CODE REFACTORING UNDER CONSTRAINTS

by

YAN LIANG

NICHOLAS A. KRAFT, CO-CHAIR
RANDY K. SMITH, CO-CHAIR
JEFFREY C. CARVER
ALLEN S. PARRISH
UZMA RAJA

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2011

ABSTRACT

Code refactoring is the process of changing the internal structure of the program without changing its external behaviors. Most refactoring tools ensure behavior preservation by enforcing preconditions that must hold for the refactoring to be valid. However, their approaches have three drawbacks that make the refactoring results far from satisfactory and reduce the utilization of refactoring tools in practice. Firstly, programmers are not sure how code will be changed by those tools due to the invisible refactoring rules hidden behind the interfaces of tools. Secondly, current refactoring tools have limited extensibility to accommodate new refactorings. Lastly, most refactoring tools lack mechanisms to allow programmer to specify their own preconditions to indicate as to which properties of a program are of interest.

We consider refactoring a code change activity that, as with other constraints imposed on code during software development and maintenance such as naming rules, should be visible, easily extensible, and adaptable. It should also combine the developers' opinions, implementation styles of existing code and other good coding practice. We propose a model-based approach to precondition specification and checking in which preconditions can be declared explicitly and dynamically against the designated program metamodel, and verified against concrete program models.

This dissertation applies the approach of model-based refactoring precondition specification and checking on C++ source code refactoring. Based on the analysis of primitive refactorings, we design a C++ language metamodel to support constraint specification and code inspection for refactoring purposes. We then specify preconditions of 18 primitive refactorings against the

metamodel, with primary concerns on syntax error prevention and semantic preservation. The impact of a programmer's perspective on these specifications is discussed. As another example to demonstrate the importance and necessities of supporting visible, extensible and adaptable precondition specification and checking, we use template method and singleton patterns to discuss how design patterns can affect refactoring decisions. We set up an experimental environment in which we build the language metamodel, develop a program model extraction tool and simulate the process of precondition specification and verification following the proposed approach.

LIST OF ABBREVIATIONS AND SYMBOLS

AST	Abstract Syntax Tree
ASG	Abstract Semantic Graph
CBTD	Component-Based Tool Development
CDT	C++ Development Tools: C++ IDE based on Eclipse platform
DMM	Dagstuhl Middle Metamodel
EMF	Eclipse Modeling Framework
ERM	Entity-Relationship Model
MDT	Eclipse Model Development Tools
MDTD	Model-Driven Tool Development
OCL	Object Constraint Language
RE	Reverse Engineering

ACKNOWLEDGMENTS

First and foremost, I would like to express thanks to my esteemed advisor Dr. Smith. It has been an honor to be in his PhD program and to participate in my favorite research area: software engineering. This dissertation would not have been possible without his professional guidance. The passion he has shown for cutting edge Software technologies has provided me with an excellent example of how to be a professional software researcher, which will benefit my career in the future.

My special thanks go to Dr. Kraft. His expertise in reverse engineering and code refactoring has added to my understanding of research in this area. I would also like to express my gratitude to my committee members, Dr. Uzma Raja, Dr. Jeffery C. Carver and Dr. Allen S. Parrish for their guidance and advice on my dissertation and research.

I am also so grateful to my father Wenjie Liang and mother Qiaoxiu Zhang for being so supportive during my graduate study. I appreciate my husband for his encouragement and support, enabling me to complete my PhD study with less stress and more enthusiasm. There are no words to show my appreciation for all the time and effort he has put forth for the family, especially for our daughter Sharon's education during these years.

Finally, I thank all the faculty and staff of the computer science department for their friendship, encouragement and kindly support. The dedication and commitment of faculty and staff make the University of Alabama a great place to study and do research.

Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 The Problem

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure (Fowler 1999). With refactoring techniques, software systems' adaptability and understandability can be improved while evolving to meet various demands. Code refactoring has become a fundamental part of the evolutionary development of object-oriented software systems.

It has been recognized that tool support is essential for the successful application of refactoring. However, the resultant proliferation of refactoring tools fails to solve the problem, making automatic refactoring an ideal rather than a reality. Most refactoring tools put significant effort into preserving the behaviors of program, which is most ensured by enforcing a set of preconditions that must hold for a refactoring to be valid. As to the implementation of precondition checking, there are several drawbacks that current code refactoring tools need to address in order to improve their applicability.

- Visibility

Ideally, refactoring tools are able to provide remarkable support in the process of software development and maintenance. However, people do not use refactoring tools as much as they could (Murphy-Hill, and Black 2007). Many programmers prefer manual refactoring since they believe that applying refactoring by hand is safer and more efficient (while it is not entirely true). Campbell et al. address one of these barriers: programmers are not sure how refactoring tools

will transform their code (Campbell, and Miller 2008). Code preview can alleviate this issue to some extent by indicating what a refactoring will do before the programmer decides whether to apply it. However, this approach provides no way to let the programmer know what code is being checked, except for warnings or error messages given when the refactoring precondition is violated. In another words, complete knowledge of what is to be checked is invisible to the programmer. We think this is the primary reason that refactoring tools lose the trust of programmers. After all, code transformation is heavily based on the result of precondition checking.

- Extensibility with Convenience

The number of possible refactorings is unlimited, so refactoring tools must be developed that are end-user programmable to accommodate new refactoring principles. Current effort towards this goal is to give users the ability to compose larger refactorings from existing smaller refactorings (Roberts 1999) (Kniesel, and Koch 2004). With this approach, programmers can integrate multiple basic operations/preconditions hard-coded in the tool to generate preconditions for new refactorings. However, identifying a set of basic conditions that fit well into composition is really a challenge, requiring two criteria that need to be satisfied. First, basic conditions must be complete enough to express any refactoring that is syntactically and semantically legal by the rules of the programming language. In some cases, a significant part of the knowledge required to perform refactorings cannot be obtained from existing code, but remain implicit in the developers' mental processes. For example, design patterns can guide decisions on which part of the software need to be refactored and which refactoring rule is most appropriate to apply. Second, basic conditions must be designed carefully to ensure that each composed precondition does not miss any necessary checking, or implements unrelated or duplicate checking. In

conclusion, predicting all primitive preconditions that can be composed to cover all possible refactoring requirements seems to be an impossible task, which limits the composition-based approach to extend the capabilities of refactoring tools.

- Adaptability

Most refactoring tools lack mechanisms that allow the programmer to specify preconditions for refactoring. In most tools, precondition checking takes place in the absence of any indication by the programmer as to which properties of a program are of interest. However, adaptability is a highly essential feature refactoring tools need to have. Firstly, refactorings named identically may have different implementation definitions from person to person. For instance, renaming refactoring is to change the name of a specific language entity and all its references. Some tools prohibit *renameMethod* refactoring if the method to be changed is a constructor because of the potential impact throughout the code. However, programmers may have differing opinions. The change impact of *renameMethod* refactoring on a constructor is actually equal to that of *renameClass* refactoring for those classes containing user-defined constructors. This is because their preconditions should be designed equally, based on the rule that the name of a constructor must be exactly the same as the containing class. Secondly, even though behavior preservation is the most important features of refactoring, it is never formally defined. This implies that a programmer needs the freedom to change preconditions that are possibly suitable to others but not to himself. These facts tell us that preconditions should not be fixed (even for a single refactoring) in refactoring tools. Instead, programmers should be able to change them to fit their needs. Adaptable precondition specification can in some cases avoid preconditions that may lead to unsound refactorings, and overly restrictive preconditions that may rule out refactoring opportunities where they could help.

Considering all of the issues current refactoring tools have, we realize that making the preconditions of refactorings visible, extensible and changeable is vital for the success of a refactoring tool.

1.2 Our Approach

We consider refactoring a code change activity that, as with other constraints imposed on code during software development and maintenance, such as naming rules, should be visible, easily extensible, and adaptable. It should also combine the developers' opinions, implementation styles of existing code and other good coding practice. To do that, it is important to represent the program in an abstract way, so that programmers can better understand the program to benefit their decisions on precondition specification and the specification process itself. Based on this idea, we propose a model-based approach to precondition specification and checking. Programmers can navigate and browse program models similar to IDE's source code browser but with viable analysis alternative. Preconditions can be declared explicitly and dynamically as constraints against the designated program metamodel, and verified against concrete program models. Program modeling is the core and foundation of our approach.

The benefits of model-based approaches on software development and maintenance have been discussed extensively in the literature, which cover code refactoring as well. In particular, model-based precondition specification and checking decouples refactoring-related concerns and language concepts from language-dependent implementation details, which help programmers better understand the structure and behavior of the program. On the other hand, this approach also provides the possibility of integrating the analysis results of other tools such as pattern and code smell detection tools. These results help programmers evaluate the program being

refactored in order to decide which properties must be checked for the current refactoring task, how to make analysis scalable while keeping acceptable precision, and so forth.

1.3 Contribution

The main contributions of this research are:

- Analyzed the potential problems of current refactoring tools on precondition checking, and proposed to use a model-based approach to specify refactoring preconditions and verify their validity. Model-based precondition specification and verification can overcome the deficiency of current refactoring tools and produce more practical refactoring results, due to its abilities to incorporate human decisions.
- Based on the analysis of primitive refactorings and the problems of C++, we designed a metamodel for the precondition specification of C++ source code refactoring. This work considers the features of the C++ language and the challenges they bring to refactoring. The metamodel is represented with UML class diagram notations.
- Specified preconditions for 18 primitive refactorings as OCL expressions against the C++ metamodel we provide. The primary efforts of these preconditions follow the traditional refactoring requirements on syntax correctness and semantic preservation, without considering other constraints which programmers may also use in refactoring. For the precondition of each refactoring, we discuss its alternatives and related issues, showing that it is important to grant programmers the capability to specify their own refactoring rules.
- Demonstrated another type of constraint that should be considered during precondition specification: retaining the design intents that implementations of design patterns imply. We use the template method pattern and singleton pattern to illustrate how design

patterns could affect the applicability of refactoring preconditions. We conclude that design pattern sensitive refactoring needs more direction from the programmer.

- Set up an experiment environment to simulate the implementation of model-based precondition specification and verification for code refactoring. We use Eclipse EMF to create the designated metamodel and generate source code for metamodel manipulation. We also use the Eclipse CDT API to extract language elements from C++ source code to build program models. Design pattern specifications are also incorporated into the C++ metamodel. We use the Eclipse OCL interpreter plugin to exercise all the constraints we have specified. All the evaluation results are presented.

1.4 Roadmap

The rest of this thesis is organized as follows:

- Chapter 2 is an overview of state-of-the-art code refactoring technologies and their related research areas.
- Chapter 3 presents a metamodel designed to perform refactoring-purpose code analysis. This design takes into account the refactoring challenges brought by C++ language itself, and is based on the analysis results of primitive refactorings.
- Chapter 4 addresses constraint specifications that formalize primitive refactorings as declarative OCL expressions against the C++ metamodel.
- Chapter 5 introduces the necessities of facilitating pattern-sensitive refactoring. We use the template method pattern and singleton pattern as examples to demonstrate how refactoring decisions could be changed when design patterns need to be considered as a type of constraint imposed on code refactoring.

- Chapter 6 presents an experimental environment that verifies the constraints specified in Chapter 4 and Chapter 5.
- Chapter 7 discusses the issues of our approach that emphasize the idea of model-based refactoring specifications and verification.
- Chapter 8 ends this thesis with a conclusion and future work.

CHAPTER 2 BACKGROUND

In this chapter, we give an overview of state-of-the-art code refactoring technologies and other techniques that are closely related to this research. For refactoring, we discuss code analysis and code transformation separately. We also introduce research work in reverse engineering that can help software developers better understand the systems and hence assist refactoring decision-making. Design patterns are another research branch we present in this chapter that has a close relationship with code refactoring. How to deploy existing design patterns for system extension or to improve software quality is one of the main tasks refactoring activities need to fulfill. In the last section of this chapter, we introduce five metamodels well documented in the literature. In particular, we focus on their features representation granularity, language dependence, metamodel representation and design purpose.

2.1 Code Refactoring

Refactoring techniques have been attracting great interest in both academic and industrial areas in the past two decades. Martin Fowler defines refactoring (Fowler 1999) as:

> *"Refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify, without changing the observable behavior of that software component"*

Mens et al. have a comprehensive overview on the art-of-state of software refactoring (Mens, and Tourwe 2004), discussing the refactoring activities, techniques and formalisms, software artifacts to be refactored, tool support issues and how refactoring fits into the software

development process. Many other studies on code refactoring techniques can be found in venues such as International Conference on Software Maintenance (ICSM), International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA), IEEE Transactions on Software Engineering, ACM SIGPLAN conference on Object-oriented programming, and in other major software engineering conferences and journals. In this section, we will introduce research achievements in refactoring object-oriented programs. Roberts defines "a refactoring is a pair R = (pre; T) where pre is the precondition that the program must satisfy, and T is the program transformation" (Roberts 1999). We use this paradigm for our discussion, separating code analysis from actual code transformation. In particular, our attention focuses on refactoring-purpose code analysis techniques.

2.1.1 Code Analysis

In his work, Opdyke introduces many ideas on which much of the current work on refactoring is based (Opdyke 1992). He proposes to use preconditions (invariants) for ensuring behavior-preservation. For instance, *removeMethod* is a refactoring that removes a user-defined method from a class. The precondition to perform it is that the method being removed is dead code, never used in the program. Arbitrary removal of a method without considering this precondition may cause the program to behave unexpectedly. Three of the most complex refactorings are discussed in his PhD dissertation: generalizing the inheritance hierarchy, specializing in the inheritance hierarchy and using aggregations to model the relationships among classes.

The next major contribution in refactoring literature is Donald Roberts' thesis "Practical Analysis for Refactoring" (Roberts 1999). Roberts extends Opdyke's previous work by creating the tool Refactoring Browser for programs written in Smalltalk. In his thesis, he addresses the

criteria that are necessary for the success of a refactoring tool. First, it must change code accurately with low time consumption. Second, a refactoring tool should be integrated into the development environment and ready for use whenever necessary. Third, a refactoring tool should be supported by a powerful program database that contains all the information required and provide the ability to search for various entities across the entire program.

Martin Fowler's book "Refactoring: Improving the Design of Existing Code" is the first to formalize refactoring principles in detail (Fowler 1999). He proposes to use code smell to diagnose problems in existing code that could be removed through refactorings. In his book, Fowler uses a standard format to describe refactorings. For each named refactoring, he addresses the situation in which it should or should not be applied, and presents a step-by-step description of how to work it out manually. Fowler's work had a strong impact on the development of object-oriented refactoring tools thereafter.

One of the main problems when applying refactoring is deciding where to apply which refactoring. While Fowler states that most code smells are based on human intuition and subjective perceptions, techniques and tools for automatic code smell detection have emerged to assist human observations on "bad smell" and guide their decisions on refactoring. Software metrics provide a means to extract useful and measurable information about the structure of a software system and have become a popular approach for automatic code smell detection (Simon, Steinbruckner, and Lewerentz 2001). Clone detection is another technique that can contribute to code smell identification. Duplicated fragments significantly increase the work to be done when enhancing or adapting code and is the primary reason to perform code refactoring (Fowler 1999). The large amount of clone detection tools and studies give evidence to this truism (Roy, Cordy, and Koschke 2009).

Typically, invariants, preconditions and post-conditions are three elements needed to formalize a refactoring, and this approach has been suggested repeatedly in research literature since they constitute a lightweight and automatically verifiable means to ensure that the behavior of the software is preserved after refactoring. As another way to formalize refactoring, Men et al. propose a direct correspondence between refactoring and graph transformation for the reason that a program can be expressed as an abstract semantic graph and refactorings correspond to graph production rules. Their approach uses two mechanisms, type graph and graph expressions, to formalize syntactically correct programs and detect the pre- and post-conditions of refactoring (Mens, and Tourwe 2004).

Code analysis for behavior-preserving refactoring can be achieved statically or dynamically. Tip et al. propose a static analysis method in which type constraints are used to verify the preconditions and to determine the allowable source code modification for a number of generalization-related refactorings such as p*ullUpMethod* refactoring (Tip, Kiezun, and Baumer 2003). Kataoka et al. developed a tool called Daikon to dynamically detect the invariants in the program that are candidates for specific refactorings (Kataoka et al., 2001). Their approach applies a test suite on the target program to trace the variables of interest. Although dynamic analysis works well in their case study, they suggest that dynamic analysis is a complementary technique to perform refactorings. Static analysis is in general sound with respect to all possible program refactorings because the invariants detected by dynamic analysis inherently depend on the design of the test suite. Most of current refactoring tools use static code analysis.

The number of possible refactorings is unlimited, which motivates the research on the reuse of existing refactorings. Kniesel et al. developed a refactoring editor to compose larger refactorings from existing ones. The core technique of this tool is automatically computing the

precondition of the composite refactoring from preconditions of the composed refactorings in a program-independent way (Kniesel, and Koch 2004). This capability and basic refactoring operations are hard coded into their tool to create composite refactorings for arbitrary programs. Liu et al. have presented work using a formal specification language object-z to define primitive and advanced refactorings (Liu, and Zhu 2008). Prete et al. also presents evidence that exhibits the feasibility of reconstruction of complex refactorings from simple ones (Prete et al., 2010).

To be useful, code refactoring needs meaningful semantic information expressing both the language entities and their relationships in a program. This information should be extracted from textual representation, stored in memory or software repository and ready for use by a search engine. Many studies seek ways to represent this semantic information in a format useful for code refactoring. Roberts states that an abstract syntax tree (AST) contains sufficient information to implement powerful refactorings (Roberts 1999)

2.1.2 Code Transformation

Code analysis ends with a result that describes the locations in code that are recommended to change, while preserving the program's behavior. Code transformation is a follow-up step in which actual changes on the corresponding code detected in the code analysis phase take place. Refactoring requires keeping track of the original source code entities to be changed, which differentiates it from pure code analysis. Some refactoring tools can automate the changes for simple refactorings. They allow developers to preview the suggested changes, and, after changes are confirmed, be able to roll back the updated code to the original one. Automatic code change typically uses AST rewriting techniques which modifies the AST built from the original code, followed by the implementation that gets the source back from the AST. In an IDE, ASTs are generally generated by the compiler front end used for program execution. Refactoring tools can

share these ASTs for code transformation. However, to preserve documentary structures that are explicitly defined to be not part of the language but can improve the readability of code like space, indent, line break and comment, there should be a lexeme engine (also called pretty printer) to store the location of each token and all comments (Li, Reinke, and Thompson 2003). Both the modified AST and the lexeme engine contribute to returning nicely formatted code.

Code transformation needs human involvement. For a detected problem, there are possibly several ways to change the code to make it more adaptable. One can simply change a small set of classes and then recompile to remove errors. Or one can do more than that. The programmer might modify the current design pattern for the program, which may be a big transformation on the existing code, and if appropriately used, would greatly improve the readability and flexibility of the code (Gamma et al., 1995). Even for a small and common refactoring, human insight on a code transformation solution can still be flexible. For example, when a method uses or is used by more features of another class than the class in which it is defined, we can perform *moveMethod* refactoring that creates a new method with a similar body in the class that uses it most. Programmers can choose to either turn the source method into a delegating method, or remove it and change all its references accordingly.

The key characteristic that distinguishes refactoring from general code transformation is its focus on behavior preservation. That is, refactoring is *conditional* code transformation. Automatic code transformation has challenges but code analysis is especially important in the context of refactoring because it determines what code will be transformed. In this dissertation, our attention will be centered on code analysis techniques.

2.2 Reverse Engineering

Reverse engineering (RE) is the reverse of the traditional software development process called forward engineering where high-level abstractions and logical, implementation-independent designs are transformed to the physical implementation of a system. RE itself does not change the subject system. It aims to provide useful evidence and knowledge to guide software maintenance. Chikofsky et al. first defined RE as follows (Chikofsky, and Cross., 1990):

> *"Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction."*

RE research has contributed many tools to support code comprehension and reduce the overall effort required in software maintenance. These tools simplify the program comprehension process by automatically generating textual and graphical reports of design, dependencies, and code structure such as call graphs and UML class diagrams. Successful applications of reverse engineering have covered areas such as clone detection, change impact, code documentation and design pattern recovery (Canfora, and Penta 2007).

While different RE tools may accompany different technologies, they generally have similar software architecture which includes an extractor, analyzer, visualizer and software repository (Holt et al., 2006). The extractor parses code, extracts facts and builds an abstract syntax tree (AST) or abstract semantic graph (ASG) for input programs. The analyzer then analyzes the extracted artifacts and changes them into higher-level representation. This representation can be used for further code analysis or visualization within the tool. The visualizer presents the user with the view of generated knowledge about the program in a graphical way. A software repository stores data conforming to different levels of metamodel, and is accessed by the

extractor, analyzer and visualizer to fulfill their functions. Maintaining the mappings between these metamodels is important for tool users to understand software at various levels of abstraction.

