corresponding *.jar* files (available in each of the corresponding GitHub[1] repositories indicated below) and place it in the *plugins* folder of USE's installation. After restarting USE, two new buttons will show up (see Figure 3.2): clicking on the red marker icon opens the OCL Highlight Plugin, and the green ruler icon will pop-up the OCL Complexity Plugin window. The user should then import a *.usefile* with the definition of the model and a *.soil* file with its instantiation (objects). Creating a class diagram view is an additional step to make use of the highlight feature.

Figure 3.2: OCL Highlight Plugin (red marker) and OCL Complexity Plugin (green ruler) icons

[1] GitHub - Available: https://github.com/ Accessed: 2020-11-28

3.1 OCL Highlight Plugin

The purpose of the OCL Highlight Plugin is to allow the user to highlight UML class diagram elements referenced in an OCL expression. These elements include classes, attributes, operations, and relationships. By using intense colors to emphasize the mentioned elements while coloring the rest in lighter colors, we aim to focus the user's attention on the components that are relevant to the expression under study.

Since USE already includes a coverage option to highlight the UML class diagram elements accessed by each invariant, pre-and post-condition, or query operation, the starting point for our work was already in place, speeding up the prototyping phase. The coverage given by USE can be seen in a discriminate way (using the elements browser to select a specific query) or in an integrated manner (displaying all the defined queries at the same time). The highlight proposed in this thesis is presented dynamically, whereas in USE (Gogolla et al. [6, 7]) it is defined as static (structural coverage when the model is compiled).

3.1.1 Requirements

A set of technical and functional requirements is described for this plugin, stating the main functionalities and some optional yet desired operations that improve its usability. As a technical requirement, it was defined that the plugin should be compatible with USE, meaning that the user can access its functionalities when working with this tool. In regards to functional requirements, the primary use case is that a user can insert OCL expressions and request the system to highlight its elements on a class diagram. As non-essentials requirements, it was stated that the user could reset the highlighting of the class diagram in order to visualize it as it was initially (default colors defined by USE). It was also suggested that the user could configure the colors of the highlight for each specific element of a class diagram (class, enum, attribute, operation, rolename, and edge). The different use cases defined for this plugin are shown in a use case diagram (Figure 3.3), and typical usage is presented in an activity diagram (Figure 3.4).

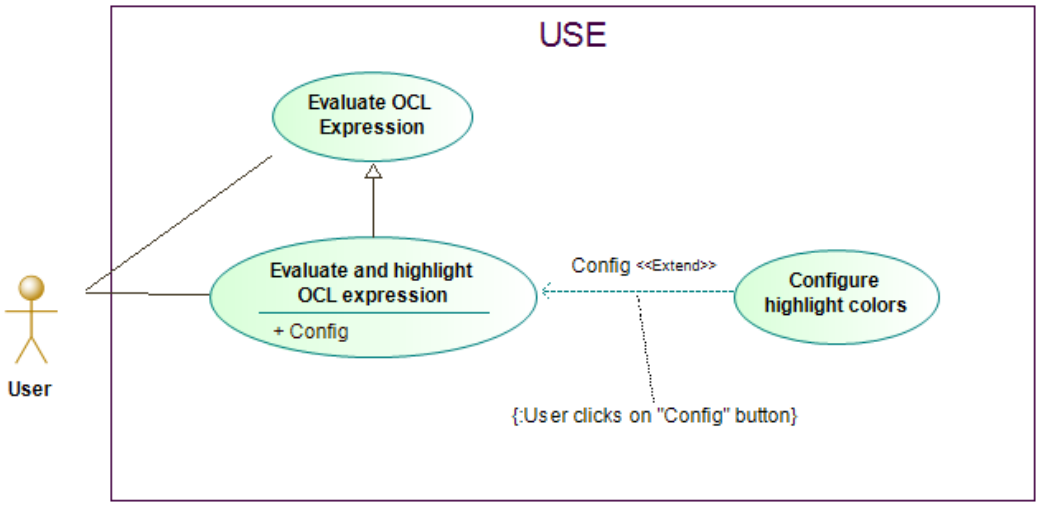

Figure 3.3: OCL Highlight Plugin: use case diagram

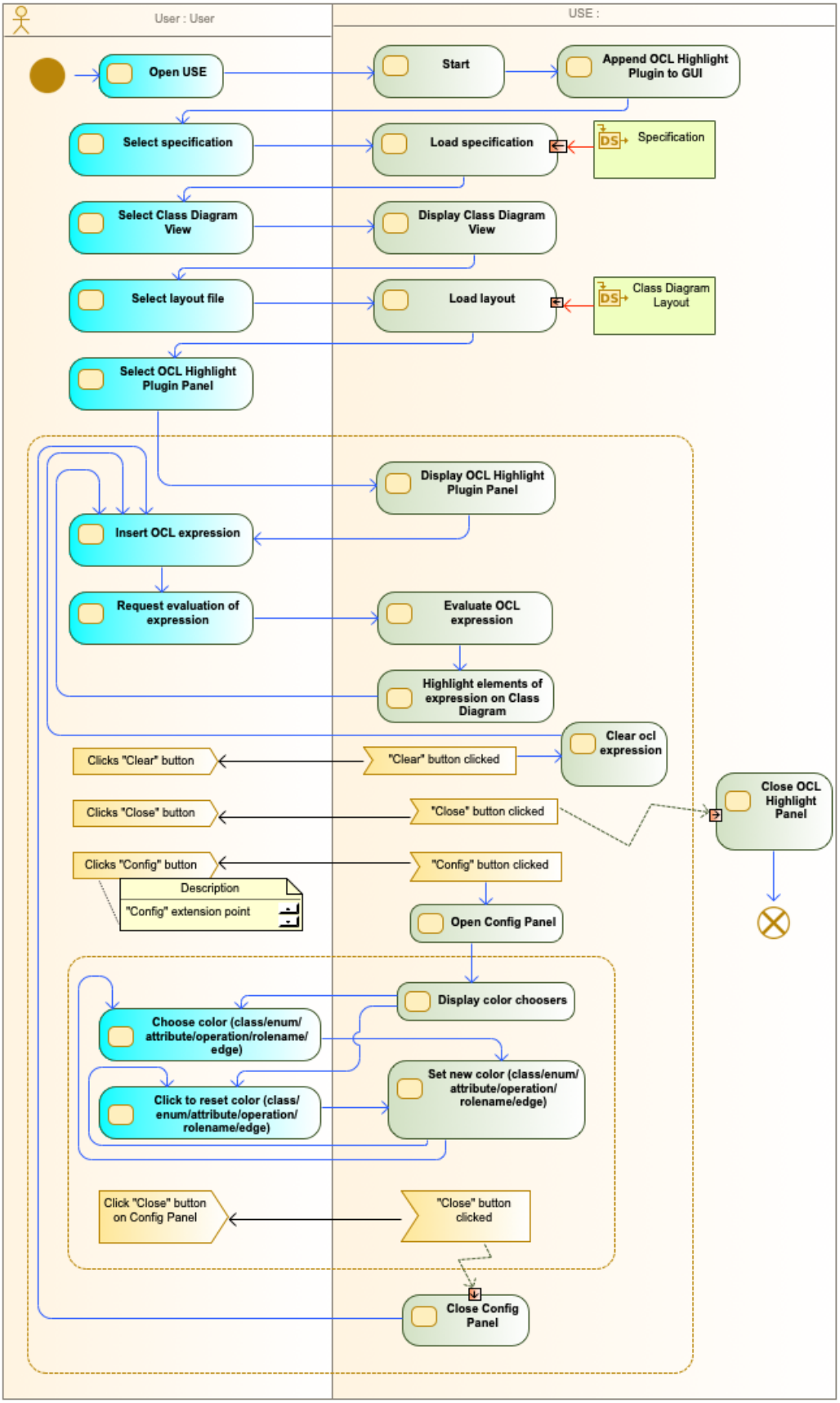

Figure 3.4: OCL Highlight Plugin: activity diagram

3.1.2 Design

In this subsection, we describe the design of the plugin. Figure 3.5 presents the class diagram, whereas Figure 3.6 shows the package diagram.

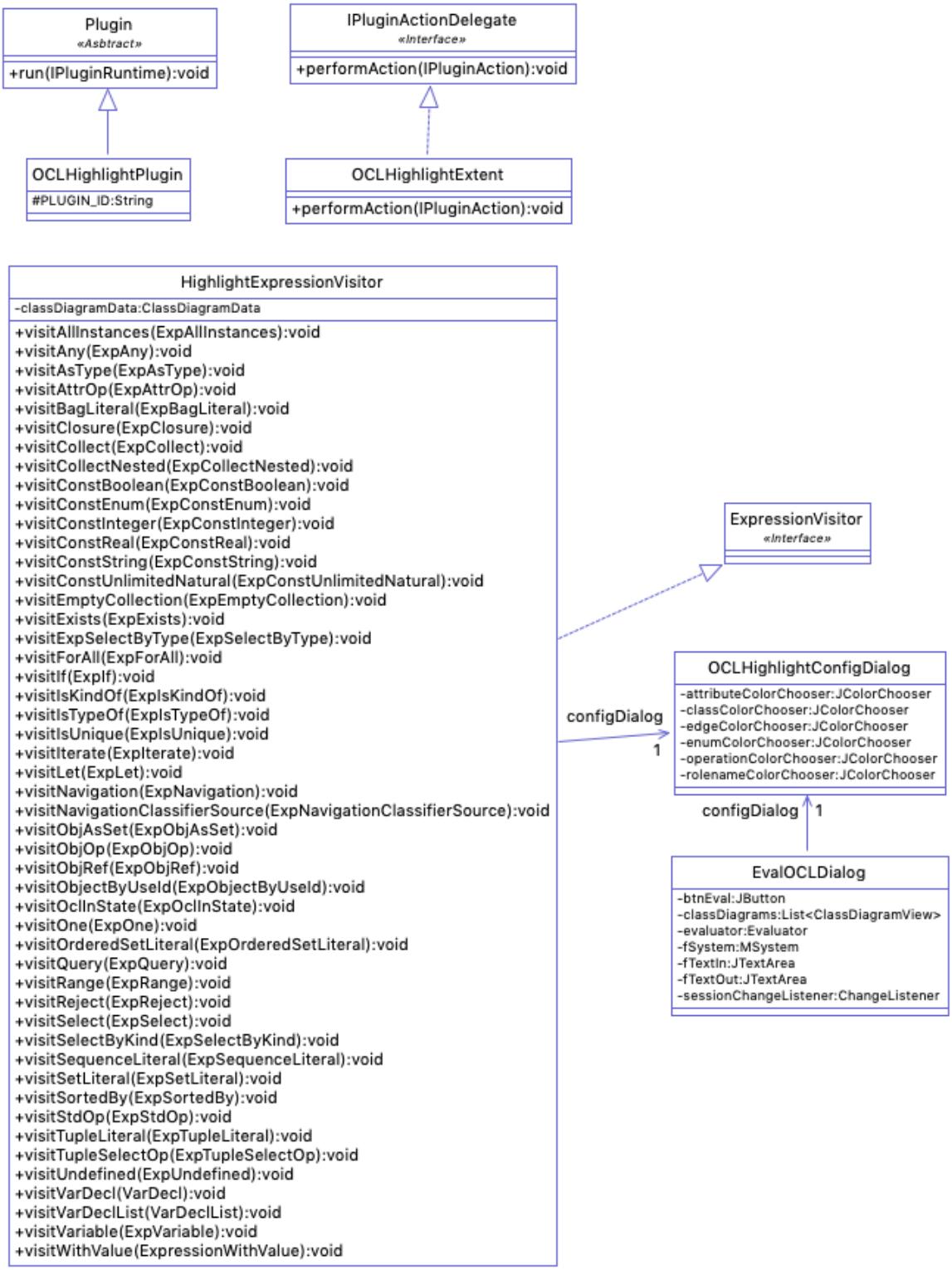

Figure 3.5: OCL Highlight Plugin: class diagram

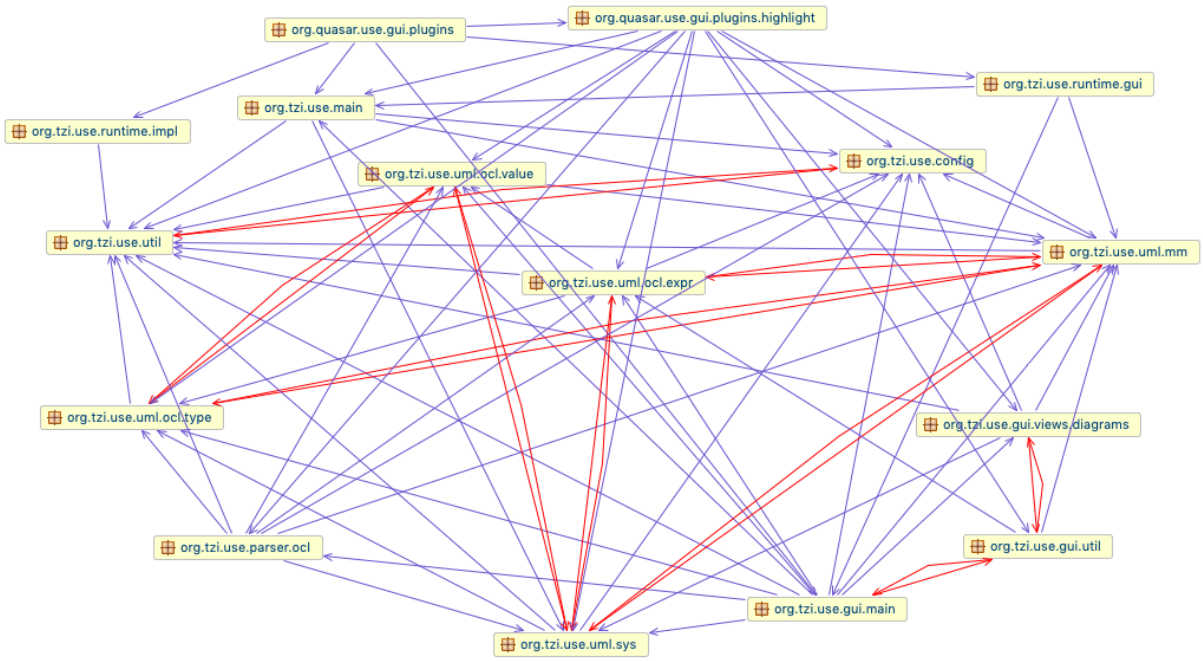

Figure 3.6: OCL Highlight Plugin: package diagram

The implementation of this plugin consisted of the development of the five classes described below:

