Language Evolution to Reduce Code Cloning

by

Marko Novakovic

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Domain-specific languages can significantly speed up the development of software applications. However, it usually takes a few iterations of the language design before it achieves such power. At the same time, many domains tend to evolve quite often today, which implies that domain-specific languages have to evolve accordingly. Thus, being able to evolve a language in a painless manner is crucial. Unfortunately, current state-of-the-art research does not provide enough answers on how to efficiently evolve domain-specific languages.

We present an approach to evolving a language in order to reduce the amount of code cloning it introduces. The approach specifically targets those languages whose design causes users to create many duplicated code segments.

We target domain-specific languages as they tend to be more challenging to evolve due to their specifics, but the approach may be applicable to general purpose programming languages as well. The approach was tested on a real-world domain-specific language that is used in a financial domain. We proposed three improvements and current users helped us evaluate them. We found that the proposed improvements would reduce code cloning, which provides evidence that the approach can be used in a real-world environment.

Furthermore, this work provides a solid basis for further research in the area of application of code cloning detection results. In particular, code cloning detection results and the ideas we presented show potential to be extended and used to facilitate domain analysis.

Acknowledgements

I would like to express my gratitude to everyone who was involved in any aspect of this thesis.

First of all, I would like to thank my supervisor, Professor Krzysztof Czarnecki, for accepting me to be his student and providing advices and guidance throughout the past two years.

I would also like to thank the thesis readers, Professors Patrick Lam and Paul Ward, for their valuable comments and feedback.

Furthermore, I would like to express my sincere gratitude to Logicblox employees for providing feedback on my ideas and finding time to answer the questions that I had.

I will also use this opportunity to thank my colleagues from the Generative Software Development Lab for making our lab a very enjoyable place, and for providing lots of useful feedback in many conversations we had.

And finally, I am infinitely grateful to my girlfriend Jelena who unconditionally supported me and encouraged to work hard during the past three years.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Research Contributions 4

1.2 Thesis organization 4

2 Background 5

2.1 Code Cloning 5

2.2 Domain Specific Languages 7

2.3 Merchandise Financial Planning 8

2.4 The Measure Language 12

3 Our language evolution approach 16

3.1 On Evolution in DSLs 16

3.2 Overview of the approach 17

3.3 Phase 1: Detect causes of code cloning 18

3.4 Phase 2: Code clone analysis 18

3.4.1 Detecting specific clone types that occur. 18

3.4.2 Measuring the importance of clone types. 21

3.5 Phase 3: Improving the language based on code cloning results. 21

4 Case Study: Code Cloning Detection **24**
4.1 The motivating example . . . 24
4.2 Introduction . . . 25
4.3 Results: relevant types of cloning . . . 26
4.3.1 Expression structure clones . . . 26
4.3.2 Corresponding metrics clones . . . 28
4.4 Importance Analysis Results . . . 28
4.4.1 Expression structure clones . . . 29
4.4.2 Corresponding metrics clones . . . 33
4.5 Implementation . . . 36

5 Case Study: Proposed improvements **37**
5.1 Improvements that target expression structure clones . . . 38
5.1.1 Domain-specific concepts as embedded functions . . . 39
5.1.2 Evaluation . . . 40
5.2 Improvements that target corresponding metrics clones . . . 40
5.2.1 Macros that accept measure components as parameters . . . 41
5.2.2 Evaluation . . . 43
5.2.3 Anonymous functions . . . 43
5.2.4 Description of the anonymous functions improvement . . . 44
5.2.5 Evaluation . . . 46

6 Related Work **47**
6.1 Code cloning . . . 47
6.1.1 Code cloning detection . . . 47
6.2 Language Design and Evolution . . . 49

7 Discussion and Future Work **50**

7.1 Discussion 50

7.2 Future Work 51

7.2.1 Extended Domain Analysis 51

7.2.2 New Measure Language as internal DSL 52

7.2.3 More Case Studies 53

8 Conclusion **54**

References **56**

List of Tables

4.1 Empirical results regarding the expression structure clones. 30
4.2 Empirical results regarding the same corresponding metrics clones. 33

List of Figures

1.1 Code cloning example from a real-world application. 2

2.1 Hierarchical ordering of roles in the Food Basics example. 9
2.2 Time and Location Dimensions with examples of concrete Dimension Levels. 10
2.3 Example of a Measure Language. 12
2.4 Example of a Measure written in Measure Language. 13

3.1 The overview of proposed methodology 19

4.1 Motivating code cloning example where expressions share many commonalities. 25
4.2 An example of expression structure clones. 27
4.3 Corresponding metrics clones: expressions have the same expression structure and corresponding metrics. 29
4.4 Distribution of frequency of occurrence for expression structure patterns. . 32
4.5 The algorithm for calculating the value of *Clone Severity*. 34
4.6 Distribution of clone severity for each of the three applications 35
4.7 Toolchain in the detection process. 36

5.1 An example of incorporating domain-specific concepts into language 39
5.2 Instances of the same corresponding metrics clone pattern. 41
5.3 Example of declaration and usages of macros. 42
5.4 Example of anonymous functions syntax: we first specify a template, and then provide a list of concrete parameters. 45

Chapter 1

Introduction

With the constant increase of usage of software in many systems and domains, together with growing demands of the industry, non-software professionals inevitably face the need to use software in ways that push its limits. These advanced requirements are mostly reflected through a need to specialize the software for specific customers. This type of work is traditionally done by software professionals, but due to the requirement for domain-specific knowledge, the professionals must collaborate with domain-experts. This poses burdens of knowledge transfer and extensive labor-resource usage. One of the answers to this problem is creation of domain specific languages (DSLs), which aim to provide the power of traditional programming languages that are specialized for a particular domain and at the same time easy to use by people who are not software professionals.

Domain specific languages, although quite useful, are not perfect. Many compromises must be made to make the syntax and semantics understandable and usable by domain-experts, which in turn might produce unwanted consequences. For example, powerful constructs used in traditional programming languages could be removed from domain-specific ones to make them less confusing and easier to learn, which might result in non-optimal code. Also, the domain and requirements for the language evolve, which implies that the language should be changed accordingly. Changing the language to adopt it to a new environment and/or to improve it is referred to as **evolution**. Evolution is very common for domain-specific languages and is extensively studied in the research community [12, 16, 8, 9].

A reason that can trigger a desire for evolving a language is code duplication. For example, if applications written in a certain language have many similar code segments the understandability might be reduced, so the users want to change such behavior. In situa-

```
RTLGM_OP_MP_APCMSUPL_AMP
        = RTLGM_OP_M_APCMSUPL_AMP * 100 / RTLSLS_OP_R_APCMSUPL_AMP
RTLGM_OP_MP_APCMSCPL_AMP
        = RTLGM_OP_M_APCMSCPL_AMP * 100 / RTLSLS_OP_R_APCMSCPL_AMP
RTLGM_CP_MP_APCMSUPL_AMP
        = RTLGM_CP_M_APCMSUPL_AMP * 100 / RTLSLS_CP_R_APCMSUPL_AMP
RTLGM_WP_MP_APCMSUPL_AMP
        = RTLGM_WP_M_APCMSUPL_AMP * 100 / RTLSLS_WP_R_APCMSUPL_AMP
```

Figure 1.1: Code cloning example from a real-world application.

tions when the language design is the cause of the code duplication, evolving a language is a solution. There are many reasons why code duplication can occur in applications written in a domain-specific language: one of them might be unexperienced or lazy users that do not want or do not know how to use abstraction mechanisms provided by the language; the domain for which the application is written might require many similar pieces of functionality, which creates a natural tendency to have similar pieces of code; and finally, a cause of code duplication that is of particular interest to us is the language design. For example, in order to make the language non-programmer friendly, the language designers have removed abstraction mechanisms that exist in other programming languages, which might result in many code segments that look very similar.

An excerpt from a real-world language whose design is the cause of code duplication (due to lack of abstraction mechanisms) is given in Figure 1.1. The language from the figure is created by our collaborators, and it represents a custom built DSL used to define arithmetic-like expressions which represent constraints in a merchandise financial planning domain [1]. The language is very specialized and is easy to be used by business consultants who do not have a computer science background. However, a drawback is code duplication: many arithmetic expressions look very similar and are created by copying and pasting existing expressions. By just looking at the code from the figure (without having to know the semantics of the language), we can spot that these expressions are almost identical. We did a survey with one group of language users, and we did informal interviews with another; their subjective opinion is that code duplication makes the language harder to read and prolongs the development process. Overall, there is a strong desire among the users to reduce code duplication causes.

However, although there exists work on domain-specific languages evolution in the literature, such work is very limited. Researchers have proposed some ideas to facilitate the evolution, but such work usually requires following certain steps/tools/ideas from the

initial stage of the language development, or targets the evolution of languages created using specific technology like UML [9, 26]. Thus, existing work is not applicable to cases where we want to evolve existing languages regardless of the technology used to create it. Evolving a DSL is, therefore, a broad unexplored area and current research does not provide enough solutions for evolving existing languages.

To better understand what challenges occur with language evolution we have decided to try to evolve the language presented in Figure 1.1, which is called the *measure language*. In particular, we have decided to understand the challenges from evolving a language to remove specific problem - code duplication (i.e. code cloning). Our initial review of the literature provided a useful background but no concrete steps or knowledge that we could directly apply to the problem. The concrete ideas that we have tried were based on common-sense and our experience, but we could not find an ad-hoc solution that worked. We gathered the following conclusions in our attempt to evolve the measure language:

- **Better understanding of code cloning that occurs in specific language is needed**. Although we had no doubt that code cloning occurs, it was obvious that to make any progress we needed to better understand what kinds of code cloning occur.
- **Code cloning detection is harder for domain-specific languages than for general purpose languages**. There is a lot of research on code cloning that explains code cloning approaches, results, and techniques. Also, a certain level of agreement on existing clone types exists [20]. However, such work covers traditional languages like Java or C in the vast majority of cases. Domain-specific languages—as they are specific—in many instances have the syntax and proposed ways of usages that are different than those found in traditional languages, so the existing work is often not directly applicable.
- **The connection between code cloning results and language improvements is not straightforward.** Although the goal is to reduce further occurrences of code cloning, given the specific types of code cloning that occur, it is not obvious what steps should be taken.

We did extensive research and experimentation to find a working approach that solves the mentioned problems and presented the results in this thesis. A generalized approach for evolving a language whose design makes it prone to code cloning is applied to the measure language as a case study, and proposed improvements are evaluated with current users of the measure language, which provides us with evidence that our approach can be applied in a real-world environment.

1.1 Research Contributions

This thesis introduces the following novel contributions:

- **An approach to language evolution as a mean of code cloning reduction.** The approach—which is the main contribution of this thesis—gives guidance on how to evolve a language in case the language design causes users to do code cloning. The goal of the approach is to reduce code cloning in new applications written in the improved version of the language.
- **A methodology for detecting code clones in domain specific languages.** We provide a list of ideas for suggesting code cloning types and a guidance on how to use them effectively to detect specific clone types that occur in a domain-specific language

1.2 Thesis organization

This thesis is organized as follows. Chapter 2 introduces concepts that we use throughout the thesis like code cloning and domain specific languages, with special focus on Merchandise Financial Planning domain, for which the language from our case study is created. The approach on how to evolve a language in order to reduce code cloning (in case the language design is the core cause of code cloning), which is the core part of this work, is presented in chapter 3. The case study is presented in chapters 4 and 5. Chapter 7 discusses potential ideas for future work that arise from the presented research, whereas chapter 6 presents an overview of the related work. Chapter 8 summarizes the work we did for this thesis and concludes it.

Chapter 2

Background

This chapter introduces the areas and the terminology used throughout the thesis. The first part of this chapter (sections 2.1 and 2.2) introduces code cloning and domain-specific languages, which are the main concepts we target in this thesis. The second part (sections 2.3 and 2.4) introduces the domain for which the language we analyze in our case study is created, together with a brief overview of the language.

2.1 Code Cloning

For the purpose of this thesis, we will use the following definition of code clones, as given by Dexter et al [19] : ***Clones are segments of code that are similar according to some definition of similarity.***

This definition is opened to interpretation as there can be more than one definition of similarity. Also, there are more terms that are related to code cloning. For example, *code duplication* is often used instead, or together with code cloning. Some researchers consider code duplication to be similar pieces of code made by a process of duplicating code (like copy/paste) [4], whereas some consider code duplication to be equivalent of code cloning regardless of the underlying reasons of similarity [19]. The research community has not reached a consensus on a definition on what code cloning and code duplications are, because it depends on the detection technique used, target clone type, programming language, or the problem one is trying to solve by code clone (or duplication) detection [20]. In the thesis, we will use both terms, and for us code duplication and code cloning are equivalent. Over the next few paragraphs we explain concepts relevant to code cloning in more details.

