Making QVTo transformations more understandable

Yu, T.

Award date:
2019

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Software Engineering and Technology

Making QVTo transformations more understandable

Master's Thesis

Tianshi Yu

Supervisor: prof. dr. M.G.J. van den Brand, dr. ir. J.G.M. Mengerink

Assessment Committee:

prof. dr. M.G.J. van den Brand
dr. ir. R.R.H Schiffelers
prof. dr. J.P.M.voeten

version 2.0

Eindhoven, December 2018

Abstract

Based on analysis of the QVT Operational Mappings (QVTo), which is a rich and imperative model transformation language, we can improve the quality of the QVTo transformations. To fulfill this task, we conduct a study which consists of two research questions. The first one is how does the QVTo transformations quality evolve over time? To address this question, we apply a set of metrics on multiple revisions of the QVTo transformation to analyze the evolution of QVTo transformation's quality. We observe that all the metrics adopted in our experiments indicate the QVTo transformation becomes less understandable over time except for the metric which is used to measure how imperative/declarative the QVTo transformations are. This contradictory suggests that the existing metric for this measurement is not reliable. Hence, we design a set of new metrics which can be reliable indicators for how imperative/declarative the QVTo transformations are. Using these new metrics to analyze the evolution, we find that the QVTo transformation becomes less declarative and less understandable. Therefore, our second research question arises: can we rewrite the QVTo transformations to make them more declarative, and how does the rewriting of QVTo transformations affect their quality? To investigate this question, we provide a rewriting pattern and apply it to the QVTo transformations and then measure the quality of the QVTo transformations before and after our rewriting with the metrics presented in the research of our first question. Based on the experiment results, we conclude that the rewriting makes QVTo transformations more declarative. Consequently, the transformations become more understandable and more analyzable.

Preface

Acknowledgments
I would like to thank Mark for introducing me to the field of model transformation, and his amazing guidance and fast feedback during the project. I also want to thank Josh for his inspiring advice for the project and help with the data collecting tool, which made it much easier to process the data. Big thanks to Ramon and Jeroen too for their feedback on the thesis. I want to thank Nan, Kousar, and Arjen who are great colleges. It was an unforgettable experience to work with them in ASML. I want to especially thank Mark for his patience when I got into trouble with the project during the bad days.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation and Problem statement

Quality is important for software, poor quality software usually consumes more cost and effort, serious defects may even cause fatal damages at a later stage of the software's life-cycle. Hence, it is important to assess the quality of software. There is a considerable amount of literature on the field of software quality measurements[2][8][9].

In recent years, there has been an increasing interest in model transformations in both science and engineering context [10][11][12]. So several studies investigating the quality of model transformation languages have been carried out[11][13], especially for QVT Operational Mappings (QVTo), which is a rich and imperative model transformation language. However, these studies focus on the quality of a single revision of the QVTo transformations. What is not yet clear is how the quality changes when the QVTo transformations evolve over time.

Therefore, the major aim of our study is to investigate the evolution of the quality in QVTo transformations. This investigation is formalized into the following research questions:

> ***Research Question 1:*** *How does the quality of QVTo transformations evolve over time?*

To address this question, firstly we adopt the metrics presented in Gerpheide's work [11] and apply them on a series of revisions of QVTo transformations to analyze the evolution of QVTo transformations' quality. Based on the results, we observe that the metric, which is used to measure how declarative the QVTo transformations are, suggests the QVTo transformations' become more understandable over time, while other metrics suggest the opposite. A possible reason for this contradictory is that the existing metric for this measurement is not reliable as it is based on an infrequent used construct (*forEach*).

Hence, we design a set of new metrics to measure how declarative the QVTo transformations are, and investigate the quality evolution of the QVTo transformations with the proposed metrics. The new metrics indicate that the QVTo transformations become less understandable and less declarative over time, which raises the second research question of our study:

> ***Research Question 2:*** *Can we rewrite the QVTo transformations to make them more declarative, and how does the rewriting of QVTo transformations affect their quality?*

To investigate this question, we provide a proof of concept, by rewriting the latest revision of a QVTo transformation. We rewrite the transformation by reducing the most frequently used imperative parts, and then measure the quality of the QVTo transformations before and after our

rewriting with the metrics presented in the research of our first question.

Additionally, when the QVTo transformations are rewritten in more declarative style, they can be more easily analyzed with some analyze tool, such as Alloy Analyzer [14] whose core is Alloy [15]. Alloy is a declarative and powerful language to describe structural properties, and can be used for expressing complex constraints. Besides, Alloy is useful to analyze model transformations to reason their correctness [16]. Therefore, making the QVTo transformations more declarative is helpful for analysis.

1.2 Methodology

1.2.1 Industry Context

This research is conducted within ASML, which provides complex lithograhpy systems for the semiconductor industry. To design these systems, a model-based developing platform named Control Architecture Reference Model (CARM) is provided, which relies on a set of Domain Specific Languages (DSL) to describe the control logic and the execution platform of lithoscanners. This developed multi-disciplinary integrated development environment (IDE) can validate the designs in early stages and construct the software components efficiently [12]. This IDE adopts the OMG [17] standard for model definition and model transformation. In terms of implementation, Eclipse modeling Framework(EMF) [18], Eclipse QVTo [19] are used. Our research focuses on these Eclipse QVTo based projects.

In this thesis, we evaluate a specific transformation named *pgapp2dsgraph*, which is a typical example of QVTo transformations in ASML. The function of this transformation is to convert an instance conforming to model *pgapp* into one conforming to model *dsgraph*. The latest revision of the QVTo transformation contains 635 lines of code, a transformation declaration, an entry point for execution, 26 *Mapping*s and 8 *Helper*s. Besides, there are other relevant files such as library files in the same folder with file *pgapp2dsgraph* to complete the function. We also refer this QVTo transformation as "our QVTo transformation" or "the specific transformation" later in this thesis.

1.2.2 Approach

The first research question is to investigate the quality evolution of the QVTo transformation. The following steps are performed to accomplish this task:

i Choose a number of metrics from Gerpheide's quality model;

ii Choose a number of revisions of the QVTo transformation from the repository;

iii Apply the selected metrics on these chosen revisions. As can be seen in Figure 1.1, Revision 1 evolves into Revision N over time, hence, a series of values can be extracted and analyzed to draw a conclusion;

iv Evaluate the results and verify whether the QVTo transformation become more imperative over time;

v Design new metrics to indicate how imperative/declarative a QVTo transformation is;

vi Repeat step ii to step iv with our proposed metrics;

vii Analyze the results to answer the first research question.

Moreover, the following procedure is undertaken to investigate the second research question

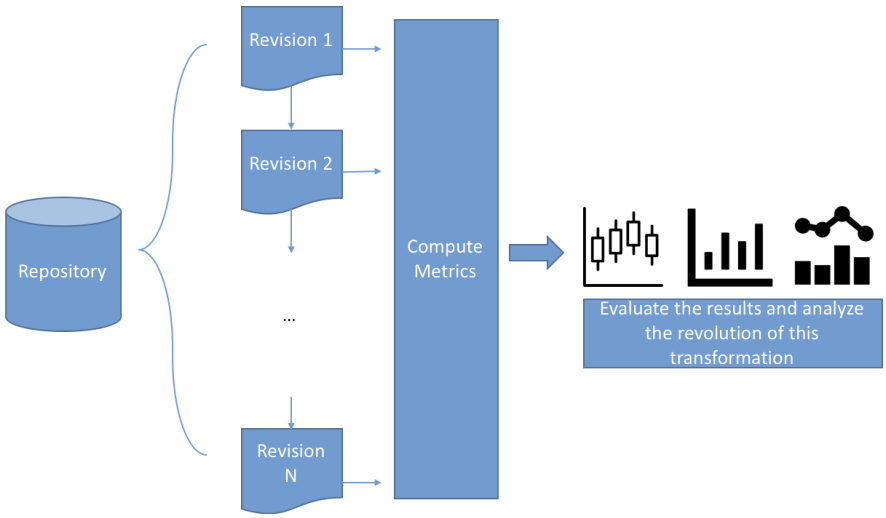


Figure 1.1: Exploring and analyze multiple revisions of the QVTo transformation

i Rewrite the latest revision of our QVTo transformation based on the proposed rewriting pattern to reduce the portion of imperative parts so that the transformation become more declarative;

ii Verify the rewritten "correctness", which means in our case the functionality is not affected by this rewriting. This verification can be done by comparing output models (test results) with the same input models (test cases), before and after the rewriting. Figure 1.2 presents how this step is done;

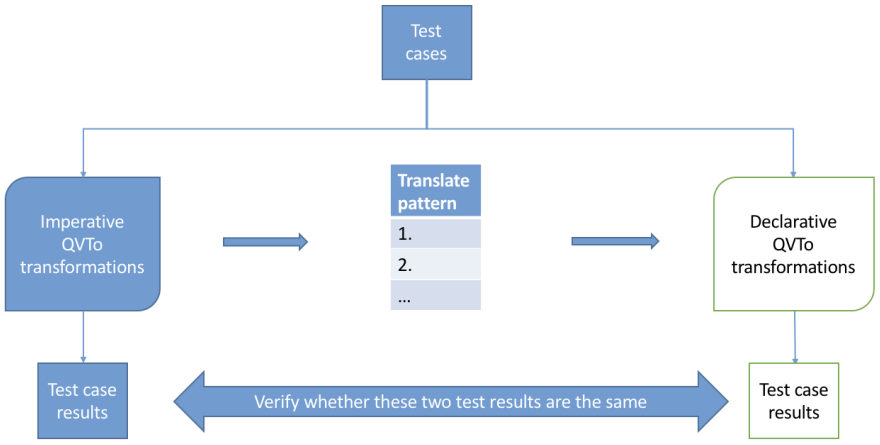


Figure 1.2: Verify the imperative/declarative rewriting with test cases

iii Apply the metrics proposed in the research of the first question on both the original transformation and the rewritten transformation;

iv Choose another two revisions of the QVTo transformation, and perform step iii on them;

v Analyze the results to answer our second research question.

1.3 Organization of thesis

The overall structure of this thesis is as follows: Firstly, Chapter 2 introduces the preliminary material of our research, including a brief description of imperative and declarative programming, an overview of QVTo, an introduction of quality and quality measurements, and a description of Lehman's Law; Chapter 3 introduces related work of software metrics, imperative-declarative rewriting and quality evolution of software systems; Chapter 4 shows the approach to investigate the first research question. Besides, in this chapter we present and analyze the results to answer the first research question; Chapter 5 describes the methodology and the experiments to explore the second research question; Finally, in Chapter 6, we conclude our work and draw the important conclusions. Besides, we mentioned possible future work in Chapter 6.

Chapter 2

Preliminaries

This chapter describes the basic concepts and notions which are discussed in this thesis. As our research is related to rewriting imperative QVTo transformations to declarative ones, Section 2.1 introduces basic concepts of imperative and declarative programming paradigms and Section 2.2 gives a brief introduction of QVTo language. Then, an overview of software quality and quality measurements is presented in Section 2.3, and Lehman's Law are described in Section 2.4

2.1 Imperative and declarative programming

Programming languages can be classified into different categories according to programming paradigms, imperative programming and declarative programming are two typical paradigms. Imperative programming languages are designed with considering the computer architecture, so variables are used to model the memory cells and assignment statements can modify the variables like the piping operations [20]. It should be noted that the memory cells are re-usable, so that the assignments could affect the state of the machine. For example, the assignment statement for $x := x + 1$ should be $x_{t+1} = x_t + 1$ in mathematical view, where x_t represents the value of x at time t. The main features of imperative programming paradigms are:

1. Imperative programming describes how the computation is done with statement sequences which change the states of the memory of computer;
2. imperative programming usually has side-effects (which means non-local variables are modified), such as input/output expressions.

On the other hand, declarative programming, which includes functional, logical and constraint programming, are built based on logics which has a soundness and preferable completeness theorem [21]. Declarative programming generally has the following features:

1. Declarative programming describes what the computation accomplishes with specifying a set of rules or equations to solve without expressing how to solve them explicitly; The computation is about evaluating these rules or equations, where the order of evaluating is usually not important;
2. Declarative programming typically has no side-effects.

Both imperative and declarative programming have their advantages. Declarative programming is suitable for formal analysis because of its mathematical nature, which means the programs behavior is easier to predict and understand, while imperative programming implementations usually has better performance. Hence, most programming languages are not purely imperative, neither purely declarative. Some typical imperative languages, like C, Python, Java also have

functional features, and declarative languages, such as SQL, support imperative style programming as well.