(1) *OCLHighlightPlugin*: Main class of the OCL Highlight Plugin. This class defines the id used to identify the plugin. The attribute of this class is shown in Table 3.1.

Table 3.1: OCL Highlight Plugin: *OCLHighlightPlugin* class attributes

Attribute	**Description**
String *PLUGIN_ID*	Id used to identify the plugin

(2) *OCLHighlightExtent*: This is the plugin action class. It provides the action which will be performed if the corresponding *Plugin Action Delegate* is called. In this case, the action consists of displaying the panel *EvalOCLDialog* (with highlight functionality). This class has no attributes.

(3) *HighlightExpressionVisitor*: This is the expression visitor class, which implements the methods from the interface *ExpressionVisitor* (available in USE), coloring the visited classes of a given OCL expression. The attributes of this class are shown in Table 3.2.

Table 3.2: OCL Highlight Plugin: *HighlightExpressionVisitor* class attributes

Attribute	**Description**
ClassDiagramData *classDiagramData*	Encapsulates all elements present in a class diagram
OCLHighlightConfigDialog *configDialog*	Dialog for configuring highlight colors

(4) *EvalOCLDialog*: This class extends the functionalities of *JDialog*[2], defining the aspect and actions for entering and evaluating OCL expressions (see Figure 3.7), including the creation of text components, labels, and buttons (and their respective action). The attributes of this class are shown in Table 3.3.

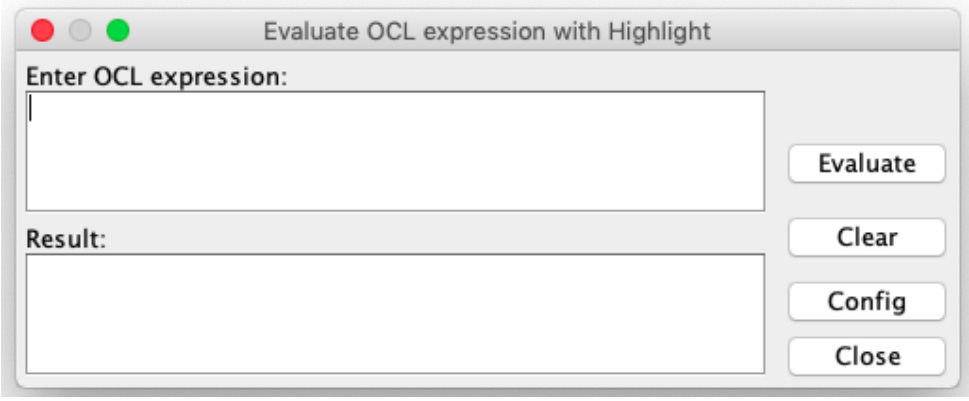

Figure 3.7: OCL Highlight Plugin: panel

Table 3.3: OCL Highlight Plugin: *EvalOCLDialog* class attributes

Attribute	**Description**
MSystem *fSystem*	Defines the system, including state and functionality
JTextArea *fTextIn*	Text area that captures the expressions introduced by the user
JTextArea *fTextOut*	Text area that displays the results of the evaluation of the expressions from *fTextIn*
Evaluator *evaluator*	Evaluates expressions
JButton *btnEval*	Button that triggers the *evaluator*
List<ClassDiagramView> *classDiagrams*	List of the available class diagrams
OCLHighlightConfigDialog *configDialog*	Dialog for configuring highlight colors
ChangeListener *sessionChangeListener*	Listener that configures the session of the *fSystem*

(5) *OCLHighlightConfigDialog*: A dialog for configuring highlight colors. Similar to *EvalOCLDialog*, this class extends the functionalities of *JDialog* (see Figure 3.8), and defines a set of default colors that can be customized by the user. The attributes of this class are shown in Table 3.4.

[2]Package javax.swing - Available: https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html Accessed: 2020-11-28

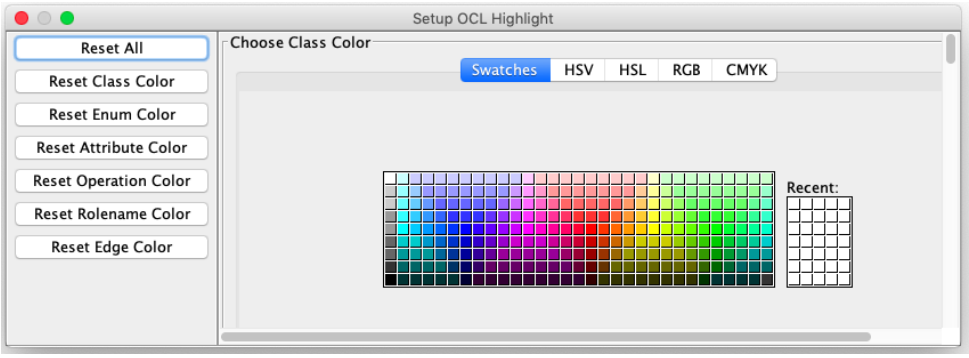

Figure 3.8: OCL Highlight Plugin: highlight color configuration panel

Table 3.4: OCL Highlight Plugin: *OCLHighlightConfigDialog* class attributes

Attribute	**Description**
Color *CLASS_COLOR*, *ENUM_COLOR*, *ATTRIBUTE_COLOR*, *OPERATION_COLOR*, *ROLENAME_COLOR*, *EDGE_COLOR*	Default color for classes, enums, attributes, operations, rolenames, and edge, respectively
JColorChooser *classColorChooser*, *enumColorChooser*, *attributeColorChooser*, *operationColorChooser*, *rolenameColorChooser*, *edgeColorChooser*	Color choosers for classes, enums, attributes, operations, rolenames, and edge, respectively

3.1.3 Implementation

The OCL Highlight Plugin was developed in Java 8[3] as an extension for USE. This plugin implements a new OCL evaluation dialog, which resembles the one already available in the tool, offering highlighting of elements in UML class diagrams when users evaluate OCL expressions. The OCL expressions are parsed and evaluated using a Visitor pattern[4], which inherits the functionalities from the interface *ExpressionVisitor* made available by USE. After the evaluation, the corresponding UML diagram elements (such as classes and properties) are highlighted, using a different color than the one provided as default (allowing them to stand out from the rest of the components). Since the API provided by the original tool did not always expose methods and properties to customize the class diagram components, it was crucial to make the necessary changes using reflection[5]. The concrete implementation of this plugin is available on Github[6], and a short demo video is accessible on Youtube[7].

[3]Java 8 - Available: https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html Accessed: 2020-11-28

[4]Visitor pattern - Available: https://sourcemaking.com/design_patterns/visitor Accessed: 2020-11-28

[5]Reflection - Available: https://www.oracle.com/technetwork/articles/java/javareflection-1536171.html Accessed: 2020-11-28

[6]OCL Highlight Plugin repository - Available: https://quasarresearchgroup.github.io/useOCLhighlight/ Accessed: 2020-11-28

[7]OCL Highlight Plugin demo - Available: https://www.youtube.com/watch?v=ZVBQ705BFi8&t=16s Accessed: 2020-11-28

3.1.4 Highlighting examples

As examples of the plugin's behavior, Expressions 3.1 and 3.2 illustrate two distinct highlighting results, using the *Royal and Loyal* model [20].

Expression 1: This expression exemplifies how to get the balance of TAP's points, which is an instance of *ProgramPartner*. The corresponding highlighting is illustrated in Figure 3.9. The highlighted element are: *ProgramPartner* class, which represents TAP; the navigation from *ProgramPartner* to *Service* (rolename *deliveredServices*), and the *Service* class; and the navigation from *Service* to *Transaction* (rolename *transaction*), and the class *Transaction*. In this expression, we want to collect the sum of *points* (property) of a *Transaction*. If a *Transaction* is of type *Earning*, the points are counted positively. If not, they are counted negatively. Both *Earning* and *points* of *Transaction* are also highlighted.

```
1 TAP.deliveredServices.transactions
2     ->collect(t | if(t.oclIsTypeOf(Earning))
3         then t.points else - t.points endif)
4     ->sum
```

Listing 3.1: OCL expression 1

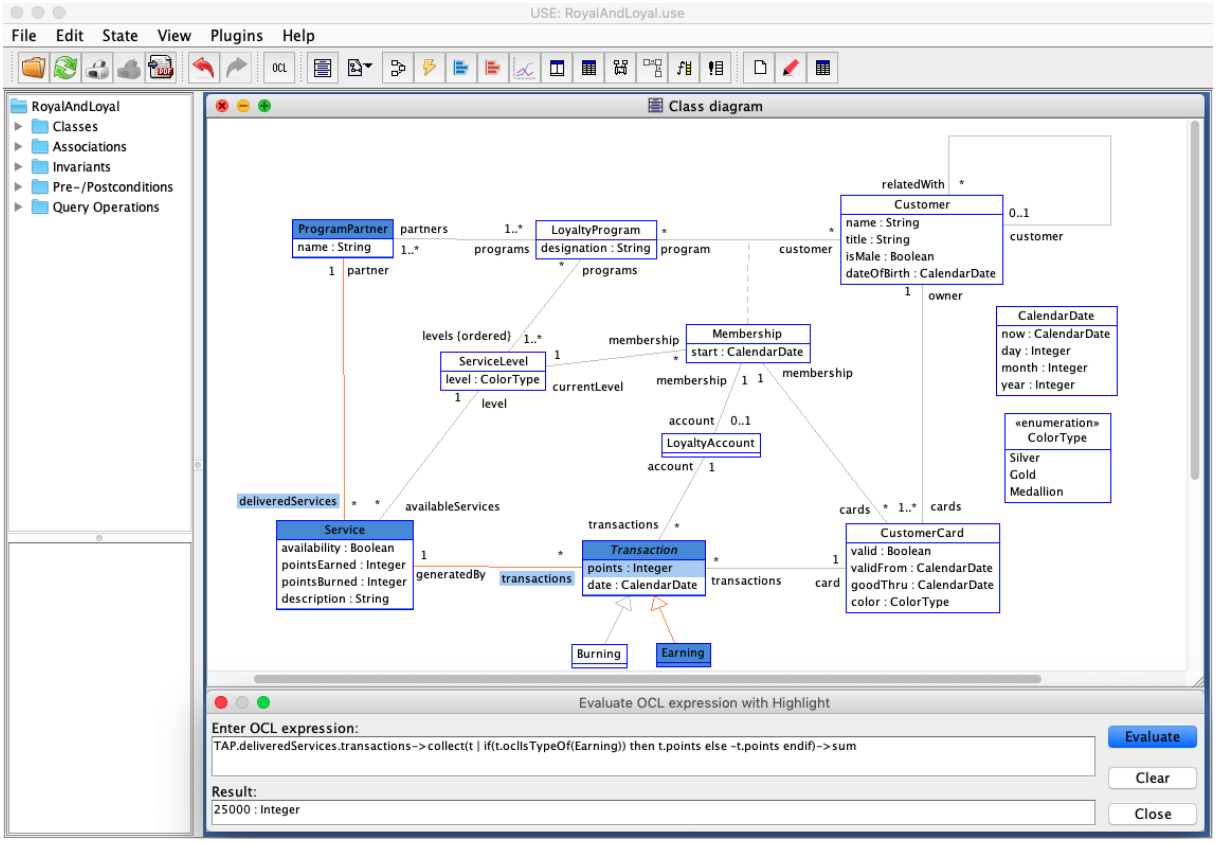

Figure 3.9: OCL Highlight Plugin: expression 1

Expression 2: This expression exemplifies how to get the number of non-Silver services provided by TAP in the participating Loyalty Programs. The corresponding highlight is illustrated in Figure 3.10. The highlighted elements are: *ProgramParner*, which again represents