Code cloning is generally considered harmful for various reasons. For example, an evidence shows that it reduces maintainability and reliability [17], and when cloned code contain bugs, the bug is replicated in multiple places [3]. On the other hand, some researchers claim that having code clones is not necessarily bad [11].

Detection techniques. There are different techniques when it comes to clone detection, and they vary in many aspects: some techniques are better suited for certain programming languages, some are more precise but less scalable. Choosing the right technique depends on the situation at hand. A good overview of existing techniques is given by Roy et al in [20], and comparison of existing clone detection techniques and tools, along with the discussion about the results is done by Shafieian and Zou in [22]. Here we present each of the techniques mentioned in [20] and [22], and give a very short description of each:

- **Text-based** technique considers a program as pure text and finds similar pieces of text.
- **Lexical/Token-based** technique transforms a program into a series of lexical tokens using compiler-style lexical analysis as a starting point, and does various operations on top of it.
- **Syntactic** technique uses the AST representation of a program as a starting point and performs various operations on trees to detect code clones. **We use this approach in our case study.**
- **Semantic** technique tries to detect whether different pieces of code have similar functionality. Some techniques use the Program-Dependency Graph [5] as a base.
- **Hybrid approach.** Combines other techniques in various ways.

Clone types. Cordy and Roy [21] summarize the types of cloning that are currently used in the research community. First, a more general classification distinguishes two types of clones: **syntactic or textual** clones, which are similar pieces of code, i.e. the text representing the application code, and **semantic or functional** clones which represent two pieces of code with the same or similar functionality, regardless of the similarity of textual representation of the code. A short summary of the classification from their paper is given in the following list. First three types are variations of syntactic/textual clone types, whereas the last type is just a synonym for the semantic clone type.

- **Type I.** Two pieces of code are identical, or there is a variation in whitespace, code layout and comments.

- **Type II.** On top of variations allowed for Type I, we allow variations in types, variable names and values of literals.
- **Type III.** This includes variations from Type II, plus variations in statements - we allow statements to be added or removed from both pieces of code.
- **Type IV.** This is a synonym for the semantic/functional cloning type.

These types are also known under different names in the community, for example Type I clones are sometimes referred to as *Exact* Clones, Type II as *Near Miss* clones. More fine-grained classification exists as well. For example, *parametrized* clones refer to pieces of code where literals and variable names are changed in a consistent way. An example of a consistent change would be refactoring in terms of changing a variable name – all occurrences of the variable are properly renamed. On the other hand, in *renamed* clones, the renaming does not have to be consistent.

2.2 Domain Specific Languages

As with code cloning, a globally accepted definition of what a domain-specific language is does not exist. We will use the definition from [27]:

> *"A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."*

Domain-specific languages are created for solving specific problems, or a set of problems from a specific domain and they usually target a narrow community from the corresponding domain. This strategy differs from that of languages like Java or C++, which are General Purpose Languages (GPLs) created for a broad community and as a means to solve broad spectrum of problems.

Two main types of DSLs (explained in [16]) are relevant to this thesis:

- **Embedded or Internal.** These languages are created as a part of a GPL (which is called a *host* language). An example of such a language is a Java library that has a very user-friendly set of methods tailored for solving a very specific set of problems,

like the one introduced in [7]. Some languages, like Scala, have very flexible syntax which allows users to create embedded languages which differ substantially from the host language [23].

- **External.** External DSLs are created independently and have their own set of tools for compilation or interpretation. Due to their independence, they have greater flexibility and power when it comes to syntax or semantics, but are more costly to develop.

Both approaches have pros and cons. When creating a new DSL, the right decision on what approach is more appropriate is specific to the goals of the language and the constraints from the environment in which the language will execute. The final decision will have to be a trade-off.

The approach on evolving a language based on code cloning detection results that we present as the main contribution of this thesis targets both types of DSL. However, the case study applies to an external DSL, which will be explained in the section 2.4.

2.3 Merchandise Financial Planning

Merchandise Financial Planning is a systematic approach to planing financial aspects of a merchandise-related business. The main purpose is to plan the business activities that will lead the company to fulfill a financial goal, which is usually expressed as a concrete profit in some currency.

The financial goal. The financial goal that a company wants to achieve is very concrete and is represented with a concrete value that should be reached, for example: *Profit for the current fiscal year should be 10% greater than the profit achieved in the previous fiscal year.* There are other possible financial goals: decreasing costs by some amount (regardless of profit), or increasing the amount of sales (without taking profit into the account). In this context, the amount of sales is the total revenue of the company, whereas the profit is what is left after paying expenses like salaries, purchased goods etc. A plan can include any combination of these parameters.

The activities that could lead to a financial goal. A company can plan multiple activities to reach the goal. For retail, it can be the budget available to be spent on purchasing the goods, which is commonly called *Open To Buy*. This plan optionally includes the list of goods or the distribution of the budget for each of the goods. Also, a company

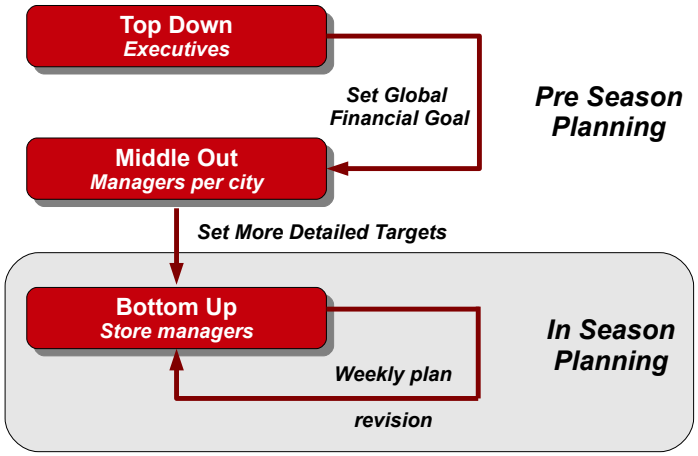

Figure 2.1: Hierarchical ordering of roles in the Food Basics example.

can plan a promotion, particularly a set of goods to put on sale in a certain period of time or under certain conditions, which might lead to increase in sales or profit. Some of the most commonly used financial aspects are Sales, Cost, Margin and Profit, and we refer to them as **Metrics**. The full set of metrics used when making a plan varies greatly among different companies and depends on the specifics of the environment in which the company does their business.

We will further explain the aspects of Merchandise Financial Planning (or MFP) relevant to this thesis by using an example. Let us consider a well known Canadian retailer, Food Basics, which is selling goods to end-users [1]. The company has many **stores** spread out across many **cities** in Ontario. It is selling different **types of goods**. Being a nationwide company, Food Basics has different types of management - executives, managers for each city in which it has presence and finally, each store has a manager.

Hierarchical planning: Roles. The overall process of planning is very hierarchical. This is best explained using an example, which is a good representative of a real-world process: executives for Food Basics set the annual financial goal, for example the profit (in CAD) that should be achieved at the end of the fiscal year. This plan does not consider the details of business at either monthly level or the city/store level. *Managers on lower hierarchical levels make more detailed plans, and the other way around.* Working with the

[1]Facts about the company are made up for the purpose of understanding the MFP domain and any similarity with the reality is purely incidental.

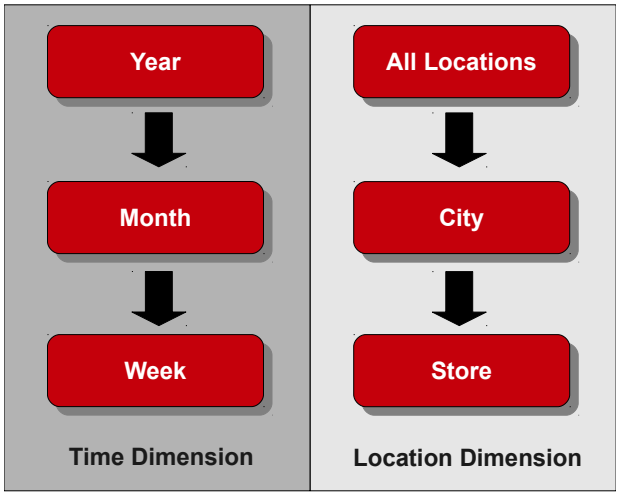

Figure 2.2: Time and Location Dimensions with examples of concrete Dimension Levels.

annual goal, managers at the city level make monthly plans for the city they are responsible for, such that the sum of profits per city reaches the annual global financial goal. Similarly, store managers make weekly plans on store levels.

Every hierarchical level is associated with its *Role*. The hierarchy depth varies by company - Figure 2.1 shows the usual setting with three roles (same as our Food Basics example). *Top Down* role represents the top hierarchical level in the company (i.e. executives), whereas *Bottom Up* role represents the lowest level of managers that do the planning. In the Food Basics example, managers at the city level would be in the *Middle Out* role while store managers would have the Bottom Up role.

Time of planning. The process of planing is not static - offer and demand change throughout the year and the plans are changed accordingly. For this purpose, there are two main types of planning - *Pre-Season* and *In-Season*, as shown in Figure 2.1. As the names suggest, Pre-Season is done before the season starts (which is usually the beginning of a fiscal year) and is done by Top-Down and Middle-Out roles. Pre-Season planning is based on the result of business in the previous season as well as predictions for the next season. In-Season planning is done during the season, by the Bottom-Up role, and it is a continuous process that results in detailed plans done usually once a week in order to adopt to the current state of the business.

Dimensions. Different roles operate with different levels of data. In the Food Basics example, we have mentioned two types of data which exist at multiple levels: (1) *Time* -

executives make annual plans, whereas store managers make weekly plans and (2) *Location* - store managers make plans for a particular store, and managers at the city level make plans for the city they are assigned to. In MFP, such multi-level data is called a **Dimension**, with concrete levels called **Dimension Levels**. The most common dimensions are Time, Location and Product but there can be more or less dimensions depending on the application. Figure 2.2 shows examples of concrete Dimension Levels for Time and Location Dimensions. Dimension Levels are associated with a Dimension and are hierarchically organized within it. We can represent this as an undirected graph with dimension levels as nodes with edges representing parent-child hierarchical relations. The nodes in two graphs in Figure 2.2 have at most one child per node but in general these graphs can have cycles and nodes can have more than one child.

Aggregation and spreading. Data on lower dimension levels **aggregates to** data on higher levels. For example, stores keep track of sales for each *week* in 2012. If we consider a single store and we want to know the amount of sales for *month* September, we will sum the values for all weeks in September to get that value. We call this operation aggregation. In this case, Week aggregates to Month. The reverse operation is called **spreading** - data on higher dimension levels spread to data on lower levels.

Versions. In practice, planners use values from past years when making plans, usually the values for the last year and two years ago. These are examples of two *versions* of values. For example, if the goal is to increase the amount of sales that was achieved last year by 10 %, planers will take the amount achieved last year (in CAD) and will multiply it by 10 % to get the goal for this year (also in CAD). Such values of metrics for past years, and the current year, are called *Versions*.

Similarly, since there are more hierarchical levels included in the planning, it often happens that managers on higher level have to approve the plan of a manager on lower level before the plan becomes official. For this purpose, there is a working plan, which is used before it gets approved and there is an approved plan, or target which is an approved version. In MFP domain, these types of plans are said to have different Versions.

Measures. Having covered some of the basic concepts in the MFP domain, we have enough background to introduce **Measures**, a concept that is among the most important ones in the MFP domain:

> ***Measure*** *is value of a* ***Metric*** *in certain* ***Unit of Measure*** *for specific* ***Version***, ***Role*** *and a set of* ***Dimension Levels***.

```
1  rule "SALES_WP_PL_CAD" {
2    formula "SALES_WP_PL_CAD" {
3      SALES_WP_PL_CAD[week, store] =
4        PROFIT_WP_PL_CAD[week, store] + COST_WP_PL_CAD[week, store]
5    }
6
7    formula "SALES_WP_EX_CAD_inverse" {
8      PROFIT_WP_PL_CAD[week, store] =
9        SALES_WP_PL_CAD[week, store] − COST_WP_PL_CAD[week, store]
10   }
11 }
```

Figure 2.3: Example of a Measure Language.

An example of metric would be: value of *Sales* in *CAD* for *Last Year* and from perspective of a *Planner*. Measure is a very precise specification of data since the value it presents is constrained by many of the concepts we have mentioned. Metric, Unit of Measure, Version and Role are sometimes called **components** of a measure. Measure is the basic building block for planning. For example, the financial goal that we have mentioned at the beginning of the section is also specified as a measure, rather than just a metric.