Overall scalability and customizability in the analysis process are two challenges in the RE domain. The scalability issue has been investigated intensively in the RE research community (Muller et al., 2000). Each RE tool should be flexible enough to adapt to different kinds of RE tasks at various levels of abstraction while pursuing better performance. Associated with scalability are efforts on metamodel (also called schema) design, techniques of fact extraction, code analysis and graphic viewing. Some research opinions about schema design issues include: multi-level schema design (Lethbridge 2003), semantic completeness of ASG-level schema (Lin, Holt, and Malton 2003) and trade-off between scalable schema and analysis precision (Kienle, and Muller 2010). In Section 2.4 we have investigations on several well-known metamodels serving RE domains. Customizability in software analysis means that engineer should have control over the analysis process and resulting precision in some way, for example, by using script languages and meta-programming (Atkinson, and Griswold 1996; Muller et al., 2000; Brichau et al., 2008; Hou, and Hoover 2006).

2.3 Design Pattern

Software design patterns encapsulate elegant ideas for solving high-level problems in modern software systems and hence take very important roles during the development and maintenance of software systems. Suitable design patterns can speed up the development process since software designers do not need to design and develop software architecture from scratch. The importance of design patterns have been well-known in reverse engineering and other research communities working on software evolution. Detection of software design patterns is a

significant effort for reverse engineering techniques devoted to help programmers understand the design intent implied by code. As a regular software evolution activity, code refactoring should be aware of effects of existing design patterns while improving the design of software for future changes.

Classification of design patterns with different criteria can help better understanding the design intent and implementation details of each pattern in various application domains. The GoF book (Gamma et al., 1995) presents a generic classification in which three categories are included: creational patterns for object creation, structural patterns for the composition of classes or objects, and behavioral patterns for interaction and distributing responsibility among classes and objects. Other classification approaches have also been proposed. For instance, design patterns can be classified according to their relation to pattern detection techniques of reverse engineering (Lee, Youn, and Lee 2008). Static structural patterns, such as composite and template method patterns, can be detected by examining relationships among classes and formalized by a UML class diagram. Dynamic behavior patterns, such as observer and abstract factory patterns, can be detected with the combination of dynamic and static analysis. Program-specific patterns, such as singleton and iterator patterns, can be detected by finding specific keywords or coding styles.

Other research is devoted to the formal specification of design patterns (France et al., 2004; Sterritt et al., 2010). Precise specification can supplement existing textual and graphical descriptions by eliminating semantic ambiguity, allow rigorous reasoning about patterns, and facilitate automation of the activities related to them (Brichau et al., 2008). The specification of a design pattern consists of a collection of roles representing structural and behavioral information

of the program, which in many studies can be modeled with UML Class and Sequence diagrams or similar concepts respectively.

Many algorithms have been proposed to improve the precision and recall of pattern detection results while discovering the design knowledge in source code (Heuzeroth et al., 2003; Tsantalis, and Chatzigeorgiou 2006; Dong, Sun, and Zhao 2008; Gupta et al., 2011). Design pattern detection can be expensive due to the large size of software system, the complexity of programming language features, and the existence of implementation variants of individual patterns (Stencel, and Wegrzynowicz 2009; Fontana, Maggioni, and Raibulet 2011). It would be helpful if detected patterns could be formally specified and integrated with development environments as pattern enforcement and reminder.

2.4 Metamodels for Code Analysis

Code analysis must rely on a repository, a shared database accumulating knowledge about the code. The most critical design aspect of a repository is its data metamodel, a structure describing the semantics of language entities to which a program model must conform. Some metamodels can be generated automatically (e.g. using language grammars to derive AST-level metamodel). Others are designed manually in order to be more abstract and to combine human judgment on the essentials of a given language. In this section we review six well-known metamodels that are designed by hand for different program analysis purposes and that inspire our research.

2.4.1 UML Metamodel

The Unified Modeling Language (UML) is a standard language created by the Object Management Group (OMG) for specifying, visualizing, constructing, and documenting the artifacts of software systems. The UML metamodel is a description of UML in UML. It

describes the objects, attributes, and relationships necessary to represent the concepts of UML within a software application. In detail, it defines the static, structural constructs (e.g., classes, components, nodes artifacts) used in various structural diagrams such as class diagrams, component diagrams, and deployment diagrams. It also specifies the dynamic, behavioral constructs (e.g., activities, interactions and state machines) used in various behavioral diagrams such as activity diagrams, sequence diagrams, and state machine diagrams. The rich notations defined by the UML metamodel allow developers and customers view a software system from a different perspective and in varying degrees of abstraction. Its extensibility and specialization mechanisms make possible to extend the core concepts for general-purpose modeling tasks. The experience of modeling practices in modern industries shows that using UML notations to model software applications is a practical and useful to represent and understand features of software systems.

The advantages of the UML metamodel make it a good candidate to represent program models and to reason about code. However, the UML metamodel has several issues that prevent it from fulfilling its responsibilities for this task.

- UML metamodel ignores the fact that there is a need to represent the structure and language-related details of the source code. For example, the UML metamodel does not handle file inclusion and name resolution. As another example, the generalization relationship in UML class diagram metamodel does not have attributes that can state if it is a virtual inheritance. This will become an issue when reverse engineering C++ code since C++ uses the keyword “virtual” to avoid the problem of ambiguous hierarchy composition (known as the "diamond problem") caused by multiple inheritance.

- Controversies exist on mappings between UML model elements and program semantic entities. A standard bridge has not appeared between UML and C++ to eradicate the inconsistent mappings implemented by different reverse engineering tools. For example, UML has distinctions between classes, interfaces and data types while C++ does not provide a rich enough vocabulary to distinguish these classifiers (Sutton, and Maletic 2007). Our previous studies on reverse engineering tool evaluation also reveal similar issues (Liang 2009; Liang, Kraft, and Smith 2009). We compared the UML class diagrams extracted by the tools *Doxygen* and *StarUML* and found several representation differences for the same language entities. For instance, *Doxygen* represents type references by stripping out pointer, reference and array symbols whereas *StarUML* keeps those symbols as part of the type name.
- Some research work proposes a set of extensions to the UML metamodel to make UML metamodel provide sufficient information to maintain the consistency between the model and the code (Gorp et al., 2003). However, the mapping issues mentioned above that are associated with core UML concepts still exist. Meanwhile, the extended concepts need additional clarifications and discussions before they will be widely accepted.

2.4.2 Columbus

The Columbus metamodel is designed for the Columbus reverse engineering framework that enables the parsing, analyzing, internal representation, filtering and exporting of information extracted from C/C++ source files into various formats including XML. It is represented with standard UML class diagram notation, which improves its comprehensibility. The Columbus project uses CAN extractor to retrieve ASG schema instances from C++ source code. Columbus

ASG schema reflects the low-level structure of the code, as well as higher-level semantic information in terms of six packages: a *base* package containing the base classes and data types for the remaining parts of the schema, a *struc* package modeling the main program elements such as objects, functions and classes, a *type* package representing elements in *struc*, *templ* and *expr* packages, *templ* package representing template parameter and argument lists, a *statm* package for modeling the statements, and an *expr* package for expressions (Ferenc, and Beszédes 2002).

2.4.3 Datrix

The Datrix metamodel is designed by Bell Canada to support C/C++ and Java source code analysis. Similar to the Columbus metamodel, Datrix includes ASG-level information to represent complete semantics of programs. There are two significant differences between Datrix and Columbus. First is the hierarchical structure to store language entities. Datrix organizes language elements in a project by compilation units. The Columbus metamodel gives namespace scope precedence over file scope. It requires that there must always be at least one namespace object representing the top-level namespace of the project files. The second is the representation of types. Datrix has a straightforward semantic representation whereas Columbus directly reflects the syntactic decomposition of the type declaration (Ferenc et al., 2001).

The Datrix metamodel is represented as Entity-Relationship model (ERM). ERM is useful for modeling relatively simple data and relationships between the data. However, it does not allow specification of interactions between the data. More importantly, it lacks constraint representations to specify complicated conceptual models more precisely. ERM also lacks standard facilities to support query and navigation across the model. Since UML has become the de-facto standard for object-oriented modeling, a metamodel represented with UML notations is preferable to improve its comprehensibility.

2.4.4 Chen's Metamodel

Chen et al. formulates a C++ metamodel for reachability analysis and dead code detection using ERM (Chen, Gansner, and Koutsofios 1998). This metamodel captures entity for all constructs not defined within function bodies, plus the nested components of any entity representing a type. In particular, it includes types, functions, variables, macro and files along with their inheritance, friendship, containment, instantiation, and reference relationships.

Chen's metamodel for C/C++ programs is designed for specific analysis tasks. It catches all necessary information and represents it with appropriate granularity. This avoids performance issues caused by model bloating while achieving analytical precision.

2.4.5 Famix

Famix is the core metamodel for modeling static information in the context of the Moose reengineering framework (Moose 2011). It is designed to be language-independent for the exchange of information about object-oriented software systems. The latest version Famix 3.0 models languages with two groups. The first group contains two types of named program entities: parameters and variables are leaf entities, namespaces, classes, methods, functions and types are container entities. The second group represents the associations between entities in terms of invocations, variable accesses, references and class inheritances.

Famix highlights core concepts that should be handled in different languages, resulting in the reuse of analysis and the reduction of language features specific to code change. The Famix metamodel has a hierarchy representation structure, which is helpful for reengineering tool builders to develop tools that are interoperable. However, Famix metamodel is not suitable to be used directly by programmers for model-based code analysis. First, as those metamodels represented with ERM, it lacks support for model navigation and constraint specification. Second,

since refactoring is language-specific code change, Famix as a language-independent metamodel must be extended for real use.

2.4.6 Dagstuhl Middle Metamodel

The Dagstuhl Middle Metamodel (DMM) is a commonly accepted and extensible metamodel for static program analysis (Lethbridge 2003). It captures program level entities and their relationships in language-independent way. It is a middle-level model representation, rather than a full low-level AST or a high-level architectural abstraction. The primary structure of DMM is composed of three hierarchies: a model element hierarchy for language conceptual entities, a relationship hierarchy for associations among entities, and a source object hierarchy for separating source chunks from abstract elements they represent.

DMM uses UML-style metamodel representation, in which each relation is represented as association class with its own attributes. This is especially useful to embody complicated relations between language entities and enable bi-directional navigations. With regard to the details captured, DMM handles neither multi-part references to members such as the expression "a.b.c()", nor local variables. This may miss some part of the relationships of invocation and access. As a language-independent metamodels, DMM merges similar concepts of different languages and represents them with unique terminologies. This could be an issue for metamodel users specialized in a particular language.

We have introduced six metamodels whose detailed descriptions are available in literature. As a summary, in Table 2.1 we make comparisons on their primary differences on metamodel representation, design purpose, granularity details and language dependence.

Table 2.1 Comparisons on Existing Metamodels

Metamodel	Representation	Design Purpose	Granularity of Details	Language Dependence
UML	UML	Software design	Member-level, extensible	No
Columbus	UML	Reverse engineering	ASG-level	Specific for C/C++
Datrix	ERM	Reverse engineering	ASG-level	No
Chen	ERM	Reachability analysis Dead code detection	Method-level	Specific for C/C++
Famix	relaxed UML	Reengineering	Method-level details with references, extensible	No
DMM	UML	Static program analysis	Middle-level, extensible	No

CHAPTER 3 A METAMODEL FOR C++ SOURCE CODE REFACTORING

This chapter presents a metamodel that will construct the knowledge base needed to specify refactoring preconditions for C++ programs. We first address the necessity of such a metamodel, based on the analysis of the cons and pros of well-known metamodels we present in Chapter 2. We discuss the characteristics of the C++ language that make refactorings on the source code a challenge, and investigate previous studies on preconditions of primitive refactorings. Both give us clues as to what information should be included in the metamodel and finally direct the design of metamodel required for our research purpose. We then present the design of the metamodel with UML class diagram notations and explain the structural details. At the end of this chapter, we propose a refactoring process where model-based precondition specification takes place using the metamodel we design.

3.1 Requirements on the New Metamodel

We need a metamodel that facilitates refactoring precondition specification and checking. However, what are the requirements for such a metamodel? Can existing metamodels meet these demands? The following discussion on the desired requirements will help to answer these questions.

(1) Metamodel Representation: The expected metamodel diagram is visualized with UML class diagram concepts to represent the static information of programs. UML graphic notation techniques have been widely adopted in the field of object-oriented software engineering. Programmers who are familiar with UML diagrams should be able to easily

understand the metamodel represented as a UML diagram. The choosing of UML as the graphical modeling language for the metamodel does not impose any limit on the type of exchange formats. Different modeling tools can have their own format to document the metamodel. In this dissertation, we use the Eclipse Modeling Framework to design our metamodel, which is then serialized into a document with XMI format. For the details, see Chapter 6.

(2) Language dependence: The subject language we focus on is standard C++ (ISO/IEC 14882:2003 2003). Particularly, we are interested in the source code instead of machine code. Although there are refactoring principles that are true for software in any language, we believe that different features of programming languages do have significant impact on refactoring decisions. Focusing on a particular language, in our case C++, means the metamodel needs to represent its language-dependent features and concepts. For ease of use, an unambiguous mapping should be established between model elements and language entities so that the metamodel design can follow the syntax and semantic definitions of C++. As a mainstream object-oriented language, C++ is more complicated than other languages, which makes the semantic preservation analysis challenging, and some seemingly straightforward refactoring surprisingly hard to implement.

(3) Design purpose. The metamodel should represent static structure information extracted directly or inferred from the C++ source code. Even though dynamic analysis may be more accurate or effective in some cases, it is more expensive and complicated than static analysis in most other cases, because of the number of test cases that need to be implemented and the large amount of data need to be collected (Kataoka et al., 2001). The metamodels we review in Chapter 2 are designed to be versatile. We believe further

analysis on the features of different refactoring activities will be helpful to generate a more suitable metamodel.

(4) Granularity of details. As we stated in Chapter 2, program ASTs provide most of the information needed to support static code analysis and transformation for refactoring. However, an AST-level metamodel is best used as an all-purpose knowledge base to serve as a higher-level abstraction of information. For our purpose of precondition specification, an AST-level metamodel is too complicated for programmers to handle in a declarative way. In addition, our research focuses on "floss refactoring", a frequent activity that interweaves with normal program development (Murphy-Hill, and Black 2007). This type of refactoring needs fast precondition checking and result generation to improve development productivity. This requirement drives us to think carefully about the granularity of information the metamodel should hold to make precondition checking scalable. Investigations on previous studies on primitive refactorings will help determine the metamodel granularity level that is detailed enough to provide the information needed and coarse grained enough for comfortable handling. We work on the question in Section 3.3.

With all the considerations above, we will design a new metamodel for precondition specifications for C++ source code refactoring. In the next section, we discuss the characteristics of C++ that challenge refactoring and their impact on our metamodel design decision, which is associated with the discussion on language dependence in this section.

3.2 The Features of C++

We decide the language features covered by the metamodel will be a major set, instead of a complete set of standard C++. The following features are not considered in our metamodel design.

(1) Preprocessor directives: preprocessor directives are not part of the C++ language, and have been regarded as a primary challenge to C++ code refactoring (Tokuda, and Batory 2001). Most studies solve this issue by implementing refactoring on preprocessed code.

(2) Templates: Templates are a popular feature of C++ facilitating generic programming. In this dissertation we do not represent templates and their instantiations. This will become part of our future work.

(3) Type casts: Explicit type casts are generally recognized as poor coding practice. Handling type casts would require more complex flow analysis, which we think is not suitable to be implemented by programmers specifying refactoring precondition.

(4) Exception handling: exception handling is a construct designed to handle the occurrence of exceptions, special conditions that change the normal flow of program execution and that cause abnormal terminations in the program. C++ uses *try* and *catch* blocks for exception handling. As we discuss in the next section, our metamodel does not contain statement level information, including *try* and *catch* blocks. However, references in both types of blocks will be stored in the function scope.

We identify some important C++ language-specific features and concepts that cannot be ignored:

(1) Inheritance: the properties of a base class can be inherited public, protected or private. C++ allows single and multiple inheritances, and uses virtual inheritance to handle

situations where member with the same names are inherited from more than one base class.

(2) Overriding: A virtual method's behavior can be overridden within an inheriting class by a method with the same signature. Overriding is the way to achieve dynamic polymorphism.

(3) Overloading: This property allows methods with same name but different signatures declared in one class.

(4) Friendship: This is a special feature of C++. A C++ class can grant friendship to classes or functions and permit them to use its private and protected members.

3.3 Analysis of Primitive Refactorings

Since the number of refactorings is consistently growing, it is necessary to scale down our search on refactoring rules to a reasonable level. To do that, it is necessary to pay attention to primitive, interface-level refactorings. Refactoring theory and tools have proposed a finite set of primitive refactorings, which can be combined into larger composite refactorings (Opdyke 1992) (Roberts 1999). For example, *ExtractClass* refactoring extracts separate concepts and corresponding members of a class into another class. It can be achieved by a sequential implementation of *addClass*, *moveField* and *moveMethod* refactorings. In several empirical studies on software evolution, researchers have found that refactoring takes a very important role on program structural evolution: over 70% of the changes pertain to structural evolution and semantic preservation, and these changes can be reproduced by a combination of primitive refactorings (Tokuda, and Batory 2001; Dig, and Johnson 2005; Xing, and Stroulia 2006). On the other hand, software development practice also shows that primitive refactorings, especially renaming and moving refactorings, are among the most frequently used refactorings in a daily

software development process. Considering the importance of primitive refactorings, we believe that a literature search on studies of primitive refactorings help us identify the required data used in the metamodel design.

Table 3.1 is a list of 19 primitive refactorings. From left to right, the first column contains symbols as names of refactorings. These names are used consistently throughout this dissertation. The second column gives the meaning and implementation of each refactoring. The third column lists the primary concerns for each refactoring precondition. The rightmost column provides the code information needed to check those concerns. All this information is included because of our study on literature, primarily (Opdyke 1992; Roberts 1999; Fowler 1999; Tichelaar et al., 2000; Tokuda, and Batory 2001). The primitive refactorings under analysis are adding, renaming, moving and deleting operations on classes, class members and parameters.

The primary concerns for all listed refactorings focus on these aspects:

(1) Name hiding: Name hiding occurs when the same name is explicitly declared in a nested declarative region or derived class (ISO/IEC 14882:2003 2003). Name hiding is the primary cause of changes in the semantics of a program being refactored (Opdyke 1992).

(2) Name redefinition/collision: This happens when duplicate names are declared in the same scope. However, this does allow method overloading. Name redefinition will cause compilation errors.

(3) Reference: An entity is referenced in the program.

(4) Reachability: When the purpose of the refactoring is to move a class member across two different classes, it may encounter a reachability problem. Reachability analysis checks if all references to the member to be moved are still resolvable with the original format. For instance, a call on non-static method via a class object will produce a syntax error if

the method is pushed down to the derived class since the method will be unreachable via its original declaring class.

(5) Control flow: Among all 19 primitive refactorings, *extractMethod* is the one that is concerned most about control flow. For example, one of its preconditions is that the statements to be extracted cannot contain a conditional statement with a return statement inside of it (Murphy-Hill, and Black 2006). To check this, control flow analysis must be implemented and statement-level information is required.

Based on the required information for precondition checking, we decide our metamodel should represent class member-level information. The statement and expression level information may bring more accurate refactoring results for each refactoring and is fundamental for *extractMethod* refactoring. However, we have two reasons for not representing statements and expressions. First, we want precondition checking that can be performed quickly, and a reduction in the size of program models is a good choice. Second, control flow analysis is too complicated to be a suitable task undertaken by programmers for refactoring purpose. Generally, this is done by special code analysis tools.

Representing class member level information does not mean we ignore the entire function body. Local variables and references are indispensable for some of primitive refactorings and will be covered by the metamodel.