TAP; the navigation between *ProgramPartner* and *LoyaltyProgram* (rolename *programs*), and *LoyaltyProgram* class; the navigation between *LoyaltyProgram* and *ServiceLevel* (rolename *levels*), and the class *ServiceLevel*; the *level* (property) of the class *ServiceLevel* and its corresponding type (*ColorType* enum); the navigation from *ServiceLevel* to *Service* (rolename *availableServices*), and the class *Service*.

```
1 TAP.programs.levels
2     ->select(l | l.level <> #Silver)
3     .availableServices->size
```

Listing 3.2: OCL expression 2

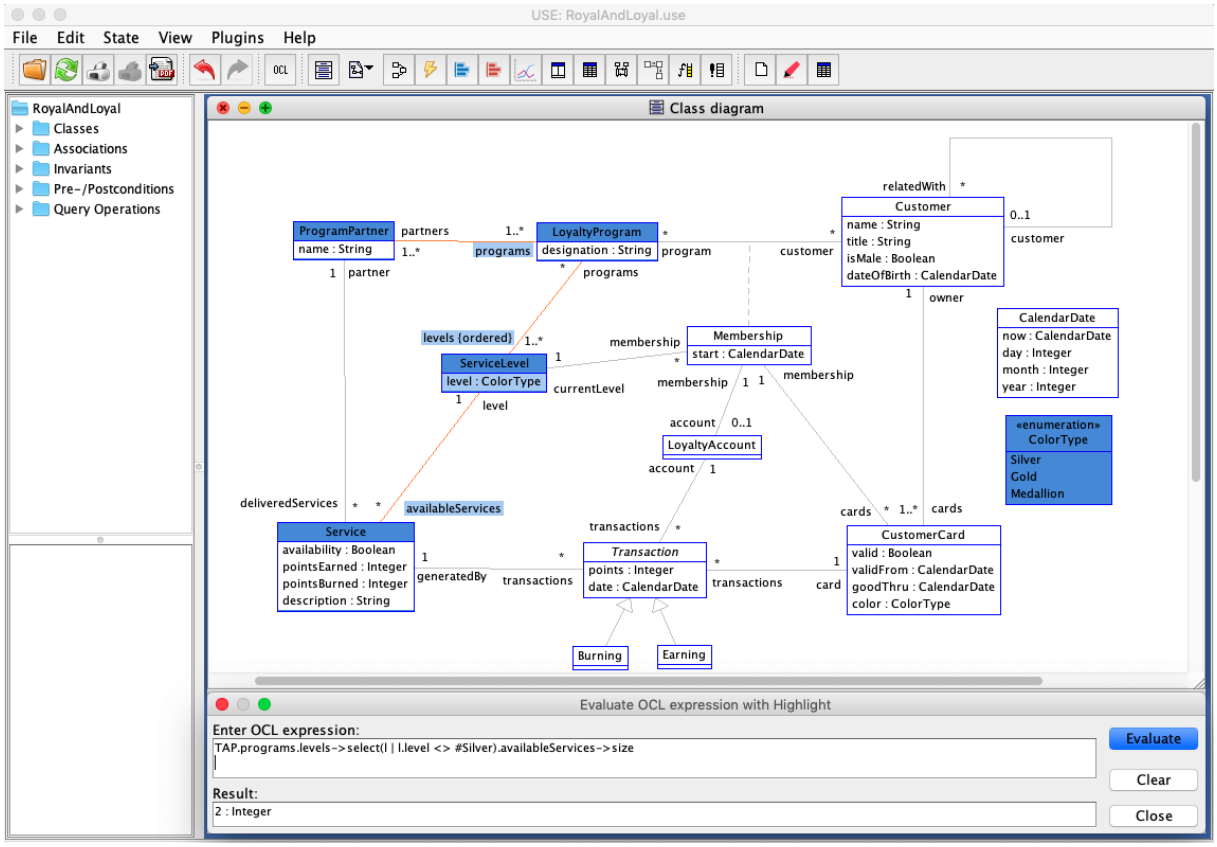

Figure 3.10: OCL Highlight Plugin: expression 2

3.2 OCL Complexity Plugin

The second and last plugin presented in this thesis is the OCL Complexity Plugin. As our investigation required to study and analyze several OCL complexity metrics (described in Section 2.3), we decided to provide them as a plugin to allow users to evaluate expressions' complexity in an automated manner.

3.2.1 Requirements

This plugin shares the technical requirement with the previous one, meaning that it should be compatible with USE. Regarding functional requirements, we defined that the user should be able to insert OCL expressions, request it's computation and display of the corresponding complexity. To improve usability, we stated that it should be possible to clear the given expression, allowing the user to insert a new one quickly. Additionally, we specified that there should be a panel with a brief explanation of each of the implemented metrics to help the user understand the results of the computation. The different use cases defined for this plugin are shown in a use case diagram (Figure 3.11), and a typical usage is presented in an activity diagram (Figure 3.12).

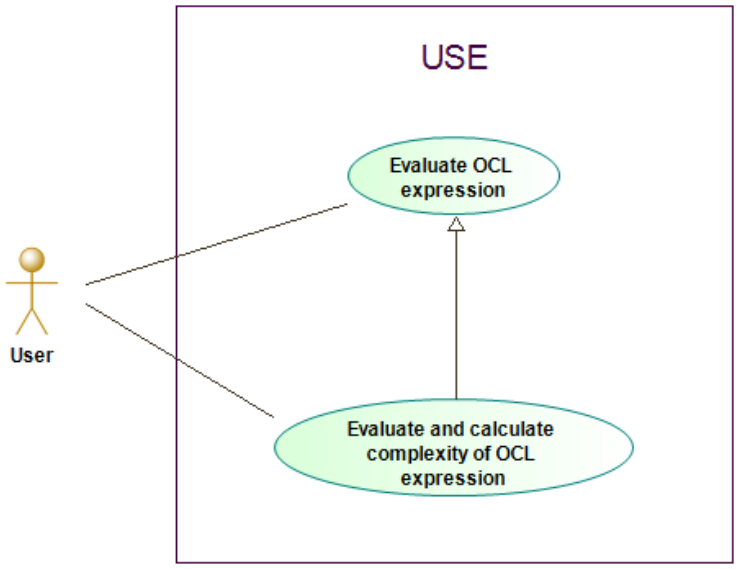

Figure 3.11: OCL Complexity Plugin: use case diagram

3.2.2 Design

In this subsection, we describe the design of the plugin. Figure 3.13 presents the class diagram of the implemented system, whereas Figure 3.14 shows the package diagram. The implementation of this plugin consisted of the development of the eight classes and two interfaces described below:

(1) *OCLComplexityPlugin*: Main class of the OCL Complexity Plugin. This class defines the id used to identify the plugin. The attribute of this class is shown in Table 3.5.

Table 3.5: OCL Complexity Plugin: *OCLComplexityPlugin* class attributes

Attribute	**Description**
String *PLUGIN_ID*	Id used to identify the plugin

(2) *OCLComplexityExtent*: This is the plugin action class. It provides the action which will be performed if the corresponding *Plugin Action Delegate* is called. In this case, the action consists of displaying the panel (with the complexity calculation functionality). This class has no attributes.