2.4 The Measure Language

The values of some measures are directly related. For example, in a retail company the result of planning can be the quantity of goods to be purchased for the next fiscal year. This amount has impact on the company's spending (i.e. Cost), whereas the amount of goods sold has impact on the sales, and accordingly the amount of sales has an impact on company's profit.

In the Merchandise Financial Planning domain those relations are called *Rules*, and they play a very important part in the process of planning. The rules can be represented as equations in terms of arithmetic expressions. An example of a very simple and often used rule is:

Profit = Sales - Cost

The interpretation of this rule is very straightforward - if we have the values for Sales and Cost, the value of Profit would be calculated using the expression. Note that this constraint does not always hold, if we have the value for Profit and Cost, we cannot use this rule to calculate the value of Sales. Equality signs resemble implication: given the values on the RHS of the expression, we calculate the value on the LHS. We will elaborate more about the semantics later.

Measure Language is a declarative DSL used to specify measures, dimensions and rules for the purpose of Merchandise Financial Planning. In the next few paragraphs, we will introduce and explain relevant parts of the language. For this purpose, we will refer to the excerpt from a real-world application in Figure 2.3.

Each measure used for planning has a corresponding variable of float type in the Measure Language. The variable name is very descriptive - it is a concatenation of names of all components that a measure consists of, separated by "_". Figure 2.4 shows an example of a measure that represents a value of Sales in a Planner's Working Plan for week and store dimension levels. From the example one can tell that the Working Plan version is represented by WP, Planner role by PL, Sales Metric as SALES and Canadian Dollars Unit of measure with CAD. The naming of components and concatenation order of components in measure name is consistent within an application written in Measure Language. The main usage of measures is to specify rules.

Rules are specified using the **rule** keyword. Each rule has a name and one or more **formulas**. Formulas contain one or more equations with a measure on the left-hand side of the equation sign and an arithmetic expression on the right-hand side. A program written in the measure language is a set of rules which describe the relations among measures (i.e. values that measures represent).

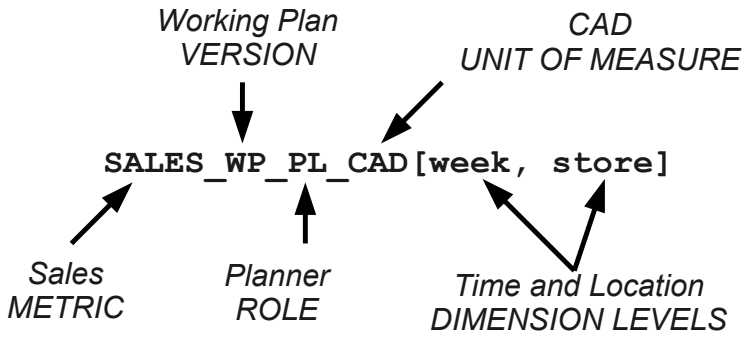

Figure 2.4: Example of a Measure written in Measure Language.

Embedded functions Although the rules mostly contain arithmetic expressions in which variables represent measures, some expressions use embedded functions provided by the measure language. The most relevant embedded function, which we will mention later in the thesis, is the the *total* function. The *total* function aggregates values from lower levels of dimensions, which we can use to, for example, calculate sales on month level given sales on week level.

The semantics of the language The set of rules in the program (i.e. the expressions in its formulas) does not act like a set of constraints. Rather, the rules are evaluated once a value of some measure changes. Once the value of a measure is changed, the interpreter will search for all the rules containing the changed measure in the right-hand side of expression in one of its formulas. Then, assuming there is only one such formula per rule, the right-hand sides will be evaluated and measures on the left-hand sides will be changed. This changed set of measures will trigger the interpretation process again and it will be repeated until no more measures are changed. Note that the system will not allow rules that introduce cycles (otherwise the process would run forever given the way it works, i.e. without fix point calculation). In case there is a rule with more than one formula that contain the changed measure in the right-hand side, only the first formula will be considered. This implies that the order in which formulas are specified within a rule matters.

The semantics we just explained can be illustrated using the rule in Figure 2.3. We have two formulas: one declaring how to calculate the value of Sales, and the other one showing how to calculate the value of Profit. Let's consider three situations:

1. The value of PROFIT_WP_PL_CAD changes. The interpreter finds that there is only one formula with this measure in the right hand side - SALES_WP_PL_CAD. The value of SALES_WP_PL_CAD is recalculated using the expression in the RHS of this formula and the process stops since there are no more rules with SALES_WP_PL_CAD in the RHS of its formulas (besides the starting rule but since it is already evaluated it will not be considered again).

2. The value of COST_WP_PL_CAD is changed. There is one rule with two formulas that contain this measure in their RHS. As mentioned earlier, the order of formulas in situation like this one matters, so only the first formula - SALES_WP_PL_CAD - will be evaluated. The rest of evaluation is executed in a way already explained.

3. The value of SALES_WP_PL_CAD is changed. In this case, the only formula that

will be considered is SALES_WP_EX_CAD_inverse. The evaluation is done in already explained manner.

Our case study that we explain in chapters 4 and 5 is done on applications that use the Measure Language.

Chapter 3

Our language evolution approach

As we have mentioned, the design of certain programming languages causes users to do code cloning. In this chapter we present an approach that can be used to improve such languages by removing the code cloning occurrences. We target domain-specific languages because they are more likely to cause code cloning: they are often less mature, and thus less polished than general purpose languages, and one of the consequences can be code cloning; they might have simpler abstraction mechanisms to make them usable by domain experts that are not programmers. They are also far more likely to be changed more often than general purpose languages: the domains evolve and the languages have to be adopted; domain-specific languages care more about the usability and thus the language has to be changed often in order to improve it.

3.1 On Evolution in DSLs

The evolution of a DSL can be roughly defined as changing the language semantics or syntax to adopt it to a changed environment or new requirements. Domain-specific languages evolve quite often in practice [26, 27].

There are multiple reasons DSLs evolve. One of them is domain evolution. Domains may evolve due to changes in laws or new technologies. Similarly, for extensively used languages, there might be requests for new features: for example, many users of the language might want to start using it for novel and unforeseen purposes. Furthermore, domain-specific languages are often developed in an iterative way, where the language is evolved in each iteration.

A possible reason for evolution of a language is existence of *code smell* ([14]), which is defined by Kent Beck in [6] as *a problem in code which is usually an indication of a much bigger problem in the system.* In certain situations, the code smell occurs as a result of a language design. For example, due to lack of abstraction mechanisms in the language (which might have not been perceived as necessary when the language was created initially) it might not be possible to extract common code and reuse it, and thus the users end up copy/pasting code.

The main contribution of this thesis is an approach that can be used in case a code smell, particularly code cloning (introduced in section 2.1), occurs. The rest of this chapter starts with an overview of the approach and is followed by detailed explanations of its main components.

3.2 Overview of the approach

As we have mentioned already, the design of a programming language might cause code cloning. The approach that we describe gives steps and advices that can be applied to improve such languages, so that they cause smaller amount of code cloning in new applications. It is consisted of three phases which are shown in Figure 3.1, and we give a brief introduction of the approach in the next paragraph.

The first thing we have to do is to investigate whether code cloning occurs and if it does to determine whether the language design is a cause. An indication that it is the case is when the analyses of language usages report code duplication, or in case the users themselves report extensive occurrences of code duplication. The next step, i.e. Phase 2 (which is applied only once we are certain that code cloning is caused by the language design) is to determine what kinds of clones exist in applications written in the target language, how often they occur, and what percentage of code is being taken by each clone type. The last step (Phase 3) is to improve the language with a goal of reducing or removing occurrences of previously discovered clone types.

We created this approach by: (1) experimenting and learning from real-world experience as explained in the case study, (2) reading existing literature in the related areas, and finally (3) brainstorming, particularly about how to generalize the lessons learned from experiments and how to apply ideas that we either found in the literature or that made sense to be applied.

The rest of the chapter explains the details of all three phases of the approach.

3.3 Phase 1: Detect causes of code cloning

The goal of this phase is to make sure that code cloning occurs and is caused by the language design. As we have mentioned earlier, code cloning can exists due to the domain itself, unexperienced users, following bad practices, etc. The idea is to apply common sense reasoning to make sure this is the case before we proceed. This can be done in few ways: by talking to experts and asking them questions related to the code cloning and causes of it; also, manually inspecting the code can give us better understanding whether and why the code cloning occurs. In case code cloning occurs and the language design is the cause of it, we can proceed.

Recall that an indication that our approach can be applied is when users report some kind of code cloning. In our case study, the users have noticed that they do a lot of copy/pasting and they have created various ad-hoc mechanisms in order to speed-up that process. This is an undesirable practice and the users are aware of that. The basic intuition behind the idea is that in cases the language design causes substantial amount of any kinds of code smells [14, 6] it is noticeable and transparent.

The hardest and the most important part is the part that follows: determining the exact types of cloning and finding the solutions to reduce it, which is what the goal of the approach that we propose is.

3.4 Phase 2: Code clone analysis

The first step towards the improved language is understanding the code cloning that occurs in the target DSL. For us, the understanding means (1) detecting clone types that occur, and (2) measuring the importance of each clone type. These two steps, along with an explanation of what the importance of a code clone is are described in the following two subsections.

3.4.1 Detecting specific clone types that occur.

The main goal of this step is to detect the concrete types of cloning that occur in usages of the target DSL. Possible cloning types could be some of the generally known clone types described in section 2.1. These categories are too general in order to come with language improvements, however. The analysis needs more precise characterization of clone types. Since we target DSLs, the identified types might not be generalizable beyond

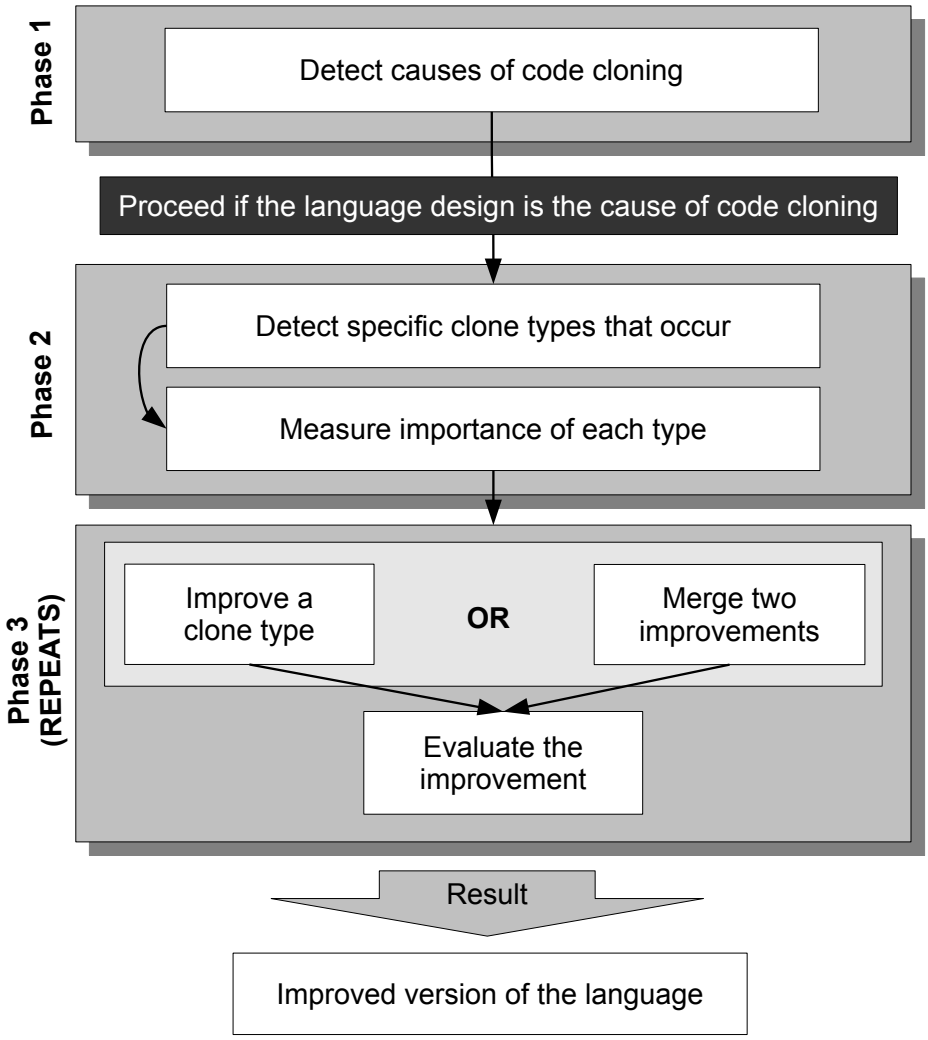

Figure 3.1: The overview of proposed methodology

the particular DSL at hand. These are the reasons why detecting clone types is DSLs is not a straightforward task, but rather a process that requires effort and creativity.