Table 3.1 Analysis on Primitive Refactorings

Refactoring	Definition	Primary Concern	Information For Precondition Checking
addClass	Add a new and empty class as a derived class of an existing class	name collision	Names of language entities, scopes
renameClass	Rename a class, its constructor and destructor; update references	name collision	Names of language entities, scopes
removeClass	Remove a class and all of its members if the class is never referenced.	reference	Class hierarchy, class members, references
addMethod	Add a method with an empty body into a class	name collision name hiding	Class hierarchy, class members
renameMethod	Rename a method (and its overriding/overridden methods, if any) and update its references	Name collision Name	Class hierarchy, polymorphism detection and references of methods.
removeMethod	Remove a method from its containing class	reference	References, methods
pullUpMethod	Methods with identical results are moved to the superclass.	Reachability Name collision Name hiding	Class hierarchy, references, accessibility, friendship
pushDownMethod	Behavior of a base class is relevant only to one of its derived classes. Move it into that derived class.	Reachability Name collision Name hiding	Class hierarchy, references, accessibility, friendship,
extractMethod	Extract a portion of statements from one method and move to another method	Control flow Reachability Name collision Name hiding	Local variables, statement-level information, references
inlineMethod	Replace the method call with the method body	Reachability	Class hierarchy, parameters, local variables, class members, references, friendship
moveMethod	Move a method from its containing class to another class in different class hierarchy	Reachability Name collision	Class hierarchy, polymorphism, method signature, references, friends
addParameter	Add a parameter into the method's parameter list	Name collision Name hiding	Class hierarchy, local variable, methods
removeParameter	Remove an unused parameter from the method's parameter list	Reference Name hiding	Class hierarchy, local variable, methods, references
addField	Add a new data member into the class	Name collision Name hiding	Class hierarchy, class members
renameField	Rename the data member and change all of its references	Name collision Name hiding	Class hierarchy, class members
removeField	Remove an unused data member from the class	Reference	Class member information
encapsulateField	Create getting and setting methods for a data member for public access	Name collision Name hiding	Accessibility, class members
pullUpField	Each derived classes declares the same field. Move it to the base class to avoid duplicates.	Name collision	Class hierarchy, member information
pushDownField	Move a data member declared in the base class to one of its derived class	Reachability Name collision	Class hierarchy, access specifier, member information, references

3.4 Metamodel Design

After the anlysis of the challenges of C++ with respect to refactoring, and the required information needed for primitve refactorings, we present the metamodel shown in Figure 3.1 for the specifications of refactoring preconditions.

In the metamodel, we use three types of UML connections to represent the relations between elements: composition is a filled diamond shape on the containing class end that connect contained class(es) to the containing class; generalization/inheritance is a hollow triangle shape on the superclass end of the line (or tree of lines) that connects it to one or more subtypes; unidirectional association represents all other relations between elements and is navigable only one direction from the end without the arrowhead to the end with the arrowhead.

Classes in the metamodel model the main program elements and are organized via UML composition relation according to their scoping structures. At the top of this metamodel is an abstract class *Member*. It has derived classes *Scope*, *NamespaceAlias*, *Enumeration*, *Enumerator* and *Object*. It also has attributes of name, access specifier (private, protected and public) and storage specifier (static, extern, mutable, auto, register). The class *Object* represents both variables and data members declared in a *Class* object. In C++ an enumeration is a user-defined type consisting of a set of named constants called enumerators. This information is represented as a class *Enumeration*, with an *Enumerator* and a containment relation. The association edge from class *Member* to *Scope* denotes the navigation direction and means that the scope of a given language entity can be obtained directly from the model.

The abstract class *Scope* has three concrete derived classes *Namespace*, *Class* and *Function*, corresponding to three of the scopes in C++. Our metamodel does not represent file scope or local scope. Each *Scope* object can contain different types of *Member* objects. For example, a

Class object can contain *Object* objects (class data members), *Function* objects (class methods), and other *Class* objects (nested classes). Each C++ program has a global scope, which is represented as a *Namespace* object and is the container of all other model objects. Each *Namespace* object may have alias. Each *Scope* object also contains a collection of *Reference* objects.

The concrete class *Class* stands for a C++ class, struct or union. A C++ class is abstract if it declares at least one pure virtual method. The Class *Inheritance* represents inheritance relations between base and derived classes and has properties of *isVirtual* and *accessSpecifier*. Each *Inheritance* object is a component of a *Class* object that is a derived class in the program, and contains a pointer to another *Class* object that is the base class of the derived class containing the *Inheritance* object. A C++ class may also have friend classes and friend functions, so we have UML association *friendClasses* and *friendFunctions*. As a type of scope, a class can declare attribute, method, inner classes and other constituents in its scope. The inheritance between class *Scope* and *Class* in the metamodel reflects this concept.

The *Function* class represents function scope and can be included by the other scopes *Class* or *Namespace.* We separate constructors (with or without keyword *explicit*), destructors, type conversion functions and operator functions from normal functions because of their special behavior. A *Function* object can contain a sequence of *Parameter* objects and can be virtual, pure virtual, inline or explicit. Through the inheritance relation between the *Scope* class and *Function* class, the accessibility and storage type are also represented for each Function object. A *Function* object may also contain local variables.

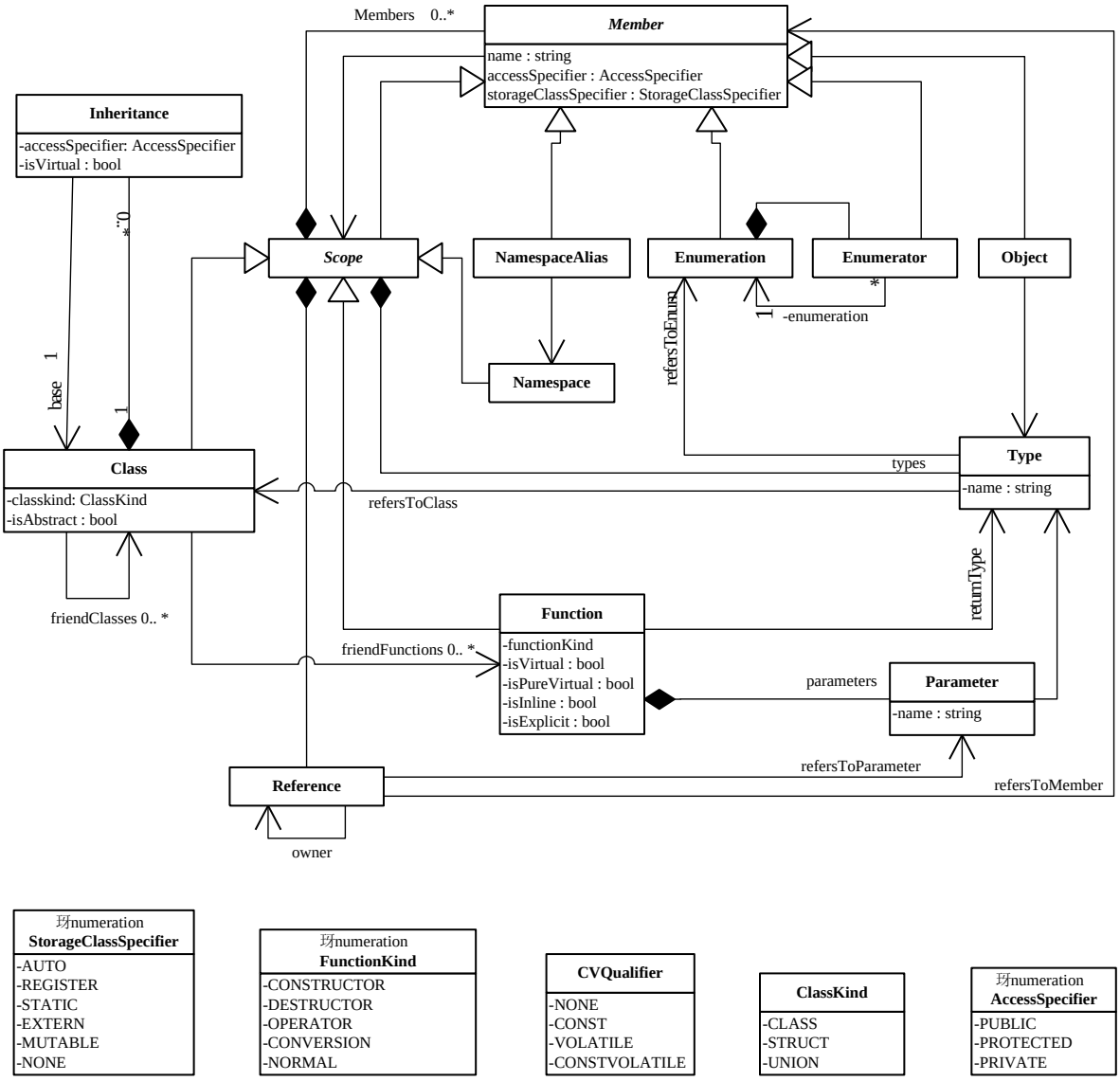


Figure 3.1 the Metamodel For Precondition Specification of Refactoring C++ Programs

The *Object* class is a concrete class derived from the *Member* class. It represents class data members, global variables or local variables in C++. The association connected from *Object* class to *Type* class represents the type of each C++ object.

The *Type* class is used to represent the types of C++ functions, objects and parameters. Each *Type* object has a string name that combines symbols such as *, &, const and volatile. For any type that is referred in a scope, a unique *Type* object will be generated and stored in that scope. A

Type object has an association connection (*refersToClass* or *refersToEnumeration*) that associates it with a user-defined *Class* object or *Enumeration* object. If a type, such as built-in type, is from a library, the program model will not generate an association.

The *Reference* class is used to represent references in C++ expressions. The reference resolution is represented by the edge *refersToParameter* if a parameter is referenced, or the edge *refersToMember* if the identifier refers to other types of C++ entities such as object, function or class. The edge *owner* corresponds to member selection operators (dot or arrow) or scope resolution operator (double colon) in C++. The object diagram in Figure 3.2 illustrates how the expression "a.b.c().d" is represented using our metamodel. We assume the symbol "a" is a variable declaration.

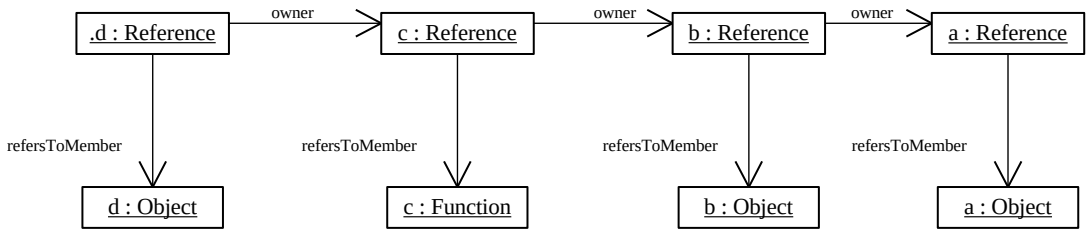


Figure 3.2 Reference Instance for the Example

3.5 Refactoring Process Featured with Model-Based Precondition Specification

In this section, we depict the refactoring process where developers can specify their own refactoring preconditions. In Figure 3.3, each parallelogram represents a data set that is the input or output of processing. Each rectangle represents a functional module that processes input data set and produce an output result. As with most refactoring tools, the source code is translated into ASTs by the compiler frontend. ASTs are stored in memory and ready for use. An abstractor works on ASTs and extracts a higher-level abstraction representation for the program. This

representation must conform to the metamodel structure that is designed for precondition specification.

Developers can specify refactoring preconditions using a precondition editor. Hence, these preconditions can reflect their opinions on refactoring as a combined result of their expertise, and understanding of the source code, and other constraints on the current code. Using the specified preconditions, the validator evaluates the program model. If the precondition is satisfied, the validation results will be passed to the code transformation engine, which rewrites the program ASTs and generates well-formatted source code from it.

Developers do not need to specify the preconditions every time they want to perform refactoring. This step can be skipped if they do not want to add new refactorings or update the preconditions of existing refactorings.

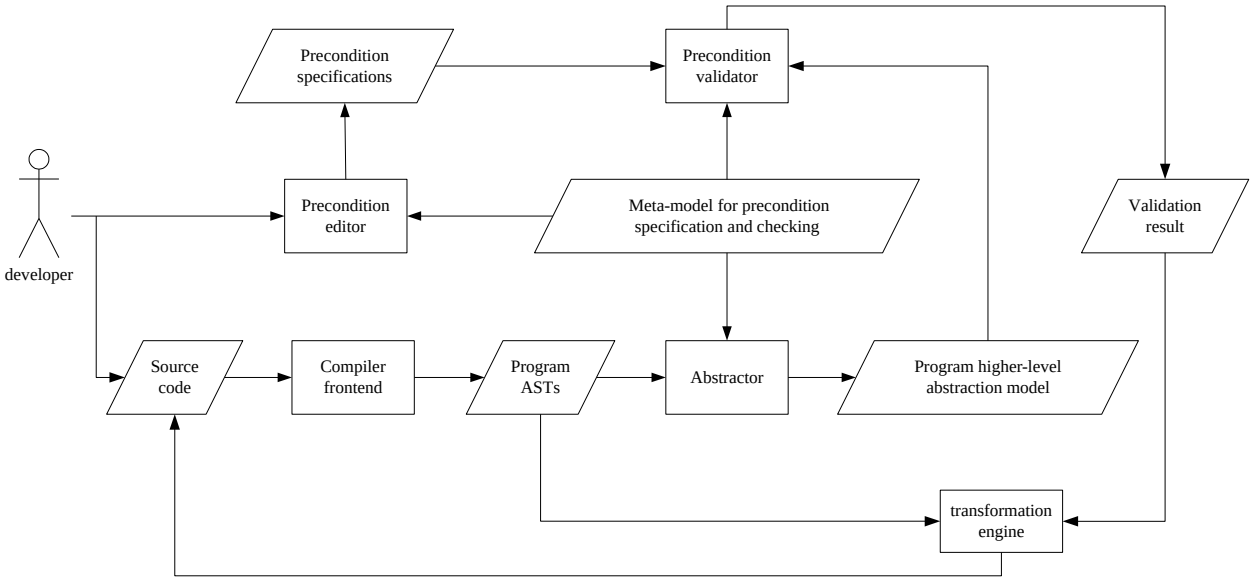


Figure 3.3 Refactoring Process Featured with Model-Based Precondition Specification

CHAPTER 4

PRECONDITION SPECIFICATIONS FOR PRIMITIVE REFACTORINGS

This chapter addresses our solutions using Object Constraint Language (OCL) to specify preconditions for primitive refactorings against the metamodel we present in Chapter 3. We first present an overview of OCL. We then formalize the way we describe each of the refactoring preconditions. The details of precondition specifications are presented in the rest of the chapter. These preconditions endeavor to guarantee syntax correctness and semantic preservation.

4.1 Object Constraint Language Overview

OCL is an expression language used to specify constraints in object oriented models (in particular UML models) to provide more relevant aspects to complete and clarify the semantics of UML diagrams of software systems. OCL has a close relationship with the metamodel designed for a specific application domain. Each OCL expression must be specified in the context of a metamodel element in order to navigate across the metamodel.

OCL is not a programming language and cannot express program logic or flow control. However, this does not influence its applicability in many domains such as querying information across models, specifying invariants on classes and types in class models, and describing pre- and post- conditions on operations and methods. OCL defines basic types such as Integer, Boolean, Real and String. It also provides collection types such as Bag, Set and Sequence. Appendix A highlights the rest of those that are used in our work. All OCL expressions presented in this dissertation conform to the OCL 2.0 specification released by OMG (OCL 2009)

Several categories of language/toolkits for source code browse, analysis, comprehension or even transformation have been developed during past two decades such as SOUL (Mens et al., 2005), JUNGL (Verbaere, Ettinger, and Moor 2006), SCL (Hou, and Hoover 2006) and Datalog (Cali, Gottlob, and Lukasiewicz 2009). However, the use of these languages/toolkits requires training, and a certain amount of skill. There are several advantages to using OCL. First, UML and OCL are de-facto standards thus learning effort is expected to be reduced. Second, OCL is powerful enough to express in a concise and elegant way very complex conditions. Studies that use OCL to query AST have also presented the capabilities of OCL on general code static analysis (Antoniol, Di Penta, and Merl 2003) (Seifert, and Samlaus 2008). Considering the benefits, we decided to use OCL to describe preconditions of refactoring.

4.2 Description Template for Precondition Specifications

In the rest of this chapter, we present the preconditions of 18 primitive refactorings that are specified against the metamodel we propose in section 3.3. These refactorings are grouped into four categories according to the types of language entities to which they apply: Section 4.3, 4.4 and 4.5 are primitive refactorings on classes; Section 4.6 through 4.10 are primitive refactorings on methods; Section 4.11 and 4.12 are on parameters; and refactorings associated with class data member (field) are introduced in Section 4.13 to 4.17. We use a template to describe the precondition for each primitive refactoring. The section label stands for the name of a refactoring operation. Each section consists of four components to describe the precondition specified for that refactoring.

(1) **Goal**: describes the requirements on the precondition specification for a specific primitive refactoring.

(2) **Parameter**: explains the required information when precondition specifications are defined in an OCL operation body. For the arguments representing model elements, we do not check their existence and relationship with other elements, assuming this information truly exists in the program model. For instance, the precondition specification for *removeMethod* refactoring needs an input argument to indicate the *Function* object to be removed from the class. We do not check the existence of this Function object in the class. For user-typed information such as strings, the precondition specification includes examinations of their validity.

(3) **Precondition specification**: the precondition of each refactoring is defined in terms of an OCL operation with a Boolean return type in the context of a particular model element. Each precondition contains OCL expressions that constitute the operation body. The precondition specification for each refactoring has two foci. The first is syntactical correctness: a refactoring should not invalidate a syntactically correct program. The second is semantic preservation. For the insertion refactorings such as *addClass, addMethod, addParameter, addField* and *encapsulateField* refactorings that insert new entities into the program, their preconditions primarily check the issues of name collision/redefinition and name hiding. These issues are also the main concerns of renaming refactorings *renameClass, RenameMethod and renameField* that try to rename language entities. The preconditions of removing refactorings that aim to delete entities, including *removeClass, removeMethod, removeParameter and removeField,* are centered on checking references. Moving refactorings *pullUpMethod, pushDownMethod, inlineMethod, moveMethod, pullUpField* and *pushDownField* are used to move entities from the current scope into another scope. They have the most

complex precondition specifications—in addition to the concerns mentioned above, they also include reachability analysis. We define a set of OCL analysis operations to simplify precondition specifications. OCL analysis operations are predicates in the first-order predicate calculus expressions. They can also be implemented as individual operations to query model elements. Implementation details of analysis functions used in this dissertation are listed in Appendix B.

(4) **Discussion**: In this part, we first explain the precondition specified by OCL expressions. Then related issues are discussed. Alternative precondition specifications for some refactorings are also presented, which illustrate the necessities of supporting various precondition specifications for individual refactorings.

For all precondition specifications in the rest of this chapter, we assume that the program under refactoring is well-formed by the rules of C++ programming language.

4.3 addClass

Goal: check if it is allowed to add into the current scope an empty class as a derived class.

Parameter: --cname: a string representing the unqualified name of the new class.

--base: the base class of the new class to be added

Precondition Specification: the precondition is defined in the context of *Scope*. See Figure 4.1.

```
context Scope::addClass(cname:String,base:Class):Boolean
body:  --current scope does not contain a member with the name cname
       self.members.name->excludes(cname)
```

Figure 4.1 Precondition Specification for *addClass* Refactoring

Discussion:

From the program metamodel we can see that a scope can be a namespace, a class or a function, therefore this precondition is applicable to any *Namespace*, *Class* or *Function* object. The precondition requires that no member is declared in the current scope to avoid type redefinition and type hiding. Syntactically, one scope cannot contain two classes with equal names. Adding a new class may cause type redefinition if the current scope already contains a class or enumeration with the same name. In addition, if a class or enumeration and an object, function or enumerator is declared in the same scope (the order does not matter), the class or enumeration name is hidden wherever the object, function, or enumerator name is visible (ISO/IEC 14882:2003 2003). Type hiding may not cause syntax errors on existing code since the new class is not referenced yet. However, this definitely makes the new class useless.

The specification above cannot detect two other cases that may cause type collision. First, members declared in other namespaces may be dumped into the current namespace via *using* directive. Second, the program model does not store all elements of included libraries but those that are referenced in the program. In both cases, adding a new class may cause issues due to ambiguous symbol after this refactoring.

4.4 renameClass

Goal: check if it is allowed to change the name of the current class into a new given name.

Parameter: -- newName: the new name for the current class.

Precondition Specification: the precondition is defined in the context of *Scope*. See Figure 4.2.

Discussion:

The purpose of condition (1) is easy to understand. The intent of condition (2) and relevant issues are basically the same as those we discuss in *addClass* refactoring. Condition (3) checks the member names of the current class. The reason is that if there is a member with a name the

same as the current class, the compiler will treat it as constructor or destructor with a return type. This is definitely an error and should be detected in the process of precondition checking.

```
context Class::renameClass (newName:String):Boolean
body: --(1) newName is not null and not equal to the name of current class
        newName.size()>0 and self.name<>newName
        and--(2) the scope containing the current class does not contain a member named newName
        self.scope.members.name->excludes(newName)
        and  --(3) current class does not contain a member named newName
        self.members.name->excludes(newName)
```

Figure 4.2 Precondition Specification for *renameClass* Refactoring

4.5 removeClass

Goal: check if it is allowed to remove an existing class from the current scope.

Parameter: -- c: a *Class* object declared in the current scope.