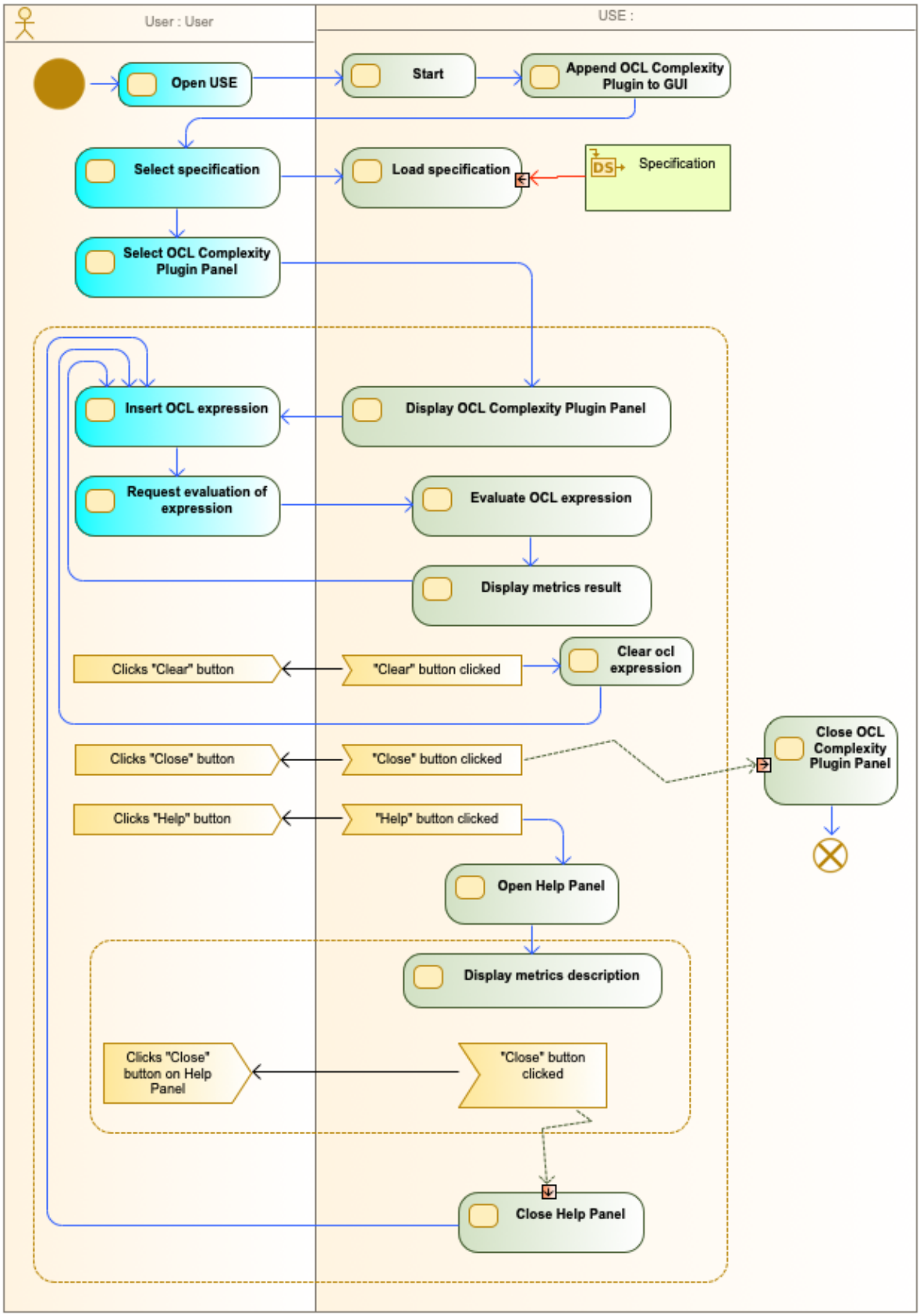

Figure 3.12: OCL Complexity Plugin: activity diagram

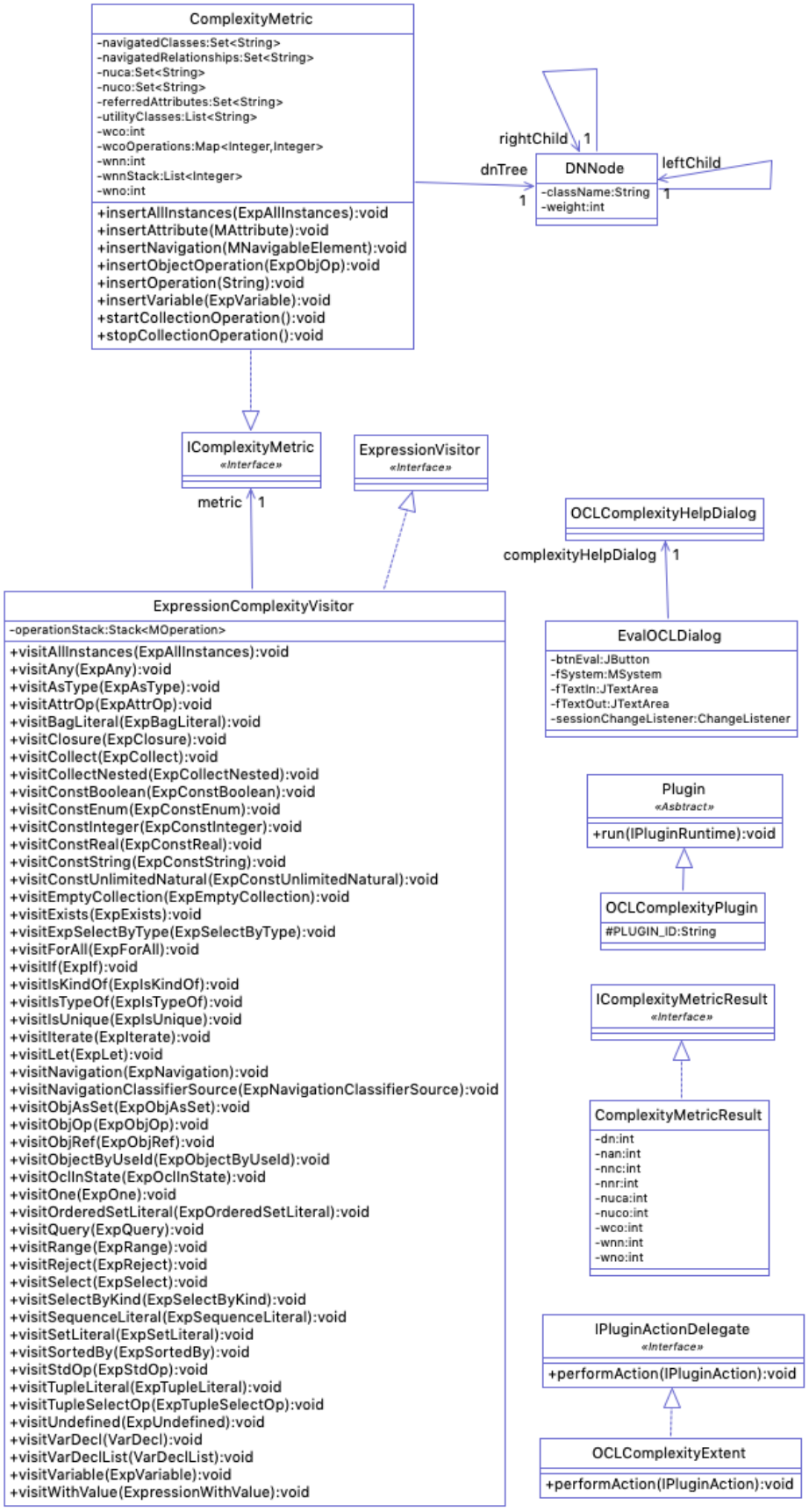

Figure 3.13: OCL Complexity Plugin: class diagram

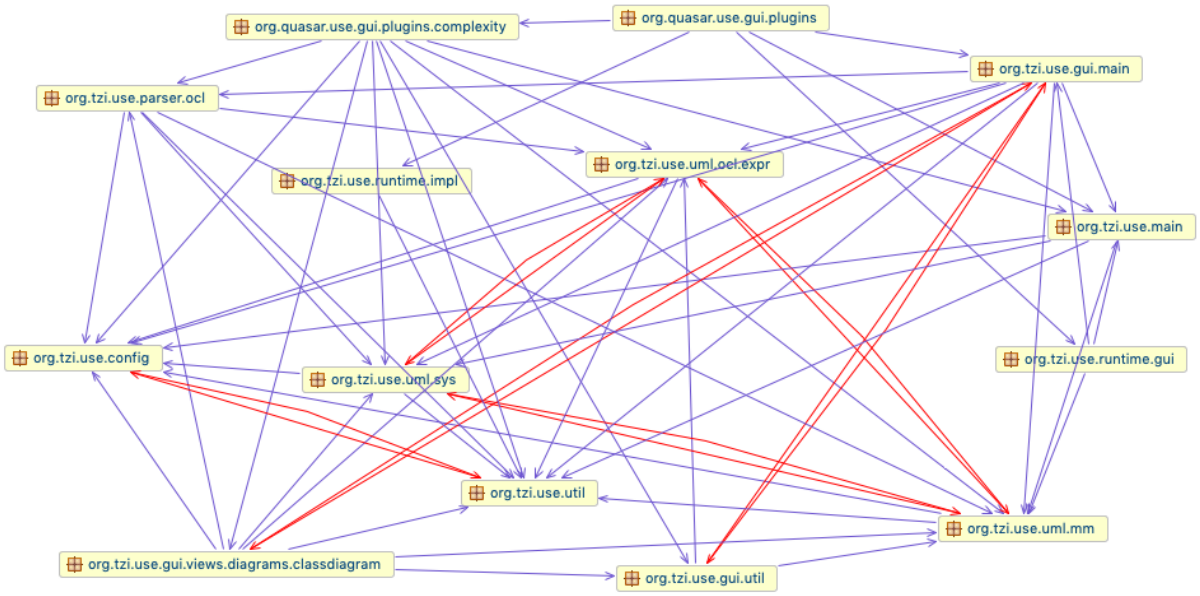

Figure 3.14: OCL Complexity Plugin: package diagram

(3) *OCLComplexityHelpDialog*: A dialog that displays a short description of each metric. This class extends the functionalities of *JDialog*, with no additional attributes.

(4) *ExpressionComplexityVisitor*: This is the expression visitor class, which implements the methods from the interface *ExpressionVisitor* (available in USE), calculating the complexity of the expression. The attributes of this class are shown in table 3.6.

Table 3.6: OCL Complexity Plugin: *ExpressionComplexityVisitor* class attributes

Attribute	**Description**
Stack<MOperation> *operationStack*	Stack that controls the navigation on nested operations.
IComplexityMetric *metric*	Processes the calculation of metrics.

(5) *EvalOCLDialog*: This class extends the functionalities of *JDialog*, defining the aspect and actions for entering and evaluating OCL expressions (see Figure 3.15), including the creation of text components, labels, and buttons (and their respective actions). The attributes of this class are shown in Table 3.7.

(6) *DNNode*: This class represents a node in the navigation tree. The attributes of this class are shown in table 3.8.

(7) *ComplexityMetricResult*: This class encapsulates the result of the different complexity metrics for an expression. The interface *IComplexityMetricResult* defines the methods of this class, and the attributes are shown in table 3.9.

(8) *ComplexityMetric*: This class calculates and stores the result of the different complexity metrics for an expression. The methods of this class are provided by the interface *IComplexityMetric*, and the attributes are shown in table 3.10.

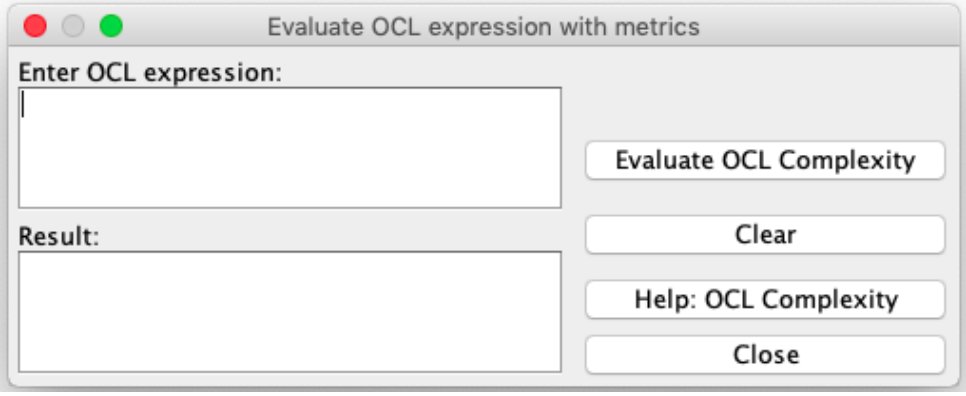

Figure 3.15: OCL Complexity Plugin: panel

Table 3.7: OCL Complexity Plugin: *EvalOCLDialog* class attributes

Attribute	**Description**
MSystem *fSystem*	Defines the system, including state and functionality
JTextArea *fTextIn*	Text area that captures the expressions introduced by the user
JTextArea fTextOut	Text area that displays the results of the evaluation of the expressions from *fTextIn*
JButton *btnEval*	Button that triggers the complexity evaluation
List<ClassDiagramView> *classDiagrams*	List of the available class diagrams
OCLComplexityHelpDialog *complexityHelpDialog*	Dialog showing an explanation of each complexity metric
ChangeListener *sessionChangeListener*	Listener that configures the session of the *fSystem*

Table 3.8: OCL Complexity Plugin: *DNNode* class attributes

Attribute	**Description**
String *className*	Name of the class represented by the node
int *weight*	Node's weight
DNNode *leftChild*	Node's left child
DNNode *rightChild*	Node's right child

Table 3.9: OCL Complexity Plugin: *ComplexityMetricResult* class attributes

Attribute	**Description**
int *nnr*, *nan*, *wno*, *nnc*, *nuca*, *nuco*, *wnn*, *dn*, *wco*	Value of each metric

Table 3.10: OCL Complexity Plugin: *ComplexityMetric* class attributes

Attribute	**Description**
Set<String> *navigatedRelationships*	Set of navigated relationships' names
Set<String> *referredAttributes*	Set of referenced attributes' names
int *wno*	Value for the metric WNO
Set<String> *navigatedClasses*	Set of the navigated classes' names
List<String> *utilityClasses*	List of available utility classes (we only considered *CalendarDate* and *Instant*)
Set<String> *nuca*	Set of referenced utility class attributes' names
Set<String> *nuco*	Set of referenced utility class operations' names
int *wnn*	Total value of the metric WNN
List<Integer> *wnnStack*	Stack of number of operations per depth (on the navigation tree)
DNNode *dnTree*	Navigation tree's root node
Map<Integer, Integer> *wcoOperations*	Map of the total of collection operations per depth (on the navigation tree)
int *wco*	Total value of the metric WCO

3.2.3 Implementation

Similar to what was described for the OCL Highlight Plugin, this one was also developed in Java 8 as an extension for USE. In this case, the decision to build it for this tool instead of another was purely based on the fact that we already had the knowledge and experience to build an extra functionality for it after the development of the first plugin. The plugins share a similar logic underneath, meaning that they both provide a new evaluation dialog that parses and evaluates expressions using a Visitor pattern. Instead of highlighting, this one calculates the complexity metrics of the given expression. The concrete implementation of this plugin is available in Github[8]. In this iteration of the plugin, we decided to exclude the metric WNM, as our models did not include Messages, and NPT since it is mainly used for pre-and post-conditions, which was not part of the questionnaires.

3.2.4 Metrics collection examples

As examples of plugin's behavior, Figure 3.16 shows the complexity values for Expression 3.1, whereas Figure 3.17 presents the values for Expression 3.2 (defined in Subsection 3.1.4).