We used different sources of ideas that helped us propose possible clone types in the analysis of the language described in the case study. The sources, together with explanations of how to use each of them is given in the following list:

1. The classification from section 2.1 – a globally recognized clone types classification.

This is a good starting point, and no matter how different the DSLs are, the classification can be customized and applied to it.

2. User input. Users of the language might have insights into common types of duplication that occur, since they write or read the code as a part of their daily job. Such feedback is very important and can facilitate the search.

3. Custom proposed types specific to the target DSL. This can be done by manual inspection of the code and looking for similar code fragments or by running existing text or token-based clone-detection tools and then analyzing the results.

Finding clone types. We propose an iterative process for finding clone types, where each iteration is consisted of using an idea source to propose a clone type. At each step, it is verified whether such clone type occurs in usages of the target DSL. If it does, we save the specification of the clone type (since it will be used in the next phase) and we repeat the process. The process is stopped when no more clone types that occur can be found. Note that this is a subjective decision, i.e. there does not exist a tool or an algorithm that can guarantee to find all possible clone types - it is up to the experience and common sense of the person or a group of a people who work on the analysis to decide when it is reasonable to claim that all the clone types have been found. At the end of this iterative process we are left with a list of clone types that exist in the usages of the target language.

An important part is verifying whether a proposed cloning type occurs and how often it occurs. To do this we propose creating an automated clone type checker (or a tool/set of scripts that produce similar results). This checker should have two inputs: the specification of a clone type and usages of the target DSL. The result of the verification step is a report of occurrences of the concrete clone type.

An example of one iteration of the process is as follows: a user of the language presents examples of a duplication, and that input is customized (by the person applying the approach) with the results of manual inspection in order to create a custom cloning type. We use the specification of such cloning type and a verifier to check whether the cloning type occurs in the applications that use the language.

From our experience many proposed clone types will prove to either not occur at all or occur quite rarely. For example, we proposed few clone types that seemed natural for the language we analyzed but found only one or two instances of them.

Once we have determined existing code cloning types, we classify them by importance, as explained in the next subsection.

3.4.2 Measuring the importance of clone types.

A relevant property of a clone type is *importance*. We define the importance of a clone type using two other properties: (1) the *frequency of its occurrence* and (2) *size*, which can be measured in different ways, for example by the number of lines in an instance of that particular clone type.

Intuitively, if a clone type occurs often and its instances are large (e.g. take many lines of code), it is important. Of course, this definition is overly general; particular thresholds for size and frequency are specific to each DSL. For example, let us consider a language in which we measure the size of code clones by the number of lines their instances take. If instances of particular clone type take up 70 % of the analyzed code base written in this language, and each instance is 20 lines of code, we would probably classify that clone type as highly important.

Determining thresholds for size and frequency is not straightforward. For example there might be a clone type that occurs quite often but takes only one line of code, and there might be a clone type whose instances are large, but does not occur often. Depending on the circumstances, we might consider both of them as more or less important. Determining what it means for a clone type to be important is an iterative process that will likely require few modifications in parameters until a satisfactory result is obtained.

This step requires a tool that can do automatic analysis of source code. It can be an extension of the tool described in the subsection 3.4.1 which only checks whether a particular type of cloning occurs, with added support for determining the size and frequency of clones.

3.5 Phase 3: Improving the language based on code cloning results.

Once we have detected clone types that occur in the target DSL, and once we know the importance of each clone, we can apply the methodology explained in this section. A simplified version of the idea that we propose can be described as:

Propose language improvements that target reduction of code cloning of specific type and evaluate each improvement with the language users. Discard improvements that do not provide positive evaluation results. Make new proposals by merging language changes introduced by different proposals and repeat the evaluation process. Make all possible combinations of merges and keep the improvement that gives the best overall results in terms of code cloning reduction.

We distinguish two types of improvements: *basic improvements*, specifically targeted to reduce code cloning of particular types and *merged improvements* - which are a result of merging of two improvements of any type. Merging two improvements means incorporating language changes proposed by both. For example, one improvement might propose adding ability to create modules in the language, and the other one to add a new operator. Merged improvement would be a language that would contain both the ability to create modules and the new operator.

Some improvements are not compatible. For example, one improvement may require changing specific concepts in the language, whereas another one may require removing them. Sometimes they are partially compatible and require additional changes to incorporate benefits of both improvements. Thus, merging two improvements might not be a straightforward process: it might be impossible or it might require substantial additional work.

Each proposed improvement requires *evaluation* to verify whether changes would introduce benefits or not. In other words, evaluation means measuring the effect of an improvement. It has *quantitative* and *qualitative* components. Quantitative component refers to the amount of code duplication that will be removed if we introduce the improvement. For example, the improvement could reduce occurrences of particular cloning types by half. Qualitative component refers to subjective opinions about the improvement: would users find it easier to write or understand the programs written in the improved version of the language? An improvement can be very effective in its qualitative component yet very hard to use. The right balance is specific to the target language and the goals we want to accomplish with the improvements.

The proposed methodology is straightforward when only one clone type occurs in usages of target language: we would propose an improvement that reduces that clone type and evaluate it. If the improvement turns out to be good, we would keep it, otherwise we would discard it. However, usually more types of cloning occur, and thus this process has to be **iterative**, as shown in Figure 3.1. We should make improvements for each clone type, and then try to merge different improvements, with evaluation at each step. The best

improvement would be determined after we apply all possible merges, which is exponential in the number of different clone types. Making all possible combinations of merges is infeasible and certain **heuristics** should be applied for particular case. For example, it might be enough just to reduce the occurrence of the cloning type that occurs most often.

The next two chapters give details about the case study, which we used both to shape the approach presented in this chapter and to apply it.

Chapter 4

Case Study: Code Cloning Detection

We have applied the approach described in chapter 3 on evolving the measure language, introduced in section 2.4. The measure language case study is done by analyzing the language usage in three real-world applications. This chapter gives a detailed overview of how we applied the code cloning detection phase (i.e., phase two) to the case study.

4.1 The motivating example

The best way to illustrate the code cloning problem we investigated is to present a concrete example Consider the code in Figure 4.1, which contains two rules (the concept we have already introduced in section 2.4). Both rules show how to calculate the percentage of sales made online (WEBSALES) in the total sales - it is done by dividing the CAD value of online sales by the value of total sales in CAD. The only difference between the two rules is that the first rule calculates the value for Target version and Executive role, whereas the other one is used for Forecast version and Manager role (roles and versions are shortened and represented using two to three letters, which is often done in the measure language).

The semantic similarity between the rules is strong, but syntactic similarity is more obvious. Given the way names of measures are created (by concatenating the names of its components), the two rules are almost identical. Examples of this kind of code similarity in usages of the measure language are numerous. The purpose of the approach we present in this thesis is to help us find clones like the one just explained and to provide improvements to the language in order to remove or reduce its occurrences. This chapter provides the details of the clone types we have found in usages of the measure language.

```
rule "WEBSALES_TG_P_EX" {
   formula "WEBSALES_|\emph{TG}|_P_EX" {
      WEBSALES_TG_P_EX = WEBSALES_TG_CAD_EX / TOTALSALES_TG_CAD_EX
        where TOTALSALES_TG_CAD_EX != 0.
   }
}
rule "WEBSALES_FC_P_MG" {
   formula "WEBSALES_FC_P_MG" {
      WEBSALES_FC_P_MG = WEBSALES_FC_CAD_MG / TOTALSALES_FC_CAD_MG
        where TOTALSALES_FC_CAD_MG != 0.
   }
}
```

Figure 4.1: Motivating code cloning example where expressions share many commonalities.

4.2 Introduction

Following the proposed approach from Figure 3.1, the goal was to find the most important code clone types. As explained earlier, this process consists of a few steps: proposing the cloning types we believe occur in the code, determining whether they actually occur, and, finally, measuring their importance. We have applied these steps in our case study.

We used a few sources to propose possible clone types by following the suggestions in chapter 3. For each proposed cloning type we created code to automatically measure how often it occurs, and, based on the results of such analysis, we decided whether certain types of clones are worth investigating further. We list the methodologies and sources used for suggesting potential cloning types:

Users feedback. As it all started with users complaining about having to do copy/paste, their feedback was a very valuable input. We took a look at examples of expressions that they copy/pasted and came up with a few potential cloning types.

Manual inspection. We manually investigated the code and looked for similar code fragments. The intuition was that if some code fragments can be spotted to be very similar by just looking at it, chances are that they will occur quite often. This observation proved to be true. We also used users feedback as a guideline on what kinds of clones to seek.

Existing literature on code cloning. We used existing literature on code cloning types to give us ideas what to look for - best summarized in [21]. Although the types listed in [21] are mostly created for imperative, general-purpose programming languages, it gave us a good idea on how to formulate equivalent or similar cloning types for the Measure Language.

We have used these ideas to generate many clone types and we explain the results in the next section.

4.3 Results: relevant types of cloning

Many of the clone types we thought existed do not exist. We have proposed more than ten different clone types but it turned out that only two of them occur often enough to be considered for further investigation. The main criterion for selecting these two clone types was their *frequency of occurrence*, i.e., the percentage of code taken by the instances of each clone type. We have also taken into account the nature of instances. Described reasoning for selecting criteria is in correlation with the approach described in chapter 3. The next sections provide more details on these two clone types and the implementations specifics of the criteria we used.

4.3.1 Expression structure clones

Expression structure is an abstraction over expression in which we ignore the specifics of measures, literals, and dimensions and keep the embedded functions and the order of operators. The example in Figure 4.2 will be used to better illustrate the details. The top rule shows how to calculate the percentage part of web sales in total sales, whereas the bottom one shows how to calculate percentage of regular sales in total sales. However, if we replace the measures with a, b, and c (in the order of their appearance from left to right) and the literal 0 with L, the two expressions are the same: $a = b \ / \ c \ where \ c \ ! = L$ and thus their expression structure is the same. If two expressions have the same expression structure, we consider them to be *expression structure clones*.

A reasonable assumption is that not all expressions in the application have the same structure. In the analysis, we first determine all different expression structures that occur, which we call *expression structure patterns*, and then we assign each expression to the corresponding pattern. That is, an application has multiple *expression structure patterns*

```
WEBSALES_TG_P = WEBSALES_TG_CAD / TOTALSALES_TG_CAD
        where TOTALSALES_TG_CAD != 0.


______________ = _______________ / _______________
        where ________________ != 0.


REGCOST_FC_P  = REGCOST_FC_CAD / TOTALCOST_FC_CAD
        where TOTALCOST_FC_CAD != 0.
```

Figure 4.2: An example of expression structure clones.

and each pattern has one or more expressions associated with it. We call those expressions *instances* of the particular expression structure pattern.

This clone type was of particular interest since the number of expression structure patterns is very small compared to the total number of expressions. This was the case with all the applications we analyzed and the concrete results will be reported in section 4.4. The results of our analyses show that some expression structure patterns are quite common and some of them are very well known in the domain. The measure language does not have any means of accelerating the creation of rules that have the same structure; we will use this fact in later sections when proposing the improvements to the language.

4.3.2 Corresponding metrics clones

We define two expressions to be *corresponding metrics clones* if they share: (1) the expression structure, and (2) metrics in corresponding measures. Dimensions are ignored for this classification. Figure 4.3 gives illustrative explanation of two expressions that satisfy these conditions. Both expressions have the same expression structure, and green color highlights the corresponding metrics, which are also the same: TOTALSALES, WEBSALES, and REGSALES (Regular Sales). Similarly, both expressions in the original cloning example in Figure 4.1 are corresponding metric clones.

Each expression in the application belongs to a certain **corresponding metrics clones pattern**, which is defined by the expression structure and a set of corresponding metrics in the expression. In case two expressions share the same expression structure and corresponding metrics, they are **instances** of the same corresponding metrics clones pattern.

This classification groups expressions and rules possibly created by copy/paste operations: we group expressions together if we suspect that one of them can be created by copy/pasting and then modifying versions/roles/uoms in the other. By interacting with the users of the measure language we have learned that many times they actually use this kind of copy/paste methodology to create new rules and expressions, which is considered to be a bad practice. However, there are good reasons why the users go for it - many rules and expressions are quite similar. The two rules in Figure 4.1 can be easily created from each other by copy/pasting and changing the versions and roles. The main cause of this is the way measure language is created: it doesn't take into account that there will be many expressions that share components.