Precondition Specification: the precondition is defined in the context of *Scope*.See Figure 4.3.

```
context Scope::removeClass(c:Class):Boolean
body: --(1) c does not have derived class
      c.getDirectDerivedClasses()->isEmpty()
      and--(2) c is never referenced
      (let publics: Set(Member) = c.members->select(m|m.accessSpecifier=AccessSpecifier::PUBLIC) in
      Reference.allInstances().refersToMember->asSet()->excludesAll(publics))
```

Figure 4.3 Precondition Specification for *removeClass* Refactoring

Discussion:

If a derived class accesses protected or public members of its base class, those references will become irresolvable or resolved incorrectly after the base class is removed. Condition (1) solves this problem by checking that the class to be removed should not have derived classes.

getDirectDerivedClasses() is a pre-defined OCL analysis operation that finds all direct derived class of current class. See Appendix B for details. Condition (2) checks if any of the public members of the class being removed are referenced. If so, the class cannot be removed. According to the checking algorithm expressed in Condition (2), if the class has been instantiated but no instance is ever referenced, precondition checking still returns true. The compiler may complain after refactoring since the type of the variable cannot be found. However, this can help programmers find those dead variables.

One may consider Condition (2) too restrictive because there should not be a problem with removing a class that references its members internally. Condition (2) can be relaxed just to check if the class being removed is referenced externally. The implement of this checking consists of finding the scopes external to the class, and then verifying that no reference contained in those scopes resolves to any of the declared members. See Figure 4.4.

```
Context Scope::removeClass(c:Class):Boolean
body: --(1) c does not have a derived class
        c.getDirectDerivedClasses()->isEmpty()
and --(2) c is not referenced externally
         (let len:Integer= c.getFullQualifiedName().size() in
          let scopes:Set(Scope) = Scope.allInstances() in
          let innerScopes:Set(Scope) = scopes->select(s| s.getFullQualifiedName().size()>=len
                    and s.getFullQualifiedName().substring(1,len) = c.getFullQualifiedName()) in
          let outsideScopes:Set(Scope) = scopes->symmetricDifference(innerScopes)->excluding(c) in
         outsideScopes.references.refersToMember->asSet()->excludesAll(c.members)
)
```

Figure 4.4 Alternative Precondition Specification for *reomveClass* Refactoring

4.6 addMethod

Goal: check if it is allowed to add a new method with empty block into the current class.

Parameter: --methodName: the name of the new method

--paramSeq: a sequence of parameter types of the new method

--returnType: the return type of the new method represented as a string

--isVirtual: a Boolean argument indicating whether or not the new method is virtual

--isPureVirtual: a Boolean value indicating whether the new method is pure virtual

--accessibility: a String indicating the access control mode of the new method

Precondition Specification: the precondition is defined in the context of *Class* and contains three conditions. See Figure 4.5.

Discussion:

By condition (1), we intentionally reject the insertion of any 'special' method via a refactoring operation to avoid complicated and imprecise checking. By condition (2), a class declared with correct syntax should not have duplicate member names, except those members that are overloading member functions. Condition (3) is dedicated to semantic preservation. First, the new method name should be different from any inherited data members. Second, if base class has methods with the same name as the new method, these methods must be overriding/overridden methods with same signature and return type as the new method. This also means that if the new method has the same name but a different parameter sequence from any of the overriding/overridden methods in the class hierarchy, this refactoring is disallowed. This consideration avoids method hiding caused by the new method. The example code in Figure 4.6 explains the issue Condition (3) takes care of.

```
context Class::addMethod(methodName:String, paramSeq:Sequence(String),
                returnType:String, isVirtual:Boolean, isPureVirtual:Boolean, accessibility: String):Boolean
body: --(1) the new method cannot be a constructor, destructor, overloaded operator, type conversion function,
      --or pure virtual method
     isPureVirtual=false and   returnType.size()>0 and methodName<>self.name
       and methodName.substring(1,8)<> 'operator' and methodName<>'~'.concat(self.name)
     and--(2) the current class does not contain a data member with duplicate name,
          --or a function member with a signature the same as the new method.
      self.members->forAll(m|m.name=methodName implies
               (m.oclIsTypeOf(Function) and m.oclAsType(Function).parameters.type.name<>paramSeq))
     and --(3) the new method does not redefine members inherited from base classes.
          -- However,  this does allow the new method to be an overriding method
       self.getAllBaseClasses()->asSet().members->forAll(m|m.name = methodName  implies
          (m.oclIsTypeOf(Function)
            and  (m.oclAsType(Function).isVirtual or m.oclAsType(Function).isPureVirtual)
            and  m.oclAsType(Function).parameters.type.name = paramSeq
            and   m.oclAsType(Function).returnType.name = returnType))
```

Figure 4.5 Precondition Specification for *addMethod* Refactoring

In Figure 4.6, adding a new method *mf1* to the *Derived* class is not allowed since it hides *Base::mf1* and therefore triggers an error in L12. As an alternative refactoring idea, one may think the method *mf1* can be inserted into the *Derived* class since the error in L12 can be avoided by making all *mf1* methods visible from *Derived*, for example, by adding the statement "using Base::mf1" to the *Derived* class.

```
L1:   class Base{
L2:    pubic:
L3:          virtual void mf1();
L4:          void mf1(int);
L5    };
L6:   class Derived: public Base{
L7:   public:
L8:           virtual void mf1(); // the new method added
L9:   };
L10:  Derived d;
L11:  d.mf1(); //call Derived::mf1
L12:  d.mf1(8); //ERROR! Derived::mf1 hides Base::mf1
```

Figure 4.6 Sample Code Demonstrating the Issue of Member Redefinition

4.7 renameMethod

Goal: check if it is allowed to assign a new name to the current method.

Parameter: --newName: the new name to assign to the current method.

Precondition Specification: the precondition is defined in the context of *Function* and includes four conditions in total. See Figure 4.7.

Discussion:

Condition (1) verifies the validity of the new method name. Condition (2) makes sure that the method to be renamed is not a constructor or destructor, or an operator overloading and conversion function, but can be virtual or pure virtual method. Condition (3) checks the member names of the containing class of the current method, ensuring that the new method name does not redefine any declared member names. However, overloading the normal method is allowed with this constraint. Condition (4) is similar as Condition (3) except that it checks all of the

polymorphic classes relevant to the current method. We require that, for *renameMethod* refactoring, all overriding/overridden methods should have their name changed if any one of them is renamed. None of these changes should cause syntax errors in any class.

```
context Function::renameMethod(newName:String):Boolean
body:
 let curClass:Class = self.scope.oclAsType(Class) in
--(1) the new name is not null and must be different from the old name
 newName.size()>0 and self.name<>newName
       and newName<>self.scope.oclAsType(Class).name
and--(2) the current method is normal method
self.functionKind=FunctionKind::NORMAL
and--(3) the containing class of the current method does not already declare data member newName,
     -- or function members with signature equal to the new one.
 curClass.members->forAll(m|m.name=newName implies
                    (m.oclIsTypeOf(Function) and
                     m.oclAsType(Function).parameters.type.name<>self.parameters.type.name))
and  --(4) each polymorphic class does not contain any data member named newName
(let pClasses: Set(Class) = self.getPolymorphicClasses() in
pClasses.members->forAll(m|m.name=newName implies
                    (m.oclIsTypeOf(Function) and
                     m.oclAsType(Function).parameters.type.name<>self.parameters.type.name)))
```

Figure 4.7 Precondition Specification for *renameMethod* Refactoring

4.8 removeMethod

Goal: check if it is allowed to remove a method from the current class.

Parameter: --method: a *Function* object to be removed from the program model

Precondition Specification: the precondition is defined in the context of *Class* and contains three conditions in total. See figure 4.8 for details.

```
context Class::removeMethod(method:Function):Boolean
body: --(1) the method to be removed is not a pure virtual normal method
        method.functionKind = FunctionKind::NORMAL  and  method.isPureVirtual = false
        and--(2) the method to be removed does not override a pure virtual method
              not self.getDirectBaseClasses().getFunctionMembers()
                    ->exists(f|f.isPureVirtual and f.hasSameSignature(method))
        and --(3) the method to be removed is never referenced
              Reference.allInstances().refersToMember ->excludes(method)
```

Figure 4.8 Precondition Specification for *removeMethod* Refactoring

Discussion:

By Condition (1), any constructor, destructor, operator overloading, type conversion function or pure virtual function is excluded from methods to which *removeMethod* refactoring can be applied . These methods provide special functions and are very often called implicitly. For example, the use of a *new/delete* operator will call a constructor/destructor which is not represented directly in the program model. It is reasonable to refuse *removeMethod* refactoring on these methods for simpler precondition checking. Condition (2) is the guarantee of correct syntax in the program. In C++, a pure virtual method declared in the base class must be overridden in each of derived classes. Therefore, we decide that the method cannot be removed if it overrides a pure virtual method, even though it is never referenced. By condition (3), a method to be removed should not be referenced.

If a method is called only by itself (recursive function), removing it will remove all of its references and hence should be allowed. However, Condition (3) cannot identify this situation.

Figure 4.9 shows the expressions in place of Conditon (3) to solve this problem. It first finds all references that are function call. If, for each function call resolved to the method to be removed, the caller is always the method itself, we can determine that all calls are invoked internally from the method.

```
--(3) the method to be removed is only called by itself .
    let functionCalls:Sequence(Reference)=
        Reference.allInstances()->select(r|r.refersToMember.oclIsTypeOf(Function))->asSequence() in
    functionCalls->forAll(fc|fc.refersToMember.oclIsTypeOf(Function) = method implies
                                method.references->includes(fc))
```

Figure 4.9 Alternative Precondition Specification for *removeMethod* Refactoring

4.9 pullUpMethod

Goal: check if it is allowed to pull up a method from the current class to one of its base class.

Parameter: --method: the *Function* object to be pulled up

--base: the base class receiving the method

Precondition Specification: the precondition is defined in the context of *Class* and includes five conditions in total. See Figure 4.10.

Discussion:

This precondition is specified based on an assumption: the method, if its access control mode is private, will become protected in the base class after refactoring. This is to ensure that the method can be called by the derived class with same format before and after *pullUpMethod* refactoring. The code in Figure 4.11 demonstrates the benefit. In the left column is the code to be refactored. In the right column is the code after *pullUpMethod* refactoring on the method *B::print()*. The access mode of *B::print()* is changed to protected such that the method

B::printTwice() can still call it without any change. This assumption also applies to *pullUpField* refactoring to be introduced later.

```
context Class::pullUpMethod(method: Function, base:Class): Boolean
body:--(1) the method to be pulled up is a normal but not pure virtual method
      method.functionKind = FunctionKind::NORMAL  and  method.isPureVirtual = false
     and --(2)current class is an immediate derived class of base
     self.inheritances->exists(i|i.base=base and i.accessibility=AccessSpecifier::PUBLIC)
     and --(3) the base class does not contain a data member  with the same name as method
          -- or a method with the same signature.
      base.members->forAll(m|m.name= method.name implies
                                (m.oclIsTypeOf(Function) and
                                m.oclAsType(Function).parameters.type.name<>method.parameters.type.name))
      and--(4) the method to be pulled up does not access members declared in its containing class.
        method .references.refersToMember->asSet()->excludesAll(self.members)
     and --(5) the method to be pulled up does not hide inherited members of the class it will go into
       base.getAllBaseClasses().members.name->excludes(method.name)
```

Figure 4.10 Precondition Specification for *pullUpMethod* Refactoring

We can now explain the details associated with the precondition specification. Condition (1) limits the method being pulled up to normal and non-pure virtual function. Condition (2) verifies the inheritance relationship between the current class and the class receiving the method. This ensures that the move is between the base class and the derived class, and that the following conditions are specified in the correct scenario. This condition only considers public inheritance. Different from protected or private inheritance, public inheritance allows all members to keep their access specifications in the derived class the same as in the base class. If the inheritance is protected or private, more reachability analysis is expected but feasible with our metamodel.

Condition (3) is to avoid syntax errors caused by duplicate members in current class. Condition (4) is part of the effort to make sure all references in the method are resolvable before and after *pullUpMethod* refactoring. This condition requires that the implementation of the method not contain references to any data or function member declared in its containing class. There are other cases that could influence reference resolution. For instance, if the current class has multiple direct base classes, compilation errors will occur if the method to be pulled up contains reference to protected members of other base classes instead of the one receiving the method. Condition (5) avoids syntax errors and semantic changes on the base class caused by name hiding.

```
class A{
};
class B:public A{
private:
    void print(){cout<<"pullUpMethod"<<endl;}
public:
    void printTwice(){
        print();
        print();
    }
};
B b;
b.printTwice();
```

```
class A{
protected:
    void print(){cout<<"pullUpMethod"<<endl;};
};
class B:public A{
public:
    void printTwice(){
        print();
        print();
    };
};
B b;
b.printTwice();
```

Figure 4.11 Sample Code Demonstrating the Accessibility Issue in *pullUpMethod* Refactoring

4.10 pushDownMethod

Goal: check if it is allowed to move a method from the current class to one of its derived classes.

Parameter: --method: the *Function* object to be moved out of the current class

--derived: the *Class* object receiving the method

Precondition Specification: the precondition is defined in the context of *Class* and includes seven conditions. See Figure 4.12.

```
context Class::pushDownMethod(method:Function, derived:Class):Boolean
body: --(1) the method to be removed is a normal , non-pure virtual method
      method.functionKind = FunctionKind::NORMAL  and  method.isPureVirtual = false
     and--(2) derived is one of the derived classes of the current class
         derived.inheritances->exists(i|i.base=self and i.accessibility=AccessSpecifier::PUBLIC)
     and --(3) pushing down method does not cause name collision in the derived class
    derived.members->forAll(m|m.name= method.name implies   (m.oclIsTypeOf(Function) and
                                m.oclAsType(Function).parameters.type.name<>method.parameters.type.name))
   and --(4)method does not directly reference the private members of the current class
     method.references.refersToMember->asSet()->
           excludesAll(self.members->select(m|m.accessSpecifier=AccessSpecifier::PRIVATE))
   and --(5) current class should have no friend
      self.friendClasses->isEmpty() and self.friendFunctions->isEmpty()
   and --(6) method to be moved is not static
    method.storageClassSpecifier<>StorageClassSpecifier::STATIC
   and --(7) no calls on the method via an object of current class
      not Reference.allInstances()->select(r|r.refersToMember = method) ->exists(r|
        (r.owner.refersToMember.oclIsTypeOf(Object) and r.owner.oclAsType(Object).type.refersToClass = self)
         or
        (r.owner.refersToMember.oclIsTypeOf(Parameter)
          and  r.owner.oclAsType(Parameter).type.refersToClass = self)
         or
         r.owner = null)
```

Figure 4.12 Precondition Specification for *pushDownMethod* Refactoring

Discussion:

The first four conditions are similar to Condition (1)-(4) specified for *pullUpMethod* refactoring. Condition (5) prohibits this refactoring if the current class has friends. The current class may grant friend functions or classes in order to allow their access to its protected and private members. However, any access to the method from the friends of current class will fail after refactoring since the method is not available any more. To be conservative, Condition (5) simply requires that the current class have no friends. One may consider a less restrictive constraint: if the method is never referenced by friends via an instance of the current class, the existence of friends should not be an obstacle to implementing this type of refactoring. Here we ignore the OCL expressions for this purpose. Condition (6) states that *pushDownMethod* refactoring cannot be applied to a static method. Otherwise the method will become unreachable via the current class. Condition (7) ensures that the method to be pushed down is never called via an object of the current class, a parameter with type relating to current class, or from within current class. Those calls will fail in the absence of the method. Calls on the method via objects of derived classes do not have this issue. Condition (7) says that for each reference to the method being pushed down, its owner reference should not refer to an object or parameter that is an instance of the current class, or its owner reference is null (the method is called from current class). For example, in the reference "a.b()", moving method "b" out of class A is not allowed if the object "a" is an instance of class A.

One of the issues relevant to Condition (7) is that, due to polymorphism and type casting, it is almost impossible to determine the type of an object in an expression accurately using static analysis. Therefore, the accuracy and effect of Condition (7) are very limited.

4.11 inlineMethod

Goal: check if it is allowed to transfer a method's body into the body of all of its callers.

Parameter: --method: the method to be lined in.

Precondition Specification: the precondition is defined in the context of *Class* and contains two conditions. See Figure 4.13.

```
Context Class::inlineMethod(method:Function):Boolean
body: --get all functions that call method
        let callers:Set(Function)=Function.allInstances()->
                select(f|f.references.refersToMember->includes(method)) in
      --get all items referenced by method that are class members
      let refdMembers:Set(Member) = method.references.refersToMember->asSet()
         ->select(m|m.scope.oclIsTypeOf(Class)) in
--(1) the method to be lined in is not static
        method.storageClassSpecifier<>StorageClassSpecifier::STATIC
--(2) all referenced member items are reachable from each caller
      refdMembers ->forAll(item|
                        item.accessSpecifier=AccessSpecifier::PUBLIC
                        or  ( item.accessSpecifier=AccessSpecifier::PRIVATE and
                              (callers.scope->forAll(s|s=self
                                 or  item.scope.friendClasses->includes(s)
                                 or item.scope.friendFunctions->includes(method))))
                        or  ( item.accessSpecifier=AccessSpecifier::PROTECTED and
                              (callers.scope->forAll(s|s=self
                                or self.getAllDerivedClasses()->includes(s)
                                 or  item.scope.friendClasses->includes(s)
                                 or item.scope.friendFunctions->includes(method)))))
```

Figure 4.13 Precondition Specification for *inlineMethod* Refactoring

Discussion:

At the beginning, we collect all functions that call the method to be inlined, and all class member items that are referenced by the method to be inlined. Condition (1) states that *inlineMethod* cannot be applied to a static method. Now the only problem with this refactoring is that the method to be inlined may reference private or protected members that the calling function cannot reference. Therefore, Condition (2) makes sure that all referenced members can be accessed by each of the callers. Condition (2) deals with three cases for this guarantee: first, a referenced member accessed publicly would still reachable by any function; second, if a referenced member is private, each caller must be either within the current class or be able to reach the referenced member via friendship; third, if a referenced member is protected, all callers must be within the current class or its derived class, or be able to reach the referenced member via friendship.

4.12 moveMethod

Goal: check if it is allowed to move a method from its containing class to another class that is not a base or derived class of the current class.

Parameter: --method: the *Function* object to be moved

--receiver: the *Class* object receiving the method

--newName: the moved method will be given a new name after refactoring

Precondition Specification: the precondition is defined in the context of *Class* and contains four conditions. See Figure 4.14.

```
context class::moveMethod(method: Function, receiver: Class, newName:String):Boolean
body:  --(1) the method to be pulled up is a normal, non-virtual, non-pure-virtual function
       (method.functionKind = FunctionKind::NORMAL  and  method.isPureVirtual = false
         and method.isVirtual=false and  method.storageClassSpecifier<>StorageClassSpecifier::STATIC)
       and –(2) the receiving class does not contain adata member with same name,
            -- nor a function member with signature same as the method to be moved in.
        (newName<>method.name and
            receiver.members->forAll(m|m.name=newName implies (m.oclIsTypeOf(Function)
                    and  m.oclAsType(Function).parameters.type.name<>method.parameters.type.name))
     and--(3) the method to be moved does not hide members that the receiving class inherit from all its base classes
          receiver.getAllBaseClasses.members->forAll(m|m.name=newName implies (m.oclIsTypeOf(Function)
                    and  m.oclAsType(Function).parameters.type.name<>method.parameters.type.name))
     and-- (4)the receiving class must be a declaring class of parameters of the method  or field type.
       (method.parameters.type.refersToClass->includes(receiver)
           or self.getDataMembers().type.refersToClass->includes(receiver))
```

Figure 4.14 Precondition Specification for *moveMethod* Refactoring

Discussion:

Condition (1) states that the method to be moved can be only a normal, non-virtual, non-pure-virtual and non-static function. Condition (2) and (3) is to avoid name collision in current and the receiving class. Condition (4) is not as simple as it looks like. Class members or functions referenced from within the method to be moved may become unreachable because of the class change. To handle this, Condition (4) states that the target class can only be a field type of the current class or a declaring class of parameters of the method to be moved. Combined with some changes on method parameter sequence, Condition (4) makes sure that, with some adjustments on the method parameters, all referenced class members are kept accessible after the

method is moved. The code in Figure 4.15 demonstrates how Condition (4) works. The original code is in the left column and refactored code is in the right column.

```
class A{
public:void printA();
};

void A::printA(){
cout<<"Print class A"<<endl;
}

class B{
public:
void printBoth(A a);//to be moved to class A
void printB();
};

void B::printBoth(A a){a.printA();printB();}
void B::printB(){
    cout<<"Print class B"<<endl;
}

 int main(){
    A a;
       B b;
       b.printBoth(a);
       return 0;
}
Output:
Print class A
Print class B
```

(a)original code

```
class B;
class A{
public:
void printA();
void printBoth_new(B a);//equals original B::printBoth
};

class B{
public:
void printBoth(A a); //delegate to printBoth_new
void printB();
};

void A::printA(){cout<<"Print class A"<<endl;}
void A::printBoth_new(B a){printA();a.printB();}

void B::printBoth(A a){a.printBoth_new(*this);}
void B::printB(){ cout<<"Print class B"<<endl;}

int main(){
    A a;
       B b;
       b.printBoth(a);
       return 0;
}
Output:
Print class A
Print class B
```

(b) refactored code

Figure 4.15 Sample Code Demonstrating the Effects of *moveMethod* Refactoring

Suppose that our refactoring task is to move method *printBoth* from class B to class A, and change its name to *printBoth_new*. Since class A is a declaring class of parameters of class B, this refactoring request passes the precondition checking. *printBoth_new* has an argument with type B to help access the members declared in Class B. *printBoth* becomes a delegate to *printBoth_new* in the new code. With these changes, the original code is refactored into a new version which produces the exactly the same output.