[8]OCL Complexity Plugin repository - Available: https://quasarresearchgroup.github.io/useOCLcomplexity/ Accessed: 2020-11-28

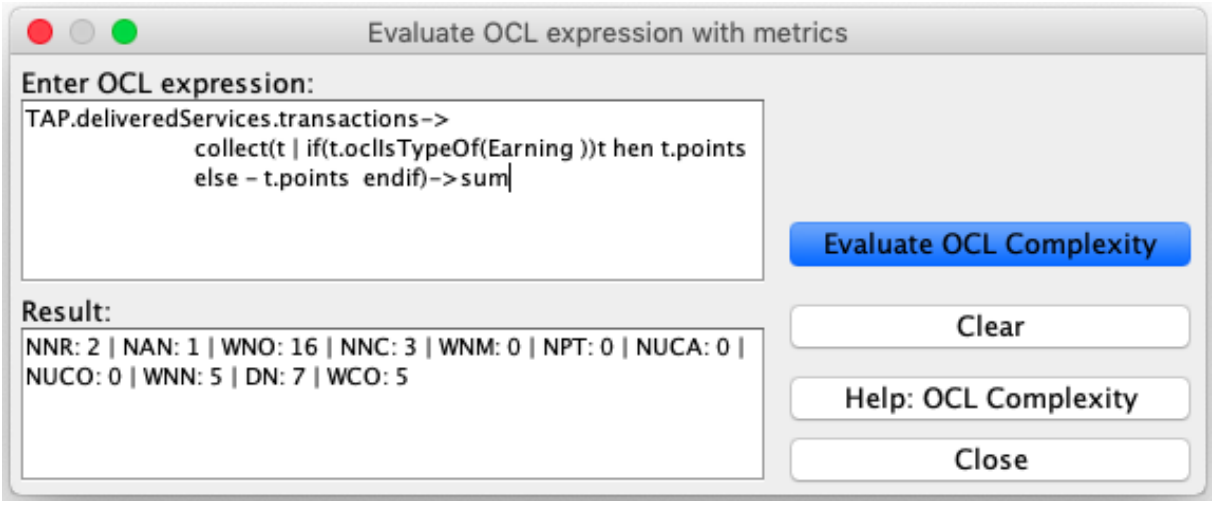

Figure 3.16: OCL Complexity Plugin: expression 1

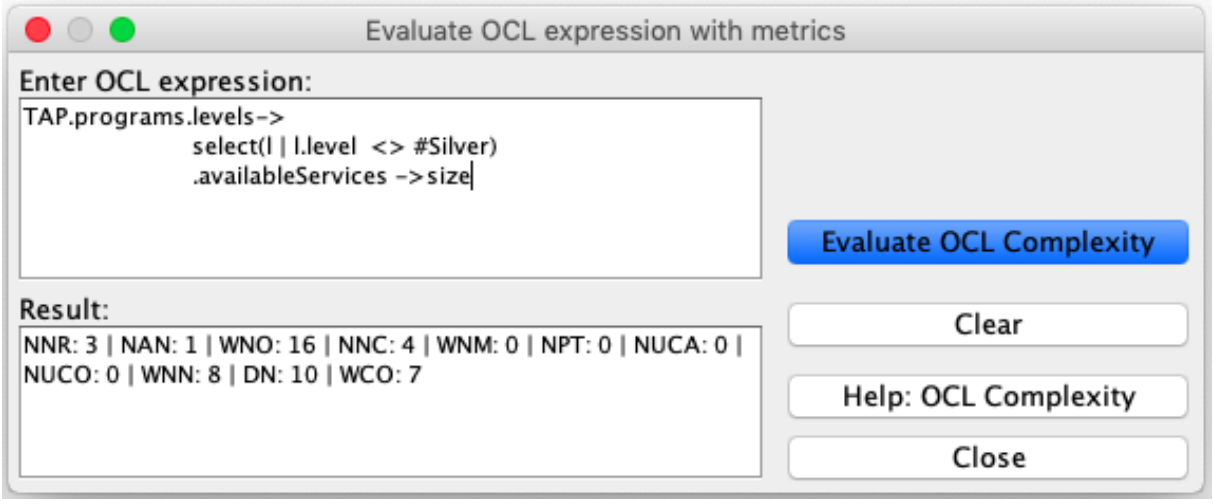

Figure 3.17: OCL Complexity Plugin: expression 2

Chapter 4

Experiment and Results

Contents

In this chapter, we present several studies conducted during our investigation. In Section 4.1, we explore the difficulty of learning OCL when compared to other subjects of SWEBOK, followed by an assessment of the variables that influence students' performance in OCL questionnaires, presented in Section 4.2. Section 4.3 concludes this chapter with the results of the OCL Highlight Plugin's influence in these same questionnaires

[This page has been intentionally left blank]

4.1 Experiment 1: the relative difficulty of learning OCL

In this section, we investigate the complexity of learning OCL compared to a set of other Software Engineering topics offered in two university courses that together span a full school year. The courses' syllabuses were organized according to the following areas of SWEBOK: Software Requirements, Software Design, Software Construction, Software Testing, Software Maintenance, Software Configuration Management, Software Engineering Management, Software Engineering Process, Software Engineering Tools and Methods, Software Quality. Although OCL was taught as part of the Software Design area, it was considered separately in this study. Other topics were kept in the Software Design area, namely the one of "Design Patterns".

After taking each area, learning was assessed through comprehensive questionnaires of closed true-false questions, through an e-learning system. Data was collected in two consecutive years, totaling around 150 students enrolled in four computer science-related graduations.[1]. Table 4.1 presents descriptive statistics for the grades obtained in OCL and the topics framed by the SWEBOK knowledge areas. The variability in the number of students per area is due to the fact that the tests for each subject took place throughout the semester, with some dropouts over time. Also, Software Requirements was taken only by students of 1 of the four graduations, and Software Construction was taken only by students of 3 of the four graduations, resulting in a smaller number of cases compared to the other topics. OCL was the topic where students obtained the worst grade, on average.

Table 4.1: Descriptive statistics for the learning grades per SWEBOK area

SWEBOK Area	N	Minimum	Maximum	Mean	Std. Deviation
OCL	156	0,00%	100,00%	50,92%	33,50%
Software Engineering Management	124	0,00%	93,33%	54,23%	22,06%
Software Quality	159	0,00%	94,44%	54,39%	21,03%
Software Requirements	59	5,00%	100,00%	58,12%	21,84%
Software Testing	141	0,00%	100,00%	60,49%	22,063%
Software Configuration Management	133	0,00%	100,00%	61,90%	19,04%
Software Engineering Process	132	0,00%	95,00%	65,15%	14,15%
Software Design	121	0,00%	100,00%	65,74%	22,70%
Software Construction	103	14,29%	100,00%	67,70%	21,93%

Our study is defined as quasi-experiment [9], as we did not extract a random sample from the population, and we had no control over the factors that could influence the subjects. For example, if some of the students had another exam the day before, that could have affected their performance. The sample is characterized as being repeated measures/within-groups, as each student was submitted to the tests from the different areas. The independent variable is the SWEBOK area, each factor being the grade the student had in the corresponding questionnaire (Software Requirements, Software Design, Software Construction, Software Testing, Software

[1] The dataset used in this section is made available at https://doi.org/10.5281/zenodo.4166133

Maintenance, Software Configuration Management, Software Engineering Management, Software Engineering Process, Software Engineering Tools and Methods, Software Quality), measured in a scale from 0 to 100. The dependent variable is the grade the students scored in the OCL questionnaire, measured in the same scale as the IV. The research question is defined as follows:

RQ1: There is no significant difference between the distribution of grades in OCL compared to other SWEBOK topics.

We started by performing a *One-Sample Kolmogorov-Smirnov* test, with *Lilliefors* significance correction. This allowed us to verify if the grades obtained in the different topics were normally distributed. The results did not allow us to assume a normal distribution for most areas. Thus, a *Non-parametric Related-Samples Wilcoxon Signed Rank* test between the grades obtained in OCL and the remaining areas (considered separately) was applied. Results are shown in Table 4.2 for a significance level of 0.05, which leads to the decision of retaining the null hypothesis, meaning no significant statistical difference between the mean rank of OCL, Software Requirements, Software Engineering Management, and Software Quality.

Table 4.2: *Related-Samples Wilcoxon Signed Rank* test results (questionnaire grades in OCL *versus* other SWEBOK topics)

SWEBOK Area	**Test Sign.**	**Decision**
Software Requirements	.719	Retain null hypot.
Software Design	.000	Reject null hypot.
Software Construction	.000	Reject null hypot.
Software Testing	.011	Reject null hypot.
Software Maintenance	.026	Reject null hypot.
Software Configuration Management	.000	Reject null hypot.
Software Engineering Management	.187	Retain null hypot.
Software Engineering Process	.000	Reject null hypot.
Software Engineering Tools and Methods	.006	Reject null hypot.
Software Quality	.487	Retain null hypot.

A *Friedman ANOVA* test confirmed that there is no significant difference between the distribution of the grades obtained in OCL and the SWEBOK topics of Software Requirements, Software Engineering Management, and Software Quality, $\chi^2 \mathrm{p}(3) = 4.86, \rho < 0.05$ (see Table 4.3). From this analysis, we conclude that, despite the claims on OCL being challenging to learn, there are other topics in SWEBOK that reveal similar difficulty. However, as stated in the objectives section, our investigation is focused on trying to soften the learning curve for OCL.

Table 4.3: *Related-Samples Friedman ANOVA* test (questionnaire grades in OCL *versus* other SWEBOK topics)

Hypothesis Test Summary

	Null Hypothesis	Test	Sig.	Decision
1	The distributions of OCL, Software Requirements, Software Quality and Software Engineering Management are the same.	Related-Samples Friedman's Two-Way Analysis of Variance by Ranks	,182	Retain the null hypothesis.

Asymptotic significances are displayed. The significance level is ,050.

Related-Samples Friedman's Two-Way Analysis of Variance by Ranks Summary

Total N	55
Test Statistic	4,859[a]
Degree Of Freedom	3
Asymptotic Sig.(2-sided test)	,182

a. Multiple comparisons are not performed because the overall test retained the null hypothesis of no differences.

4.2 Experiment 2: assessing OCL comprehension

The analysis presented in this section is based on the dataset[2] containing the data collected during five different school years (identified in Figure 4.1, with numbers from 1 to 5). During these years, more than six hundred students of three different undergraduate university courses were given OCL questionnaires (with a limited duration of 40 minutes) built as follows, to guarantee a true-false answer: the same UML class model with no more than 20 classes (to fit legibly in a computer screen) was made available to students in advance, and its semantics explained in detail throughout the semester. On the day of the questionnaire, a large set of objects and links were given to them right before they start answering the questions. Students proceeded to instantiate the class model in USE and could make free use of it while answering the questionnaire. Each of its 10 NL questions (extracted randomly from a large collection, to prevent cheating) could be answered by formulating a quantitative OCL expression upon the instantiated model. For instance, in *Royal and Loyal* the correct answer to *"how many services are provided by TAP?"* would be 42. This would require the students to develop an OCL expression similar to Expression 4.1. To be considered a correct answer, students had to insert the number 42 in the e-learning platform. Each year, a different UML model was given: *FootballCup* on years 1 and 3, *AirNova* on years 2 and 5, and *IULTrain* on year 4. Models were created with similar complexity (comparable number of classes, including utility classes and enumerations) to allow a fair assessment across the different school years. As an important note, the OCL Highlight plugin was introduced to the students on school year 5, not only for learning during the classes but also as a tool to assist them during the assignments.

```
1 TAP.programs.levels.availableServices->size
```

Listing 4.1: OCL expression 3

[2]The dataset used in this section is made available at https://doi.org/10.5281/zenodo.4287564

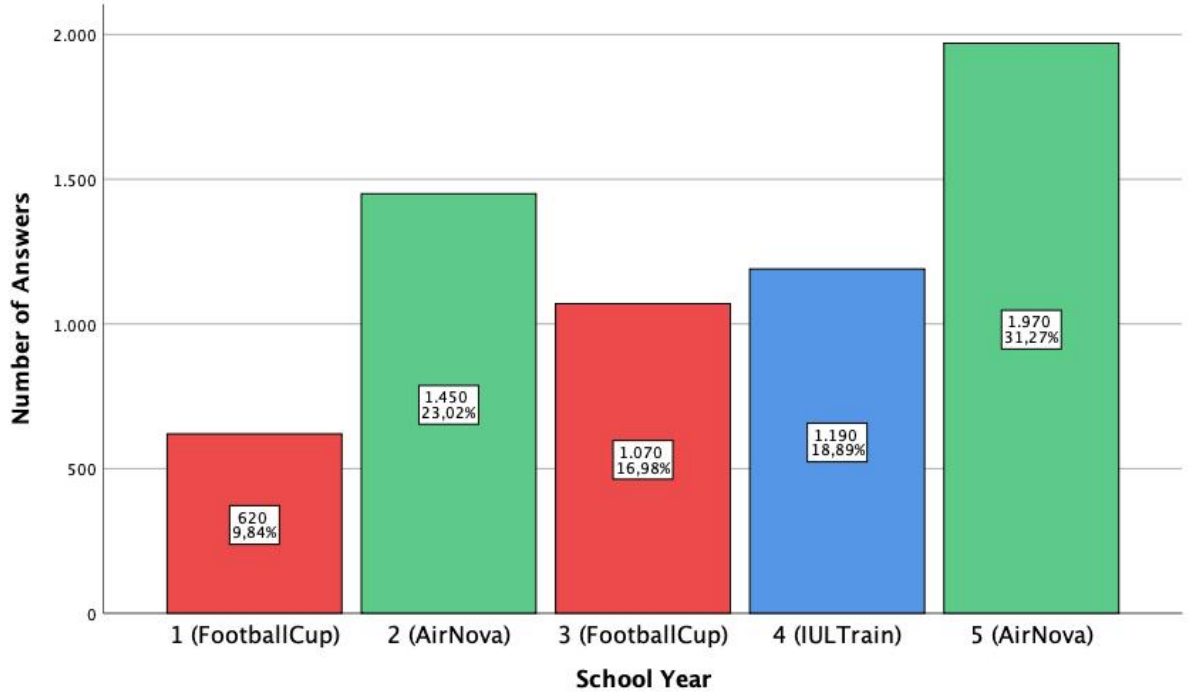

Figure 4.1: *Barchart* of total answers per school year

Table 4.4 shows a *Crosstabs* of the percentage of correct answers across the different years. Years 3 and 5 have the highest percentage of correct answers, with 50.1% and 54.5% correct answers, respectively. Year 2, despite using the same model as in year 5, showed the smallest value of correct answers, with just 36.3%. Years 1 and 3 studied the same model (FootballCup), but results do not show a significant difference (only 4% more correct answers on Year 3, whereas from year 2 to 5 the difference is 18.2%).