It can be noted that two expressions can have more similarities than just the corresponding metrics; they can have the same corresponding versions and/or roles/uoms. Expressions in Figure 4.3 also have the same corresponding UOMs. We do analysis for each of the combinations (of which there is $2^3 = 8$, the number of elements in the power set of {Version, Role, UOM}), but we report only the same metrics situation, since that is still a valid corresponding metrics clone but it reports the biggest amount of cloning (which is expected since other variants are subsets of this one).

4.4 Importance Analysis Results

We analyzed usages of the measure language to understand the importance and specifics of the two mentioned clone types. The results and the analysis of the results are given in the next two subsections.

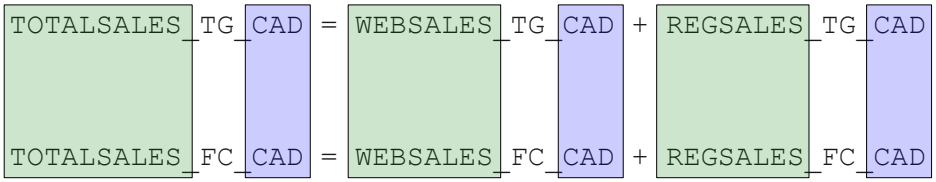

Figure 4.3: Corresponding metrics clones: expressions have the same expression structure and corresponding metrics.

4.4.1 Expression structure clones

To determine the importance of this clone type we analyzed each application individually. We considered the main component of the importance to be the frequency of occurrence; we have also analyzed the nature of instances of this clone type to make sure they are not too simple and deserve further investigation.

The frequency of occurrence of this clone type is defined as the percentage of expressions for which there exists at least one more expression with the same expression structure, (We have mentioned earlier that if two expressions have the same expression structure, they are considered to be expression structure clones.) This implies that in case an expression has a unique structure in the application, it will not be considered a clone. If we use the terminology defined in subsection 4.3.1, the frequency of occurrence is the percentage of expressions that are instances of those expression structure clone patterns that have at least two instances.

We did an additional check to make sure that these instances deserve further investigation. For example, there can be 100 expressions in the application and 50 different expression structure clone patterns which are distributed such that each pattern has two instances. This situation would imply that there are many patterns but they are duplicated only once, which would make the process of improving such duplication harder. For our purposes, the situation where the number of patterns is small compared to the total number of expressions would be ideal. The results described in the next paragraph show that this is the case.

The results for the described criteria for each of the three applications we analyzed can be found in Table 4.1. The first column in the table shows the total number of expressions in the application; the second column shows the frequency of occurrence and, as we can see, the substantial percentage of expressions have a non-unique expression structure; the last column shows the number of patterns with 2+ instances - the number of patterns is very small compared to the number of expressions, which means that many expressions

	Total number of expressions	Frequency of occurrence	Number of patterns with 2+ instances
App H	295	98.3 %	29
App T	659	98.5 %	25
App C	1896	99.9 %	43

Table 4.1: Empirical results regarding the expression structure clones.

have the same structure. The results show that the users of the measure language quite often create expressions that resemble some expressions that have already been created before. And yet, the measure language does not contain any means to speed up creation of the common patterns.

The presented results show us that this clone type occurs frequently and is worth investigating. However, since our goal is to provide language improvements, we need to better understand instances of this clone type. Better understanding this clone type has a goal to provide answers to the questions similar to the following ones: are the instances of expression structure clone patterns distributed evenly or do some patterns contain a majority of instances? Are the patterns with the most instances shared among different applications? What kind of patterns contain the most instances? Our improvements that target reduction of code duplication will be different depending on the answers to those questions. The second analysis has a goal to provide those answers.

For each of the reported patterns we further analyzed how many instances it has and the complexity of those instances (i.e, the complexity of instances that represent each pattern). The complexity of an expression is correlated to the number of leaf nodes in the AST representation of an expression, or, simply, a numeric value correlated to the number of measures, embedded functions, and integer literals in it. Intuitively, more complex expressions contain more measures, embedded functions, and/or literals. The procedure for calculating the complexity is simple: we traverse the expression and increase the complexity each time we encounter a leaf node or an embedded function. Each measure or integer literal increases the complexity by 1, and each embedded function by 3. The reason we value embedded functions more is because embedded functions usually contain more than one measure in it, and thus contain at least 2 pieces of information, or 3 on average. For example, if we have the following pattern: '$m1 = m2 + m3$', the number of measures in it (and the complexity) is 3. Similarly, the complexity of '$m1 = m2 + m3 + 1$' is 4 and the complexity of '$m1 = 0$' is 2. When it comes to embedded functions, as we mentioned earlier, the complexity of '$m1 = total[m2]$' is 4.

Since our goal is to provide improvements to the language, we decided to focus on

patterns that occur often (i.e, have many instances) and are of complex structure. The reasoning is simple: we want to make improvements that impact many expressions and thus it is better to target improving those patterns that occur often; we also want to improve the way people represent expressions, which is impossible if expressions are simple: for example, expression $m1 = m2 + m3$ is already represented in a very concise way and it is unlikely that it can be represented in a simpler way.

Particular thresholds are as follows: we considered only the top five occurring patterns among those patterns whose instances have complexity greater than 3. The threshold values are determined by manually looking at the patterns and their distribution. We set the complexity threshold to > 3 because patterns like $m1 = m2 + m3$ are too simple (as already mentioned). The threshold for frequency is set to the top five because the patterns below that limit occur (on average) in less than 2.5 % of expressions; thus, even if we find a way to improve their creation it will be a minor overall impact. The next few paragraphs discuss our findings regarding the investigation of interesting patterns. We have analyzed them and found that they represent domain-specific concepts.

Frequencies of patterns satisfying these two requirements are given in Figure 4.4. The chart represents the frequency distribution for the top five frequent patterns that have complexity greater than 3. The results for each application are presented in its own chart. For the purpose of the chart, the frequency of occurrence (Y axis) is defined as the number of instances of a particular expression structure pattern divided by the total number of expressions in a particular application. The important thing to mention is that the patterns from the chart are either instances of domain-specific concepts that will be discussed in the next section or instances of *total* embedded function discussed in section 2.4. Few patterns, and those are the ones that occur in less than 5% of expressions, are not discussed since they are very application-specific.

Domain-specific concepts as patterns. There are very few patterns that we perceived as "dangerous" because most of them are either not complex enough or occur infrequently. That is why we were able to manually look at all the expressions and investigate deeper whether those patterns group expressions around the same purpose. We additionally talked to the domain experts and we were able to identify that some of these patterns actually represent domain-specific concepts. Particularly, such patterns occur most frequently in each of the applications.

Two domain specific concepts are represented as patterns:

- *Variance* can be represented with the following pattern: $m1 = (m2/m3)$ - 1 *where* $m3! = 0$. It represents the percentage of increase of value $m2$ over value $m3$. For

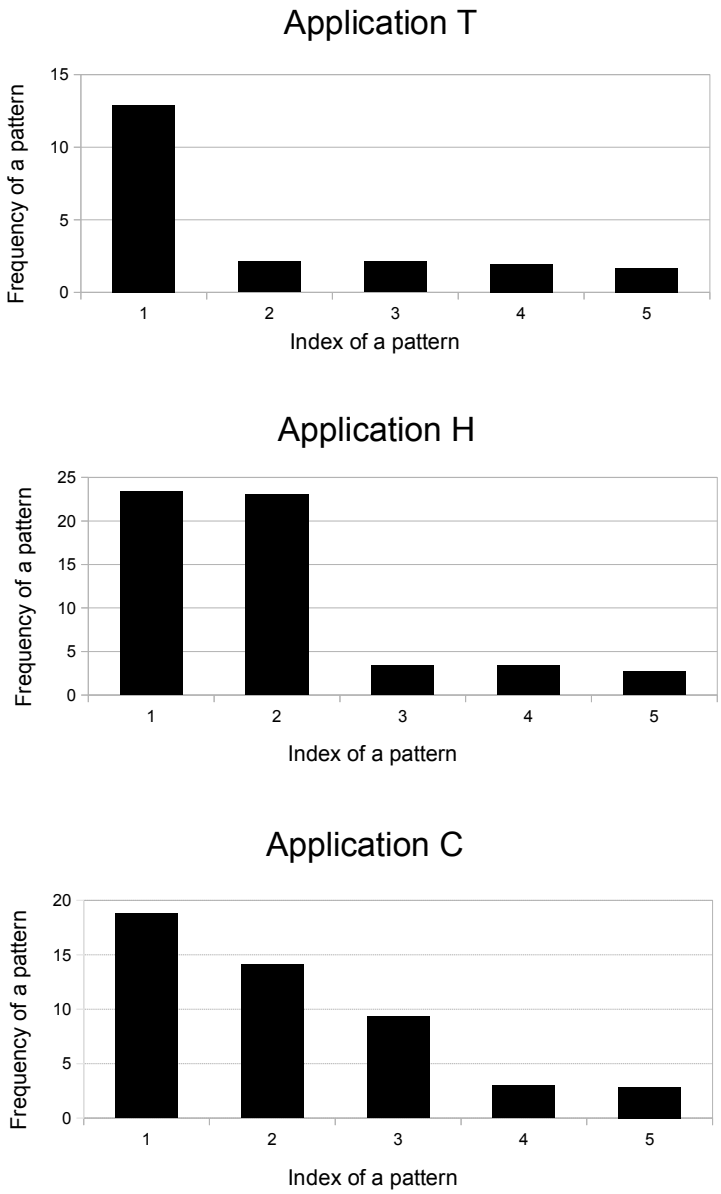

Figure 4.4: Distribution of frequency of occurrence for expression structure patterns.

example, if the value of sales for this year is represented by $m2$ and is 1100$, and the value of sales for the previous year is represented by $m3$ and is 1000$, the variance is (1100 / 1000) - 1 = 10%.

- *Inverted Variance* can be represented with the following pattern: $m1 = 1 - (m2/m3)$ *where* $m3 != 0$. It is similar to variance, but represents the percentage of **decrease** of value $m2$ over $m3$.

4.4.2 Corresponding metrics clones

The importance for this clone type is determined in the manner similar to that from the previous subsection: the main component was the *frequency of occurrence*, but we did additional analyses to make sure instances of this clone type are not too simple and are worth further investigation.

Frequency of occurrence is defined as the percentage of expressions that belong to those corresponding metrics clones patterns which have at least two instances. In other words, it is the percentage of expressions for which there exists at least one more expression with the same expression structure and corresponding metrics. This analysis considers expressions within a single application.

Table 4.2 shows the results related to the frequency of occurrence. Column one shows the total number of expressions per application. Column two shows the number of patterns that contain two or more instances, and as we can see, this number is multiple times smaller than the the total number of expressions, which implies that on average there are many expressions that have the same corresponding metrics and expression structure. And finally, the last column shows what is the frequency of occurrence as we defined it already. Frequency of occurrence shows that very high percentage of expressions is perceived as possibly made by copy/pasting some other expressions. This just proves that code cloning is a significant problem for the applications written in Measure Language.

	Total number of expressions	Number of patterns with 2+ instances	Frequency of occurrence
App H	295	67	81 %
App T	659	74	92 %
App C	1896	157	99.5 %

Table 4.2: Empirical results regarding the same corresponding metrics clones.

However, if we want to propose the improvements that will remove occurrences of this clone type, we need to better understand its instances. Also, more analysis is needed to better understand, describe, and classify corresponding metrics clones. For example, if we have two applications with the same number of expressions that are of similar complexity, how can we tell which one has a bigger amount of code cloning? If one of these two applications has expressions evenly distributed across many corresponding metrics clones patterns, whereas in the second application a majority of expressions belong to a single pattern, can we argue that the second application has a bigger amount of code cloning? Intuitively, having many large, complex expressions that differ slightly is more problematic than having few similar and simple expressions. In the first case we have a much bigger overlap that repeats across all the expressions.