Following is an example which shows the differences between these two program paradigms: The first piece of code generates the nth Fibonacci number with imperative programming and the second one accomplishes the same task with declarative programming.

```
int fib = 0;
for(int i = 1; i<=n; i++)
{
  if(i<3){
    fib = 1;
    pre_fib = 1;
  }else{
    fib += pre_fib;
    pre_fib = fib;
  }
}
return fib;
```

Listing 2.1(a) Generates Fibonacci numbers with imperative programming language

```
fib :: Int->Int{}
fib n
    | n < 2 = 1
    | otherwise = fib(n-1)+fib(n-2)
```

Listing 2.1(b) Generates Fibonacci numbers with declarative programming language

In this examples, a Fibonacci numbers generator is written in imperative language C and declarative language Haskell. As can be seen, the C version 2.1(a) program defines the generator by describing the instructions which are executed step by step. The Haskell version 2.1(b), in contrast, does not give any instructions but specifies the rules should be satisfied, which are:

1. The first two numbers are 1;
2. Since the third number, every number is the sum of its two preceding ones.

2.2 Overview of QVTo

In the field of MDA(Model-driven architecture), the model transformation is an important technique which converts source models into target models. QVT (Query/View/Transformation) defined by OMG (Object Management Group) [4] is a set of model transformation languages that operate on models which conform to MOF 2.0 meta-model(Meta Object Facility). QVT is consist of three languages: QVT Operational Mapping Language (QVTo), QVT core (QVTc) and QVT relation language (QVTr). The character which distinguishes three QVT languages is that QVTo is imperative while the other two are declarative. As our research only focus on QVTo, the major features and concepts are presented in the following section:

1. **Transformation framework**: consists of a transformation declaration, an entry point for execution and the model type definition. The transformation declaration indicates the source and target models, which are the input and output of the transformation, the main() function is the actual entry point for execution and model type definition is a reference to the model type used in the transformation; besides, when- and where- clauses extended the transformation with pre- and post- conditions.
2. **Imperative operations**: include mappings, helpers, queries and constructors, which are operators to access or modify the elements;

- **Mappings**: consist of a mandatory populate section, an optional init section and an optional end section, are the core of transformations. They specify how the the elements of input model are transformed into elements of output model.
- **Helpers and Queries**: helpers perform computation on source elements and return the result while queries are helpers without any side-effects.
- **Constructors**: define how to create an instance of a class and populate the properties of the instance. They are called with the keyword new.

3. **Resolving**: Trace resolution provides a method to extract a source or target object of the last execution from the trace record. With the method, a source element doesn't need to be transformed again if a transformation has already been performed. There are four related functions, **resolve** returns a set of results from the last mapping of the source element; **resolveone** however returns only one target element (the last one if multiple results are available). **resolveIn** and **resolveoneIn** are similar with the first two functions while they specify the mapping's name;
4. **Intermediate data**: intermediate classes and properties can be defined in a transformation for the purpose of computation, hence they are not returned as the final results.
5. **Inherits and merges**: inheritance for mappings is possible with inherits and merges, the keyword inherits indicates the inherited mapping is executed between the the *init* and *population* section while merges indicates the merged mapping is executed after *end* section. Meanwhile, polymorphism for mappings is specified with keyword disjuncts. Disjuncts means only the first mapping which matches the source object type is performed (if any preconditions exists, they should also be satisfied).
6. **Basic operators**: Table 2.1 shows the basic operators that manipulate the meta data, which are similar to other languages; Similarly, forEach, while Loop, if/else statements and switch/case statements are constructors for control flow. Additionally, log() and assert() are useful functions while debugging.

Table 2.1: Basic operators

Operator	Description
:=	assignments
=	equals
<>	not equals
.	dot accessor
->	arrow accessor

Following is a simple example, Figure 2.1 shows two metamodels *BOOK* and *PUB* representing books and publications, which are the input and output model respectively. Listing 2.2 describes a transformation *Book2Publication* that coverts a *BOOK* object to a *PUB* object, where a entry point *main()* and a mapping rule *book_to_publication* are declared.

```
//defining model BOOK
metamodel BOOK {

  class Book {title: String; com{}poses chapters: Chapter [*];}
  class Chapter {title : String; nbPages : Integer;}
{}
}

//defining model PUB
metamodel PUB {
```

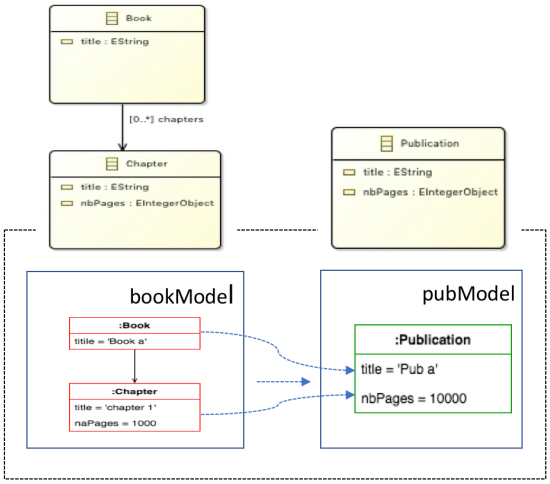


Figure 2.1: Transformation from metamodel *BOOK* to metamodel *PUB*

```
  class Publication {title : String; nbPages : Integer;}

}


//Transformation declaration with one input model BOOK, one output model PUB
transformation Book2Publication(in bookModel:BOOK, out pubModel:PUB);

//Entry point
main() {

  bookModel->objectsOfType(Book)->map book_to_publication();

}

//Mapping the input Class to Publication class
mapping Class::book_to_publication () : Publication {

  //Assign properties to target object
  title := self.title;
  nbPages := self.chapters->nbPages->sum();

}
```

Listing: BOOK2PUB transformation (Annex A.2.1 in [4])

2.3 Software Quality and Quality measurements

2.3.1 Software Quality

Quality is important for software, poor quality software generally consumes more cost and effort, a serious defect may even cause fatal damages at a later stage of the software's life-cycle. Software quality can be observed from various perspectives, such as functional quality, non-functional quality, product quality, process quality, etc. Quality models are proposed to describe these different aspects of software quality, a classic example is the ISO 25010 [1] quality model, which provides two models to describe the quality of softwares: product quality model (which is shown in Figure 2.2) and "quality in use" model. The product quality model consists of eight characteristics

that are all observed from the software products. It combines the internal quality model and the external quality model, both specified in ISO 9126 [22] (the predecessor of ISO 25010). The internal quality focuses on software artifacts itself, e.g. source code, design and architecture of the software, while external quality concentrates on attributes in software process, e.g. execution time. Another model provided in ISO 25010 [1] is "quality in use" model, which is composed of five characteristics that are related to the interaction between software products and stakeholders. All these characteristics can be further divided into several sub-characteristics.

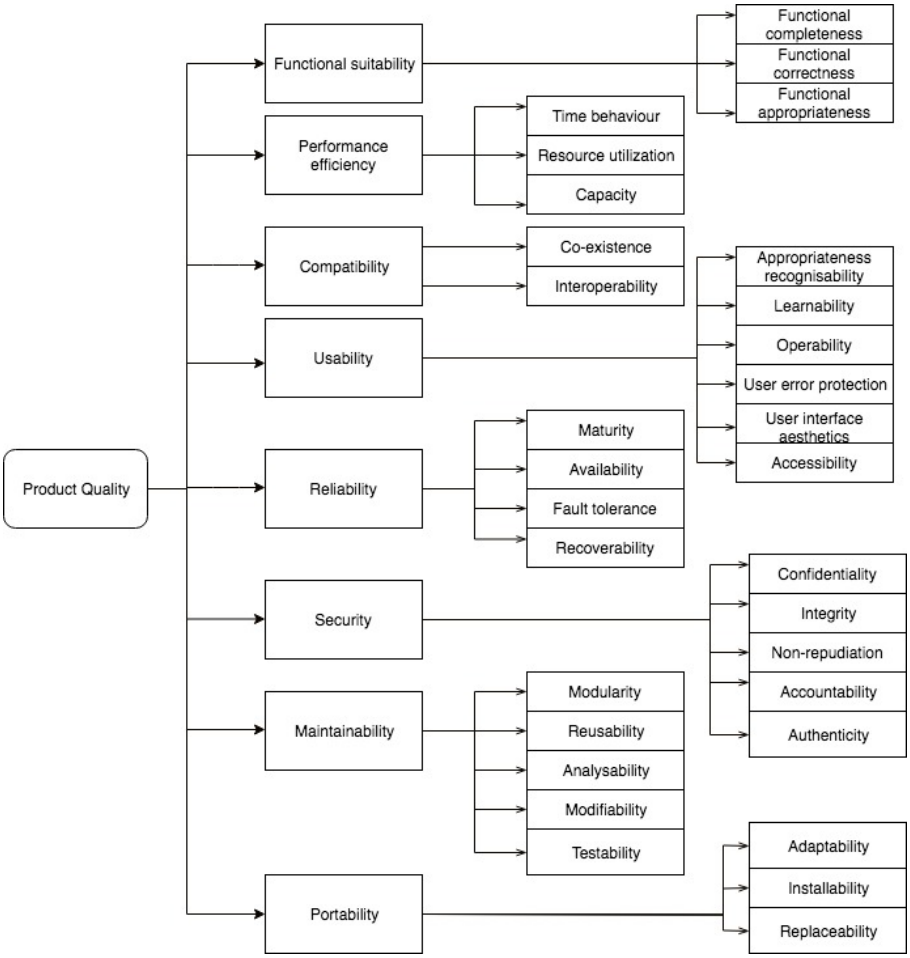


Figure 2.2: Product quality model. This model is described in ISO 25010 [1]

It is notable that there are characteristics with different meanings given similar names in these two models. For instance, **context completeness** defined in "quality in use" model means how usable a software is in certain context (low network bandwidth, non-expert users, etc), but the **function completeness** specified in the product quality model describes to which extent the tasks and goals of the users are covered by the functions of software products. Hence, it is important to distinguish theses characteristics when the quality models are used for analyzing requirements, identifying design and test objects, defining quality control standard and acceptance criteria. Additionally, some quality name are refined in the latest revision, for example, "Understandability" is refined as "Appropriateness recognizability". Considering all these mentioned product development activities, establishing measurements is the common foundation.

2.3.2 Quality measurements

A quality measurement reference model is defined in ISO 25020 [2], it illustrates the relationship between quality and measurements, as shown in Figure 2.3. The basic element is defined as quality measurement element, including base measurements which are directly observed from the software artifacts and derived measurements defined as the function result of two or more base measures. The quality measurement elements construct a high-level quality measures that can be used as indicators for the characteristics of the software. A considerable amount of work has

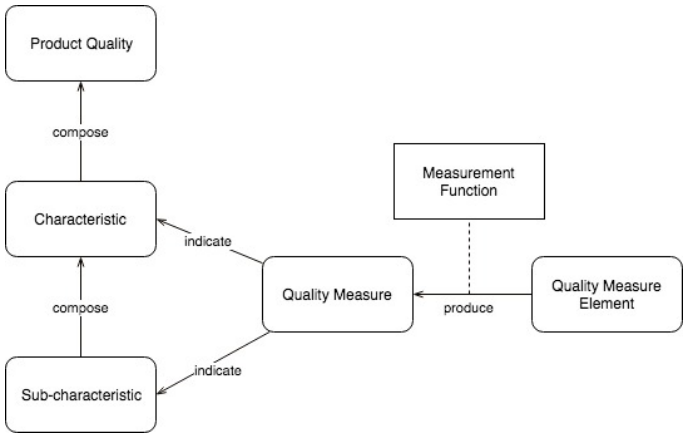


Figure 2.3: Relationship between quality and measurements. Demonstrate the model described in ISO 25020 [2]

been done to investigate measurements and metrics for various characteristics of software quality. For example, internal metrics (static product metrics), such as lines of code, complexity measures and number of faults in the program, are proposed for quantifying internal quality. On the other hand, external metrics (process metrics) are specified to measure the characteristics of external quality. For instance, the number of failures during testing, which is different with number of faults in the program, as a fault can lead to multiple failures and testing may not find any faults.

The prime issue of measurements is how to construct metrics to measure the quality of software. Kitchenham [8] proposed a structural model of software measurement with identifying the key elements:

1. **Entities and attributes**: entities are products, processes or resources, while attributes are the properties of entities;
2. **Units (metrics) and Scale Types**: units define how a quality characteristic is mapped to the mathematical world. Like Celsius is used when measuring temperature, Lines (of code) is adopted for measuring the size of code. Different units can lead to different scale types, which includes nominal, ordinal, interval and ratio. Scale type is a key factor when manipulating the metrics, for example, nominal metric {0,1,2} that represents the fault categories can not be summed or averaged;
3. **Values and properties of values**: the values are collected by applying particular measurement units to the software. In most cases, values are numerical while non-numerical values, such as a set of labels, can be mapped to numbers. The properties of values refer to the valid set over which values are defined, they are finite or infinite, discrete or continuous, bounded or unbounded, etc;
4. **Measurement instrument**: instruments are used to obtain values, for example a software program can be used to count the lines of code of a program;

5. **Indirect measures, properties and compound units**: Indirect measures are the measures that are calculated from one or more other measures, like lines of code per hour and this type of measures has its corresponding properties and compound units;
6. **Measurement validation**: the validation presents the way in which an attribute is related to the characteristic of quality.