In the following subsections, we present the studies conducted to analyze the influence of different metrics on students' assessments across the years. Both experiments are defined as quasi-experiments, and the sample characterized as between subjects, as in each year the students were taught only one model. The independent variable in the first experiment is the OCL complexity, each factor being the value of the corresponding OCL metric (presented in Section 2.3), defined as scale. The independent variable in the second experiment is the natural language complexity, each factor being the value of the corresponding readability formula (presented in Sub Section 4.2.2, also defined as scale. The dependent variable is the success of the students in the OCL questionnaire, measured as nominal (0 for incorrect answers, 1 for correct ones). The research questions are defined as follows:

RQ1: There is no significant correlation between the complexity of OCL expressions, given by OCL complexity metrics, and the correctness of students' answers.

RQ2: There is no significant correlation between the complexity of natural language questions, given by readability formulas, and the correctness of students' answers.

Table 4.4: *Crosstabs* of answers' correctness per school year

Case Processing Summary

	Cases					
	Valid		Missing		Total	
	N	Percent	N	Percent	N	Percent
year * correct	6300	100,0%	0	0,0%	6300	100,0%

year * correct Crosstabulation

			correct		
			0	1	Total
year	1	Count	334	286	620
		% within year	53,9%	46,1%	100,0%
		% within correct	9,8%	9,8%	9,8%
		% of Total	5,3%	4,5%	9,8%
	2	Count	923	527	1450
		% within year	63,7%	36,3%	100,0%
		% within correct	27,2%	18,1%	23,0%
		% of Total	14,7%	8,4%	23,0%
	3	Count	534	536	1070
		% within year	49,9%	50,1%	100,0%
		% within correct	15,7%	18,5%	17,0%
		% of Total	8,5%	8,5%	17,0%
	4	Count	707	483	1190
		% within year	59,4%	40,6%	100,0%
		% within correct	20,8%	16,6%	18,9%
		% of Total	11,2%	7,7%	18,9%
	5	Count	897	1073	1970
		% within year	45,5%	54,5%	100,0%
		% within correct	26,4%	36,9%	31,3%
		% of Total	14,2%	17,0%	31,3%
Total		Count	3395	2905	6300
		% within year	53,9%	46,1%	100,0%
		% within correct	100,0%	100,0%	100,0%
		% of Total	53,9%	46,1%	100,0%

4.2.1 OCL complexity metrics

As a first attempt, we analyze OCL complexity metrics' influence on students' assessments' success rate. From the dataset described above, we extracted the unique OCL expressions, in a total of 140, calculating for each of them its complexities with the support of the OCL Complexity Plugin (Section 3.2). Since many metrics were proposed in the literature (Section 2.3), we executed a variable reduction, to avoid obtaining an overspecified model.

Using a *Kolmogorov-Smirnov* test (see Table 4.5), we examined the normality of the given metrics. For NNR, the results indicated that OCL metrics do not follow a normal distribution, $D(140) = .19$, $p < .05$ (other metrics showed similar results).

To evaluate if these metrics can explain the success of students' answers, and given the fact that the independent variables (metrics) do not follow a normal distribution, we applied a *Spearman's rho correlation coefficient* to evaluate their correlation and their effect on the dependent variable (students' success). Table 4.6 presents the result of this test. Our decision was to exclude metrics with strong correlation, as they can be used to explain the same results, keeping the metrics that are more dissimilar (identified in dark blue). The resulting metrics were NAN, WNO, NUCO, and WCO, as they reveal a greater influence on the dependent variable.

Table 4.5: *Kolmogorov–Smirnov* test on the normal distribution of OCL complexity metrics

One-Sample Kolmogorov-Smirnov Test

	N	Normal Parameters[a,b] Mean	Std. Deviation	Most Extreme Differences Absolute	Positive	Negative	Test Statistic	Asymp. Sig. (2-tailed)
NNR	140	2,40	1,516	,190	,190	-,140	,190	,000[c]
NAN	140	,74	,903	,300	,300	-,207	,300	,000[c]
WNO	140	23,79	30,527	,258	,258	-,218	,258	,000[c]
NNC	140	3,59	1,569	,153	,153	-,145	,153	,000[c]
NUCA	140	,16	,626	,528	,528	-,401	,528	,000[c]
NUCO	140	,04	,203	,541	,541	-,417	,541	,000[c]
WNN	140	10,33	8,504	,112	,112	-,112	,112	,000[c]
DN	140	8,91	6,037	,110	,110	-,073	,110	,000[c]
WCO	140	12,74	15,761	,240	,240	-,228	,240	,000[c]

a. Test distribution is Normal.

b. Calculated from data.

c. Lilliefors Significance Correction.

Table 4.6: *Spearman's rho correlation coefficient* of OCL complexity metrics

Spearman's rho correlation coefficient

	NNR	NAN	WNO	NNC	NUCA	NUCO	WNN	DN	WCO	Average Success
NNR	1,000	-,389**	-,332**	,937**	-0,099	-0,056	,832**	,830**	,493**	-0,011
NAN	-,389**	1,000	,668**	-,313**	,172*	0,132	-,357**	-,253**	-0,108	-0,130
WNO	-,332**	,668**	1,000	-,259**	,415**	,358**	-,285**	-,194*	-,208*	-,242**
NNC	,937**	-,313**	-,259**	1,000	-0,140	-0,093	,849**	,844**	,546**	-0,034
NUCA	-0,099	,172*	,415**	-0,140	1,000	,786**	-0,114	-0,026	-,196*	-0,013
NUCO	-0,056	0,132	,358**	-0,093	,786**	1,000	-0,138	-0,037	-,240**	0,106
WNN	,832**	-,357**	-,285**	,849**	-0,114	-0,138	1,000	,955**	,789**	-0,066
DN	,830**	-,253**	-,194*	,844**	-0,026	-0,037	,955**	1,000	,742**	-0,045
WCO	,493**	-0,108	-,208*	,546**	-,196*	-,240**	,789**	,742**	1,000	-,179*
Average Success	-0,011	-0,130	-,242**	-0,034	-0,013	0,106	-0,066	-0,045	-,179*	1,000

**. Correlation is significant at the 0.01 level (2-tailed).

*. Correlation is significant at the 0.05 level (2-tailed).

After reducing from nine to only four metrics, we performed a *Linear Regression* to assess the new set's capability to explain students' success, as *Spearman's rho* can evaluate relative monotonies whether the models are linear or not. The results of this test are shown in Table 4.7. A significant regression equation was found $F(4, 135) = 4.313$, $p < .01$, $R^2 = .113$, $R^2 adjusted = .087$. The resulting linear model revealed a poor explanatory power on the dependent variable (students' success), meaning that there is a monotony relationship, but it is not linear. The regression coefficient $B = .041$ indicated that an increase of one point in NAN corresponded, on average, to an increase in students' success of .041 points (analogous analysis to other metrics).

Given the weak results of the first variable reduction, we opted for a *Principal Component*

Table 4.7: *Linear Regression* using NAN, WNO, NUCO, and WCO to explain the success

Model Summary

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate	Change Statistics			
					R Square Change	F Change	df1	df2	Sig. F Change
1	,337[a]	,113	,087	,20126249	,113	4,313	4	135	,003

a. Predictors: (Constant), WCO, WNO, NAN, NUCO

ANOVA[a]

Model		Sum of Squares	df	Mean Square	F	Sig.
1	Regression	,699	4	,175	4,313	,003[b]
	Residual	5,468	135	,041		
	Total	6,167	139			

a. Dependent Variable: Average Success

b. Predictors: (Constant), WCO, WNO, NAN, NUCO

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.	Correlations			Collinearity Statistics	
		B	Std. Error	Beta			Zero-order	Partial	Part	Tolerance	VIF
1	(Constant)	,513	,027		18,656	,000					
	NAN	,041	,029	,178	1,452	,149	-,138	,124	,118	,439	2,280
	WNO	-,005	,001	-,756	-3,605	,000	-,103	-,296	-,292	,149	6,692
	NUCO	,717	,193	,692	3,717	,000	,083	,305	,301	,189	5,280
	WCO	,000	,001	-,029	-,335	,738	-,073	-,029	-,027	,887	1,127

a. Dependent Variable: Average Success

Analysis on the initial metrics that resulted in three components with a cumulative explanatory power of 90.986% (Table 4.8). The weight of each metric on the components can be seen in Table 4.9. Using a *Binary Logistic Regression*, we tried to predict students' success considering the three extracted components. Different input methods were tested, including forward and backward stepwise, but we only achieved poor results with R^2 around 3%.

Table 4.8: *Principal Component Analysis* for OCL metrics

Total Variance Explained

Component	Initial Eigenvalues			Extraction Sums of Squared Loadings			Rotation Sums of Squared Loadings		
	Total	% of Variance	Cumulative %	Total	% of Variance	Cumulative %	Total	% of Variance	Cumulative %
1	4,003	44,476	44,476	4,003	44,476	44,476	3,881	43,127	43,127
2	2,648	29,425	73,901	2,648	29,425	73,901	2,604	28,935	72,062
3	1,538	17,085	90,986	1,538	17,085	90,986	1,703	18,924	90,986
4	,466	5,180	96,166						
5	,163	1,808	97,974						
6	,090	,998	98,972						
7	,044	,489	99,462						
8	,029	,317	99,779						
9	,020	,221	100,000						

Extraction Method: Principal Component Analysis.

None of the tests described above allowed us to conclude that there is a significant correlation between OCL complexity metrics and students' success. These results imply that it is not possible to predict if students will answer correctly or not to a given question, considering the complexity of the OCL expression that provides with the appropriate answer.

Table 4.9: *Component Matrix* for the resulting *PCA* components

Component Matrix[a]

	Component		
	1	2	3
NNR	,817	-,038	-,491
NAN	,127	,295	,866
WNO	,097	,943	,189
NNC	,930	-,020	-,207
NUCA	-,096	,895	-,292
NUCO	,005	,930	-,226
WNN	,964	-,048	,074
DN	,983	,022	-,052
WCO	,735	,036	,569

Extraction Method: Principal Component Analysis.

a. 3 components extracted.

4.2.2 Readability metrics

In the previous section, we pursued without success to explain the correctness of students' answers considering the complexity of the solution, i.e., the OCL expression. In this section, we investigate if the justification can be found instead in the complexity of the problem, i.e., if the complexity of the question posed to the students has a significant influence on the responses' correctness.

To measure the complexity of the questions, we decided to apply several of the available readability metrics proposed by experts in Literature and Readability over the years [18], which are available in an online tool named Readable[3]. For example, for the question *"how many services are provided by TAP?"* each metric's values are as seen in Table 4.10. For each metric, the results should be interpreted as follows:

Definition 12. *Flesch-Kincaid Grade Level: Readability tests developed by the U.S. Navy that confers a U.S. grade level score. The higher the score, the higher level of education needed.*

Definition 13. *Flesch Reading Ease: Similar to the Flesch-Kincaid Grade Level, scoring the text between 1 and 100 (highest readability score).*

Definition 14. *Gunning Fog Index: Readability formula that estimates the degree of education needed to understand a given text, with a scale from 0 to 20. A value of 8 classifies the text as readable for someone in the eighth grade, and above 17 for a graduate level.*

Definition 15. *Coleman-Liau Index: Readability formula similar to Flesch-Kincaid Grade Level. It is most suitable for educational or medical texts.*

Definition 16. *SMOG Index: 'Simple Measure of Gobbledygook' (SMOG) Index measures the number of years of education a person needs to understand a text.*

[3]Readable - Available: https://readable.com/ Accessed: 2020-11-28

Table 4.10: Readability metrics for the given question

Metric	Value
Flesch-Kincaid Grade Level	7.4
Flesch Reading Ease	54.7
Gunning Fog Index	14.2
Coleman-Liau Index	6.0
SMOG Index	11.2
Automated Readability Index	2.9
FORCAST Grade Level	11.4
Powers Sumner Kearl Grade	6.5
Rix Readability	6
Lix Readability	36
New Dale-Chall Score	2.6
Spache Score	3.8
Linsear Write	85.7

Definition 17. *Automated Readability Index: Similar to the Coleman-Liau Index. The difference in this metric lies in the fact that it counts characters rather than syllables, and also counts sentences.*

Definition 18. *FORCAST Grade Level: Readability formula similar to Flesch-Kincaid Grade Level. Unlike other formulas, it is not designed to run on complete sentences, making it suitable for multiple-choice quizzes.*

Definition 19. *Powers Sumner Kearl Grade: Measures the U.S grade level in a similar way as Flesch-Kincaid Grade Level, using another formula.*

Definition 20. *Rix and Lix Readability: Two variations of the same readability formula based on letter counting. Results of the Rix formula can be interpreted using the grade level system. Lix's results have a corresponding value in this same scale (the higher the value, the higher the grade level).*

Definition 21. *New Dale-Chall Score: Measures a text against several words considered familiar, on a scale from 0 to 10. The lower the value, the easier it is to understand a text.*

Definition 22. *Spache Score: Similar to New Dall-Chall, but best suited for texts up to fourth-grade level. It uses a smaller familiar words' set.*

Definition 23. *Linsear Write: Readability formula that scores the text on a scale from 0 to 100. The higher the score, the more simplistic the text is.*

Similarly to the OCL complexity metrics, and considering that we were again facing a high number of variables (total of 13), we decided to assemble a variable reduction. A *Kolmogorov-Smirnov* test indicates that the majority of the proposed readability metrics do not follow a normal distribution, $D(140) = .089$, $p < .05$ (for Flesch–Kincaid Grade Level, other metrics

show similar results). After executing a *Spearman's rho correlation coefficient* to evaluate their correlation, and excluding the metrics with strong correlation, we ended up with a smaller set containing the Automated Readability Index, Forest Grade Level, New Dale-Chall, and Lensear Write. Applying a *Linear Regression* to assess the capability of the resulting metrics to explain the correctness of the answers, a significant regression equation was found $F(4, 135) = 6.311$, $p < .001$, with an R^2 of .158. The resulting linear model revealed once again a poor explanatory power on the success of the students.