There are other ways to check the reachability of all referenced items with the method to be removed. However, the expressions in Condition (4) are comparatively simple.

4.13 addParameter

Goal: check if it is allowed to add a new parameter to the current method and all of its overriding/overridden methods.

Parameter: --pname: the name of the parameter to be added

--typeName: the type of the parameter

--pos: the index of the parameter in the parameter sequence

Precondition Specification: the precondition is defined in the context of *Function* and contains three conditions. See Figure 4.16.

Discussion:

If the current method has overriding/overridden methods, our refactoring intent is to add a parameter to not just current method, but all of its overriding/overridden methods. At the beginning are variables that are initialized for further reuse. *paramSeq* is the type sequence of parameters of the current method , *newParamSeq* is the new type sequence when adding a new parameter, *pClasses* is the set of polymorphic classes relevant to the current method, and *methods* represents overriding/overridden methods of the current method. Conditions (1) and (2) work on syntax-level examination. Condition (1) makes sure that the name of new argument *pname* does not collide with another variable (parameters or locally declared variables) in the same function scope. By condition (2), the new method will not cause signature collision with other methods within individual class.

```
context Function::addParameter(pname:String, typeName:String, pos:Integer):Boolean
body:   let paramSeq:Sequence(String) = self.parameters.type.name in
        let newParamSeq:Sequence(String) = paramSeq ->insertAt(pos,typeName) in
        let pClasses: Set(Class) = self.getPolymorphicClasses() in
        let methods :Set(Function) = pClasses.getFunctionMembers().oclAsType(Function)->select(m|
                                    m.hasSameSignature(self) and m.isVirtual)->including(self)->asSet() in
       --(1)current method and its overridden/overriding methods do not contain
          --a parameter or a local variable pname
       methods->including(self)->forAll(m|m.parameters.name->excludes(pname)
                                and  m.members.name->excludes(pname))
      and --(2) polymorphic classes and the class containing the current function have not already declare
          -- a method with its signature same as the new method.
      (not pClasses->including(self.scope.oclAsType(Class)).getFunctionMembers()->
                exists(f|f.name = self.name and f.parameters.type.name=newParamSeq))
      and --(3) polymorphic classes and the class containing current function do not declare
          --any members with the same name as the new parameter
      (not pClasses->including(self.scope.oclAsType(Class)).members->exists(m|m.name=pname))
```

Figure 4.16 Precondition Specification for *addParameter* Refactoring

We assume that the new parameter will be referenced in its containing method immediately after this refactoring. If the current method contains references to a data member with name same as the new parameter, this data member will be hidden behind the new parameter. As a result, the references that are originally resolved to that data member will be resolved to the new argument, which definitely changes the semantics of the original code. In fact, name hiding is one of the primary causes of semantic change. On the other hand, a new parameter that has a name clash with any method of the polymorphic classes will invoke a compilation error, which is not what

we want either. Condition (3) prevents the occurrences of these two problems by requiring new argument with a name different from any member names in the polymorphic classes. The sample code in Figure 4.17 explains the name hiding issue we discuss (Line 3).

```
1:class A{
2:    public:
3:          void print(int v){ //the added argument v will hide data member v
4:                cout<<" v is "<< v <<endl;
5:          }
6:    private:
7:    static const int v = 5;          //newly added data member
8:    };
```

Figure 4.17 Sample Code Demonstrating the Issue of Name Hiding Caused by Parameter

One may have a different opinion on condition (3). For example, if a member with same name as the new argument does exist but is never referenced in current method, adding this argument should be allowed. This consideration is reasonable. To do that, we can further check the references contained in each method to be updated.

4.14 removeParameter

Goal: check if it is allowed to remove a parameter from the current method and all of its overriding/overridden methods.

Parameter: --pos: the index of the parameter in the parameter sequence of the current method

Precondition Specification: the precondition is defined in the context of *Function* and contains two conditions. See Figure 4.18.

Discussion:

This precondition specification is based on a refactoring intent similar to the *addParameter* refactoring we introduce in Section 4.11: more than one method could be influenced by

removeParameter refactoring. The same is also true of the initialization of variables. A parameter to be removed must be unreferenced, which is guaranteed by Condition (1). Condition (2) has the same purpose as Condition (2) in the precondition specified for *addParameter* refactoring. It prevents methods with identical signature from being in a single class in order to produce a well-formed program.

```
context Function::removeParameter(pos:Integer):Boolean
body:   let paramSeq:Sequence(String) = self.parameters.type.name in
        let newParamSeq:Sequence(String) = paramSeq ->excluding(paramSeq->at(pos)) in
        let pClasses: Set(Class) = self.getPolymorphicClasses() in --get polymorphic classes
        let methods :Set(Function) = pClasses.getFunctionMembers().oclAsType(Function)->select(m|
                                        m.hasSameSignature(self) and m.isVirtual)->including(self)->asSet() in
       --(1) the parameter to be removed is unreferenced
        methods->including(self)->forAll(m|m.references.refersToParameter->isEmpty())
        and --(2) polymorphic classes and the class containing the current function do not already
            --declare any method with the new signature
       (not pClasses->including(self.scope.oclAsType(Class)).getFunctionMembers()->
                exists(f|f.name = self.name and f.parameters.type.name=newParamSeq))
```

Figure 4.18 Precondition Specification for *removeParameter* Refactoring

4.15 addField

Goal: check if it is allowed to add a new data member into the current class.

Parameter: --field: the name of the new data member

--type: the type of the new data member

--storage: the storage specifier of the new data member

Precondition Specification: the precondition is defined in the context of *Class* and contains two conditions. See Figure 4.19.

```
context Class::addField(field: String, type:String, storage: String): Boolean
body:  --(1) no member named field is declared in the current class
        self.members.name->excludes(field)
        and --(2) the new data member does not cause the redefinition of members inherited by the current class
        self.getAllBaseClasses().members.name->excludes(field)
```

Figure 4.19 Precondition Specification for *addField* Refactoring

Discussion:

Redefining a data member locally causes syntactic errors. This problem can be detected by Condition (1). Condition (2) prevents the inherited member from being redefined when the new data member is added. Redefinition of inherited members is not good design practice, and has a significant possibility of causing semantic changes in the code.

4.16 renameField

Goal: check if it is allowed to rename the current data member to a given name.

Parameter: --newName: the new name of the current data member

Precondition Specification: the precondition is defined in the context of *Object* and contains three conditions. See Figure 4.20.

Discussion:

The implementation of *renameField* refactoring is unnecessary if the given name is the same as current name of the data member. This is where Condition (1) is useful. Condition (2) and (3) are very similar to Condition (1) and (2) in the precondition specification for *addField* refactoring in Section 4.13.

```
context Object::renameField(newName:String)
body:  --(1) newName is not identical to the current name
        newName<>self.name
        and--(2) the new name does not cause the redefinition of members in the current class
        self.scope.oclAsType(Class).members.name->excludes(newName)
        and --(3) the new name does not cause the redefinition of members inherited by the current class
        self.scope.oclAsType(Class).getAllBaseClasses().members.name->excludes(newName)
```

Figure 4.20 Precondition Specification for *renameField* Refactoring

4.17 removeField

Goal: check if it is allowed to remove a data member from the current class

Parameter: --field: the *Object* instance as a data member to be removed from the current class

Precondition Specification: the precondition is defined in the context of *Class* and contains only one condition. See Figure 4.21.

```
context Class::removeField(field:Object): Boolean
body: --field is never referenced
        Reference.allInstances().refersToMember->excludes(field)
```

Figure 4.21 Precondition Specification for *removeField* Refactoring

Discussion:

Our strategy for this refactoring is to make sure the data member to be removed is never referenced. According to our metamodel, if we can find at least one reference that is resolved to the data member *field* (resolution is represented by *refersToMember*), removing the data member will be prohibited. The precondition is specified for this purpose.

4.18 encapsulateField

Goal: check if it is allowed to add the getter and setter methods into the current class for a data member.

Parameter: --field: an Object instance as a data member to be encapsulated.

--getterName: the name of the getter method

--setterName: the name of the setter method

Precondition Specification: the precondition is defined in the context of *Function* and contains three conditions. See Figure 4.16.

```
context Class:: encapsulateField(field:Object, getterName:String, setterName:String):Boolean
body: --(1)No member name clashes with getterName
      self.members->forAll(m|m.name= getterName implies
                           (m.oclIsTypeOf(Function) and  m.oclAsType(Function).parameters<>null))
       and --(2) No member name clashes with setterName
      self.members->forAll(m|m.name= setterName implies
                           (m.oclIsTypeOf(Function) and
                            m.oclAsType(Function).parameters.type.name<>self.name.asSequence()))
       and –(3) adding getterName and setterName does not hide inherited members
      self.getAllBaseClasses().members.name->asSet()->excludesall(Set{getterName,setterName})
```

Figure 4.22 Precondition Specification for *encapsulateField* Refactoring

Discussion:

Condition (1) and (2) check the name collision issue. Condition (3) checks the name hiding issue. For the setter method, its input argument is an object of the current class.

4.19 pullUpField

Goal: check if it is allowed to move a data member from current class to the base class.

Parameter: --field: an *Object* instance as a data member to be moved from the current class

--base: a *Class* object receiving the data member

Precondition Specification: the precondition is defined in the context of *Class* and contains two conditions. See Figure 4.23.

```
context Class:: pullUpField(field:Object, base:Class):Boolean
body: --(1)current class is one of direct derived class of base
      self.inheritances->exists(i|i.base=base and i.accessibility=AccessSpecifier::PUBLIC)
      and --(2)no member with same name is declared in the base class
      base.members.name->excludes(field.name)
```

Figure 4.23 Precondition Specification for *pullUpField* Refactoring

Discussion:

Careful readers may have found the precondition above is similar to that specified for *pullUpMethod* refactoring but simpler. It is. Furthermore, they are specified based on the same assumption: the pulled up member, if declared as private, will become protected in the refactored code. Condition (1) verifies the inheritance relationship between the current class and the class receiving the data member. This ensures that the move happens between the base class and the derived class. This condition is the same as Condition (2) in the precondition specification for *pullUpMethod* refactoring. Condition (2) states that the base class cannot already have a member named *field* for syntactical correctness.

4.20 pushDownField

Goal: check if it is allowed to move a data member from the current class down to one of its derived classes.

Parameter: --field: an *Object* object that is a data member to be moved out of current class

--base: a *Class* object receiving the data member

Precondition Specification: the precondition is defined in the context of *Class* and contains five conditions. See Figure 4.24.

```
context Class::pushDownField(field: Object, derived:Class):Boolean
body: --(1) derived is one of the derived classes of current class
      derived.inheritances->exists(i|i.base=self and i.accessibility=AccessSpecifier::PUBLIC)
    and --(2)pushing down field does not cause name collision in the derived class
      derived.members.name->excludes(field.name)
    and –(3) current class does not have friend classes or friend functions
      self.friendClasses->isEmpty() and self.friendFunctions->isEmpty()
    and --(4) field is not static
      field.storageClassSpecifier<>StorageClassSpecifier::STATIC
    and--(5) the data member is not referenced via an object of the current class
      not Reference.allInstances()->select(r|r.refersToMember = field) ->exists(r|
        (r.owner.refersToMember.oclIsTypeOf(Object)
          and r.owner.oclAsType(Object).type.refersToClass=self) --
        or
        (r.owner.refersToMember.oclIsTypeOf(Parameter)
          and  r.owner.oclAsType(Parameter).type.refersToClass=self)
        or
        r.owner=null)
```

Figure 4.24 Precondition Specification for *pushDownField* Refactoring

Discussion:

This precondition borrows the idea of *pushDownMethod* refactoring, but is comparatively simple. Condition (1) verifies the inheritance relationship between the current class and the class receiving the data member. Condition (2) avoids member name clash in the derived class. Condition (3) addresses the reachability problem caused by the existence of friend classes and friend functions. Condition (4) excludes a class variable from this refactoring. Condition (5) is similar to Condition (7) of the precondition for *pushDownMethod* refactoring, ensuring that there is no resolution issue due to the moving of the field.

We have presented the precondition specifications written with OCL for the 18 primitive refactorings listed in Chapter 3. The only one we do not cover is the *extractMethod* primitive refactoring. We have explained the reason in Section 3.3. Among all 18 primitive refactorings, removing unreferenced entities or adding new entities is comparatively simple, since there are no existing references in the program. Renaming and moving refactorings is harder. Each refactoring may have variant preconditions. More checking means more time consuming, but could be more semantic preservation. Developers can make choices by balancing the advantages and disadvantages of different precondition specifications for a particular refactoring.

CHAPTER 5 DESIGN PATTERN SENSITIVE CODE REFACTORING

The use of design patterns has become widespread in the design, development and maintenance of modern object-oriented software systems. The implementations of design patterns impose additional constraints on code evolution. In this chapter, we discuss design pattern sensitive refactoring in which pattern-related constraints as well as constraints for syntax correctness and semantic preservations are considered. We use the template method and singleton patterns and their specifications to demonstrate the issues that may be encountered in the implementation of design pattern sensitive code refactoring. The possible solutions to these problems give more evidence to the importance of programmer-directed precondition specification.

5.1 Design Pattern Sensitive Code Refactoring

Most often, each software system has its own fundamental design patterns that should be kept in its source code during software evolution in order to retain the design intents of the developers. Breaking these design patterns unintentionally will adversely affect the quality of code in terms of reusability, extensibility and comprehensibility, in violation of the original purpose of refactoring. Therefore, to avoid serious maintenance issues, it is important for code refactoring activities to protect the properties of source code critical to the implementations of design patterns from being violated. We call it design pattern sensitive code refactoring.

For some patterns, their properties are representative of static structures that can be identified easily by developers. Other patterns can only be detected with the aid of design pattern detection

tools, performing expensive dynamic analysis. In either case, if the existence of design patterns is only in developers' memory instead of being transformed into fixed principles in the process of code refactoring, developers may gradually lose this knowledge. As a result, it is necessary to create rules governing the correct treatment of design patterns in existing code during refactoring, unless these patterns are regarded inappropriate and the developers decide to discard them purposely.

To keep design intents in terms of the implementations of design patterns, we need suitable specifications of pattern properties established over the multiple artifacts in the source code that participate in the pattern implementation. These specifications describe the structural and behavioral properties of the code that must remain the same before and after refactoring. Many approaches for pattern specifications have emerged in the literature (France et al., 2004; Sterritt, Clarke, and Cahill 2010). Our approach follows most of the design pattern specification principles where each pattern is a collection of different roles and their interactions. The key points are listed below:

- Each type of design pattern is represented as a metamodel that encloses all roles involved in that particular pattern. A role is a type of language entity that participate the design pattern implementation. We visualize the metamodel as UML class diagram.
- Each design pattern metamodel has a collection of OCL expressions attached to it that reflects the interactions between its different roles. For our examples in the next two sections, we group all constraints imposed on a pattern implementation into a logic expression and put it into an OCL operation body.
- Each concrete design pattern is an instance of the metamodel designed for a particular design pattern and should satisfy the OCL constraints attached to that metamodel.

- An instantiation of a design pattern metamodel is actually a binding process that associates a set of source code entities in a concrete program (e.g. classes, methods and class hierarchy) with the roles in the pattern. The concrete program model is in conformance with the metamodel we present in Chapter 3. Therefore, design pattern specification can be treated as another type of query across the program model.

We want to make clear that our approach to design pattern specification may not be as precise as those pattern specifications used in pattern detection algorithms. Our approach largely relies on static structures such as class inheritance, function calls and method overriding. Most pattern detection algorithms collect both static and dynamic information about the program for analysis. Some patterns, composite and decorator patterns for example, have similar structure but different intents. It is impossible to distinguish them from one another by simply specifying the structural features of their implementations. Some patterns require dynamic analysis and control flow analysis to verify their behavioral features. For instance, one requirement in the implementation of the observer design pattern is that an observer object is updated only if it has been attached to the observer list of the subject object. Verification of this constraint is possible only with dynamic analysis. Therefore, what we expect from the pattern specifications using our approach is that they cover the essential language entities and relations in order to prevent code changes from violating the implementation intents of the patterns.

The template method and singleton patterns are among the most widely used design patterns. In the next two sections, we specify these two patterns using the approach we present above. We also use them as examples to illustrate the implementation of design pattern sensitive refactoring and discuss relevant issues.

5.2 Template Method Design Pattern and Its Specification

In this section, we first have a brief introduce the template method design pattern. We then address its specification which contains a metamodel and a set of constraints describing the properties of that pattern.

5.2.1 Template Method Design Pattern

The template method pattern defines the skeleton of an algorithm in a base class, and defers some of the algorithm's steps to a subclass by method overriding. This pattern ensures that the implementation of the algorithm follows the consistent steps while allowing different behaviors for objects of different classes. The structure and a C++ source code example of the template method pattern are shown in Figure 5.1 and 5.2 (Gamma et al., 1995).

Figure 5.1 shows the template method pattern implemented in a class hierarchy. We identify two primary roles that are critical to the implementation of the template method design pattern:

- *TemplateMethod*: This role represents a method declared in the base class for the implementation of the algorithm. In Figure 5.2, the method *Application::openDocument* takes this role and defines each step of opening a document. We call the method taking *TemplateMethod* role **template method**.
- *PrimitiveMethod*: a collection of methods that are declared in the base class, called by the template method, and overridden by each derived class. In Figure 5.2, the methods *doCreateDocument*, *canOpenDocument* and *aboutToOpenDocument* declared in the *Application* class take this role and we call them **primitive method**.

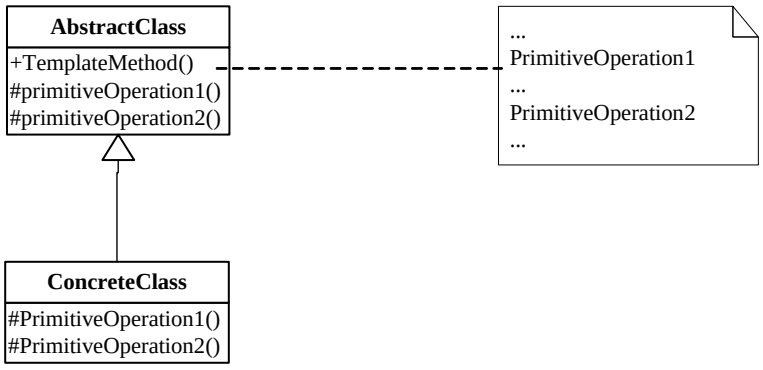


Figure 5.1 Structure of Template Method Design Pattern

```
class Application {
public:
        void addDocument(Document*);
        void openDocument(const char* name);
protected:
        virtual bool canOpenDocument(const
char* name)=0;
         virtual Document*
doCreateDocument()=0;
        virtual void
aboutToOpenDocument(Document*)=0;
        //omit other members
};

class Application1:public Application{
protected:
        bool canOpenDocument(const char*
name);
       virtual Document* doCreateDocument();
        virtual void
aboutToOpenDocument(Document*);
        //omit other members
};
```

```
class Application2:public Application{
protected:
        bool canOpenDocument(const char*
name);
       virtual Document* doCreateDocument();
        virtual void
aboutToOpenDocument(Document*);
        //omit other members
};

void Application::openDocument(const char*
name){
        if(!canOpenDocument(name)) {
                return;
        }
        Document* doc = doCreateDocument();
        if (!doc){
                addDocument(doc);
                aboutToOpenDocument(doc);
                doc->open();
                doc->doRead();
        }
}
```

Figure 5.2 Implementation Example of Template Method Design Pattern

5.2.2 Template Method Design Pattern Specification

As we said in Section 5.1, the design pattern specification in our approach starts with the design of a metamodel that is a container of required roles in the template method design pattern. Figure 5.3 is the UML class diagram representing the metamodel of this pattern. The abstract class *MethodRole* represents roles taken by C++ class methods. It has one inherited attribute *name* from the abstract class *Role* and a parameter sequence *paraSeq*. Attribute *name* is a method's fully qualified name. Here we use the fully qualified name and parameter sequence in place of the compiler-dependent mangled name to ensure a unique name for the function. Attribute *name* and *paraSeq* can be used as entry points to find corresponding methods in a program model. For the template method pattern, the *MethodRole* class has two derived classes, *TemplateMethodRole* and *PrimitiveMethodRole*. At the bottom of Figure 5.3 is another abstract class Pattern that derives the *TemplateMethodDesignPattern* class. Each pattern instance is assigned a unique name at the time of pattern instantiation. Each *TemplateMethodDesignPattern* contains a *TemplateMethodRole* object and one or more *PrimitiveMethodRole* objects.