To answer these questions and to further and more precisely describe the corresponding metrics clones, we introduced a concept of *Clone Severity*, which is an integer value assigned to a corresponding metric clones pattern that describes the intuitive perception of the amount of cloning the instances of that pattern introduce. To calculate this number, we take into account three parameters: (1) the complexity of instances of particular pattern (as defined in the previous subsection), (2) the number of pattern instances, i.e. the number of expressions that are of certain pattern, and (3) the number of different corresponding uoms/versions/roles in the group. Concrete algorithm is given in Figure 4.5. The complexity and the number of instances is directly correlated to the value of Clone Severity. The last parameter decreases this value - bigger variety in components of measures will decrease it more. The value is calculated only for those patterns whose instances

```
input  : A set of instances of a particular pattern
output: The value of Clone Severity
head ← input.head ;
result ← NumOfMeasures (head) * NumOfDimensions (head) * input.size / 2 ;
correspondingMeasures ← SetsOfCorrespondingMeasures (input) ;
foreach setOfMeasures in correspondingMeasures do
    setsOfComponents ← CorrespondingComponents (setOfMeasures) ;
    foreach setOfComponents in setsOfComponents do
        result = result  - NumberOfDifferentComponents (setOfComponents)
    end
end
return result
```

Figure 4.5: The algorithm for calculating the value of *Clone Severity*.

have the complexity greater than three (This value is based on the same reasoning as the one described in the previous subsection.)

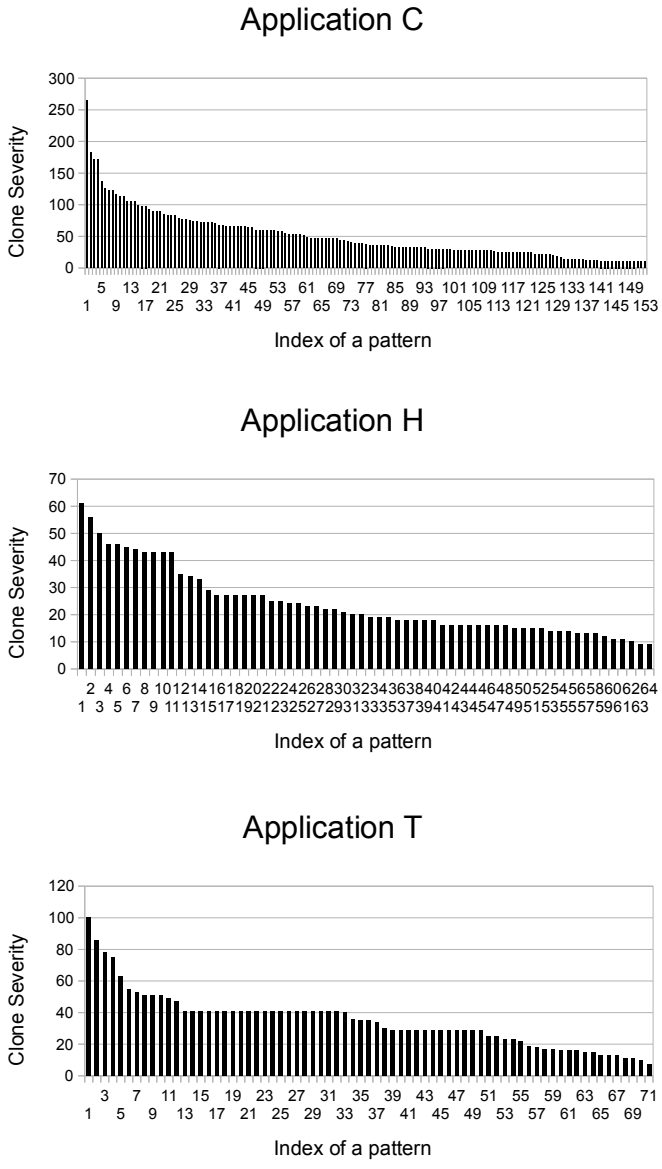

Figure 4.6: Distribution of clone severity for each of the three applications

The distribution of clone severity values is given in Figure 4.6. The X axis contains the value of a clone severity, whereas the value on Y axis shows the pattern index. The first information that can be inferred from the figure is that the complexity of expressions is proportional to the number of expressions in the application. The outlier group, which has a complexity of 265, is a group of 28 expressions that execute the *total* function. We used the patterns with the biggest clone severity values as a guide when we proposed the improvements.

4.5 Implementation

We implemented code cloning detection in Scala [2]. Parsing results in a set of custom-made AST objects, which are further processed using our core framework. The core framework returns a data-structure with relevant facts about the application code. To extract results we made Scala scripts that operate on the data-structure returned by the core framework. Scala's functional features like pattern matching and its functional-style collections framework made the development process very elegant, enjoyable and easy. Figure 4.7 shows the toolchain used for the case study.

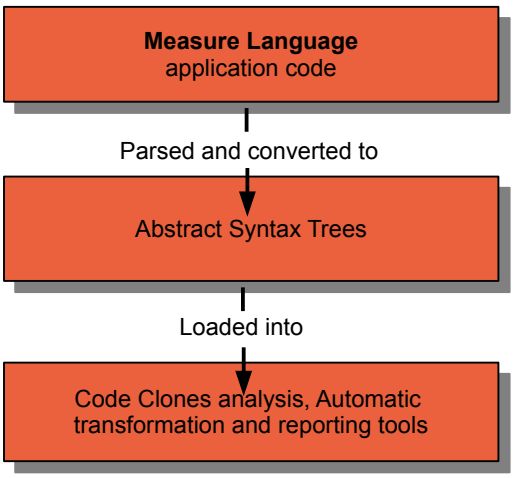

Figure 4.7: Toolchain in the detection process.

Chapter 5

Case Study: Proposed improvements

The last phase of the approach presented in chapter 3 is consisted of the following activities: proposing improvements to the target language, evaluating them, and choosing the best improvement. We have followed the same steps in our case study. The improvements we proposed in this chapter are based on the results of code cloning detection described in the previous chapter.

The approach suggests proposing two types of improvements: the ones that target reduction of specific clone types and the ones that are created by merging changes to the language introduced by other improvements. Each improvement requires evaluation. For this case study, we have focused on the first type of improvements, i.e. we only propose improvements that target removal of specific clone type. We did not merge the improvements due to practical limitations (time and effort needed to do the evaluation with current users of the language), so we left that for the future work.

The evaluation. The evaluation of each improvement we proposed is done in two ways: (1) by doing automatic translation of existing applications to the improved version of the language, and (2) by doing a survey with the current users of the measure language.

The goal of the automatic translation was to measure the reduction of the code size in terms of number of characters. We have translated each of the three applications to the improved syntax and compared the number of characters in such version of the application to the applications in their current state (i.e. the unchanged measure language). The basic assumption was that any improved version of the language should use fewer characters, since our main goal was to reduce code duplication.

As for the second type of evaluation, our goal was to get current users' feedback on the proposed improvements. Even in case our improvements prove to be great in terms

of code reduction, if the users feel uncomfortable with either its syntax or semantics, it becomes unusable. To get the feedback, we have created an online questionnaire and got two participants to complete it. The questionnaire provided the details of new versions of the language, and we asked the users to compare each of the new versions of the language with the current version and answer the following questions: (1) how would the new syntax contribute to the understandability, (2) how would it impact the effort to develop new applications and (3) would it introduce unrelated problems.

In the questionnaire, the participants were provided with the scale from 1 to 7 for the first two questions - 1 representing a much better solution, 7 representing a much worse solution, and 4 representing a solution that would neither be better nor worse compared to the current version of the language. When presenting the results, we do not mention the numbers but instead we use equivalent descriptions that are easier to understand. For example, if a participant answered with 3, we would present it as **slightly improved** if we talk about understandability or **slightly reduced** if we talk about the effort needed to write new applications. Similarly, 2 is presented as **improved** or **reduced**, 5 as **slightly worse** or **slightly increased** and 6 as **worse** or **increased**. None of the subjects answered with 1 or 7. In the following sections, we refer to the two participants as Participant A and Participant B

The next few sections explain the improvements and the evaluation in more details.

5.1 Improvements that target expression structure clones

This improvement takes into account the results of expression structure clones analysis presented in section 4.4.1, which shows that some patterns were detected to come from the MFP domain itself. The results have also shown that such patterns occur quite often (particularly in more than 20% of expressions for each of the applications we analyzed), and yet the measure language does not have any means of speeding up their creation. As our goal was to improve the language in order to reduce these kinds of redundancy, we proposed incorporating those concepts in the language itself. The next subsection explains the details of our proposal.

5.1.1 Domain-specific concepts as embedded functions

Description of the improvement. Two domain-specific concepts that we detected to occur often are *Variance* and *Inverted Variance*. Accordingly, the two functions that we propose are ***variance*** and ***invertedVariance***. The notion of function in this case is that instead of using the measure language to create the expressions that represents domain-specific concept, users would just use the function that represents the same concept and provide appropriate parameters. For example, instead of writing $m1 = (m1 \ / \ m2)$ - 1 where $m1$!= 0, users would write $m1 = variance(m1, m2)$. Figure 5.1 shows examples of usage for both concepts in the way we proposed: on top there is an expression written using the measure language, whereas below is the equivalent expression, that uses proposed domain-specific functions.

Problems targeted by the improvement. Introducing domain-specific concepts to the language would reduce the amount of time needed to create corresponding domain-specific expressions - at least the **where** part of the expression is avoided. But the biggest advantage would be understandability since it is much easier to spot that the expression

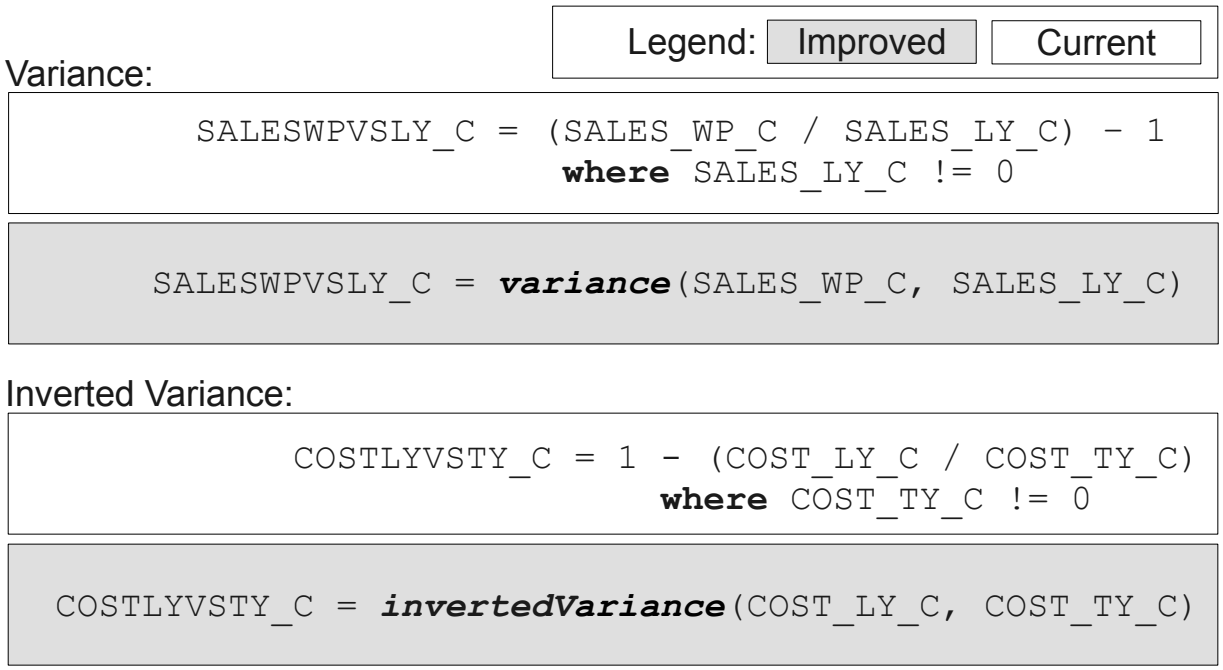

Figure 5.1: An example of incorporating domain-specific concepts into language

represents a domain-specific concept if it is represented by function with the appropriate name than by reading the whole expression and inferring from it.

Problems not solved/introduced by the improvement. In case we have many variances with the same corresponding metrics (i.e. many instances of the same corresponding metrics clone pattern that represent variance expressions), we would still have commonalities in terms of repeating the same metrics (as parts of measures) and the function name. The solution to this problem is given by the improvements proposed in the next section.

5.1.2 Evaluation

Results of the automatic translation. Since there is only around 20% to 30% expressions that represent domain-specific concepts, we did not expect huge changes in terms of code size reduction. For App H the reduction was 14%, which was the biggest reduction achieved. For App T the reduction was 13%, whereas App C had 9% reduction.

Results of the survey. Participant A thinks that there would be no change in understandability and effort needed to write new applications when compared to the current version of the measure language. The reason for this is the naming convention - term *variance* is used for other purposes in the domain and it might be confusing. Participant B has a different opinion - according to him the effort to write new applications would be **slightly reduced** and understandability would be **slightly improved**. The problem with this syntax, according to this participant, is that it is a bit more technical since it introduces a notion of a function in a way that is not already seen in the language.