In conclusion, our tests show that neither the complexity of the questions, given by Readability metrics, nor the complexity of the answers, given by OCL complexity metrics, can explain the students' success when answering the questionnaires.

4.3 Experiment 3: on the effect of using the OCL Highlight Plugin

In this last section of experiments, we focus on understanding whether the OCL Highlight Plugin was able to soften the learning curve when studying OCL and therefore produced the desired effect for which it was intended. Before requesting the students to use the plugin on their final questionnaires, it was important to have a stable version. The development of the plugin required several iterations, which allowed for the correction of bugs and the introduction of "nice-to-have" features, such as highlight colors' configuration. The feedback on the stability and usefulness of the plugin was first questioned to experts in the field. After making the necessary corrections, the students were able to experiment with the plugin during the semester. On the day of the exam, we collected their answers, and by comparing the results to the ones obtained in previous school years, we were able to discern the positive impact of the plugin.

4.3.1 Qualitative evaluation: experts

In this subsection, we present the focus group study we conducted to obtain feedback and experiences from UML/OCL experts (teachers) when using the OCL Highlight Plugin to teach OCL to undergraduate students at the university. This experience followed the available guidelines on how to plan and run focus groups [10]:

Defining the research problem: The study' objective was to provide insights into the utility of the developed plugin to ease the learning of OCL from the perspective of UML/OCL experts (teachers). Furthermore, we sought to obtain suggestions for improvement and the teachers' opinions on how students reacted to it.

Selecting the participants: Six teachers from the Department of Computer Science at ISCTE were invited by e-mail to participate in the focus group discussions. The selected invitees are Software Engineering experts, and they teach UML and OCL to bachelor students.

Planning and conducting the focus group session: We designed the focus group session to consist of two parts. In the first part, teachers were asked to individually observe their respective students during a week of classes, where students were submitted to their first contact with USE

and the plugin. At the end of the week, teachers were invited to attend a one-hour meeting to discuss their observations and fill out a previously prepared questionnaire about the usefulness of the plugin and the reaction of students to it.

Analysis: Results in Table 4.11 indicate that UML/OCL experts agreed on the plugin's usefulness and that it helps students understanding how OCL transverses a class diagram. This assessment showed that students could efficiently operate with the plugin, as we did not want to aggravate the difficulty in learning by introducing another element that they would need to manage. As a last remark, some teachers also referred that students thought that the highlighting was something obvious, which might indicate that it provides a natural integration with the tool.

4.3.2 Quantitative evaluation: students

As a last set of experiments, we explore the impact of using the plugin on students' answers. To consider it a success, we aspire to observe a significant increase in correct answers, when comparing to previous school years where the courses were given without the plugin. We are also interested in examining how the duration of the questionnaires was affected. For this analysis, we only considered the school years 2 and 5 for the reason that they studied the same model in an attempt to isolate the effect of using different models. Since the e-learning platform did not allow to collect the time the students took per question in year 2 (only the cumulative time of the test was available), we grouped the total of correct answers per test. Our experiments are defined as quasi-experiments, and the sample characterized as between subjects, as in year 2 the plugin was not used, but in year 5 it was used. The independent variable on both experiments is the usage of the plugin, defined as nominal (0 for no plugin, and 1 representing usage of the plugin). The dependent variable of the first experiment is the test duration (in seconds), defined as scale. The dependent variable of the second experiment is the success of the students in the OCL questionnaire, measured as nominal (0 for incorrect answers, 1 for correct ones). The research questions are defined as follows:

RQ1: There is no significant difference in the time needed to complete the questionnaires when using and not using the plugin.

RQ2: There is a significant improvement between the grades of the questionnaires where the plugin was not used compared to the grades when the plugin was used.

On the matter of test duration, a *Levene* test found that the assumption of homogeneity of variance between the groups (test duration and usage of the plugin) was violated, $p < .01$; therefore we carried out a *Two-Tailed Independent Samples T-Test* based on unequal variances. The test duration with plugin (M=2012.91, SD=398.554) may be considered similar when compared to not using the plugin (M=1942.59, SD=257.579), t=-1.968, $p = .05$ (Table 4.12). With a p-value at the limit of rejection, we assume from the calculated average values that the time consumption is very similar, despite being slightly superior when using the plugin.

A similar analysis was made, this time considering the correct answers. The equality of variances was met using a *Levene* test, $p > .05$. An *Independent Samples T-Test* for equal variances showed that the number of correct answers was significantly higher when using the plugin

(M=5.45, SD=2.807) than when not using it (M=3.63, SD=2.879), t=-5.836, $p < .05$ (Table 4.13). The difference observed for the number of correct answers can be credited to the plugin, as the homogeneity of variances is met.

The analyses presented above allow us to conclude that even though there was a slight increase in the amount of time needed by the students to complete the questionnaires, it was not significant. On the other hand, the grades (amount of correct answers) revealed a notable improvement. With these tests, we inferred that using the plugin was proved beneficial to students when learning OCL and did not introduce an additional effort that would reduce the speed of response for the questionnaires.

A *Chi-Square* test of independence confirmed this conclusion, showing that there is a significant association between using the plugin and the correctness of the answers, $\chi^2(1, N = 3420) = 110.177$, $p < .01$ (Table 4.14). We obtained a similar result when performing the same test against the whole dataset (years 1 to 5), $\chi^2(1, N = 6300) = 80.538$, $p < .01$.

As the last test, it is crucial to prove that there was no difference between the complexity of the questions and answers in both years, as that would jeopardize our experiments. Assuming equal variances between the complexity of the questions in years 2 and 5 (*Levene* test with $p > .01$), a *T-Test* shows that there was no significant difference between the groups, $p > .01$ (see Table 4.15). The same test applied to OCL complexity metrics requires a more in-depth analysis, as the results for each metric are dissimilar (see Table 4.16): assuming equal variances for NNR, NNC, and DN (*Levene* test, $p > .01$), there is no significant difference between both groups, $p > .01$; assuming equal variances for NAN (Levene's test, $p > .01$), the complexity of answers in year 2 is significantly higher when compared to year 5, $p < .01, t > 0$; assuming unequal variances for WNO, NUCA, and WNN (*Levene* test, $p < .01$), there is no significant difference between groups, $p > .01$; assuming unequal variances for WCO (*Levene* test, $p < .01$), the complexity of answers in year 2 is again significantly higher than in year 5, $p < .01, t > 0$.

Table 4.11: Qualitative validation of the OCL Highlight Plugin

Expert	**Q1: What is your opinion, as a UML/OCL expert, on the usefulness of this plugin? Considering that you have used the tool without the plugin, do you think it can help students when learning OCL?**	**Q2: How did the students in your class(es) react to the plugin? In particular, the assimilation of the visual metaphor was easy, that is the correspondence between the textual expression in OCL and the highlighted elements in the class diagram?**
Expert 1	An indispensable and missing plugin, so far! While learning OCL, it is critical to understand the relationship between the often complex OCL expressions and related modeling elements defined in a class diagram – often complex as well. OCL Highlight plugin delivers a simple, yet complete, graphical representation of those relationships, thus facilitating the understanding and learning of OCL expressions.	The students quickly understood the usefulness and ease of use of the plugin. The visual matching between the OCL expression and the related modeling elements in the class diagram was considered very intuitive. The graphical feedback also motivated the students to try out and understand increasingly complex OCL expressions.
Expert 2	I consider that the plugin is of great utility since it illustrates the navigation of the queries in OCL. It facilitates the understanding of the expressions, as well as the understanding of OCL language.	The reaction of the students to the plugin was something natural as if it was something obvious that the tool "painted the way". I took the opportunity of asking them if the path was not painted in the class diagram would be more difficult to understand the semantics of queries, and most students said yes.
Expert 3	The plugin is extremely important as students have some difficulties to understand OCL and, more importantly, to build OCL expressions. This visual insight helps to grasp how OCL operates over the model.	Yes, they have immediately realized which parts of the model were involved.
Expert 4	It is a very useful plugin since it helps the students understand what classes are being used in each OCL expression. It introduces traceability and shows a better insight to the students, making it easier for them to learn.	Yes, they understood quickly how OCL expressions worked.
Expert 5	This plugin is mostly a good facilitator and a really useful tool for the understanding of UML/OCL, mainly when we have class diagrams with a considerable size. Especially for students, it can ease and improve the process of learning OCL queries.	In class and with this plugin, students were able to follow the OCL expressions they were writing by immediately visualizing the result of those queries in terms of the used classes, associations and attributes that were highlighted. Students accepted the use of this tool really well, compared with the previous year, it not only provided a more engaging learning experience for the students but also, as a lecturer, it helped to demonstrate the process of querying.
Expert 6	I believe that, in fact, this plugin can facilitate the learning of OCL by the students. The selective visualization allows a greater concentration and effectiveness of the analysis of the pathway performed by the queries.	The students in my class seemed to assimilate well the connection between the path highlighted in the diagram and the path made by the OCL queries.

Table 4.12: *Independent Samples T-Test* between test duration and usage of the plugin

T-Test

Group Statistics

	withPlugin	N	Mean	Std. Deviation	Std. Error Mean
TestDuration	no	145	1942,59	257,579	21,692
	yes	197	2012,91	398,554	28,396

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
TestDuration	Equal variances assumed	31,714	,000	-1,838	336	,067	-70,320	38,261	-145,581	4,941
	Equal variances not assumed			-1,968	332,825	,050	-70,320	35,733	-140,611	-,028

Table 4.13: *Independent Samples T-Test* between the total of correct answers and usage of the plugin

T-Test

Group Statistics

	withPlugin	N	Mean	Std. Deviation	Std. Error Mean
correct	no	145	3,63	2,879	,239
	yes	197	5,45	2,807	,200

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
correct	Equal variances assumed	,656	,418	-5,836	340	,000	-1,812	,311	-2,423	-1,201
	Equal variances not assumed			-5,814	305,976	,000	-1,812	,312	-2,426	-1,199

Table 4.14: *Chi-Square* test for the association between answer's correctness and usage of the plugin

Case Processing Summary

	Cases					
	Valid		Missing		Total	
	N	Percent	N	Percent	N	Percent
correct * withPlugin	3420	100,0%	0	0,0%	3420	100,0%

correct * withPlugin Crosstabulation

Count

		withPlugin		
		no	yes	Total
correct	0	923	897	1820
	1	527	1073	1600
Total		1450	1970	3420

Chi-Square Tests

	Value	df	Asymptotic Significance (2-sided)	Exact Sig. (2-sided)	Exact Sig. (1-sided)
Pearson Chi-Square	110,177[a]	1	,000		
Continuity Correction[b]	109,450	1	,000		
Likelihood Ratio	111,118	1	,000		
Fisher's Exact Test				,000	,000
Linear-by-Linear Association	110,144	1	,000		
N of Valid Cases	3420				

a. 0 cells (0,0%) have expected count less than 5. The minimum expected count is 678,36.

b. Computed only for a 2x2 table

Table 4.15: *Independent Samples T-Test* between the readability metrics of years 2 and 5