If metrics are designed for a specific purpose, an effective approach is Goal/Question/Metrics (GQM) proposed by Basili [9], which defines the measurements in a top-down way. The GQM model focus on three levels:

1. **Conceptual Level (Goal)**: a goal is set for a measurable objects (products, processes or resources), considering different quality models and viewpoints;
2. **Operational Level (Question)**: questions interpret a goal, each question is based on a selected quality and viewpoint;
3. **Quantitative Level (Metric)**: metric is the method to capture values answering questions. The values can be objective or subjective.

2.4 Lehman's Law

Analyzing the evolution of QVTo transformations is one research question in this thesis. In our research, we validate whether the evolution of QVTo transformation conforms to the classic laws of software evolution Lehman's Law. In this section, a description of Lehman's Law is presented. In software engineering, software evolution is a domain which investigates how software adapts to the change of requirements and environment, it also analyzes the change process and different versions of software in the repository to predict trends. One classical view of software evolution is proposed by Lehman [5]. In Lehman's study, the observation of software evolution is concluded

Table 2.2: Lehman's Law [5]

Name	Law
Continuing change	An E-type system must be continually adapted otherwise it becomes progressively less satisfactory in use
Increasing complexity	As an E-type system is evolved its complexity increases unless work is done to maintain or reduce the complexity
Self Regulation	Global E-type system evolution is regulated by feedback
Conservation of Organizational Stability (invariant work rate)	The work rate of an organization evolving an E-type software system tends to be constant over the operational lifetime of that system or segments of that lifetime
Conservation of Familiarity	In general, the incremental growth (growth rate trend) of E-type systems is constrained by need to maintain familiarity
Continuing Growth	The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over the system lifetime
Declining Quality	Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining
Feedback System	E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems

as a set of laws. It should be noted the laws only applies to software belong to E-type software,

which always changes because of the continuing modification of their requirements. Table 2.2 shows the total list of Lehman's law. In our research, we check whether the evolution of our QVTo transformation goes with the following laws *Continuing change, Continuing Growth, Declining Quality.*

Chapter 3

Related work

In this chapter, we describe several studies related to our research. First of all, Section 3.1 discusses the existing metrics, based on which we design new metrics according to our need. Then, in order to find methods for analyzing the quality evolution of our QVTo transformation, Section 3.2 presents relevant research about the evolution of software quality. At last, Section 3.3 describes related work for imperative/declarative rewriting as bases for proposing our rewriting pattern for QVTo transformation.

3.1 Metrics

In order to design suitable metrics, we have a literature review of the existing software metrics. The research of software metrics has evolved with developments in software engineering. For instance, from traditional software metrics to metrics for objected oriented design. Since Model Driven Software Engineering has been embraced by the industry, recent research has focused on metrics for models and model transformations. In particular, we focus on QVTO transformations [23]. This section describes and discusses these classic metrics as well as the state-of-art metrics.

3.1.1 Metrics for traditional software

First of all, we present a brief overview of the existing work about classic metrics for traditional software, which inspire the research for designing metrics for objected oriented softwares and models. A significant amount of research has been done to evaluate the software quality and various of metrics have been proposed. Typical metrics used for estimating the quality of software systems are related to the size and complexity of the software artifacts[24], which are described and discussed in this section.

Lines of code

Size is an important attribute to be considered when measuring an software artifact. Lines of codes (LOC), which measures the size of a software, is a widely used metrics, because of its easy computation and high correlation with the software quality and development effort. A higher LOC generally leads to more errors and costs more effort to develop and maintain the software. Hence, many cost estimation models, such as Constructive Cost Model (COCOMO) [25] uses LOC as an input, however the definition of SLOC is ambiguous, for instance, whether this metric counts white spaces and comments, moreover the relationship between lines and statements may not be

one-to-one. Measurements with LOC are inconsistent and incomparable among different companies and organizations[26]. To solve this problem, the Software Engineering Institute (SEI) has provided a framework for defining LOC based on a checklist of the counted attributes of software so that all involved stakeholders can confirm the definition of LOC [27], which is extended in [28] and applied using a tool set CodeCount. Another major disadvantage of LOC is that it ignores the structure of a software, which leads to the study of metrics related to complexity.

Halstead complexity

A well-known metric that analyzes software's complexity is Halstead's metrics. The Halstead [29] metrics, which is defined in a manner analogy to thermodynamics, was designed for analyzing the complexity of the source code regardless of the language is used. Based on the hypothesis that the general structure of a program obeys the physical law, operators and operands are identified as the only two independent properties of a program . Assuming the number of unique operators and operands are η_1,η_2, and the number of accumulated operators and operands are N_1, N_2, then the size of a program V is defined as

$$V = N * log_2\eta$$

where N is the sum of N_1 and N_2, η is the total occurrences of operators and operands. A program can have multiple implementations, which is distinguished by a level indicator L. The program level is calculated with the following formula

$$L = \frac{2}{\eta_1} * \frac{\eta_2}{N_2}$$

The maximum value of L is 1, when the program is implemented in the highest level using only 2 distinct operators and every operand has only one occurrence. Combining V and L, a metric $E = \frac{V}{L}$ is defined to indicate the amount of mental effort required for understanding and maintaining the program.

McCabe complexity

Since the Halstead's metric has been proposed, more research has been conducted to explore the complexity of a software. A classic metric is Cyclomatic Complexity(CC) that was introduced by McCabe [30] based on graph theory. The main goal of cyclomatic complexity is to compute the maximal linearly independent paths in a graph, which can be a basic set to construct the graph. Based on this strategy, the cyclomatic complexity $V(G)$ of a graph with n vertexes, e edges, and p connected components is defined as $V(G) = e - n + 2p$. For example, Figure 3.1 shows a simple graph and its five linearly independent paths $p1 \dots p5$, which means any path of the graph can be represented with the linear combination of $p1 \dots p5$. It is notable that $V(G)$ only depends on the number of decision made, so that McCabe [3] gives another definition of cyclomatic complexity $V(G)$ = the number of decision points +1.

Dependency complexity

Another significant metric proposed in this software's complexity research is *fan-in* and *fan-out* [31], which was introduced to describe the information flow between procedures. Henry [31] suggested that there are two major factors affecting the complexity of a software, one is the complexity of the code, and the other is the relations to its environments. However, Halstead [29] metrics and McCabe [3] metrics shows little concern for the latter. Additionally, programming languages have been developed to support functions and procedures. This evolution and the introduction of

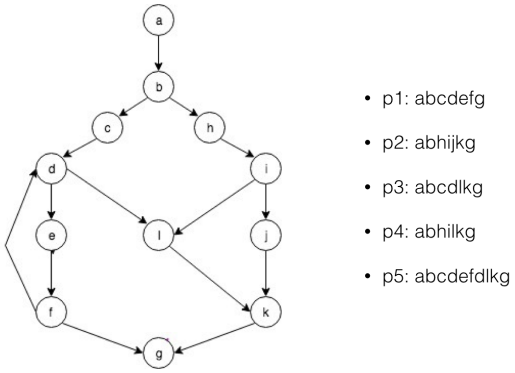


Figure 3.1: a sample graph and its maximum linearly independent paths, described in [3]

modularity motivated new ways to calculating program's complexity. Therefore, Henry presented an approach to estimate this complexity by counting the number of local information flows entering in a module or a procedure as the *fan-in*, and the number of local information flows exiting out as the *fan-out*. The key to this strategy is identifying the local information flow, for instance, information flow from module A to module B includes:

1. A calls B;
2. B uses the data returned from A when calling A;
3. Another module C calls A and B, and passes data from A to B.

The relations between Halstead, McCabe and the *fan-in and fan-out* metrics were also demonstrated in [31] with a series of experiments undertaken on Unix operating system. Henry applied these three metrics to 165 procedures in the UNIX operating systems and calculated the correlation coefficient between measured values. Based on the correlation coefficient Henry [31] concluded Halstead's metrics and the Cyclomatic Complexity metric are highly correlated to each other, while the *fan-in* and *fan-out* metrics is independent of the other two metrics.

3.1.2 Metrics for object oriented paradigms

Modeling languages are built on the basis of object oriented paradigms, for instance, studies for OO design including Booch's method [32], Rumbaugh's OMT (object modeling technology) [33] and Jacobson's approach[34] lead to UML [17], a standard object oriented modeling language. Hence, we discuss the existing studies about metrics for OO design in this section before further exploring research about metrics for model transformations.

With the rise of object oriented technology, the previously discussed traditional metrics do not reflect these object oriented paradigms properly. Object oriented programs solve problems by simulating the real-world objects and their behavior. There are four fundamental characteristics that make OO programs effective and convenient, which are encapsulation, data abstraction, inheritance and polymorphism.

1. Encapsulation: OO programs hide the irrelevant information and provide a single unit which contains data and methods. This characteristic protects data from misusing through restricting its access.

2. Data abstraction: this means capturing the objects essential features and ignoring the implementation details. Only concerning the attributes which are relevant to the context makes it possible to describe the complex real-world objects.

3. Inheritance: inheritance allows different objects sharing the same method implementation, therefore it makes the programs more effective with code reuse.

4. Polymorphism: means methods of an object with the same name can have different implementation, in this way objects respond differently depending on messages, for instance, number of input arguments or their data type. This characteristic also makes programs more effective.

The following section presents study about metrics designed for object oriented softwares. The most popular OO metrics are the suite metrics proposed by Chidamber and Kemerer[35], which consists of six metrics that focus on key OO concepts, These six metrics are listed as follows:

1. Weighted Method Per Class (WMC): The complexity of a class is highly depends on the complexity of its methods, since methods comparing to variables are considered by developers to be more time consuming. Therefore, the complexity of a class C is evaluated through measuring these methods.

$$WMC = \sum_{i=1}^{n} c_i$$

where c_i is the complexity of the ith method defined in class C. For the purpose that WMC can be used more generally, the definition of c_i is decided by developers. Any complexity metric which are summable can be adopted, for instance, McCabe's cyclomatic complexity. In the case that complexities of all methods are the same, WMC is equal to the number of methods. A higher WMC indicates a more complex class which needs more effort to develop and maintain.

2. Depth of Inheritance Tree (DIT): Inheritance is an important characteristic for OOD, so its impact must be considered when estimating the complexity of OO software. The DIT of a class is defined as the depth of the longest path from the class node to the root in the inheritance tree. A higher DIT metric value tends to indicate a more complex design, as more classes and methods may affect the class which makes it harder to interpret its behavior.

3. Number of Children (NOC): NOC is another metric related to inheritance. Different from DIT which focuses on the relation between a class and the whole design, NOC value concerns the relation between a class and its immediate subclasses. The NOC of a class C is defined as the total number of the classes which inherits from class C. When observing the class C in the inheritance tree, NOC are the values measured in a horizontal view, while DIT are values in a vertical view. Figure 3.2 illustrates NOC and DIT metrics with a very simple instance. The NOC indicates how many subclasses can be impacted by the class, a higher NOC value means the class is highly reused and has more influence in the design, hence the class requires more testing effort.

4. Coupling between object classes (CBO): Dependency between classes is a crucial factor when estimating the complexity of a software. For traditional software, *fan-in, fan-out* values, which measure the number of information flows, are used for this purpose. Likewise, CBO for a class is defined as the total number of its coupled classes to indicates this dependency. By definition, a class is coupled to class C, when it uses methods or instance variables that defined in class C, therefore, CBO for class C can be computed by summarizing the total number of classes which coupled to class C and the number of classes that class C is coupled to. Because a high CBO value indicates a higher complexity of the design, a minimal CBO value is suggested to improve modularity and reduce the maintenance effort.

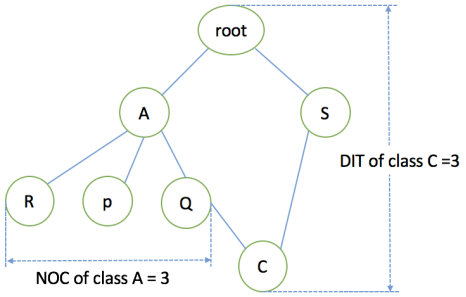


Figure 3.2: A sample inheritance tree

5. Response For a Class (RFC): this metric is designed to measure the maximal methods that can be executed when using an instance object of the measured class. The value is extracted by counting the number of methods contained in the response set RS of the class, which is formalized as follows,

$$RS = \{M\} \cup_{\forall i} \{R_i\}$$

where $\{M\}$ is the set of methods defined in the class, and $\{R_i\}$ is the methods called by method i in the class. With this definition, RFC can also be seen as a measurement of dependency between different classes, but differing from CBO, RFC measures at the method level rather than the class level, in other words, CBO counts the number of corresponding classes while RFC counts the number of methods. A class with a higher RFC value means the class can invoke more methods, therefore it is more difficult while debugging and costs more time for maintenance.