5.2 Improvements that target corresponding metrics clones

We propose two measure language improvements that target removing this specific type of code cloning. The first improvement proposes introducing macros, whereas the second one proposes introducing anonymous functions. The problems that each of the improvements aims to remove, together with the specifics of each improvement, is described in the next two subsections.

```
1   ...
2   formula "SHIPMENTSFQ_WP_L" {
3     SHIPMENTSFQ_WP_L[week, store] =
4         SHIPMENTSGROSS_WP_L[week, store] −
5         (SHIPMENTSIRRG_WP_L[week, store] + SHIPMENTSCLOSEOUTG_WP_L[week, store]).
6   }
7   formula "SHIPMENTSFQ_WP_U" {
8     SHIPMENTSFQ_WP_U[week, store] =
9         SHIPMENTSGROSS_WP_U[week, store] −
10        (SHIPMENTSIRRG_WP_U[week, store] + SHIPMENTSCLOSEOUTG_WP_U[week, store]).
11  }
12  formula "SHIPMENTSFQ_LY_L" {
13    SHIPMENTSFQ_LY_L[week, store] =
14      SHIPMENTSGROSS_LY_L[week, store] −
15      (SHIPMENTSIRRG_LY_L[week, store] + SHIPMENTSCLOSEOUTG_LY_L[week, store]).
16  }
17  formula "SHIPMENTSFQ_LY_U" {
18    SHIPMENTSFQ_LY_U[week, store] =
19      SHIPMENTSGROSS_LY_U[week, store] −
20        (SHIPMENTSIRRG_LY_U[week, store] + SHIPMENTSCLOSEOUTG_LY_U[week, store]).
21  }
22  ...
```

Figure 5.2: Instances of the same corresponding metrics clone pattern.

5.2.1 Macros that accept measure components as parameters

There are many expressions that do not represent domain-specific concepts, and even those that do belong to a corresponding metrics structure clone pattern. In case the pattern has many instances, this implies that there may be many common components (at least metrics) which are duplicated and specified for each expression-instance of the pattern. The bigger the number of instances and the number of same corresponding components, the bigger is the overall duplication. In this subsection we present an improvement that targets this type of a problem.

To better explain the problem let us take a look at the set of expressions in Figure 5.2. The figure shows four expressions - they have the same structure, corresponding metrics and dimensions. Variable parts for these expressions are Versions (LY - Last Year and WP - Working Plan) and UOMs (L - litars and U - units), whereas other components and

```
1  // DEFINITION OF A MACRO
2  def SHIPMENTSFQ(Version, Uom) as
3    SHIPMENTSFQ_<Version>_<Uom>[week, store] =
4      SHIPMENTSGROSS_<Version>_<Uom>[week, store] −
5      (SHIPMENTSIRRG_<Version>_<Uom>[week, store] +
6        SHIPMENTSCLOSEOUTG_<Version>_<Uom>[week, store]).
7
8
9  // USAGES OF A MACRO
10 formula "SHIPMENTSFQ_WP_L" {
11   SHIPMENTSFQ(WP, L)
12 }
13 ...
14 formula "SHIPMENTSFQ_LY_U" {
15   SHIPMENTSFQ(LY, U)
16 }
```

Figure 5.3: Example of declaration and usages of macros.

dimensions are the same. This implies that many components will be specified four times for this particular example, and we have no abstraction mechanism that would facilitate creation of these expressions. To propose improvements, we decided to take this fact into account.

Description of the improvement. The idea is to introduce a notion of a macro that accepts measure components as parameters. The example can be seen in Figure 5.3. Basically, we define a macro that contains commonalities, whereas variable parts are provided as parameters to the macro. Instead of copy/pasting the existing rule and changing variable parts (which is the common practice with the current language), the users can just specify a macro and provide variable parts as parameters.

The approach is very easy to explain in case there is only one variable component across instances of the same corresponding metrics clone pattern.

Problems targeted by the improvement. Compared to the advantages of introducing domain-specific concepts as functions, this approach increases the amount of code that we abstract over. In cases we have a pattern with many complex expressions, using this approach will drastically remove the cloning introduced by repeating common parts of expressions.

Problems not solved/introduced by the improvement. There are few disadvantages to this approach. First of all, the understandability and readability is influenced. When users see a macro and its parameters specified instead of the actual expression, they would need to lookup the declaration of a macro in order to understand what expression does certain formula provide. This layer of abstraction enables reduction of code cloning, but adds more time needed to understand the code. On top of that, we would still need to specify corresponding rules and formulas names.

5.2.2 Evaluation

Results of the automatic translation. The reductions of the code size are greater than in the previous improvement – as expected – since more than 90% of the rules in average are perceived as potentially made by copy/paste. For App H, the reduction was 24 %; App T had reduction of 44 % and in App C the number of characters is reduced by 36 %.

Results of the survey. Participant A thinks that this version of the language would be slightly better than the current version: the understandability would be **slightly improved** and effort needed to write new applications would be **slightly reduced**. Participant B thinks that there would be **no change** when it comes to effort needed to write new applications, whereas the understandability would be **worse**.

5.2.3 Anonymous functions

This approach presented in this section is an answer to two types of problems: the ones introduced by the solution from section 5.2, and the ones that previous approaches have not dealt with.

Let us summarize the disadvantages introduced by/ignored by the previous solutions:

- Users would still have to write rules and formula names; none of the solutions proposed anything regarding the way rules/formulas names are specified.
- Names of macros or functions would be repeated for each expression in copy/paste suspects group, or for each domain-specific concept.

- Definition of a macro would be separated from its usage. This would add a layer of abstraction and users would have to go to declaration of a macro in order to understand it, which might be a problem for many users.

In the next few paragraphs we discuss each of the three points in more detail.

Users would still have to write rules and formula names. The vast majority of rules have only one formula: 92% - 95% across the three applications we analyzed. For such rules, a majority of the time the names of both rule and the formula are the same; we did manual inspection for cases where this doesn't hold and the conclusion is that the differences are in capitalization, truncation, typo, or the name of the rule is just rephrased in a different way. Thus we can freely abstract over such cases and claim that in case we have one formula inside of a rule, their names are the same (or at least are intended to be the same). Furthermore, these names are either the same as the name of the measure in the left-hand side of the expression in the formula or they differ from it for the same reasons already mentioned.

In the anonymous functions syntax, we take advantage of this fact by providing defaults for formulas and rules names and give users the ability to not specify these parameters every time. Of course, since there are cases where these names have to be different, we allow users to specify these names manually.

Repeating of functions and separation of a definition from its usage. If we have many instances of a corresponding metrics clone pattern, we would have to specify the function many times. One can think of this as cloning. The anonymous functions approach avoids this by binding the declaration of a common pattern to the declarations of variable parameters so the problem of separate definition is avoided.

5.2.4 Description of the anonymous functions improvement

The four expressions specified in Figure 5.3, rewriten using the anonymous functions syntax are shown in Figure 5.4. In the next few paragraphs we will describe the relevant parts of proposed syntax using the example from Figure 5.4.

Specification of expression template and variable parameters. The central place in the new syntax is given to expressions: we specify an expression in a manner similar to expressing a template, and we annotate variable parts with "<" and ">", as in Figure 5.4. This "template" part is specified between the **to** and **apply** keywords. In the example, variable parts are *Version* and *Uom*.

```
1  to
2    SHIPMENTSFQ_<Version>_<Uom>[week, store] =
3      SHIPMENTSGROSS_<Version>_<Uom>[week, store] −
4      (SHIPMENTSIRRG_<Version>_<Uom>[week, store] +
5        SHIPMENTSCLOSEOUTG_<Version>_<Uom>[week, store]).
6  apply {
7
8  Version = WP, Uom = L;
9  Version = WP, Uom = U;
10 Version = LY, Uom = L;
11 Version = LY, Uom = U,
12         in rule "SHIPMENTSFQ_LY_U", in formula "SHIPMENTSFQ_LY_U", with priority 1;
13 }
```

Figure 5.4: Example of anonymous functions syntax: we first specify a template, and then provide a list of concrete parameters.

Providing parameters Concrete instances of variable parts are meant to be specified after the **apply** keyword and in between opened and closed curly braces: this way we do not separate anything. Variable parts can be the following components: Versions, Roles, or UOMs. Metrics cannot be specified as variable part,re since this syntax is meant to be used for same corresponding metrics clone patterns, which share the same metric.

Rule and Formula names. As explained earlier, the most probable scenario is that each rule will have only one formula and the name would be the same as the measure name in LHS of the expression in the formula. Thus, the new syntax creates a default rule and formula name for each expression. For example, the rule and formula name for the expression in line 8 would be "SHIPMENTSFQ_WP_U", which is the name of the measure from the expression. However, for example in cases when a rule has more than one formula, default names we proposed will not be enough, so we provide a way for the users to specify these names. The example of such syntax, where we provide our own names for rules and formulas, can be seen in line 12.

Priorities. In the current version of the measure language, priorities of formulas are specified by the order in which they are declared within a rule: first formula has top priority, second rule has priority 2, etc. (priorities are described in section 2.4). In case there is only one formula, this is straightforward: the formula has priority 1. In new syntax, the priority is specified with **priority** keyword. In case the specification of the priority is

left out, the default value is 1.

5.2.5 Evaluation

Results of the automatic translation. With this improvement, App H would reduce code size by 26%, App T by 28% and App C by 33%.

Results of the survey. Participant A thinks that the understandability would be **improved** and the effort needed to write new applications would be **slightly increased**. This user sees a problem with locating the order of formulas within the rules in this syntax. On the other hand, Participant B sees **no change** in understandability and thinks that the effort needed to write new applications would be **increased**.

Overall, if we consider both the results of the questionnaire and automatic translation, the best improvement would be the first one presented - introducing domain specific concepts as embedded functions.

Chapter 6

Related Work

This thesis combines work from several research areas but to our knowledge is the first work applying cloning detection results to language improvements. However, there is a lot of related work in areas that this thesis focuses upon: code cloning - with particular focus on clones detection and application of detection results, and programming languages - more specifically analysis, design and evolution of domain-specific programming languages. The next few sections connect the work in these areas to our work.

6.1 Code cloning

There are at least two related areas when it comes to code cloning: how to do the actual code cloning detection, and what to do with the results. A very through review of existing literature that covers work done in both areas is given in [21]. A list of relevant papers grouped by categories and authors can be found at at [24]. More information about the work on each of the two areas is given in the next subsections.

6.1.1 Code cloning detection

In our case study we detect code clones that occur in a domain specific language using operations on Abstract Syntax Trees, and we propose generalized code cloning detection process. Despite a lot of research in the area of code cloning detection, we have found only few papers where clone detection is done using the AST-based approach. Also, very few

papers do code cloning detection on domain specific languages, and even fewer propose generalized approaches for code cloning detection that targets DSLs.

The Clone Digger tool, developed by Minea and Bulychev and described in [18] finds clones that range from exact to structural clones (types 1, 2 and 3 from section 2.1). They proposed an approach that uses *anti-unification* and is programming language agnostic. Anti-unification is based on calculating the distance between subtrees in ASTs (similar to the Levenshtein distance). By parameterizing the algorithm and setting different thresholds, detected subtrees (and thus pieces of code) can be made more or less similar. The difference from our approach is that we used a detection process that is very specific for our purposes, and thus their work is more general, although we could have used this approach for our case study.

Baxter et all ([3]) presented a tool and an approach that compares ASTs to detect exact and near miss clones, which can be applied on large C programs. Doing tree comparisons in a straightforward way would be too expensive for large programs due to the polynomial complexity (in particular $O(m * n)$ to compare two trees with m and n nodes) of such algorithm – the analysis would take a lot of time for large programs but the precision gains might not be worth it. That is why the paper presents an algorithm that assigns a hash function to subtrees to speed up the process. Using carefully selected hash functions, Baxter creates an efficient algorithm for capturing near-miss clones. Since our programs – and thus the corresponding ASTs – were small, we were able to apply the straightforward algorithm without having to worry about the performance.

Robert Tairas and Jordi Cabot created a generalizable approach for code cloning detection in DSLs, which they evaluated on OCL ([25]). Their approach does comparisons on DSL metamodel (abstract syntax) level and is more general than ours.

No work was found on application of detection results to language improvement. However, there is work on program analysis and program understanding which is a prerequisite for language improvement. The basic idea is that the knowledge about code clones can help understanding the program as a whole. For example, Johnson et al ([10]) created a prototype of a visualization tool to aid in understanding the contexts where detected clones reside.