Independent Samples Test

		Levene's Test for Equality of Variances				
		F	Sig.	t	df	Sig. (2-tailed)
fleschKincaidGradeLevel	Equal variances assumed	1,715	,195	-,014	64	,989
	Equal variances not assumed			-,014	63,973	,989
gunningFogIndex	Equal variances assumed	,847	,361	1,637	64	,107
	Equal variances not assumed			1,638	60,420	,107
colemanLiauIndex	Equal variances assumed	5,585	,021	,494	64	,623
	Equal variances not assumed			,529	57,203	,599
smogIndex	Equal variances assumed	,560	,457	,829	64	,410
	Equal variances not assumed			,820	57,417	,416
automatedReadibilityIndex	Equal variances assumed	,518	,474	1,517	64	,134
	Equal variances not assumed			1,500	57,417	,139
forecastGradeLevel	Equal variances assumed	,148	,702	-1,293	64	,201
	Equal variances not assumed			-1,288	59,380	,203
powersSumnerKearlScore	Equal variances assumed	,865	,356	1,468	64	,147
	Equal variances not assumed			1,474	61,159	,146
rix_score	Equal variances assumed	,020	,888	2,587	64	,012
	Equal variances not assumed			2,583	59,941	,012
fleschReadingEase	Equal variances assumed	3,800	,056	,870	64	,388
	Equal variances not assumed			,934	56,194	,354
spacheScore	Equal variances assumed	,446	,507	-,836	64	,406
	Equal variances not assumed			-,812	52,226	,420
newDaleChallScore	Equal variances assumed	,297	,588	-1,252	64	,215
	Equal variances not assumed			-1,231	55,756	,224
lix_score	Equal variances assumed	2,016	,160	1,575	64	,120
	Equal variances not assumed			1,597	62,886	,115
lensear_write	Equal variances assumed	,929	,339	-1,370	64	,175
	Equal variances not assumed			-1,353	57,129	,181

Table 4.16: *Independent Samples T-Test* between the OCL complexity metrics of years 2 and 5

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
NNR	Equal variances assumed	1,296	,259	,260	64	,795	,07176	,27569	-,47900	,62252
	Equal variances not assumed			,259	58,634	,797	,07176	,27757	-,48372	,62725
NAN	Equal variances assumed	4,201	,045	2,935	64	,005	,56011	,19082	,17892	,94131
	Equal variances not assumed			2,753	41,560	,009	,56011	,20349	,14933	,97089
WNO	Equal variances assumed	25,019	,000	2,888	64	,005	11,92824	4,13066	3,67630	20,18017
	Equal variances not assumed			2,654	36,301	,012	11,92824	4,49367	2,81729	21,03919
NNC	Equal variances assumed	2,938	,091	,917	64	,363	,23392	,25514	-,27578	,74362
	Equal variances not assumed			,894	53,229	,376	,23392	,26177	-,29106	,75891
NUCA	Equal variances assumed	13,818	,000	-1,721	64	,090	-,17428	,10128	-,37661	,02805
	Equal variances not assumed			-1,847	56,549	,070	-,17428	,09438	-,36331	,01475
WNN	Equal variances assumed	9,319	,003	1,569	64	,122	1,65517	1,05521	-,45286	3,76320
	Equal variances not assumed			1,481	43,426	,146	1,65517	1,11797	-,59879	3,90914
DN	Equal variances assumed	,983	,325	,300	64	,766	,24977	,83386	-1,41606	1,91559
	Equal variances not assumed			,295	56,582	,769	,24977	,84589	-1,44437	1,94390
WCO	Equal variances assumed	10,922	,002	4,416	64	,000	4,23206	,95826	2,31771	6,14641
	Equal variances not assumed			4,146	41,878	,000	4,23206	1,02074	2,17195	6,29217

[This page has been intentionally left blank]

Chapter 5.

Conclusions and Future Work

Contents

In this chapter, we present the outcomes of our experiments, including open issues, limitations, as well as future work directions.

[This page has been intentionally left blank]

5.1 Conclusion

This dissertation was triggered by claims made in the literature stating that OCL is difficult to learn. We collected data systematically during two consecutive school years, with the help of an e-learning platform, on the outcome of the learning process of a collection of SWEBOK topics, from a large set of undergraduate students, through extensive closed questions questionnaires. OCL appeared in the group of challenging topics, but it did not emerge as the most difficult one.

We also performed inferential statistics to identify the cause of students' success/failure when learning OCL, based on similar questionnaires to the ones mentioned above, where students had to translate NL clauses to OCL expressions to answer the questions appropriately. The explanatory variables that we explored were: (i) the linguistic complexity of the questions in NL (problem domain), ranked by different readability formulas calculated using Readable, and (ii) the complexity of the OCL clauses (solution domain), measured by a set of metrics proposed in the literature calculated with the help of OCL Complexity Plugin, required to answer those questions. Neither individually nor in conjunction were the aforementioned variables able to provide an acceptable explanatory power, as denoted by low R-squared values in the experiments.

In addition to shedding light on the factors that influence the OCL learning process, we proposed a tool-based learning feature, dubbed "OCL Highlight", reified as a plugin on the USE tool, which highlights how an OCL clause traverses a UML Class Diagram. We validated this feature by combining an action-research observation period on more than six hundred students in lab sessions, with semi-structured interviews whose conclusions were consolidated by a focus group of experts. We were able to compare students' success between years where the plugin was used and the years where it was not available. A full consensus was reached that the highlighting feature had a positive effect on the OCL learning process, improving the overall grades without a significant increase in the time needed to complete the OCL questionnaires.

5.2 Future work

With extensive use of the plugin by students in evaluation moments, we were able to corroborate its utility and confirm that it was beneficial when learning OCL. Nonetheless, we would like to extend this validation with more tests using different models with similar complexity. Future experiments should also control for the origin of students since they were following three distinct graduations ("Computer Science", "Telecommunications and Computer Engineering" and "Computer Science and Business Management"). Although the courses' syllabus has remained constant during the observation period, variations in students' background might have occurred.

Additionally, some improvements to the plugins can be considered. First of all, they have many internal USE dependencies (as seen in the package diagrams). In a future implementation, it would be recommended to use a Facade pattern[1] to reduce these dependencies and isolate

[1] Facade pattern - Structural software-design pattern that hides the underlying system complexities by providing an interface to the client

changes resulting from USE itself. Second, for the OCL Highlight Plugin we only considered the utility classes presented in our models. Meaning that if a new model introduced a different one, it would not be acknowledged, and therefore, not highlighted in the class diagram. Therefore, an upcoming iteration should include an automatic mechanism to detect new utility classes. And last, for the OCL Complexity Plugin the implementation of WNM and NPT metrics was not included, since they were not relevant for our experiment. A future iteration could consider this implementation.

Bibliography

[1] C. Atkinson and T. Kühne. "Model-driven development: A metamodeling foundation." In: *IEEE Software* 20.5 (Sept. 2003), pp. 36–41. DOI: 10.1109/MS.2003.1231149.

[2] L. C. Briand, Y. Labiche, H. D. Yan, and M. Di Penta. "A controlled experiment on the impact of the object constraint language in UML-based maintenance." In: *Proceedings of the IEEE International Conference on Software Maintenance, ICSM.* 2004, pp. 380–389. DOI: 10.1109/ICSM.2004.1357823.

[3] L. C. Briand, Y. Labiche, M. Di Penta, and H. Yan-Bondoc. "An experimental investigation of formality in UML-based development." In: *IEEE Transactions on Software Engineering* (2005). DOI: 10.1109/TSE.2005.105.

[4] M. Gogolla. "Benefits and Problems of Formal Methods." In: *Proceedings of the Reliable Software Technologies conference (Ada-Europe 2004).* Ed. by A. Llamosí and A. Strohmeier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–15. DOI: 10.1007/978-3-540-24841-5_1.

[5] M. Gogolla, F. Büttner, and M. Richters. "USE: A UML-based specification environment for validating UML and OCL." In: *Science of Computer Programming* 69.1-3 (2007), pp. 27–34. DOI: 10.1016/j.scico.2007.01.013.

[6] M. Gogolla, L. Hamann, and F. Hilken. "On Static and Dynamic Analysis of UML and OCL Transformation Models." In: *Proceedings of the Workshop on Analysis of Model Transformations co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), Valencia, Spain, September 29, 2014.* Ed. by J. Dingel, J. de Lara, L. Lucio, and H. Vangheluwe. Vol. 1277. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 24–33.

[7] M. Gogolla, L. Hamann, F. Hilken, and M. Sedlmeier. "Checking UML and OCL Model Consistency: An Experience Report on a Middle-Sized Case Study." In: *Proceedings of the 9th International Conference, TAP 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 22-24, 2015. Proceedings.* Ed. by J. C. Blanchette and N. Kosmatov. Vol. 9154. Lecture Notes in Computer Science. Springer, 2015, pp. 129–136. DOI: 10.1007/978-3-319-21215-9_8.

[8] Y.-G. Guéhéneuc. "TAUPE: Towards Understanding program comprehension." In: *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research (CASCON'06).* New York, New York, USA: ACM Press, 2006, p. 1. DOI: 10.1145/1188966.1188968.

[9] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. "Reporting Experiments in Software Engineering." In: *Guide to advanced empirical software engineering*. Springer, 2008, pp. 201–228. DOI: 10.1007/978-1-84800-044-5_8.

[10] J. Kontio, L. Lehtola, and J. Bragge. "Using the focus group method in software engineering: Obtaining practitioner and user experiences." In: *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE 2004)*. IEEE, 2004, pp. 271–280. DOI: 10.1109/ISESE.2004.1334914.

[11] R. Mehta and R. Zhu. "Blue or red? Exploring the effect of color on cognitive task performances." In: *Science* 323.5918 (Feb. 2009), pp. 1226–1229. DOI: 10.1126/science.1169144. arXiv: arXiv:1106.5958.

[12] G. K. Rambally. "The influence of color on program readability and comprehensibility." In: *Proceedings of the 17th SIGCSE technical symposium on Computer science education (SIGCSE'86)*. Vol. 18. New York, New York, USA: ACM Press, 1986, pp. 173–181. DOI: 10.1145/5600.5702.

[13] L Reynoso, M Genero, M Piattini, and E Manso. "Assessing the impact of coupling on the understandability and modifiability of OCL expressions within UML/OCL combined models." In: *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*. 2005, 10 pp.–14. DOI: 10.1109/METRICS.2005.12.

[14] L. Reynoso, M. Genero, and M. Piattini. "Measuring Ocl Expressions: an Approach Based on Cognitive Techniques." In: *Metrics for Software Conceptual Models*. Imperial College Press, Distributed by World Scientific Publishing Co., Jan. 2005, pp. 161–206. DOI: 10.1142/9781860946066_0005.

[15] L. Reynoso, E. Manso, M. Genero, and M. Piattini. "Assessing the influence of import-coupling on OCL expression maintainability: A cognitive theory-based perspective." In: *Information Sciences* 180.20 (2010), pp. 3837–3862. DOI: https://doi.org/10.1016/j.ins.2010.06.028.

[16] A. Sarkar. "The impact of syntax colouring on program comprehension." In: *Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2015)*. Ed. by M. Coles and G. Ollis. Bournemouth, UK: Psychology of Programming Interest Group, July 2015, pp. 49–58.

[17] B. Selic. "The pragmatics of model-driven development." In: *IEEE Software* 20.5 (Sept. 2003), pp. 19–25. DOI: 10.1109/MS.2003.1231146.

[18] S. Stajner, R. Evans, C. Orasan, and R. Mitkov. "What can readability measures really tell us about text complexity?" In: *Proceedings of Workshop on natural language processing for improving textual accessibility* (Jan. 2012), pp. 14–22.

[19] A. Toval, V. Requena, and J. L. Fernández. "Emerging OCL tools." In: *Software and Systems Modeling* 2.4 (Dec. 2003), pp. 248–261. DOI: 10.1007/s10270-003-0031-0.

[20] J. Warmer and A. Kleppe. *The Object Constraint Language Second Edition, Getting Your Models Ready for MDA, by Jos Warmer and Anneke Kleppe*. Vol. 2. Addison-Wesley, 2003, p. 139. DOI: 10.5381/jot.2003.2.6.r1.

[21] S. Yusuf, H. Kagdi, and J. I. Maletic. "Assessing the comprehension of UML class diagrams via eye tracking." In: *Proceedings of the IEEE International Conference on Program Comprehension (ICPC'07)*. 2007. DOI: 10.1109/ICPC.2007.10.

[22] A. Zamansky, G. Rodriguez-Navas, M. Adams, and M. Spichkova. "Formal Methods in Collaborative Projects." In: *Proceedings of the Evaluation of Novel Approaches to Software Engineering conference (ENASE'2016)*. 2016, pp. 396–402. DOI: 10.5220/0005937403960402.

[This page has been intentionally left blank]