6. Lack of Cohesion in Methods (LCOM): Similar to RFC, LCOM also investigates relations between methods, but this metric is aimed at measuring the intra-class methods relationships instead of inter-class methods relationships. Assuming M_i is a method defined in class C, and a set $\{I_i\}$ is the corresponding instances variables set of M_i. The definition of LCOM is

$$LCOM = \begin{cases} |P| - |Q| & \text{if } |P| > |Q| \\ 0 & \text{otherwise} \end{cases}$$

where for all methods in class C, $P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\}$ and $Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$. On the basis of the definition, a high LCOM suggests that the class may be divided into two or more separate classes. For example, Figure 3.3 shows two classes with different LCOM values, where M_i are methods, and V_i are variables, when a variable belongs to the instance variables set of a class, the corresponding block is marked with A. It can be observed that class A is more likely divided into two classes due to its higher LCOM value.

CLASS A	V_1	V_2	V_3	V_4
M_1	A	A		
M_2	A	A		
M_3			A	A
M_4			A	A

CLASS B	V_1	V_2	V_3	V_4
M_1	A	A		
M_2		A	A	
M_3			A	A
M_4	A			A

Figure 3.3: A comparison between class A (LCOM = 2) and class B (LCOM = 0)

3.1.3 Metrics for QVT

Having discussed the traditional metrics and the OO metrics, we finally present the relevant work about metrics for QVTo in this section. Like traditional metrics, these metrics can be categorized into three classes: size metrics, complexity metrics, and dependency metrics.

1. **Size metrics**: The metrics related to the size of transformation mostly count the number of elements of a transformation. Kapova [36] presents a series of metrics including *lines of codes* that measures the code size as the original LOC, *number of relations, number of top level relations, number of relations without when- clause* that measure the transformation rules in QVTr cases.

   In Nguyen's study [13], several different metrics related to size for various elements for QVTo are also proposed, like *number of overloaded mappings, number of helpers, and number of modules, number of input models, number of output models* etc. Gerpheide [11] adapted the metrics in [13][37] to quality attributes and constructed a quality model with these attributes, such as *small transformation size, more mappings than helpers, more queries than helpers, few nested if statements.*

2. **Complexity metrics**: In model transformation, the *when- and where-* clauses, *forEach and while Loop* lead to multiple path for execution, therefore *average cyclomatic complexity per Mapping* and *average cyclomatic complexity per Helper* are computed in [37] analogy to Cyclomatic Complexity metric for traditional software.

3. **Dependency metrics**: Dependency between different elements is a key aspect for estimating the quality of model transformations. Hence, *fan-in* and *fan-out* can be adapted to several useful indicators for the complexity of information flows of model transformation.

   In the case of QVTr, Kapova [36] defines the information flow on two levels, number of variables which are called in the when- or where- clauses are extracted as *val-out* metric, and *val-in* is the average number of arguments in the form of its domains, which is always the same as *number of domains* in QVTr. On the relations level, counting the number of times relation r uses other relations or queries as *fan-out*, and *fan-in* is the number of times that other relations uses r.

   For QVTo, dependency is measured with various metrics. Nguyen [13] proposed a series of metrics to estimate various types of dependencies including dependencies between different mappings, different helpers, different modules, and dependencies between mappings and helpers. Nguyen further derived more complex metrics, such as *number of calls from mappings in other Modules per Module, number of calls from Helpers in other Modules per Module, number of calls to resolve expressions per module, number of calls from mappings to resolve expressions per mapping, and number calls from helpers to resolve expression per Helper* to describe dependencies in details. Figure 3.4 demonstrates a selected set of metrics of the sample QVTo code, which is partial *SimpleUml2RDB* code [4].

Additionally, metrics can be collected in different ways to assess the external quality. Gerpheide [11] assessed the QVTo transformation quality by using quality attributes, which combined metrics and a preferable direction, such as *small transformation size, few dependencies between modules and more mappings than helpers.* Metrics can also be embedded to the systems, Saeki [39] adopted OCL to express the metrics and binded them to the systems. Table 3.1 and Table 3.2 presents a set of major metrics for model transformation.

Table 3.1: Metrics related to Size or Complexity

Metric	**Transformation Language**	**Reference**
lines of code	QVTr	Kapova[36], Amstel[38]
# When- clauses	QVTr, QVTo	Kapova[36]
# Where- clauses		
# Relations	QVTr	
# Top level relations		
# Starts		
# OCL queries		
# Metamodels in transformation		
Avg. domains per relation		
Avg. domain pattern nodes per relation		
Avg. when- clauses per relation		
Avg. where- clauses per relation		
Avg. local variables per relation		
# Mappings	QVTo	Amstel[37]
# Mappings with Condition		
# Helpers		
# Elements per Mapping		
# Parameters per Mapping		
# Operations on collections per Mapping		
# Sub-Objects per Helper		
# Parameters per Helper		
# Overloaded mappings		Nguyen[13]
# Unused mappings		
# Abstract mappings		
# Mapping Inheritances		
# Mapping Mergers		
# Mapping Disjunctions		
# Unused Helpers		
# Overloaded Helpers		
# Modules		
# Library Modules		
# Operational Transformation Modules		
# Input models		
# Output models		
# Imported Metamodels		
# Intermediate Classes		
# Intermediate Properties		
# End section		Gerpheide [11]
# Blackboxes		
# Configuration properties		
# Queries with side-effects		
# When and where clauses		
# ForEach Loops		
Size of init sections		

Table 3.2: Metrics related to Dependency

Metric	**Transformation Language**	**Reference**
Val-in per relation	QVTr	Kapova[36]
Val-out per relation		
Fan-in per relation		
Fan-out per relation		
Fan-in per mapping	QVTo	Amstel[37], Nguyen[13]
Fan-out per mapping		
Fan-in per Helper		
Fan-out per Helper		
Calls to resolve expressions		
Avg. resolve from mapping		
# Calls to log()		
# Calls to assert()		
# Calls from Mappings in other Modules per Module		
# Calls from Helpers in other Modules per Module		
# Calls from Other Modules per Module		
# Calls to Mappings in Other Modules per Module		
# Calls to Helpers in Other Modules per Module		
# Calls to Other Modules per Module		
# Calls from Mappings to Mappings per Mapping		
# Calls from Mappings to Helpers per Mapping		
# Calls from Mappings to Mappings/Helpers per Mapping		
# Calls to Mappings from Mappings per Mapping		
# Calls to Mappings from Helpers per Mapping		
# Calls to Mappings per Mapping		
# Calls from Helpers to Helpers per Helper		
# Calls from Helpers to Mappings per Helper		
# Calls from Helpers to Mappings/Helpers per Helper		
# Calls to Helpers from Mappings per Helper		
# Calls to Helpers from Helpers per Helper		
# Calls to Helpers per Helper		
# Calls to log() per Module		
# Calls from Mappings to log() per Mapping		
# Calls from Helpers to log() per Helper		
# Calls to assert() per Module		
# Calls from Mappings to assert() per Mapping		
# Calls from Helpers to Resolve Expressions per Helper		

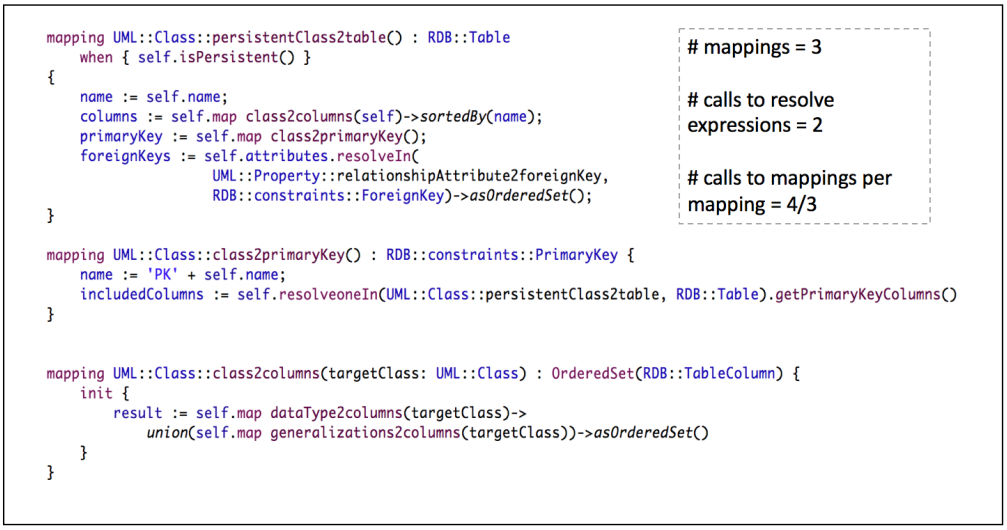


Figure 3.4: A example of QVTo metrics.(SimpleUml2RDB code presented in [4])

3.2 Quality evolution

Another related topic for our thesis is quality evolution because the first step in our research is to analyze the evolution of QVTo transformations. Software systems evolve over time because of the adaptation to requirements or environment. Software evolution is a key role in the whole life-cycle of the software since the maintenance of the software needs the most effort and cost. Therefore, it is important to assessing the quality of the software during the evolution process. There has been an increasing investigation into the quality evolution of software since the awareness of its importance [40][6][41].

In Men's work [6], a list of research topics related to metrics for analyzing software evolution is presented. These topics can be classified into two categories: predictive analysis before evolution or retrospective analysis after evolution. Among all these topics, we focus on *Measuring software quality* and *Long term evolution* in our research. Table 3.3 presents all the topics.

Drouin's [41] study presents an empirical analysis of software evolution using metrics. A novel metric named *Qi* which integrates different OO (Object Oriented) metrics described in the previous section is proposed and used to observe how the software systems evolve. The value of metric *Qi* is normalized, and a lower value indicates the complexity of the software is higher and needs more quality assurance effort. An empirical study is conducted on multiple versions of three open source software: two Eclipse components (JDT.Debug, PDE.UI) and Tomcat project. With CVS (Concurrent Versions System), values of metrics are collected from historical data of several years (4 years for the first two software, and 7 years for the other one). By analyzing the collected data, Drouin states that metric *Qi* reflects the quality evolution of software systems properly.

Hecht's research [40] tracks the quality evolution of Android applications with metrics. The authors use a tool called PAPRIKA to detect antipatterns (aka. poor design choices) from 3568 versions of 106 Android applications and further analyze the quality evolution based on antipatterns.

Table 3.3: Topics of software evolution metrics [6]

No.	Topic	Discription
1	Measuring software quality	Using metrics to measure the quality of software and whether the quality has improved between different revisions of the software
2	Coupling/cohesion metrics	Using metrics to quantify coupling and cohesion, which can further measure a system's structural complexity
3	Scalability issues	Using metrics to analyze the scalability of software. Considering the explosion of number of measurements to be interpreted, the key problem is how to visualize the measurements.
4	Empirical validation and realistic case studies	Empirical research are important topic since we can only validate the evolution metrics when they are tested on a sufficient number of examples
5	Long term evolution	Long term evolution of software are important as it may show different nature of the software
6	Data gathering	Avoiding the lost of data, we need tools to maintain precise logs about the changes applied between revisions
7	Detecting and understanding different types of evolution	Using metrics to detect the types of evolution
8	Process issues	Using metrics to predict changes in evolution, such as cost estimation, effort estimation and programmer productivity
9	Language independence	Using evolution metrics in a language-independent way to extract results about software evolution that are not bound to a specific language

3.3 From imperative to declarative

Since we need to convert the original imperative QVTo transformations to more declarative ones in our research, a review of related work is necessary before we try to propose our rewriting pattern to fulfill the task for imperative-declarative conversion. There have been a considerable number of published studies describing methods which can accomplish the task of rewriting or optimizing for both general programming language, like Java [42][43], and modeling language, such as UML/OCL [7][44].

3.3.1 RImperative/declarative conversion for Java

In a study which coverts imperative parts of Java code to declarative parts, Gyori [42] proposed and implemented two refactoring patterns: the first one can replace the AIC (Anonymous Inner Class) with Lambda expressions and the second one converts *for Loops* which iterate over an instance of *Java.util.Collection* to Lambda expressions. The Lambda expression refers to a function defined without binding to an identifier, and it consists of three parts: a parameter, an arrow, and a body. For example, the following Lambda expression

$$p \rightarrow p.getGender() == \text{Person.Sex.MALE}$$

checks the gender of p (an instance of Class *Person*), where p is the parameter and expression $p.getGender() ==$ Person.Sex.MALE is the body of this Lambda expression [45]. With Lambda expressions, the imperative code can be rewritten based on the refactoring patterns described in Gyori's study.