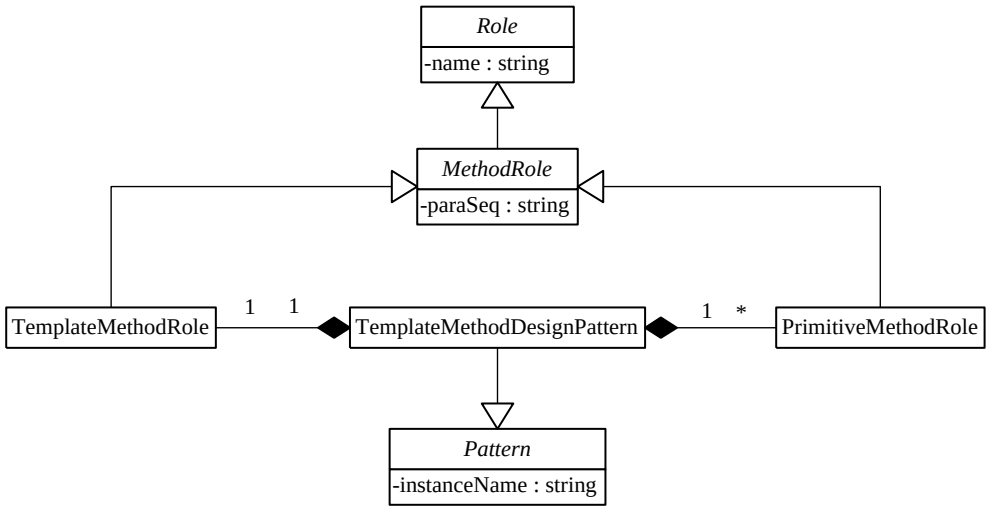


Figure 5.3 Metamodel for Template Method Design Pattern Specification

As the second step of the pattern specification, we define OCL constraints in the context of the *TemplateMethodDesignPattern* class of the metamodel designed for the template method pattern specification. See Figure 5.4.

```
context TemplateMethodDesignPattern::notViolated(): Boolean
body:
let template: Source::Function = self.templateMethodRole.getMethod() in
self.primitiveMethodRoles.name ->iterate(n:String; primitives:Set(Source::Function) = Set{} |
                primitives->including(n.getMethod()))
if (template=null or primitives->size()<>self.primitiveMethodRoles->size())
then false
else (
template.accessSpecifier=AccessSpecifier::PUBLIC                                    --(1)
and template.scope.oclIsTypeOf(Class) and template.isVirtual=false and template.isPureVirtual=false
and primitives.isPureVirtual->excludes(false) and primitives->forAll(p|p.scope=template.scope)
and template.references.refersToMember->asSet()->includesAll(primitives)             --(2)
and (let derivedClasses:Set(Class)=template.scope.oclAsType(Class).getDirectDerivedClasses() in
      derivedClasses.members.name->excludes(template.name)                           --(3)
      and primitives->forAll(p|p.getPolymorphicClasses()=derivedClasses              --(4)
         and p.getOverridingMethods()->forAll(m|m.accessSpecifier=AccessSpecifier::PROTECTED))
    )
```

Figure 5.4 Constraints on Template Method Design Pattern Metamodel

We first obtain the participating methods from the program model. Verification stops if any of them is not available. Otherwise, we check if all of the four constraints are satisfied.

(1) The template method and all primitive methods from the same template pattern instance must be declared in the same class. The template method must be public, non-virtual and non-pure-virtual. The primitive methods must be pure virtual.

(2) The template method must call all of the primitive methods.

(3) The template method cannot be redefined in any of the derived classes.

(4) Primitive methods must be overridden in each derived class of the abstract base class, and must be declared protected instead of public, to ensure they are called only by the template method.

5.3 Singleton Design Pattern and Its Specification

This section follows the layout of Section 5.2. The singleton design pattern is first introduced, followed by its specification in terms of a metamodel and constraints.

5.3.1 Singleton Design Pattern

The singleton pattern is used to make sure that one class can be instantiated only once in the entire program and to provide a global access point to that instance. Usually it is used for centralized management of the internal and external resources of the system. The implementation details of singleton pattern vary greatly, whereas its intent is straightforward. The C++ source code in Figure 5.5 presents a popular implementation style that applies the singleton pattern to the class *Logger*. The bolded code highlights the keys of this implementation. This sample code uses eager instantiation in which a static data member is declared and instantiated without considering whether it is accessed thereafter. A static method *Instance()* is declared to uniquely access the *Logger* object.

```
L1:   class Logger{
L2:    public:
L3:     static Logger& Instance();
L4:     bool openLogFile(std::string logFile);
L5:     void writeToLogFile();
L6:     bool closeLogFile();
L7:    private:
L8:      Logger(){};
L9:      Logger(Logger const&){};
L10:      Logger& operator=(Logger const&){};
L11:     static Logger theLogger;
L12:  };
L13:  static Logger& Logger::Instance(){
L14:      return theLogger;
L15:  }
```

Figure 5.5 Implementation Example of Singleton Design Pattern

5.3.2 Singleton Design Pattern Specification

As in the template method design pattern, a metamodel is first presented for the singleton pattern specification. See Figure 5.6. The concrete class *SingletonDesignPattern* is derived from the abstract class *Pattern*. Each *SingletonDesignPattern* instance contains only one *SingletonRole* object that stores a name string used to bind this role to a C++ class object.

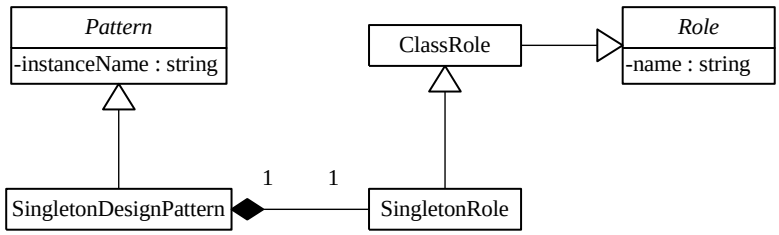


Figure 5.6 Metamodel for Singleton Design Pattern Specification

Figure 5.7 shows OCL expressions that describe three constraints for the singleton pattern implemented by a static data member, shown in Figure 5.5. These expressions are defined in the context of the *SingletonDesignPattern* class in the metamodel designed for the singleton pattern specification.

At the beginning of the OCL operation *notViolated()*, we first get the singleton class from the program model via the fully-qualified name stored in the *SingletonRole* object. If such a language entity does not exist, we do not need to check the rest of constraints. Otherwise, the verification ensures that the essential structures of the singleton pattern are not violated by a code change. Condition (1) states that constructors and the copy assignment operator of the singleton class must be explicitly declared as private. So the creation, destruction and copy of the singleton object can only occur internally. This also prohibits the singleton class from deriving any other classes. To check Condition (2) and (3), we need to find all static objects (OCL variable *objects*) whose type is the single class and all methods (OCL variable methods) in the singleton class that have a return type of the singleton class. By Condition (2), exactly one static singleton class object in the entire program is allowed and it must be a static private data member of the singleton class. By Condition (3), the singleton class must declare one and only one static method that is called by client code to access the unique static singleton class object.

5.4 Issues of Design Pattern Sensitive Refactoring

As we discuss in Section 5.1, design pattern sensitive refactoring considers structural and behavioral features of patterns as types of constraints. These constraints are different from those that strive for syntactical correctness and semantic preservation we discuss in Chapter 4. We find that design pattern related constraints have three primary concerns:

```
context SingletonDesignPattern::notViolated():Boolean
body:   let s:Source::Class = self.singletoRole.getClass() in
        if s = null
        then false   --singleton class never exists.
        else (
        --(1)the constructor and assignment operator must be private
        c.getFunctionMembers()->select(f|f.functionKind=FunctionKind::CONSTRUCTOR
                    or  ( f.functionKind=FunctionKind::OPERATOR  and f.name='=')).accessSpecifier
                                ->asSet() = Set{ AccessSpecifier::PRIVATE}
        and
        (let objects:Sequence(Object)  =  Object.allInstances()->select(o|
                    o.storageClassSpecifier = StorageClassSpecifier::STATIC
                    and o.type.refersToClass = s)  in
         let methods:Sequence(Function)  = s.getFunctionMembers()->select(f|
                        f.returnType.refersToClass = s
                        and f.storageClassSpecifier = StorageClassSpecifier::STATIC) in
         --(2) one and only one static singleton class object can exist in the entire program
             --which is a static data member of the singleton class
            objects->size() = 1 and s.members->includes(objects->first())
            and objects->first().accessSpecifier = AccessSpecifier::PRIVATE
            and objects->first().type.name = s.name
     and  --(3) the singleton class declares one and only one static method
           --that is called by client code to solely access the static singleton class object.
            methods->size()=1
            and methods->first().returnType.name = s.name
            and methods->first().accessSpecifier = AccessSpecifier::PUBLIC
            and methods.references.refersToMember->includes(object->first)
    )
   endif
  )
```

Figure 5.7 Constraints on Singleton Pattern Implementation

(1) Names of language entities. These entities take roles in the pattern implementation. Their names are stored in the *Role* objects of pattern instances and used to retrieve language entities from the program model.

(2) Cardinality of entities. For example, the singleton design pattern requires that the singleton class declare one and only one static method for client code to access the singleton class object.

(3) Implementation dependency among entities such as inheritance, typing and function calls. For example, the template method pattern requires that the template method must call all of the primitive methods, and each derived class must override all of the primitive methods declared in the base class. We think these types of constraints have the most complex specification, and are most likely to be violated due to the number of entities involved.

When programmers try to implement pattern sensitive code refactoring, they need to take care of constraints to preserve the behaviors of the program as well as the implementation intents of design pattern. This gives rise to an issue: pattern related constraints are not always compatible with the constraints we specify in Chapter 4 addressing semantic preservation. Why? Because these two types of constraints have different concerns even though they may share some common points: as we have addressed in Chapter 3 Table 3.1, constraints towards behavior preservation care most about name hiding, name collision, reference, reachability and control flow. It is necessary to further explore this incompatibility issue and its possible solutions.

For the naming constraints brought by pattern related constraints, if a renaming refactoring applies to a language element such as class or method that takes a role in a particular pattern instance, verification of that pattern instance will fail since the corresponding entities do not exist

anymore. For example, Figure 5.5 presents a singleton class *Logger*. Verification of the singleton pattern instance for the class *Logger* will fail if *Logger* is renamed. That is to say, renaming refactoring may violate the name constraints imposed by pattern instances. The good thing is that this conflict can be easily resolved by changing the fully-qualified name of the entity stored in the pattern instance.

For the constraints associated with cardinality and dependency relationships, conflicts may be caused by removing, adding, and moving refactoring. Table 5.1 and 5.2 are lists of refactorings that violate constraints imposed by the template method and singleton patterns while passing the refactoring precondition checking. In both tables, the first column is the name of the refactoring whose precondition violates the pattern constraints, the second column explains the refactoring intent and the language entities this refactoring is invoked on. The numbers in the third column correspond to the numbered constraints specified in Figure 5.4 and 5.7 that are violated by particular refactorings.

Table 5.1 Refactorings Violating the Template Method Pattern

Refactoring	Refactoring Intents	Violated Pattern Constraints
pushDownMethod	Move the template method down to one of the derived classes	(1)
inlineMethod	Inline the template method	(1)
addClass	Add an empty class as the derived class of the base class declaring the template and primitive methods	(4)

Table 5.1 shows that *pushDownMethod* and *inlineMethod* refactorings on the template method violate Condition (1) of the template method pattern. Condition (1) requires that the template method and all primitive methods must be declared in the same base class. However, these two refactorings move the template method out of the abstract base class so that it loses the

role as a template for each of the derived class. Table 5.1 also shows that adding an empty derived class will break Constraint (4) imposed by the implementation of the template method pattern. This is because according to Condition (4), each of the derived classes must override all of primitive methods declared in the base class.

Table 5.2 Refactorings Violating the Singleton Pattern

Refactoring	Refactoring Intent	Violated Pattern Constraints
addMethod	Add a static method with return type of the singleton class	(3)
removeMethod	Remove the static method used to uniquely access the singleton class object	(3)
addField	Add a new static data member with type of the singleton class	(2)

Table 5.2 presents three refactorings which pass the syntax and semantic checking but violate Constraints (2) or (3) imposed by the singleton pattern. These two constraints state that the number of static singleton class objects and its accessing methods existing in the program should be one and only one. As a result, any refactoring that changes these cardinality restrictions will likely violate the implementation intent of the singleton pattern.

What does it mean if the constraints on cardinality and dependency are violated? We believe this means that either implementing the refactoring is not suitable since it breaks the design pattern, or the design pattern was applied inappropriately, in that code smell or other issues are present for such a refactoring to be considered, or both. In any case, programmers need to balance the benefits and disadvantages in deciding how to change the code. They can stop implementing the refactoring to avoid structural regression or inappropriate usages of design patterns. They may choose to allow this particular refactoring and then discard or make some changes on the existing pattern. They may reconsider their constraint specifications and adjust some of them. They may also want to specify preconditions for a new refactoring to handle a

particular situation. We do not have answers for this issue and believe that this is an interesting topic for future research on code refactoring. However, we believe that, when pattern-related constraints are considered, the enabling conditions of refactorings become more complicated and flexible. Facilitating visible, extensible and adaptable precondition specifications for code refactoring becomes more and more important. The model-based approach for precondition specification provides a feasible way to help.

CHAPTER 6 EXPERIMENT

In this chapter we build an experiment to verify the constraints specified as refactoring preconditions in Chapter 4 and Chapter 5. We first introduce the tools required to conduct this experiment. Using these tools, we create metamodels, metamodel instances of sample programs and OCL expressions that will be used in the process of precondition verification. We choose six representative refactorings and present their verification results.

6.1 Experiment Environment Setup

We configure the experiment environment in Eclipse 3.6 where several Eclipse plug-in tools are integrated to facilitate constraint specification and verification. Below is an overview of our tool set.

6.1.1 EMF

EMF is the fundamental part of Eclipse modeling project and provides a basic framework for modeling in the Eclipse IDE. Given the Ecore model of a domain application, the EMF framework and code generation facility can generate XML, UML and Java code to represent and manipulate the domain metamodel (Steinberg et al., 2008). In this experiment we use EMF for:

Modeling: Our metamodel is modeled as an instance of the EMF Ecore metamodel. The Ecore metamodel is the core of EMF, and is designed to map cleanly to Java implementation.

Code generation: The EMF code generation engine generates classes that correspond directly to the packages, classes and enumerated types defined within an application domain metamodel. The generated code has two basic packages: the interface package contains the

set of Java interface representing the client interface to the model, and the implementation package contains corresponding implementation classes. Other useful packages adhering to the metamodel can also be generated such as a utility package and a validation package.

XML serialization and visualization: EMF facilitates a highly customizable resource implementation in support of model object persistence. A model editor plug-in automatically generated by the EMF code generation engine can be used to edit and visualize the content of an XML file that is organized as a tree structure.

One of the prominent features of EMF is its integration with the Eclipse Model Development Tools (MDT) OCL component, with which modelers and developers can specify constraints directly by attaching OCL expressions as annotations to Ecore model elements. EMF provides an extensible code generator to convert OCL expressions such as invariants, operations, derived attributes to Java code. In this way, the imposing of constraint and metamodel design are seamlessly integrated in EMF. Associating this feature with code refactoring, we can bundle OCL invariants with certain classifiers in our metamodel to represent those features that must remain before and after refactoring.

6.1.2 CDT API

While the Eclipse framework is implemented using Java, it has been used to implement development tools for other languages. CDT is a C++ IDE based on the Eclipse platform. It also provides an API for C/C++ code introspection (Schorn 2009). CDT uses three different models to organize the syntactical and semantic information of C/C++ code:

C-Model intends to populate the project navigator and outline view without including local declarations.

C-Index captures each name and binding. It works as connections between declarations and references.

Abstract Syntax Tree (AST) contains every detail about the code such as scope, preprocessor directives, return types and parameters of functions, macro expansions, comments and types of variables.

The CDT API provides client programs with access to multi-view representations of C/C++ code to handle the scalability issue of model creation. Retrieval of C/C++ source code artifacts via the CDT API is expected to perform efficiently by accessing proper models.

6.1.3 OCL Parser/Interpreter

The Eclipse MDT project provides an OCL component as an implementation of the OCL OMG standard for EMF-based models. This component contains a very useful tool as a console for the interactive evaluation of OCL expressions. We customize this tool to measure and display the parsing and evaluation time for each OCL expression.

6.1.4 Program Model Generator

We develop a program model generator to generate a C++ program model against the metamodel we design for refactoring precondition specification. This development reuses the approach in our previous work where the ideas of Component-Based Tool Development (CBTD) and Model-Driven Tool Development (MDTD) are combined for a lightweight RE tool development (Liang 2011). CBTD is also called compositional reuse, and emphasizes the reuse of existing code in the form of components. MDTD moves the level of abstraction from implementation details to model design with the intent to solve issues that code-driven development generally has. Kienle and Muller name CBTD and MDTD as compositional and generative reuse (Kienle, and Muller 2010).

The CDT API and EMF are two primary tools for this development task. We use the CDT API to extract artifacts from C++ source code and avoid the burden of a complicated parsing process. We use EMF to handle the creation and management of models. Figure 6.1 shows the development process of our program model generation tool. It starts with creating a metamodel as .ecore file. On top of .ecore file, a code generation model with the file extension .genmodel is created to describe a code generation pattern that does not belong to Ecore model itself, but affects the code to be generated. Based on the .genmodel file, the EMF generator produces corresponding Java source code for model manipulation. We rewrite some of the generated methods to meet our needs. When the code for manipulating the metamodel is readily available, we extract language elements from the C++ source code via the CDT API, map them into corresponding model elements, and serialize the mapping result into an XMI file as the final output. The mapping process is implemented using the visitor pattern to traverse multiple times on the CDT C-Model and AST model of the program (Gamma et al., 1995). The entire mapping algorithm is constructed by a number of visitors, each of which iterates all translation units (TU), extracts code entities and fills the output model in an incremental way.

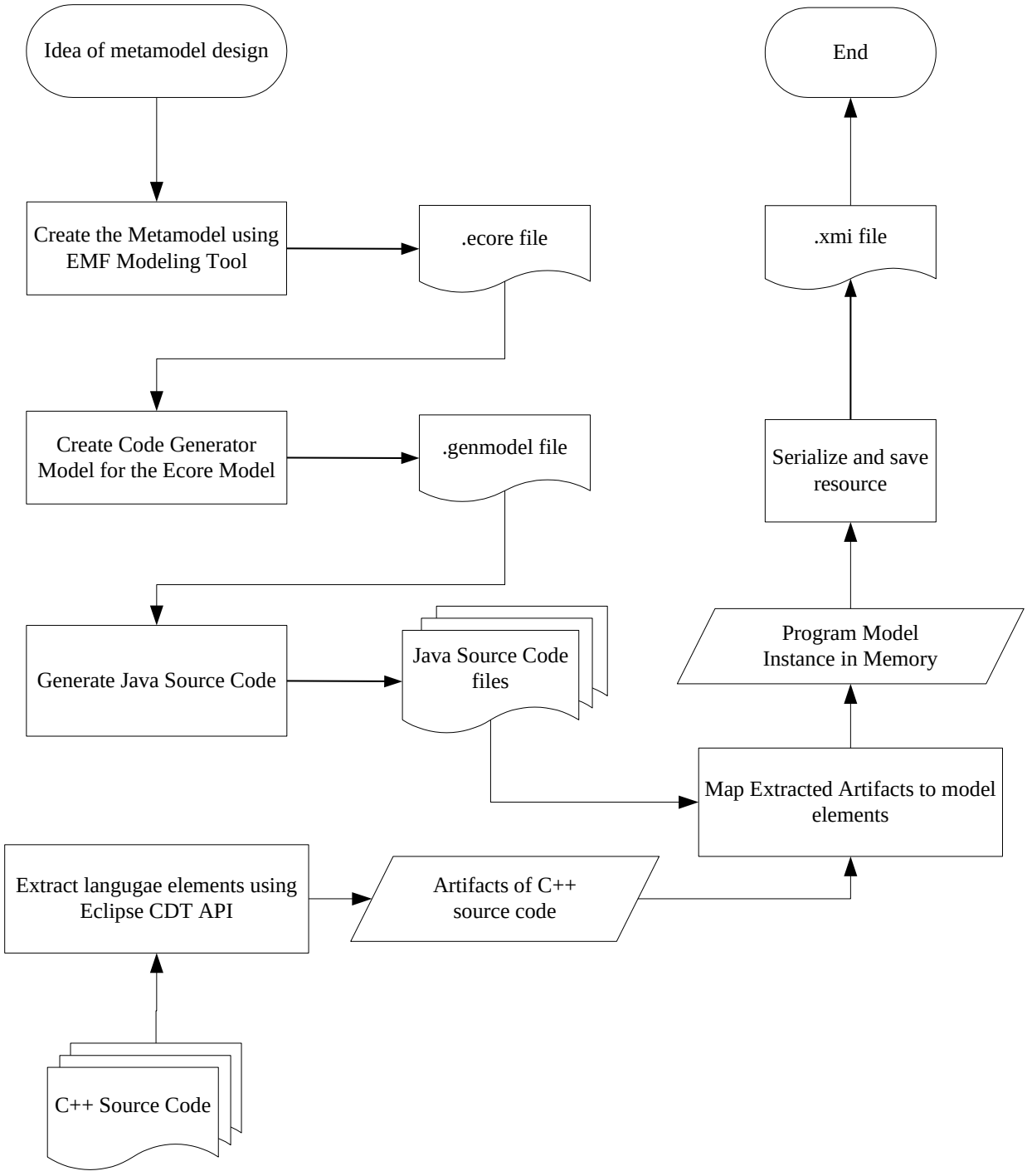


Figure 6.1 Abstractor Development Workflow

We have introduced all of the tools that we adopt to conduct our experiment. Table 6.1 summarizes their basic information, including the version and the usage.