Another application, related to our future work is on doing domain analysis using the results of code cloning detection. Ma and Woo perform domain analysis of device drivers using this methodology ([13]).

6.2 Language Design and Evolution

Our main focus is on the evolutions of domain-specific languages. Although intuitively evolving a language is not the same as redesigning it from scratch, sometimes it means creating exactly that, which is what we did in one of the suggested improvements. Thus, our work is related to both those areas that are focused on **how to create** a language, and those that describe approaches on **how to evolve** a DSL.

There has been a lot of research on how to create a Domain-specific language and going into details on existing work is outside the scope of this work. Mernik et all [16] and Van Deursen [27] give very detailed summaries of existing work both on creating DSLs, and on DSLs in general.

We list two papers focusing on evolution of a DSL in case it is created as an embedded DSL. Some experiences from that work match our experiences while evolving a language, for example the one concerning user friendliness of the syntax.

Freeman and Pryce ([7]) describe their experience with the creation and evolution of an embedded domain-specific language in Java. Their language originated as a library, but has evolved into a domain-specific language after a few iterations of usage analysis and feedback from users. They give many guidelines on how to create and evolve a Java-embedded DSL.

Another work describing how to evolve an embedded language is by Tratt el al ([26]). Tratt argues that using Converge and its macro-like functionality is a good choice for creating an embedded DSL from the point of view of evolution of a language. He gives an example of a language that is used to specify state machines (their states and transitions) and goes through a few iterations of changes in the syntax and semantics and then shows how to evolve it fit new requirements without too much effort.

Chapter 7

Discussion and Future Work

7.1 Discussion

We look back at and highlight our challenges we had and what we learned while working on this thesis.

Challenges. The most challenging part of the work was to detect that improving the language could lead to reduction of the copy/paste practices. This conclusion might seem obvious at this point, but it was not when we started solving the problem. To better explain why this was so challenging, let us highlight the problem that initiated the research presented in this thesis: *the users of the measure language have reported many similar parts of the code in the applications that they developed*. Since the measure language is used to represent relations among measures, the first hypothesis was that there exist a generic mathematical model that represents all or at least most of the relations among measures in applications (we will talk more about this in the future work section).

The preliminary test of the hypothesis was done by a group of domain (MFP) experts who made such a mathematical model. The model seemed to work on a simple set of requirements, but detailed testing proved that it does not cover even basic cases that occur in existing applications, i.e., the domain turned out to be too complex with too many corner cases and variability. The original plan for the thesis research was to try to make this model again, but this time with thorough investigation. This investigation was performed manually, but the graph representing the relations among metrics was too complex for manual understanding, given that there are at least 300 expressions per application. A lot of work was done in this direction without success. It was only after many brainstorming

sessions that we came up with a hypothesis that language improvements might reduce code cloning. Afterwards, the set of conclusions we had while improving the language has led us to the idea that the approach could be generalized, which shaped the direction of this thesis.

We had a few more notable challenges related to the work this thesis presented. For example, it was not easy to determine the actual cloning types that occur, i.e., we have looked for many different types of possible cloning unsuccessfully. Finally, proposing the improvements was a challenging task, since it is a creative work that cannot be done using some set of predefined rules: we have also tried many different suggestions without success.

Lessons learned. The biggest lesson we have learned is that the progress means being able to do **fast iterations**, at least when it comes to doing this kind of research. This refers to both clone detection and improvements phases: the time from proposing an improvement (or the possible code clone type) to evaluating the proposal (for either improvement or the clone type) should be as short as possible. For example, in the beginning we spent a few days polishing the specification of a possible clone type, and we have also spent a lot of time using a generic framework that we made for analyzing the measure language. In such environment, it took us more than a week to get the results of occurrences for a single clone type we proposed. Since many of the clone types we predicted did not actually occur, the overall process was taking longer than we felt it should. Later we decided to be less specific in our proposals and use smaller scripts for verifying the occurrences and we were able to shorten the process to a couple of hours. With this kind of pace we were able to execute many more iterations and in the end obtain better results.

7.2 Future Work

7.2.1 Extended Domain Analysis

Our first improvement proposed introducing functions corresponding to domain-specific concepts. As we explained previously, these domain-specific concepts were detected by analyzing the results of code cloning detection. This fact shows the potential of code cloning detection as a tool for doing domain analysis.

Similar work was done by Ma and Woo, who did domain analysis of device drivers using code cloning detection in [13]. They focused their work on device drivers and evaluated it on the drivers from the Linux kernel. Their conclusion was that similar pieces of code among different drivers represent implementation details of the domain-specific concepts.

We believe that similar ideas can be applied to our domain with a potential to discover much more than the two concepts we presented.

The prerequisite for generalizing the domain-analysis approach is to have a bigger set of applications to analyze, since a bigger pool of applications would give us a better way to test any hypotheses we might have. One direction to focus our future work on domain analysis is to do the same analysis we did so far - mapping expressions to domain-concepts. However, we see another direction with a potentially large impact: detection of patterns in relations among measures.

An example of a pattern (i.e., a universal relation) is the following: sales is always (or at least very often) calculated as a sum of costs and profits. Domain experts from the company we collaborated with believe that there are many relations that are universal for the whole domain. Given that there are many measures and metrics that are involved in applications, figuring out patterns in expressions would be difficult. However, a tool like the one we used for our case study can be useful. In particular, a tool that can accept a relation or a set of metrics as an input and search across multiple applications for either a particular pattern or expressions in which a particular set of metrics occur would speed up getting feedback on the hypotheses. The results produced by such a tool could be used to evaluate the hypotheses and further modify them. The final outcome would ideally be a domain-specific set of patterns of relations among measures.

7.2.2 New Measure Language as internal DSL

The improvements that we have proposed assume complete freedom in terms of the syntax and the underlying language interpretation mechanism, which can be achieved only if the language is implemented as an external DSL. The freedom that we assumed is justified by the fact that the measure language itself is implemented as an external DSL.

A possible approach is to apply the same ideas and propose the improvements as a new, internal language hosted by DatalogLB. DatalogLB is a declarative, logic programming language developed by the same company that developed the measure language, but it has much higher priority and usage within the company.

One of the obvious benefits of having an internal language is better integration with the existing tools. Since DatalogLB is being constantly improved and developed, an internal language would inherit all the benefits of the improved version of DatalogDB for no cost.

An obvious drawback of having an internal language is the fact that both the syntax and the interpretation would be constrained by the host language. For example, the ability

to have syntax that is close to the one that we proposed depends on how much flexibility DatalogLB allows for such tasks.

Fortunately, there are mechanisms in DatalogLB that make it very internal-language friendly. There are currently few DatalogLB libraries that leverage the internal-language friendliness – those libraries have the look and feel of a different language but are executed within the DatalogLB.

Thus, the next step would be to create an internal language that implements the improvements that we proposed. Achieving this step would require effort in two directions: (1) experimenting with the DatalogDB language embedding features in order to make the syntax as close to our proposals as possible, follwed by (2) understanding the impacts of the fact that the new language could be integrated with existing tools and customizing the language accordingly if needed.

7.2.3 More Case Studies

The approach to evolving a language presented in this thesis was applied to only one case study, which was at the same time the only real world experience and input to the approach. Although we have tried to make the approach as generic as possible, by providing just high-level guidance, it is possible that this approach would be hard to apply for some domain-specific languages.

More case studies could help us make the approach better in different ways: we could better evaluate and validate the correctness of it, we could adjust it in cases it proves not useful, and it could help us make some parts more specific and easier to apply. For example, the part on the proposing the improvements to the language (along with evaluation and merging with different improvements) would require substantial amount of time to apply the way it is presented now in cases the language contains many clone types. This is due to the fact that the worst-case number of improvements that could be proposed is exponential in terms of the number of important clone types that occur in the target language. Perhaps by having more experience with the approach and with different languages, we can provide some best practices or some more detailed and easier ways to apply it.

Chapter 8

Conclusion

In this thesis we presented an approach to improving a domain-specific language to reduce the amount of code cloning in its usage, which can be applied in cases the language design is the cause of the code cloning. The approach proposes two main steps: detecting the types of code cloning that occur in usages of the language and proposing improvements based on the detection results. The main purpose of the first step is to understand what kinds of code cloning occur and how often they occur. The goal of the second step is to propose language improvements that target reduction of specific code clones, and to combine such clone-specific improvements to maximize overall cloning reduction.

We have applied the approach to a proprietary DSL, called the **measure language**, which is used to represent arithmetic expressions in the merchandise financial planning domain. The users of the measure language reported that they often copy/paste code while developing applications, which was a perfect case study for the approach.

The improvements we proposed were evaluated in two ways: (1) by doing automatic translation of current applications to the improved version of the language, and (2) by doing a questionnaire with the users of the measure language. The automatic translation was used to measure potential improvement in terms of code size, particularly the number of characters, where the most successful improvement reduced the code size by 50%. In the questionnaire, the users were asked to compare the effort needed to understand and create new applications between the current and new versions of the measure language. The responses we received from the questionnaire were positive, i.e., the users believe that the proposed improvements would reduce efforts for the two aspects. These evaluations have shown that the approach can be useful in a real-world environment.

A potential future application of the ideas presented in the approach is extended domain

analysis. Particularly, in the case study we detected that certain types of cloning indicated domain-specific concepts, i.e., similar expressions represented implementation of concepts that are well-known in the domain. We believe that code analysis can be extended to detect patterns of expressions that are common across different applications in the domain. Such information can be used to speed up the creation of new applications, where developers would, instead of creating all the expressions from scratch, choose patterns that correspond to their particular case, and have the applications (or at least parts of it) automatically generated.

References

[1] Merchandise financial planning documentation. `http://docs.oracle.com/cd/B31329_01/merchfinplan/index.html`. Accessed: 10/11/2012.

[2] The scala programming language. `http://www.scala-lang.org`. Accessed: 3/11/2012.

[3] I D Baxter, A Yahin, L Moura, M SantAnna, and L Bier. Clone detection using abstract syntax trees. *Proceedings International Conference on Software Maintenance Cat No 98CB36272*, 98:368–377, 1998.

[4] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 109–, Washington, DC, USA, 1999. IEEE Computer Society.

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[6] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code.* Reading, MA: Addison-Wesley., 1999.

[7] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in java. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 855–865, New York, NY, USA, 2006. ACM.

[8] Antoine Savelkoul Gerardo De Geest and Aali Alikoski. Building a framework to support domain-specific language evolution using microsoft dsl tools. Master's thesis, Delft University of Technology, 2008.

[9] P. Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 134–, Washington, DC, USA, 1998. IEEE Computer Society.

[10] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1*, CASCON '93, pages 171–183. IBM Press, 1993.

[11] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Paul Laird and Stephen Barrett. Towards dynamic evolution of domain specific languages. In *Proceedings of the Second international conference on Software Language Engineering*, SLE'09, pages 144–153, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] Yu-Seung Ma and Duk-Kuyn Woo. Applying a code clone detection method to domain analysis of device drivers. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, APSEC '07, pages 254–261, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Softw. Engg.*, 11(3):395–431, September 2006.

[15] Mark Marcel van Amstel, Marcel van den Brand and Luc Engelen. An exercise in iterative domain-specific language design. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 48–57, New York, NY, USA, 2010. ACM.

[16] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[17] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 87–, Washington, DC, USA, 2002. IEEE Computer Society.

[18] M. Minea P. Bulychev. Duplicate code detection using anti-unification. In *SYRCoSE (Spring/Summer Young Researchers' Colloquium on Software Engineering)*, 2008.

[19] Koschke Rainer. Survey of research on software clones. 2006.

[20] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *SCIENCE OF COMPUTER PROGRAMMING*, page 2009, 2009.

[21] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEENS UNIVERSITY*, 115, 2007.

[22] Saeed Shafieian and Ying Zou. Comparison of Clone Detection Techniques - Technical Report 2012-593. Technical report, 2012.

[23] Anthony Sloane. Experiences with domain-specic language embedding in scala. In *2nd Int'l Workshop on Domain-Specic Program Development. Proceedings*, 2008.

[24] Robert Tairas. Code clones literature, students.cis.uab.edu/tairasr/clones/literature/, 2012.

[25] Robert Tairas and Jordi Cabot. Cloning in dsls: experiments with ocl. In *Proceedings of the 4th international conference on Software Language Engineering*, SLE'11, pages 60–76, Berlin, Heidelberg, 2012. Springer-Verlag.

[26] Laurence Tratt. Generative and transformational techniques in software engineering ii. chapter Evolving a DSL Implementation, pages 425–441. Springer-Verlag, Berlin, Heidelberg, 2008.

[27] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.