Listing 3.3.1 shows a simple example for the first refactoring pattern, which implements a function that adds a listener to a button. The one shown in (a) uses an inner class, and its equivalent Lambda expression is presented in (b).

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    ui.dazzle(e.getModifiers());
  }
});
```

Listing 3.3.1(a) Anonymous inner class [42]

```
button.addActionListener(
  (ActionEvent e)->{ui.dazzle(e.getModifiers())};
);
```

Listing 3.3.1(b) Equivalent lambda expression [42]

For the second pattern, here is an example. Given a collection of blocks, we try to change all blue color to red. Instead of iterating the blocks with a *for* loops, a lambda expression is adopted

$$blocks.stream().filter(b \rightarrow b.getColor() == \text{BLUE}).forEach(b \rightarrow b.setColor(\text{RED}))$$

3.3.2 Imperative/declarative conversion for UML/OCL

Unlike Radoi's work, Cabot's [7] investigates into the imperative/declarative conversion of UML/OCL code instead of Java code. Additionally, the goal of this investigation is to find a rewriting method between declarative and imperative UML/OCL specifications. Before discussing these methods, it is necessary to have a brief description of UML (Unified Modeling Language).

Since the emergence of OO programs, various notations for OO design had been proposed including Booch's method [32], Rumbaugh's OMT (object modeling technology) [33] and Jacobson's approach[34]. Their studies lead to UML [17], a standard object oriented modeling language that is controlled by Object Management Group (OMG). UML provides a variety of model elements, such as classes, states and relationships to represent artifacts of software systems:

1. Classes: represent a group of objects that have the same attributes;
2. States: capture a set of conditions of objects in time, the objects transit through different states responding to events;
3. Relationships: describe the associations between UML model elements. There are four widely used types of relationships: associations (a general link relates objects two each other, it also specifies how many objects are evolved), generalizations (describes the inheritance), dependency and aggregation.

OCL (Object Constraint Language) is a declarative language which specifies the rules applied to UML models. In OCL expressions, constructs like *forAll, select* are very powerful when iterating over collections. For instance, a rule "A person is younger than its parents" can be expressed as follows,

$$context\ Person\ inv : self.parents \rightarrow forAll(p|p.age > self.age)$$

where *self* is the object which the rule is applied on and *parents* is an attribute of object *self*.

Cabot's study presents the target pattern in the original UML/OCL code and the corresponding rewritten pattern. The rewriting pattern only covers the structural events in UML for simplicity. Table 3.4 shows examples, where B_i is a boolean expression, o refers to an object, and $o.r$ stands for a navigation from object o to a related object of an associate class.

Table 3.4: Rewriting Pattern for UML/OCL code [7]

N	Expression	Rewritten code	Description
1	B1 and ... and Bn	Translate(B1); ... Translate(Bn);	A set of boolean expressions linked by ANDs are transformed by translating each single expression sequentially.
2	o.r→includes(Y)	CreateLink(r.association,o,Y);	A link is created between o and the single object returned by Y
3	o.r→excludesAll(Y)	foreach o in Y DestroyLink(r.association,o,o) endfor;	All links between o and the objects in Y are destroyed.

3.4 Conclusion

Having discussed the relevant work, which includes an overview of existing software metrics, a description of related work for quality evolution and the introduction of methods for imperative-declarative programming conversion. With knowledge of these techniques, we can analyze the quality evolution of our QVTo transformation and design new metrics in Chapter 4, and furthermore, we can propose methods to rewrite the imperative QVTo transformation in more declarative style in Chapter 5.

Chapter 4

Quality evolution of QVTo transformation

This section investigates the first research question. The exploration is divided into two parts: in Section 4.1, inspired by the methods described in Section 3.2, we analyze the evolution of the quality of the QVTo transformation with a set of metrics. The results show that the old metrics, which are used for measuring how imperative/declarative the QVTo transformation is, are not suitable in our case, hence in Section 4.2 we design a set of new metrics and validate these new metrics.

4.1 Approach

The first research question in this thesis is explore how the quality of QVTo transformations evolve over time. The goal of this question is to analyze the evolution of the quality of the QVTo transformation quantitatively. As discussed in Chapter 3, metrics can be used to perform quantitative analysis. First of all, we need to select suitable metrics for analyzing the quality evolution.

Selecting the suitable metrics

As discussed in Section 3.1.3, a considerable number of metrics for QVTo are presented in the literature. Gerpheide's research [11] provides a useful quality model for QVTo transformation, within the model multiple metrics are adopted as indicators for understandability, which is suggested as the most ubiquitous quality goal by Gerpheide. Hence, in our study, these metrics are applied to a series of revisions of the QVTo transformations to analyze how the quality evolves. Table 4.1 presents the total list of metrics related to understandability. As described in Section 2.2, *forEach, init, where, when* are notions for blocks in QVTo transformations and *mapping, helper, query* notions for operations.

Quality attribute refers to quality-carrying properties of the target objects, which are QVTo transformations in our research. All the quality attributes presented in Table 4.1 improve the quality (understandability in our case) of QVTo transformations based on Gerpheide's quality model. However, in Gerpheide's work, a survey of developer's point of view is performed to confirm the relationship between the quality attributes and their corresponding quality. Based on the analysis of the results of this survey, *Little imperative programming, Small init section*, and *Few queries with side-effects* are considered as the validated attributes, and the attribute

Table 4.1: Metrics for Understandability

Metric	Quality attribute
#*forEach* loops	Little imperative programming
Size of *init* section	Small *init* section
#*where* clauses	Few *where* clauses
#*when* clauses	Few *when* clauses
Ratio #mappings to #helpers	More mappings than helpers
Ratio #queries to #helpers	More queries than helpers
#Configuration properties	Few configuration properties
#Intermediate properties	Few intermediate properties
#*end* sections	Few *end* sections
#Queries with side-effects	Few queries with side-effects

Few when clauses is considered positive but less important than the three attributes mentioned previously for understandability. Therefore, we select their corresponding metrics to explore the quality evolution of the QVTo transformation in our experiments.

Mining the repository

The second step is to decide which revisions of the QVTo transformation are analyzed in the experiments. The number of revisions of the QVTo projects in the repository is so large that it is necessary to find a sufficient number of revisions to make analyzing the quality evolution reasonable. As our research focuses on the quality of a specific QVTo transformation, we track the file which contains this QVTo transformation.

4.1.1 Results and Analysis

We apply the validated metrics from Gerpheides model to the chosen revisions of the QVTo transformation. Overall, 60 revisions where the file *pgapp2dsgraph.qvto*, which have been discussed in Section 1.2, has been modified are chosen out from 4419 revisions and the selected metrics are extracted from each revision of the QVTo transformation. Additionally, we extract values of basic metrics, like LOC and *Number of Mappings* from these 60 revisions of the QVTo transformation. The metrics are analyzed based on the commit time of each chosen revision, Figure 4.1 and Figure 4.2 illustrates how the values of metrics evolve from the earliest to the latest revision. As can

Figure 4.1: Evolution of basic metrics

be seen in Figure 4.1, the size of our QVTo transformation is relatively stable over time, but the total number of *Mapping*s increases over time. These observations conform to the rule *Continuing change, Continuing Growth* of Lehman's Law which has been introduced in Section 2.4.

- Continuing change: both the size and the metric *Number of Mappings* change over time;
- Continuing Growth: metric *Number of Mappings* increases along the evolution of the QVTo transformation. As the *Mapping*s are crucial for the functionality of a QVTo transformation, the rise of this metric indicates the QVTo transformation is continually enhanced over time in terms of functions.

Figure 4.2: Evolution of metrics for understandability

Quality evolution trends

Based on the results shown in Figure 4.2, the following conclusion can be made.

1. **Overall Rise**: the metrics *Size of init section*, *Number of when clauses* and *Number of queries with side effects* all increase along the evolution. As stated in Table 4.1, the overall rise of these metrics indicate that the understandability of the QVTo transformation decreases over time;
2. **Overall Decline**: however, there is a clear trend of declining for the metric *Number of forLoops*, which implies improvement of understandability along the QVTo transformation's evolution;
3. **Sudden Rise**: as can be seen, there are cases where the value increases sharply at a specific revision for metric *Size of init section, Number of when clauses*and *Number of queries with side effects*. This sudden rise infers a turning point where the understandability of the QVTo transformation drops rapidly;
4. **Sudden Decline**: In contrast, *Number of forLoops* shows sudden decreases at some revisions, therefore, at these turning points, the understandability improves suddenly.

It is notable that the first conclusion represents an opposite trend of quality evolution with the second conclusion, and the last two findings are contradictory as well. The reason for this contradiction is that metric *Number of forLoops* leads to different conclusions with other metrics, which indicate a decline of the understandability of the QVTo transformations. Moreover, according

to Lehman's Law, the quality of a software system should decline over time. Hence, we further investigate metric *Number of forLoops* in the following section.

As presented in Table 4.1, metric *Number of forLoops* are used for quality attribute *Little imperative programming* which has been validated to improve the understandability in Gerpheide's evaluation. However, the relationship between the metric and quality attribute has not been confirmed. As can be seen in the results, *forLoops* are not used frequently in our QVTo transformation, they are only used three times in the latest revision. Therefore, metric *Number of forLoops* is not reliable to indicate *Little imperative programming*, this could be the reason why there are conflicting conclusions drawn from the results shown in Figure 4.2. To address this problem, we need to design more suitable metrics to measure how imperative/declarative a QVTo transformation is. The following section discusses how to design suitable metrics and proposes our solution.

4.2 Metrics

4.2.1 Design metrics

Using the Goal/Question/Metrics model discussed in Section 2.3.2, we decide a goal and interpreted with several questions and finally design the metrics to answer these questions. The three steps are performed as follows:

- Goal: improving understandability for the QVTO transformation
- Question 1: how imperative is the QVTo transformation?
  Metric 1: number of imperative operations contained in the QVTo transformation;
- Question 2: how imperative are these imperative operations which are counted in Metric 1?
  Metric 2: number of imperative expressions contained in the imperative operations.