Table 6.1 the Tool Set for Refactoring Precondition Specification and Checking

Tool	Version	Usage
Eclipse	3.6	IDE
EMF	2.6	Modeling, code generation for model manipulation, OCL constraint specification, model serialization
CDT API	7.0	Extract C++ entities to build program models
OCL Parser/Interpreter	3.0	Parse and interpret OCL expressions
Program Model Generator	Development	Generate program model against the metamodel designed for precondition specification

6.2 Preparation for Constraint Verification

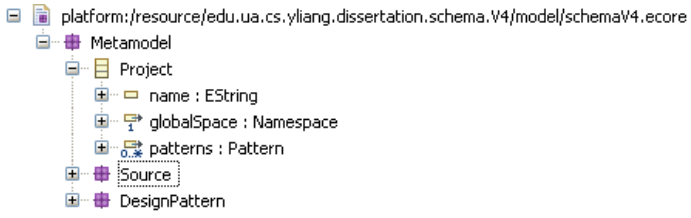
We use EMF to create the metamodels and OCL constraints we design and specify in Chapter 4 and Chapter 5. Utilizing the program model generator we develop, we generate the program model for the sample code that demonstrates the implementations of the template method and singleton patterns in Chapter 5. We edit the metamodel instances of design patterns manually and insert them into the generated program model. Later we present these output results.

Figure 6.2 displays the skeleton of the metamodel and the structures of some important elements we have created. Figure 6.2 (a) is the root package that declares the *Project* class and two packages *Source* and *DesignPattern*. Each *Project* object stores the project name, the *Namespace* object that corresponds to the global space of the C++ project, as well as a set of design pattern instances. The *Source* package corresponds to the metamodel we design and present in Chapter 3, which models the C++ language entities and their relations. Figure 6.2(b) is the list of these modeled entities defined in the *Source* package. Figure 6.2(c) shows the structure of the *DesignPattern* package which includes two pattern metamodels and their constraints for

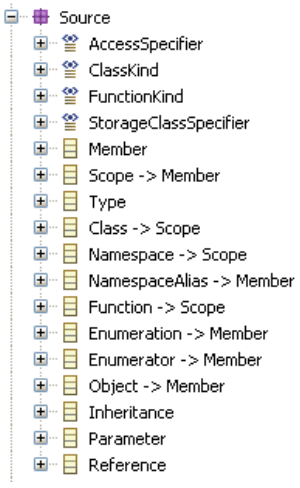
the specifications of the template method and singleton patterns. Figure 6.2 (d) (e) and (f) display the details of three primary classes *Scope, Class* and *Function* in the program metamodel. Each class has its attributes, the OCL analysis functions, and functions for precondition checking defined in the context of that class.

EMF generates Java source code automatically for each element of the metamodel. Figure 6.3 (a) and (b) displays the structures of the interface class and implementation class for the *Object* class. Generally, the interface class contains a getter and a setter method for each class attribute. The method *renameField* in Figure 6.3 (a) is generated for the OCL operation that specifies the precondition of *renameField* refactoring. The corresponding OCL expressions are stored in the implementation class as a static string variable *renameFieldBodyOCL*. Other classes are also generated for particular uses but not displayed here.

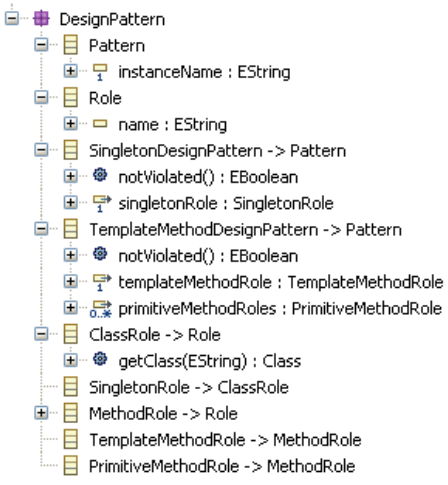
We feed into our program model generator the two sample classes we use in Chapter 5 and produce a model for it. Figure 6.4 is the screenshot of this model displayed as a tree structure rooted at the *Project* object. We can see that the entire model can be divided into two parts. The first part models the source code where the five C++ classes *Logger, Document, Application, Application1* and *Application2* are declared in the global namespace. The second part are the instances of the template method and singleton design patterns. In the next section we will implement the constraint verification of this model.



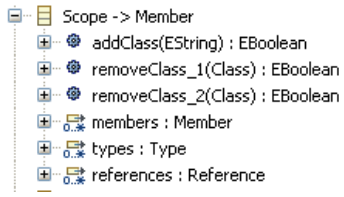
(a) Root Package



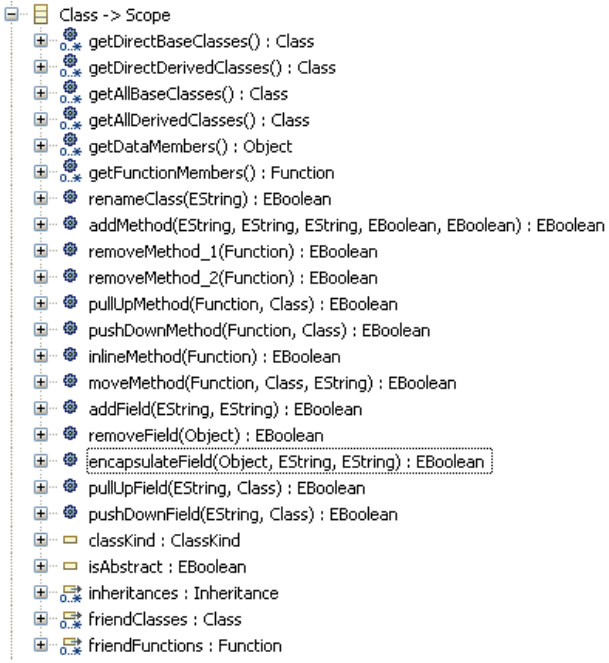
(b) *Source* Package



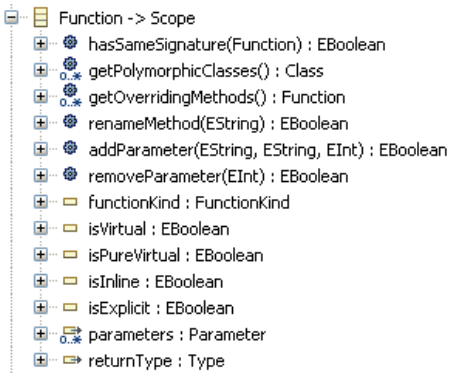
(c) *DesignPattern* Package



(d) *Scope* Class

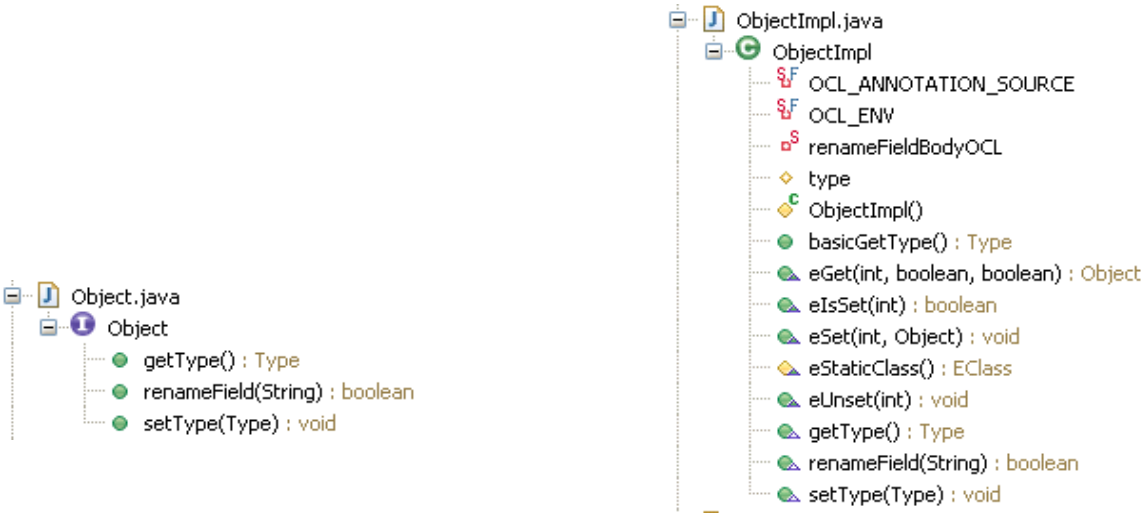


(e) *Class* Class



(f) *Scope* Class

Figure 6.2 Screenshots of the Metamodel Created with EMF



(a) Interface Class Generated for *Object* class

(b) Implementation Class Generated for *Object* class

Figure 6.3 a Portion of Code Generated for *Object* Class

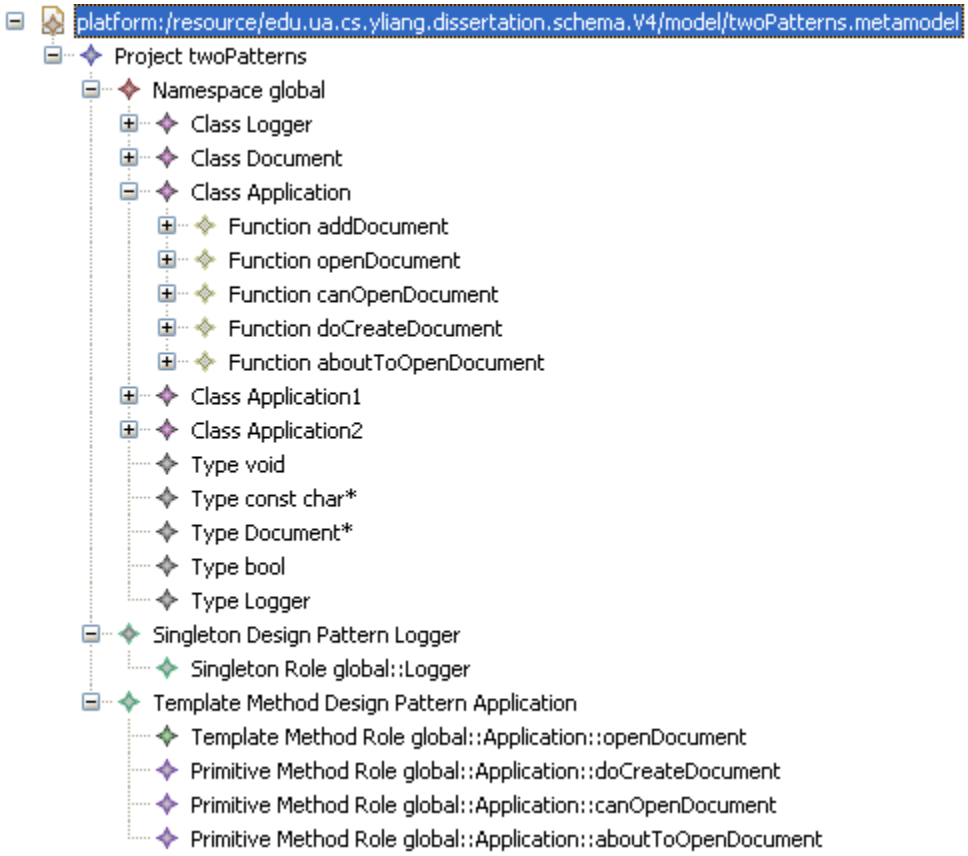


Figure 6.4 the Program Model Used for Constraint Verification

6.3 Verification Results

Suppose that we are implementing a list of refactorings on the program model shown in Figure 6.4. Recall that we define preconditions as OCL operations. Therefore, to check whether the precondition for a particular refactoring is satisfied or not, we type into the OCL interpreter the operation to invoke evaluation. For instance, if we invoke *renameClass* refactoring on the *Application* class, the evaluation result would be similar to what is shown in Figure 6.5. The output indicates that the *renameClass* refactoring on the *Application* class is allowed and the process of precondition checking takes 62 milliseconds for parsing the input OCL expression and 140 milliseconds for evaluation.

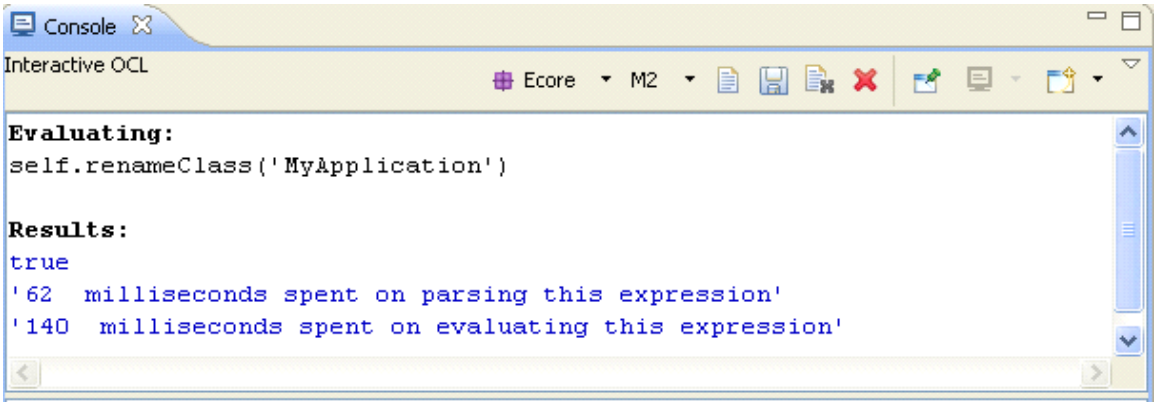


Figure 6.5 Result of Precondition Evaluation on *renameClass* Refactoring

We implement two categories of constraint verifications. The constraints specified in Chapter 4 are verified before the code changes. That is, the verification is to check the status of current programs with efforts on behavior preservation. We present this type of verifications in Section 6.3.1. The constraints specified in Chapter 5 are verified on the code with suggested changes. This type of verification is to check if the suggested changes violate any design pattern implemented in code. We present this type of verifications in Section 6.3.2.

6.3.1 Verification on Constraints for Behavior Preservation

Table 6.1 lists the refactorings that we implement on the program model in Figure 6.4. In this experiment, we do not intend to cover all the types of refactoring we mention in Chapter 4 (there are 18 refactoring preconditions in total). Instead, we select those refactorings that help understand the problems when different types of constraints are considered. For a concrete program, some refactorings will break design patterns but others will not. Table 6.2 is an overview of all refactorings that we have implemented.

Table 6.2 Verification Results for Preconditions Considering Behavior Preservation

No	Entity	OCL Expression	Refactoring Intent	Result
Figure 6.6	*Application*	let method:Function = Function.allInstances()->select (m|m.name='openDocument')->asSequence()->first() in let derived:Class = Class.allInstances()->select (c|c.name='Application1')->asSequence()->first() in self.pushDownMethod(method, derived)	Push the method *openDocument* down to the class *Application1*	true
Figure 6.7	Global namespace	let base:Class = Class.allInstances()->select (c|c.name='Application')->asSequence()->first() in self.addClass('Application3',base)	Add *Application3* as a derived class of the class *Application*	true
Figure 6.8	Global namespace	let base:Class = Class.allInstances()->select (c|c.name='Application1')->asSequence()->first() in self.addClass('Application3',base)	Add *Application3* as a derived class of the class *Application1*	true
Figure 6.9	*Logger*	let method:Function = Function.allInstances()->select (m|m.name='Instance')->asSequence()->first() in self.removeMethod_1(method)	Remove the method *Instance* from the class *Logger*	true
Figure 6.10	*Logger*	self.addField('myLogger','Logger&','static')	Add a static data member *myLogger* into the class *Logger*	true
Figure 6.11	*Logger*	self.addField('curLogFile','std::string','static')	Add a static data member *curLogFile* into the class *Logger*	true

In Table 6.2, the first column is the evaluation screenshot for corresponding refactoring. The second column contains the language entities on which refactorings are invoked. In the third column are the OCL expressions entered into the interactive OCL parser/interpreter to invoke the evaluation. The *let* statement is used to retrieve an object from the metamodel. The fourth column explains the refactoring intent. The last column has the evaluation result. See Chapter 4 for the details of each refactoring precondition.

For all six refactorings, the evaluation results are true, which means all the refactorings are allowed only if we consider constraints that strive for behavior preservation. However, as we will see in Section 6.3.2, among them some refactorings violate pattern-related constraints while others do not.