Metric 1 and 2 can be defined differently according to which imperative operation is measured. As described in Section 3.1.3, there are various types of metrics designed for measuring the quality QVTo transformation, but they are not suitable in our case. Following the GQM model presented previously, metrics proposed in our study can be divided into three categories:

- Operation metrics: the imperative operations including *Mapping, Constructor, Helper*, and *Query* are fundamental elements for QVTo transformations. Hence, metrics belong to this category are related to operations;
- Expression metrics: the operations consist of multiple expressions, which can further be measured using metrics in this category;
- Occurrence metrics: the occurrence of an expression should also be measured, the metrics classified in this category solve this problem.

Table 4.2 presents all the metrics designed for measuring how imperative/declarative a QVTo transformation is.

4.2.2 Apply metrics

In this section, we apply the metrics proposed in Table 4.2 on those revisions of QVTo transformation that are analyzed in Section 4.1.1 and then compare the new results with those demonstrated in Section 4.1.1.

Table 4.2: List of metrics for measuring how imperative/declarative a QVTo transformation is

Category	Metric	Description
Operation metrics	#Mappings which contain VariableInitExp	The number of Mappings which contain one or more variable initial expressions
	#Constructors which contain VariableInitExp	The number of Constructors which contain one or more variable initial expressions
	#Helpers which contain VariableInitExp	The number of Helpers which contain one or more variable initial expressions
	#Queries which contain VariableInitExp	The number of Queries which contain one or more variable initial expressions
Expression metrics	Total #VariableInitExp in Mappings	The total number of variable initial expressions in Mappngs
	Avg. #VariableInitExp in Mappings	The average number of variable initial expressions in Mappngs
	Total #VariableInitExp in Constructors	The total number of variable initial expressions in Constructors
	Avg. #VariableInitExp in Constructors	The average number of variable initial expressions in Constructors
	Total #VariableInitExp in Helpers	The total number of variable initial expressions in Helpers
	Avg. #VariableInitExp in Helpers	The average number of variable initial expressions in Helpers
	Total #VariableInitExp in Queries	The total number of variable initial expressions in Queries
	Avg. #VariableInitExp in Queries	The average number of variable initial expressions in Queries
Occurrence metrics	Total #Occurrence of variables in Mapping	The sum of the times of all variables are used in Mappings
	Max #Occurrence of a variable in Mapping	The max of the times of all variables are used in Mappings

Metrics related to *Mapping*

Figure 4.3 shows the values of metrics related to *Mapping* in Table 4.2 extracted from the chosen 60 revisions of QVTo transformation. The values are demonstrated according to the commit time from earliest to latest. As can be seen, values of *Number of Mappings which contain VariableInitExp* and *Total number of VariableInitExp in Mappings* increase overall, which implies that the QVTo transformation becomes more imperative regarding *Mapping* operations. Meanwhile, the rise of *Total number of occurrence of variables in Mapping* over time indicates the *Mapping*s become more imperative during the evolution of the QVTo transformation.

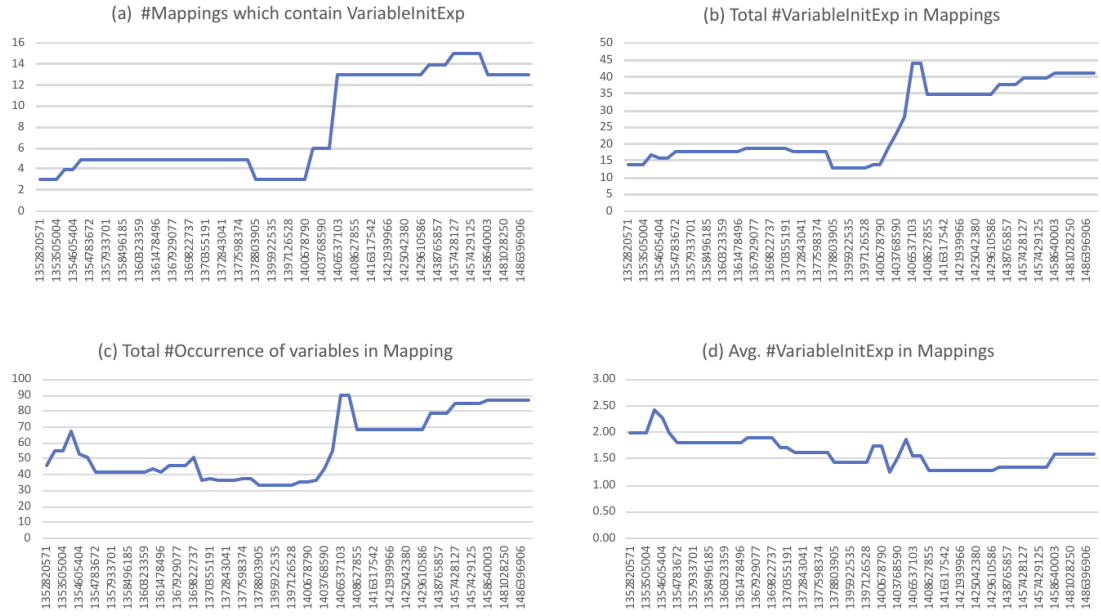


Figure 4.3: Evolution of metrics related to *Mapping*

Metrics related to *Helper*

On the other hand, Figure 4.4 presents the values of metrics related to *Helper* in Table 4.2. According to the decrease trend of values of metric *Number of Helpers which contain VariableInitExp* and *Total number of occurrence of variables in Mapping*, we can conclude that the QVTo transformation are less imperative in terms of *Helper* operations, while the overall decline of values of metric *Total number of occurrence of variables in Mapping* and *Average number of VariableInitExp in Helpers* indicates the *Helper* operations become less imperative along the evolution of the transformation.

It is notable that metrics related to *Mapping* indicates the QVTo transformation become more imperative, but metrics related to *Helper* draws the opposite conclusion. The evolution of *Mapping* operation are more important because they are the critical elements of QVTo transformation, we, therefore, focus on the evolution of *Mapping* which shows an overall growth of imperative programming. However, as discussed in Section 3.2, *Little imperative programming* can improve the understandability of QVTo transformation, hence the increase of imperative programming of the QVTo transformation results in a decline of understandability, which are not desirable. To improve the quality, we explore how to make the QVTo transformation more declarative in the next chapter.

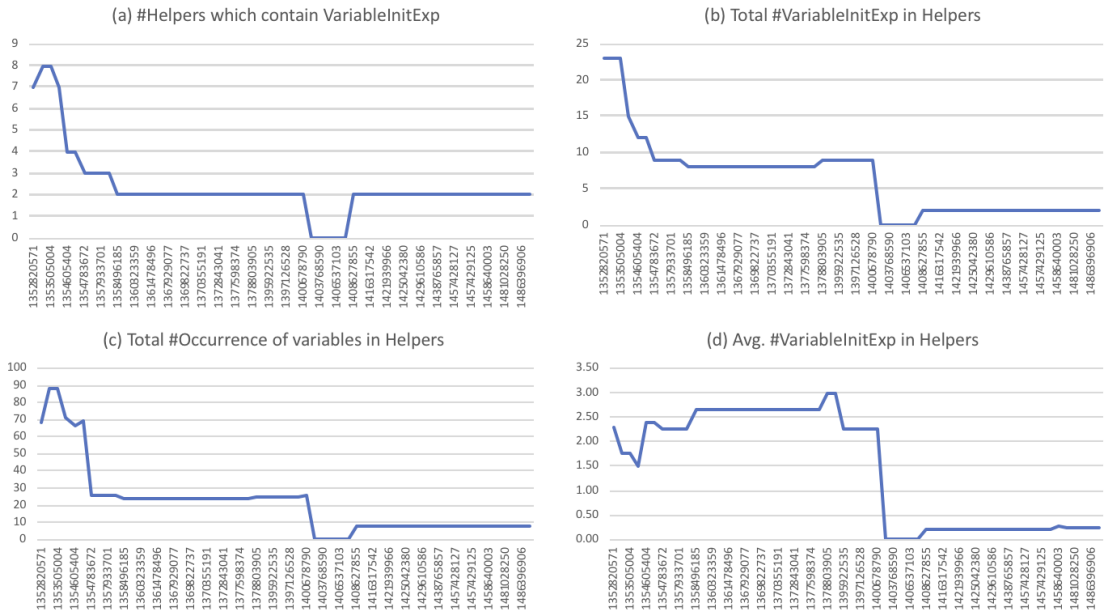


Figure 4.4: Evolution of metrics related to *Helper*

4.3 Conclusion

We investigate the quality of our QVTo transformation using the quality model provided by Gerpheide [11], and analyze the changes of QVTo transformation's quality over time by applying a set of metrics on 60 revisions of the *pgapp2dsgraph.qvto* file. Based on the results, we can observe that for understandability, which is the most ubiquitous quality goal suggested in Gerpheide's model, all the relevant metrics indicate a declining except metric *Number of forLoops*, which is used for indicating *Little imperative programming*. However, unlike *Little imperative programming* improving understandability is confirmed in the quality model, the relationship between metric *Number of forLoops* and *Little imperative programming* is not validated. Hence, we design a set of new metrics to measuring how imperative/declarative the QVTo programming is, and find that these novel metrics indicate our QVTo transformation becomes less declarative over time.

Chapter 5

Rewriting QVTo transformation

In this chapter, we propose a rewriting pattern to convert the imperative QVTo transformations to declarative ones and present the experiments where this proposed translate pattern is applied to three different revisions of the QVTo transformation.

5.1 Approach

The second research question of the thesis is to find a method to rewrite imperative QVTo transformations to declarative ones. As discussed in Section 3.3, there are various approaches to accomplish the rewriting task. Among these methods, the pattern-based method adopted in Cabot's research [7], which investigates rewriting of DSL (UML/OCL in their study), is the most suitable for our study. Hence, first of all, we identify the imperative parts which should be rewritten into declarative QVTo expression. Secondly, a list of patterns are proposed, where each pattern indicates the target imperative QVTo expressions and its corresponding declarative QVTo expressions. At last, the list of patterns are applied to the QVTo transformation.

5.1.1 List of target expressions

In Query/View/Transformation specification [4], OMG defines QVT Operational with two packages: *QVTOperational* and *ImperativeOCL*. Within the *ImperativeOCL* package, the imperative expressions are defined as the base for all side-effect oriented expressions in QVTo specification. The imperative expressions extend OCL expressions and allow expressing complex transformations in a comfortable way while in the meantime keep the functional features of OCL. Figure 5.1 shows the hierarchy of imperative expressions in the QVT specification.

As can be observed in the hierarchy tree, multiple classes inherit the *imperativeExpression* class. These classes are designed for various usages:

- **VariableInitExp** a *variable initial expression* declares a variable and possibly initialized it;
- **AssignExp** an *assignment expression* represents the assignment of a variable, a new value is assigned to a variable if the variable is mono-valued, otherwise the value the variable can be reset or appended based on whether the *isReset* property is true;
- **InstantiationExp** an *instantiation expression* creates an object of a class;
- **AssertExp** an *assert expression* checks if a condition is satisfied, and sends a message if the condition fails;

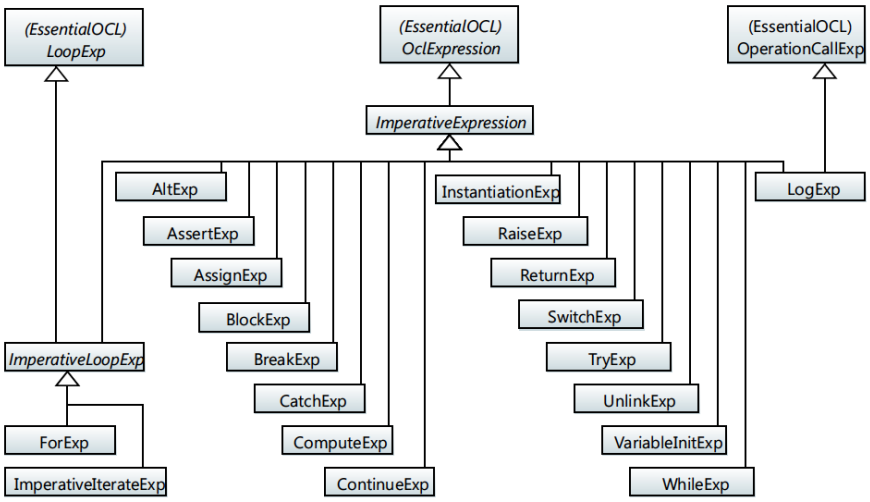


Figure 5.1: Imperative expressions hierarchy [4]

- **LogExp** a *log expression* prints a log to the environment;
- **ReturnExp** a *return expression* is the exit of an imperative operation, it assigns the value to the result of the imperative operation if a value is indicated;
- **ImperativeLoopExp** an *imperative loop* expression executes an imperative loop statement over a collection, the predefined constructors *forEach, forOne, xcollectselect, xcollectselectOne* are all defined based on the *imperative loop* expressions;
- **WhileExp** a *while* expression iterates until the condition is no longer satisfied;
- **ForExp** a *for* expression iterates over a collection for each element that satisfies a given condition;
- **SwitchExp** the *switch* expression executes alternatives according to the give conditions, it acts like the nested *if* expressions in OCL but can be interrupted by the control expressions;
- **BlockExp** *block expressions* execute a list of expressions sequentially, but the execution can be interrupted by control expressions, like *break expressions*;
- **ComputeExp** a *compute expression* is where a variable is declared and possibly initialized followed with a body that updates the variable. The return value of a *compute expression* is the variable value;
- **TryExp, CatchExp, and RaiseExp** these three expressions are used for exceptions;
- **ContinueExp and BreakExp** these two types of expressions are used within an iteration, a *continue expression* is used to jump to next iteration and a *break expression* is used to terminate the iteration.

By analyzing the usages of these classes, we can observe that using instances of certain two or more classes together could be imperative in most cases. For example, using a variable initial expression in an assignment expression is a typical imperative pattern in most scenarios. In the next section, we find more combinations of this kind and then propose them as the target imperative patterns.

5.1.2 List of rewriting patterns

The rewriting focus on *Variable Initial expressions* and *Assignment expressions* because they are the most frequently used expressions in QVTo transformations of CARM. These two types of QVTo expressions can form an imperative pattern with each other or with other expressions, like *AssertExp, LogExp*. Table 5.1 presents a list of rewriting patterns, the rewriting uses a simple method to replace a variable with its initialization expression if possible so that imperative patterns which related to *VariableInitExp* and *AssignExp* can be reduced.

Table 5.1: Rewriting patterns

No	Imperative expression	Rewritten declarative expression	Description
1	var a:= initExpression ; lhs := f(a) ;	lhs := f(initExpression) ;	When a variable is used in a function which is the right hand side expression of an assignments, replace the variable with its initial expression when the variable is declared.
2	var a:= initExpression ; assert(a) ;	assert(initExpression) ;	When a variable is used in an assertion operation, replace the variable with its initial expression when the variable is declared.
3	var a:= initExpression ; log(a) ;	log(initExpression) ;	When a variable is used in a log operation, replace the variable with its initial expression when the variable is declared.

5.1.3 Apply the rewriting patterns

Using the patterns listed in the previous section, a QVTo imperative operation can be rewritten into declarative operation. Listing 5.1 shows an example of an imperative mapping, where two *variable initial expression*s and two *assignment expressions* are defined within the *init* section. By applying pattern 1 in Table 5.1, the mapping is specified in declarative style as presented in Listing 5.2.

```
mapping Block::createBlockDependency(src: Block, con : Connection) : Dependency
{
  init {
      var target := self.resolveone(ds_graph::Task);
      var source := src.resolveone(ds_graph::Task);
      result := source.map createDependency(target);
      result.connection := con;
  }
}
```

Listing 5.1: Original imperative code

```
mapping Block::createBlockDependency(src : Block, con : Connection) : Dependency
{
  init {
      result := src.resolveone(ds_graph::Task).map createDependency(self.resolveone
          (ds_graph::Task));
      result.connection := con;
  }

}
```

Listing 5.2: Rewritten declarative code

5.2 Results and Analysis

The purpose of our experiments is to measure the changes on quality when a QVTo transformation is rewritten in a more declarative style. The following section describes the setup of experiments and presents the results, and then analyze the results.

5.2.1 Experiments Setup

The experiments are conducted with ASML code. First of all, we rewrite the latest revision of a specific QVTo transformation *pgapp2dsgraph*. When applying rewriting pattern to the QVTo transformation, it is necessary to validate whether the rewritten transformation is equivalent to the original one. In this experiment, two QVTo transformations are considered equivalent as long as they produce the same results when tested by the same test cases suite which contains 73 test cases.

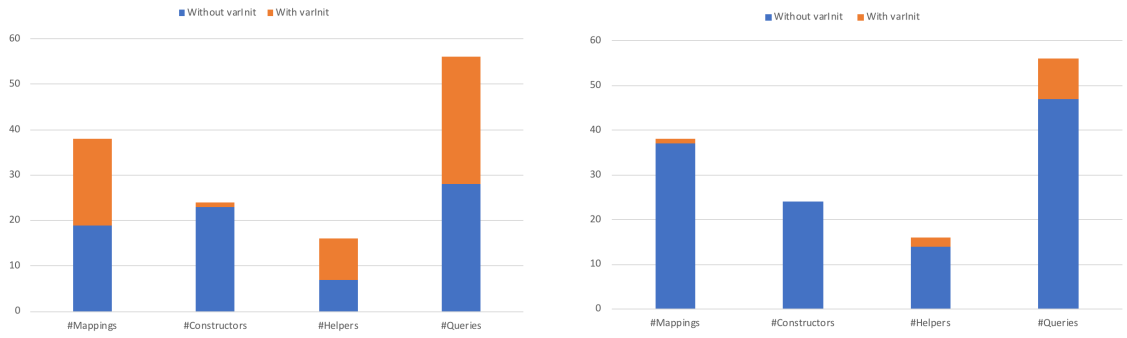
Secondly, a suite of metrics which are proposed in Section 3.1.3 is applied to the latest revision of these *.qvto* files. Furthermore, based on the observation in Section 4.2.2 that the metrics are changed significantly at two revisions (revision α and revision β), the measurement is also applied on these two revisions of our QVTo transformation. However, validating the functionality of an inactive revision of QVTo transformation is unnecessary. The experiments are carried out as the following steps:

- Step 1: Rewrite the QVTo transformation and its relevant Library *.qvto* files manually;
- Step 2: Validate the equivalence of the original QVTo transformation and the rewritten one;
- Step 3: Extract the value of metrics of both original QVTo transformation and the rewritten one with EMMA [23]
- Step 4: Execute Step 1 and Step 3 on revision α and revision β.

5.2.2 Imperative versus Declarative

Firstly, we try to figure out whether the rewriting effectively makes the QVTo transformations more declarative. Hence, we extract a series of values of our proposed metrics, which are used to measure how imperative/declarative the QVTo transformation is in our experiments. Figure 5.2 shows the results of the experiments on imperative operations level, where the blue portion of the bar represents the number of imperative operations (*Mappings, Constructors, Helps*, and *Queries*) which contains *variable initialization expression*. On the other hand, the orange portion represents the number of these imperative operations which does not have any *variable initialization expression* in it.

As can be seen in Figure 5.2 (a), for *Mapping, Helper, and Query*, more than half of these operations use *variable initialization expressions*. In contrast with the other three imperative operations, *Constructors* in our QVTo transformations only use one *variable initialization expression*. The major function of a *Constructor* is to initialize a given class, hence the body of a constructor typically consists of instantiated assignments, where introducing new variables are unnecessary. Figure 5.2 (b) shows the metrics value of rewritten QVTo transformations. As can be observed,

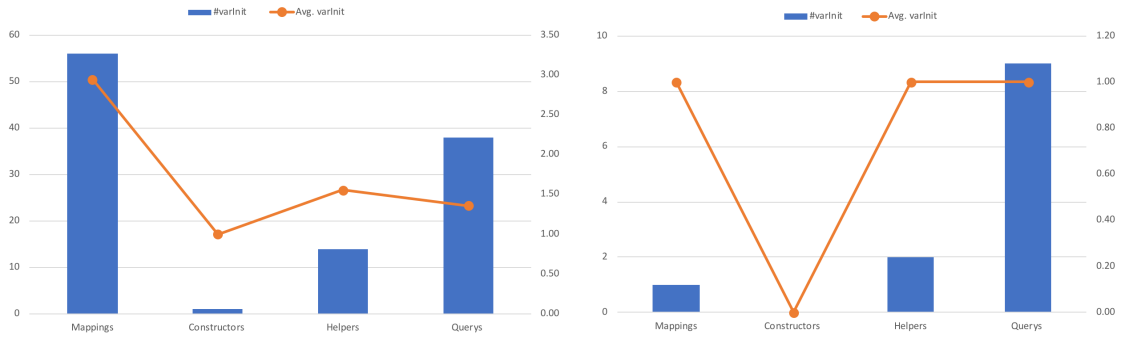


(a) Number of operations which contain *VarInitExp* in imperative transformations

(b) Number of operations which contain *VarInitExp* in declarative transformations

Figure 5.2: Comparison of operations using *variable initialization expression*s in imperative and rewritten declarative QVTo transformations

there are no *variable initialization expressions* in *Constructor*s and only one in *Mapping*s, two in *Helpers* and less than 10 in *Queries*. Comparing the two results in Figure 5.2 (a) and (b), it can be seen that the orange parts (also known as the imperative parts) are reduced significantly after the rewriting. Hence, we can conclude that our proposed imperative-declarative rewriting makes the QVTo transformation more declarative in terms of operations.



(a) Number of VarInitExp used in imperative transformations

(b) Number of VarInitExp used in declarative transformations

Figure 5.3: Comparison of *variable initialization expression* usage in imperative and rewritten declarative QVTo transformations

Moreover, in our experiments not only the imperative operations which contain *VariableInitExp* are investigated, but also these variable initial expressions themselves are explored. Figure 5.3 presents the results on expressions level, the blue bar indicates the total number of *variable initialization expressions* defined in operations, while the orange dots represent the average number. As Figure 5.3 (a) shows, *Mapping*s use more *variable initialization expressions* than other operations both in total and on average, which means *Mapping*s are the most imperative operations before rewriting. On the other hand, Figure 5.3 (b) illustrates that except for *Constructor*s which no longer contains any *variable initialization expression*, *Mapping, Helper, Query* all has the same average number of *variable initialization expression* usage. While for the total number of *variable initialization expressions*, there is a sharp decrease for all operations. As a result, *Queries* has the most *variable initialization expressions*. A comparison of Figure 5.3 (a) and (b) reveals that our rewriting makes operations of QVTo transformation more declarative, whilst the rewriting has the greatest impacts on *Mappings* than all the other operations. Overall, the observations can be concluded as follows:

1. The rewriting reduces the number of imperative operations; (Observation 1)
2. The rewriting reduces both the total and the average number of imperative expressions; (Observation 2)
3. The rewriting affects *Mapping*s more than other operations. (Observation 3)

To evaluate our rewriting, we apply the pattern to another two revisions of the QVTo transformation. Figure 5.4 (a) and (b) shows the comparison of operations using *variable initialization expression*s before and after rewriting revision α of the QVTo transformation, and Figure 5.4 (c) and (d) presents the comparison of *variable initialization expression* usage in these two QVTo transformations. Figure 5.5 shows the results for revision β.

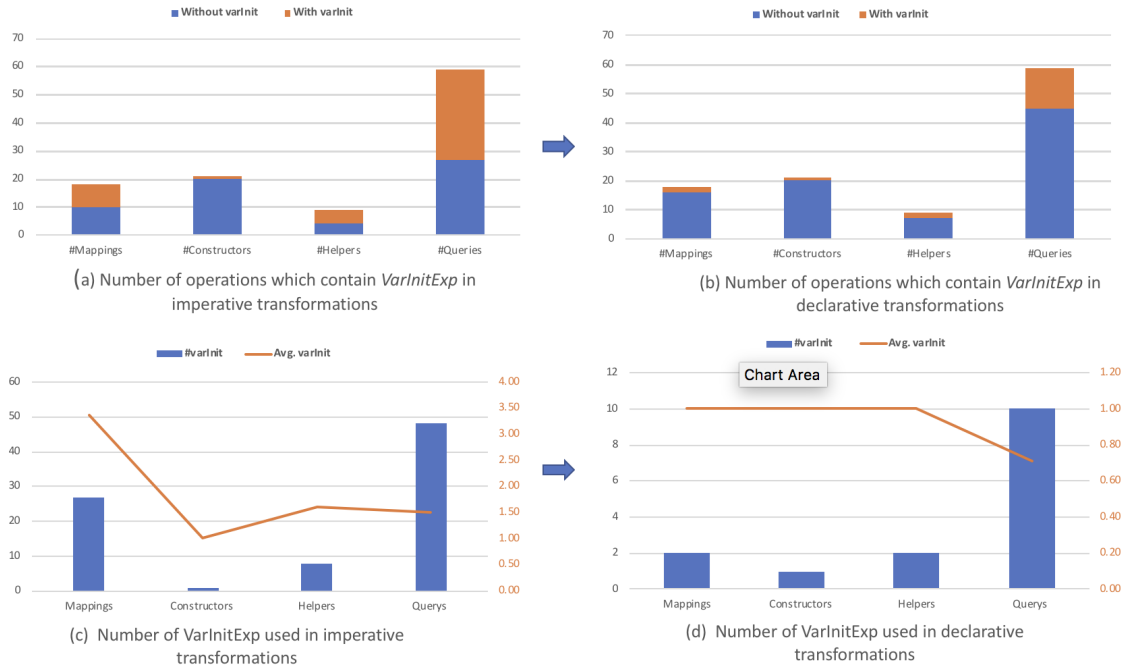


Figure 5.4: Evaluate rewriting patterns on revision α

As can be seen in Figure 5.4 and 5.5, we find that the major differences among revision α, β, and the latest revision is the total number of *Mapping* operations. Furthermore, we can check whether the previous observations hold for revision α and β. Table 5.2 shows that the first two observations hold for all three revisions, but *Observation 3* does not hold for revision α, due to its relatively small number of *Mapping* operations.

Table 5.2: Observation for revision α and β

Revision	Observation 1	Observation 2	Observation 3
α	Yes	Yes	No
β	Yes	Yes	Yes

Based on the previous discussion, we can conclude that our imperative-declarative rewriting makes the QVTo transformations more declarative because of the decrease in the number of imperative operations. Meanwhile, the rewriting also makes the operations more declarative since they contain less imperative expressions both in total and on average after rewriting.

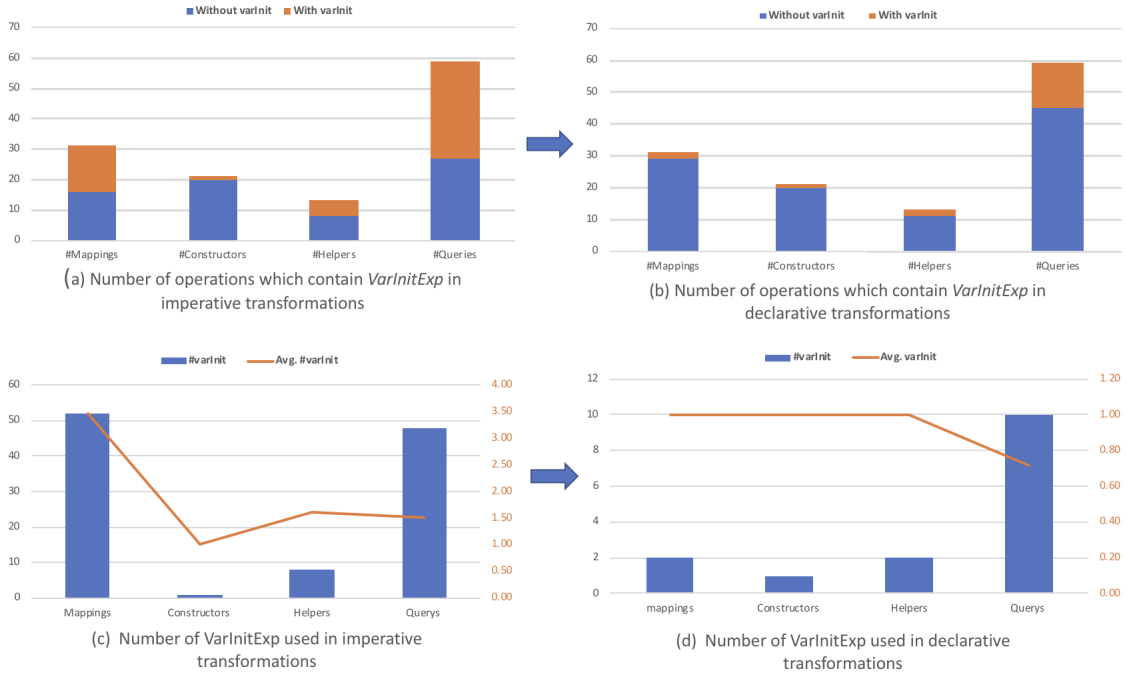


Figure 5.5: Evaluate rewriting patterns on revision β

5.2.3 Quality

The previous section shows that our rewriting can make the QVTo transformation more declarative. In this section, we explore how this rewriting affects the quality of the QVTo transformation.

Understandable

Because the metrics are designed to improve understandability, we firstly investigate how the imperative-declarative rewriting affect the understandability of QVTo transformations. We apply our metrics designed in the research of the first question to the QVTo transformation before and after rewriting. Furthermore, the metrics proposed as indicators for quality of QVTo in Gerpheide's research [11] are used as a comparison. Since *Mapping* operation plays a key role in a QVTo transformation, first of all, we explore how *Mapping*s' quality changes because of the rewriting. We extract values of metric *Size of init* , which is a metric measuring *Mapping* operation in Table 4.1, from 12 *Mapping*s in the latest revision of QVTo transformation *pgapp2dsgraph*. Moreover, our proposed metrics related to *Mapping* are applied as well. Figure 5.6 shows the results, where the blue bars refer to the metrics before rewriting (imperative) and the orange bars represent the ones after rewriting (declarative).

We can draw the following conclusions from Figure 5.6:

- The average number of both metrics from Gerpheide's and our proposed metrics dropped after rewriting;
- The orange box (the values after rewriting) is lower than the blue one (the values before rewriting);
- The values of our proposed metrics are reduced to zero after rewriting;

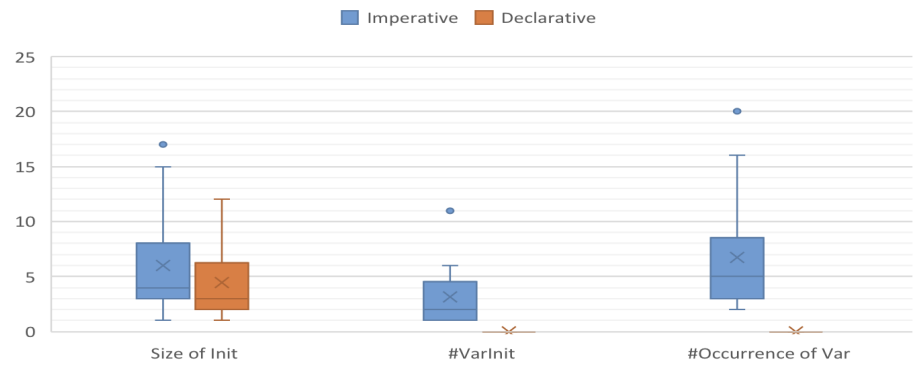


Figure 5.6: The metrics of understandability

- The blue box of metric *Number of VariableInitExp* is lower and comparatively short than that of metric *Size of init*.

Overall, we can observe that size of **init** section in *Mapping* reduces and considering Gerpheide's model states that smaller size of **init** section improves the understandability of QVTo transformations, we can draw the conclusion that our rewriting makes the QVTo transformation more understandable in this perspective.

Analyzable

Multiple tools have been developed to analyze or assess the quality of software. For example, Alloy Analyzer [14] is a widely used and discussed tool to ensure the quality of modeling by finding structures to satisfy all the constraints of a model. If models and model transformation are analyzed with Alloy Analyzer , they need to be represented by declarative language, because of the declarative nature of Alloy [16], which is the core of Alloy Analyzer. Based on the discussion in Section 5.2.2, we draw a conclusion that the QVTo transformation become more declarative after rewriting. Hence, it is easier to analyze the QVTo transformation using Alloy Analyzer.

5.3 Conclusion

In this chapter, inspired by methods proposed for rewriting JAVA or UML/OCL, we design a rewriting pattern to convert the QVTo transformation into more declarative one. Based on the analysis using metrics presented in the previous chapter, we find that our rewriting makes the QVTo transformation more declarative by reducing the imperative parts in the transformations. Besides, the QVTo transformation becomes more understandable according to Gerpheide's quality model and easier to be analyzed by Alloy Analyzer.

Chapter 6

Conclusions

Our main contribution in this thesis to provide insight into software metrics (especially for QVTo transformations) and imperative/declarative rewriting. We design a suite of metrics to measure how imperative/declarative a QVTo transformation is and quantitatively analyze the quality evolution of our target QVTo transformation with metrics. Additionally, we propose a rewriting pattern to convert imperative QVTo transformations into declarative ones so that the QVTo transformations become more understandable. In this chapter, we present the major conclusions we draw from the research and mention some ideas for future work.

In this thesis, we first check whether the QVTo transformations conform to Lehman's Law, and then try to improve the quality of QVTo transformations. The first research question is formulated as follows:

RQ 1: *How does the quality of QVTo transformations evolve over time?*

Conclusion for RQ1 We analyze the quality evolution of our QVTo transformation by collecting values of a series of metrics proposed in Gerpheide's model [11]. We find that the trend of all metrics is declining over time except for the metrics which is designed to indicate how imperative/declarative a QVTo transformation is. Hence, a set of novel metrics are presented and applied to the QVTo transformations to get new values, we further observe that the trend of the new proposed metrics is consistent with the other metrics' in Gerpheide's model, which is our QVTo transformation becomes more imperative and less understandable over time. Additionally, it is notable that this trend also conforms to the rule of *Continuing Change, Continuing Growth and Declining Quality* in Lehman's Law.

To tackle the problem of quality declining, we try to find methods to make the QVTo transformation more declarative. The second research question is stated as follows:

RQ 2: *Can we rewrite the QVTo transformations to make them more declarative, and how does the rewriting of QVTo transformations affect their quality?*

Conclusion for RQ2 In the research of the second question, we rewrite our QVTo transformation according to the pattern we described. To validate whether the QVTo transformation is more declarative after rewriting, we applied the metrics designed in the research of the first question. These metrics are applied to three major revision of our QVTo transformation to collect the values. The results show that our rewriting pattern makes the QVTo transformation more declarative. Because Gerpheide's model suggests that quality attribute *Little imperative programming* improves the understandability, we can also conclude that the QVTo transformation becomes more understandable after rewriting. Additionally, the rewriting is helpful when the QVTo transformation is

analyzed by some tool (such as Alloy) that only suitable for declarative language.

Future work Firstly, we manually rewrite the imperative QVTo transformation into declarative one, which can be further automatically done. Secondly, more rules can be added so that more imperative parts can be rewritten, and then the QVTo transformation can be fully declarative without any imperative patterns. At last, declarative QVTo transformations can be more easily analyzed with Alloy [15], hence representing QVTo transformations with Alloy and analyzing QVTo transformations with Alloy Analyzer [14] can be done in future work.

Bibliography

[1] ISO/IEC 205010:2011. https://www.iso.org/standard/35733.html, 2011. ix, 8, 9

[2] ISO/IEC 25020:2007. https://www.iso.org/standard/35744.html, 2007. ix, 1, 10

[3] Thomas J McCabe and Charles W Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1989. ix, 14, 15

[4] QVT Omg. Meta object facility (mof) 2.0 query/view/transformation specification. *Final Adopted Specification (November 2005)*, 2008. ix, ix, 6, 8, 18, 21, 33, 34

[5] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In *Software metrics symposium, 1997. proceedings., fourth international*, pages 20–32. IEEE, 1997. xi, 11

[6] Tom Mens and Serge Demeyer. Future trends in software evolution metrics. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 83–86. ACM, 2001. xi, 21, 22

[7] Jordi Cabot. From declarative to imperative uml/ocl operation specifications. In *International Conference on Conceptual Modeling*, pages 198–213. Springer, 2007. xi, 23, 24, 33

[8] Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman Fenton. Towards a framework for software measurement validation. *IEEE Transactions on software Engineering*, 21(12):929–944, 1995. 1, 10

[9] Victor R Basili. Software modeling and measurement: the goal/question/metric paradigm. Technical report, 1992. 1, 11

[10] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. 1

[11] Gerpheide Christine. Assessing and improving quality in QVTo model transformations. Master's thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2014. 1, 18, 19, 25, 31, 39, 41

[12] Ramon RH Schiffelers, Wilbert Alberts, and Jeroen PM Voeten. Model-based specification, analysis and synthesis of servo controllers for lithoscanners. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, pages 55–60. ACM, 2012. 1, 2

[13] Phu hong Nguyen. Quantitative analysis of model transformations. Master's thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2010. 1, 18, 19, 20

[14] Alloy. http://alloytools.org/, 2018. 2, 40, 42

[15] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002. 2, 42

[16] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M Küster. Analysis of model transformations via alloy. In *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007. 2, 40

[17] OMG. Unified modeling language (omg uml). *Superstructure*, 2007. 2, 15, 24

[18] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008. 2

[19] Eclipse QVT Operational. `https://projects.eclipse.org/projects/modeling.mmt.qvt-oml`, 2018. 2

[20] Robert W Sebesta and Soumen Mukherjee. *Concepts of programming languages*, volume 8. Addison-Wesley Reading, Massachusetts, 1999. 5

[21] John W Lloyd. Practical advtanages of declarative programming. In *GULP-PRODE (1)*, pages 18–30, 1994. 5

[22] ISO/IEC 9126-1:2001. `https://www.iso.org/standard/22749.html`, 2001. 9

[23] Josh GM Mengerink, Alexander Serebrenik, Ramon RH Schiffelers, and Mark GJ van den Brand. Automated analyses of model-driven artifacts. 2017. 13, 36

[24] Ian Sommerville et al. *Software engineering*. Addison-wesley, 2007. 13

[25] Barry W Boehm et al. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981. 13

[26] IEEE. Standard for Software Productivity Metrics (IEEE Std 1045 1992). *The Institute of Electrical and Electronics Engineers, Inc)*, 1993. 14

[27] Robert E Park. Software size measurement: A framework for counting source statements. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1992. 14

[28] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A SLOC counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16, 2007. 14

[29] Maurice H Halstead. Natural laws controlling algorithm structure? *ACM Sigplan Notices*, 7(2):19–26, 1972. 14

[30] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976. 14

[31] Sallie Henry, Dennis Kafura, and Kathy Harris. On the relationships among three software metrics. *ACM SIGMETRICS Performance Evaluation Review*, 10(1):81–88, 1981. 14, 15

[32] Grady Booch. *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991. 15, 24

[33] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William E Lorensen, et al. *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991. 15, 24

[34] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004. 15, 24

[35] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994. 16

[36] Lucia Kapová, Thomas Goldschmidt, Steffen Becker, and Jörg Henss. Evaluating maintainability with code metrics for model-to-model transformations. In *International Conference on the Quality of Software Architectures*, pages 151–166. Springer, 2010. 18, 19, 20

[37] Marcel Van Amstel, Steven Bosems, Ivan Kurtev, and Luís Ferreira Pires. Performance in model transformations: experiments with ATL and QVT. In *International Conference on Theory and Practice of Model Transformations*, pages 198–212. Springer, 2011. 18, 19, 20

[38] MF Van Amstel, CFJ Lange, and MGJ van den Brand. Metrics for analyzing the quality of model transformations. In *12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering*, 2008. 19

[39] Motoshi Saeki. Embedding metrics into information systems development methods: An application of method engineering technique. In *International Conference on Advanced Information Systems Engineering*, pages 374–389. Springer, 2003. 18

[40] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the software quality of android applications along their evolution (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 236–247. IEEE, 2015. 21

[41] Nicholas Drouin, Mourad Badri, and Fadel Touré. Analyzing software quality evolution using metrics: An empirical study on open source software. *Journal of software*, 8(10):2462–2473, 2013. 21

[42] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 543–553. ACM, 2013. 23

[43] Cosmin Radoi, Stephen J Fink, Rodric Rabbah, and Manu Sridharan. Translating imperative code to mapreduce. In *ACM SIGPLAN Notices*, volume 49, pages 909–927. ACM, 2014. 23

[44] Jesús Sánchez Cuadrado. Optimising ocl synthesized code. In *European Conference on Modelling Foundations and Applications*, pages 28–45. Springer, 2018. 23

[45] The Java Tutorial, Lamda expression. `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`, 7/19/2016. 23