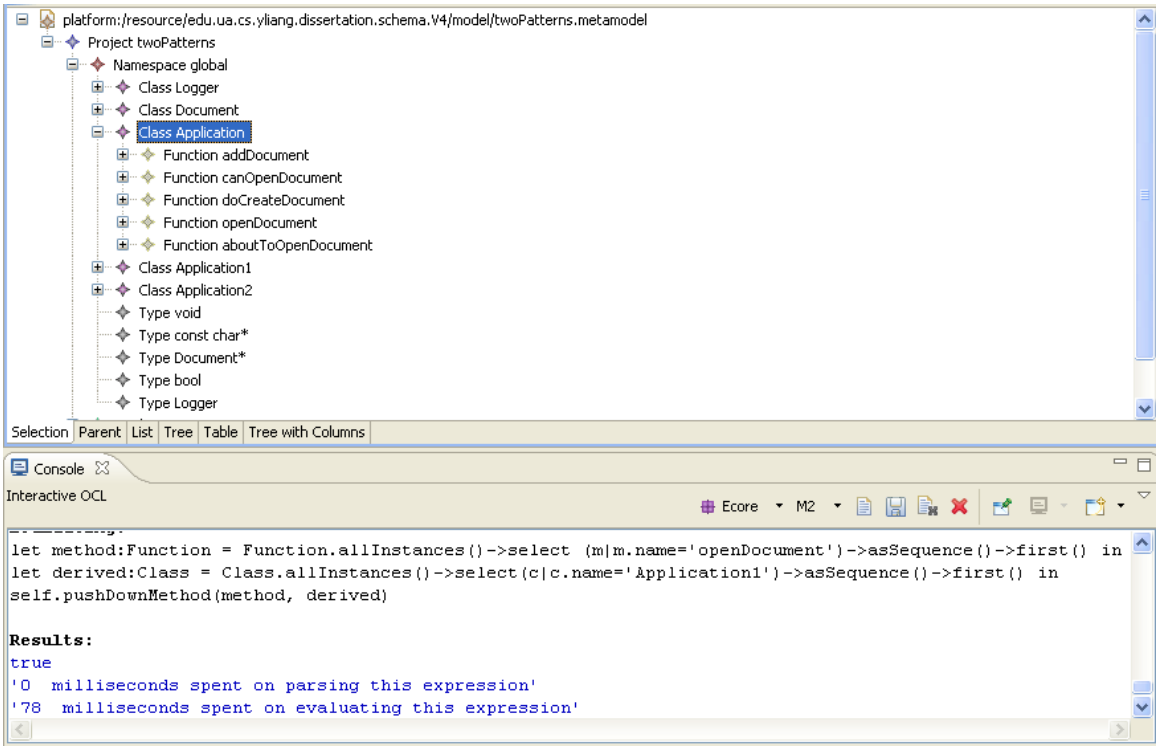


Figure 6.6 Push Method *openDocument* **Down to Class** *Application1*

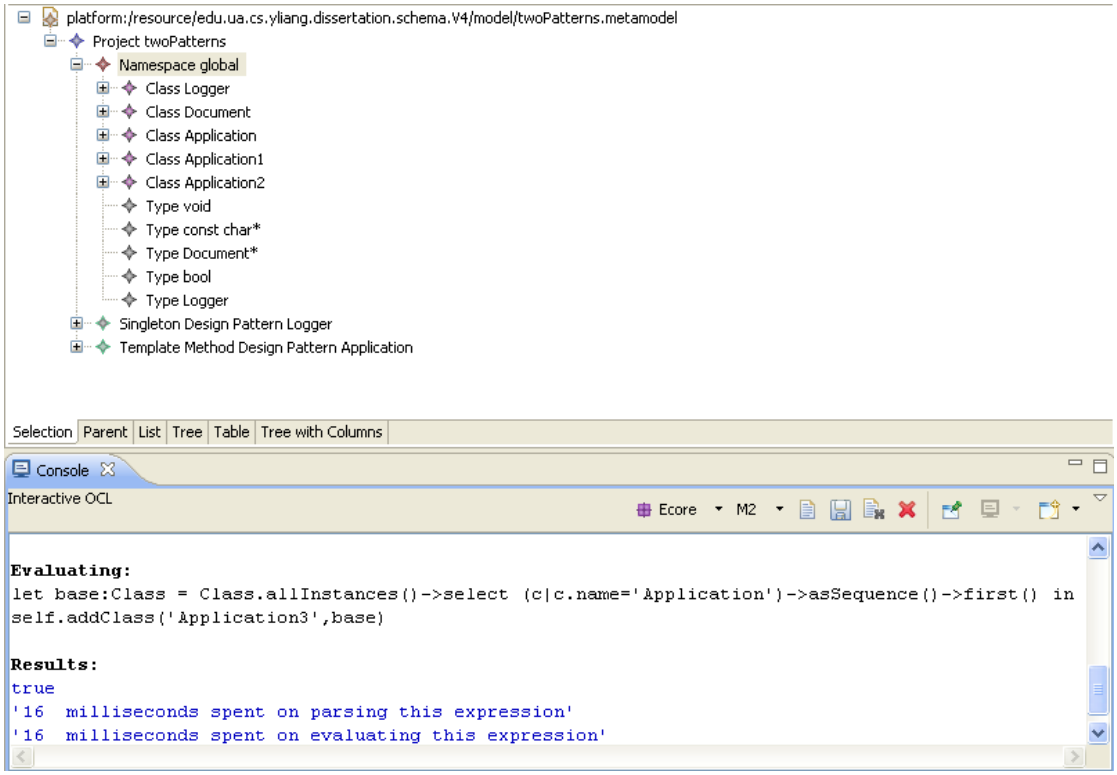


Figure 6.7 Add *Application3* as a Derived Class of Class *Application*

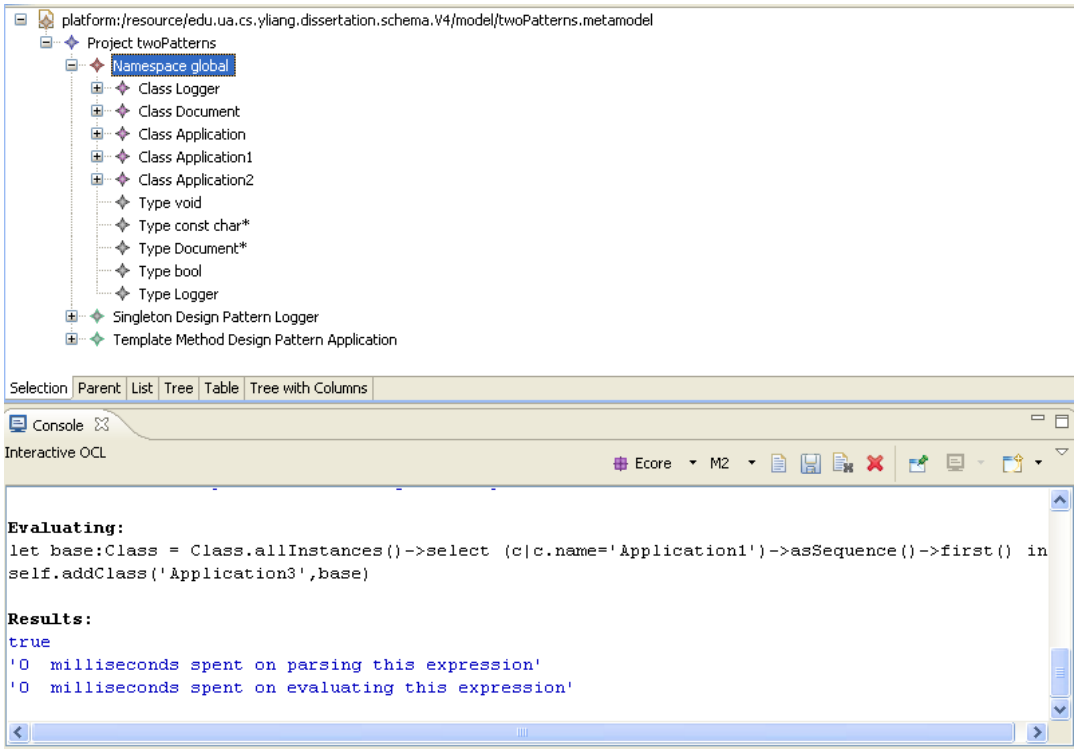


Figure 6.8 Add *Application3* as a Derived Class of Class *Application1*

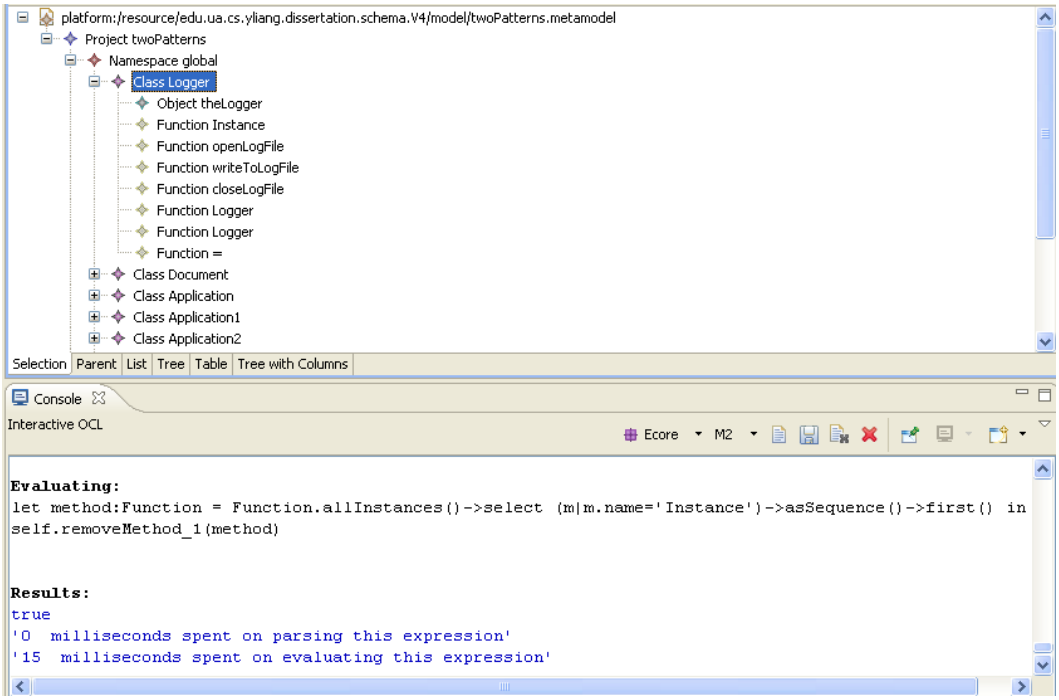


Figure 6.9 Remove the method *Instance* **from the class** *Logger*

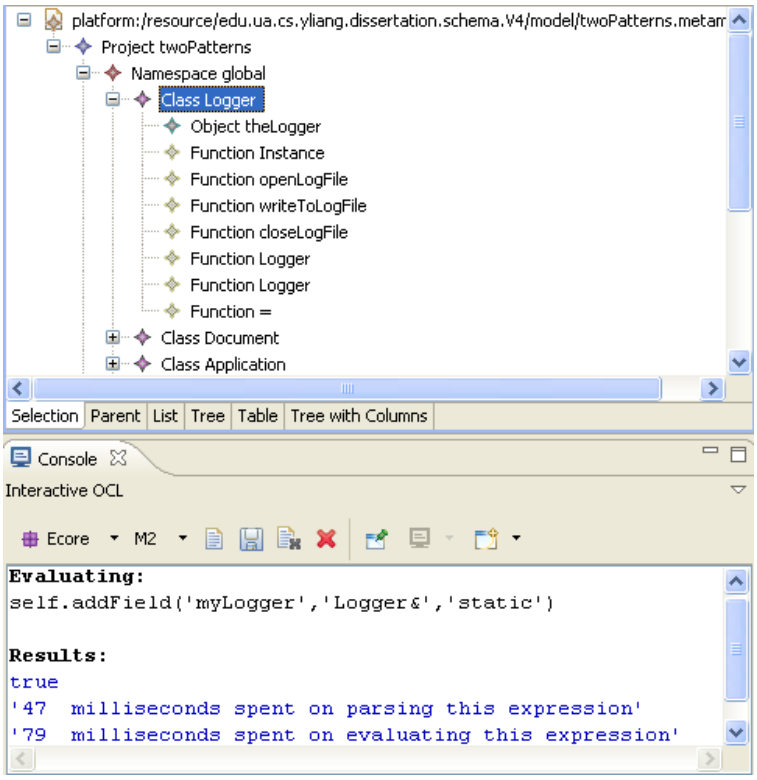


Figure 6.10 Add Static Data Member *myLogger* **into Class** *Logger*

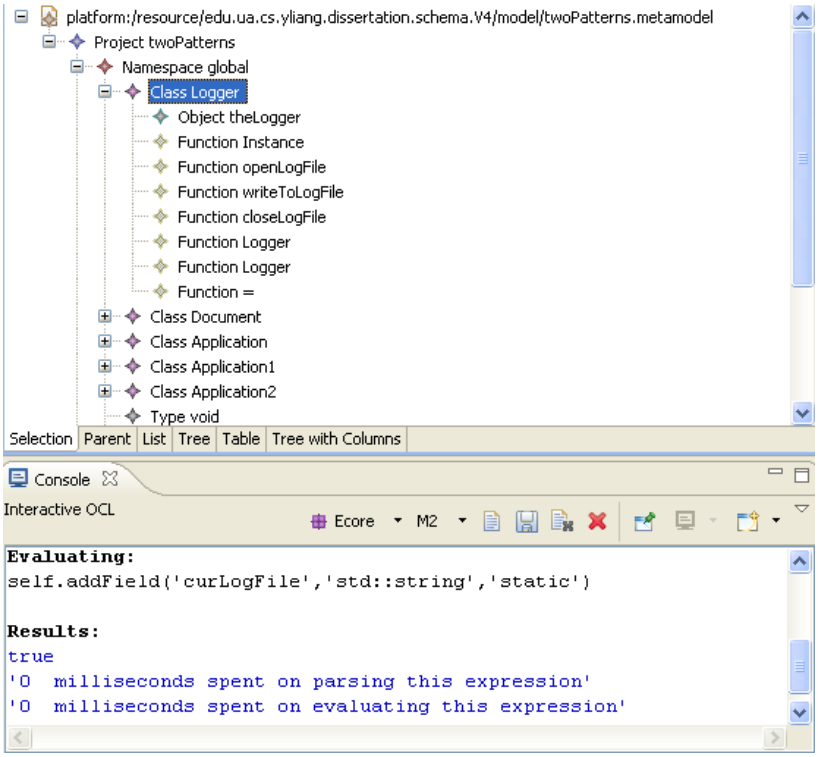


Figure 6.11 Add Static Data Member *curLogFile* **into Class** *Logger*

6.3.2 Verification of Constraints for Design Pattern Preservation

In this section, we demonstrate verification results on constraints that could break pattern Integrity. We simulate each evaluation invoked on the code that is changed by a refactoring operation listed in Table 6.2. Figures 6.12 to 6.17 are screenshots for each evaluation result. In each figure there are three windows. The window on top shows the changed program element and we shade it in blue. The middle window are the properties of the changed element. The window at the bottom is the evaluation result, along with the OCL expression to check the pattern integrity.

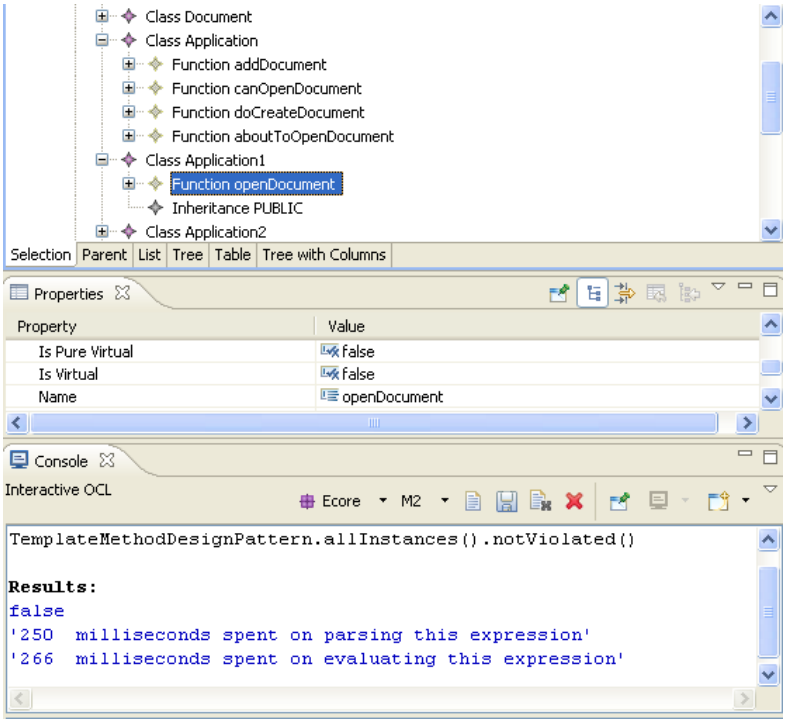


Figure 6.12 Pushing Method *openDocument* Down to Class *Application1* Violates Template Method Pattern

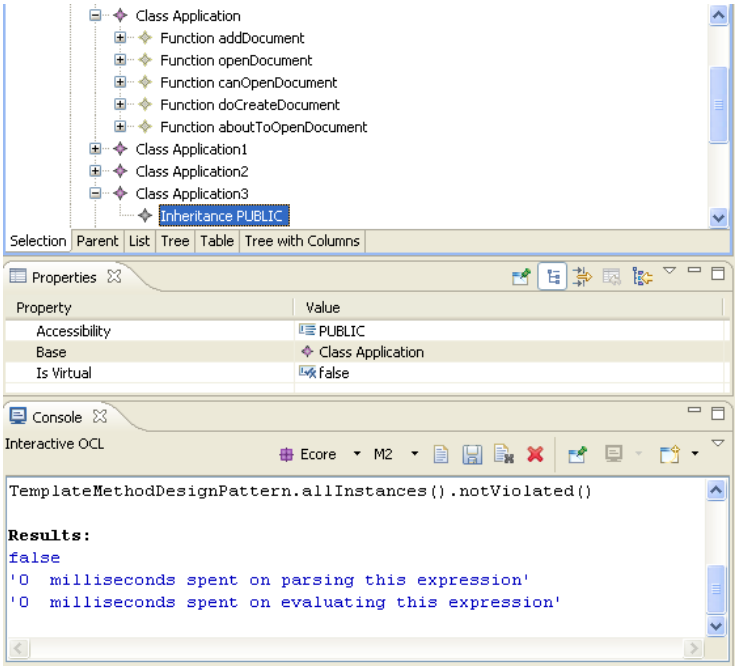


Figure 6.13 Adding *Application3* as a Derived Class of Class *Application* Violates the Template Method Pattern

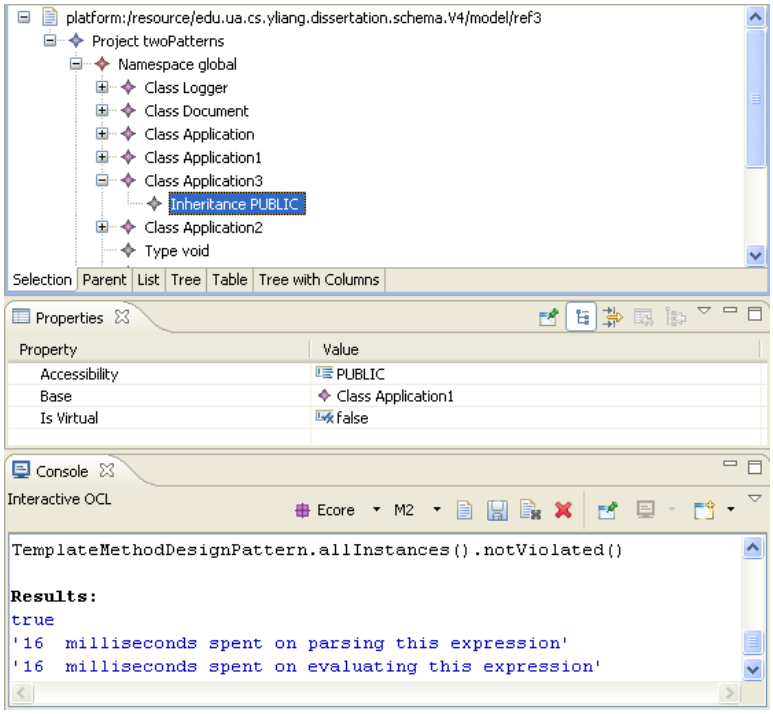


Figure 6.14 Adding *Application3* as A Derived Class of Class *Application1* Does Not Violate Template Method Pattern

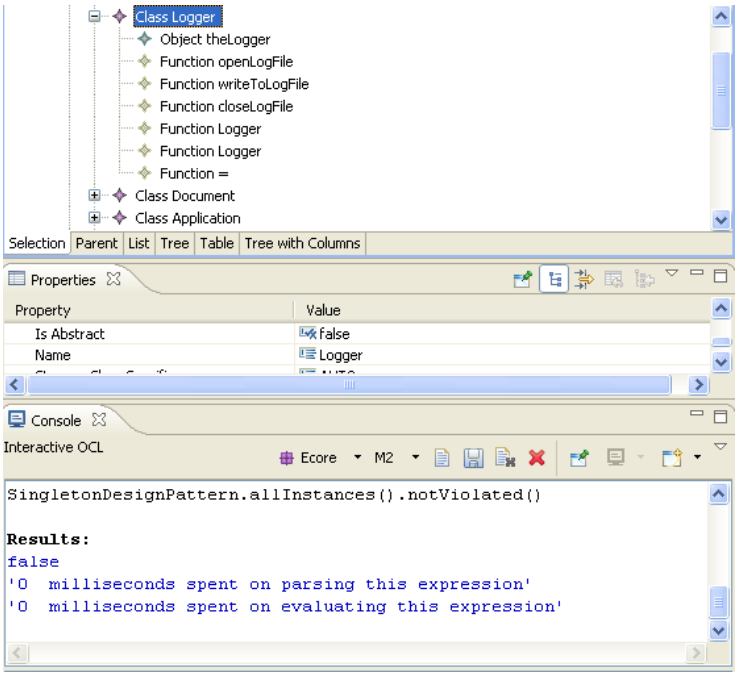


Figure 6.15 Removing Method *Instance* from Class *Logger* Violates Singleton Pattern

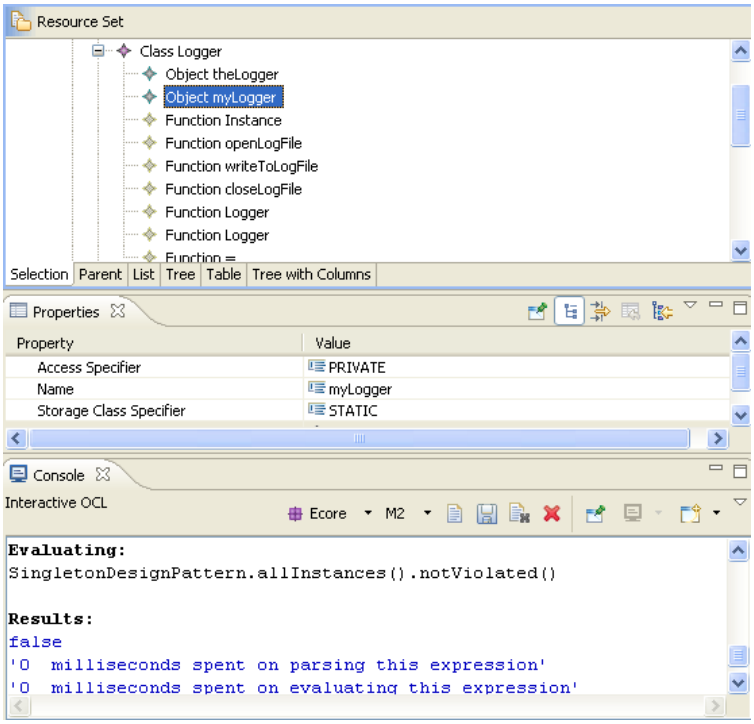


Figure 6.16 Adding Static Data Member *myLogger* into Class *Logger* Violates Singleton Pattern

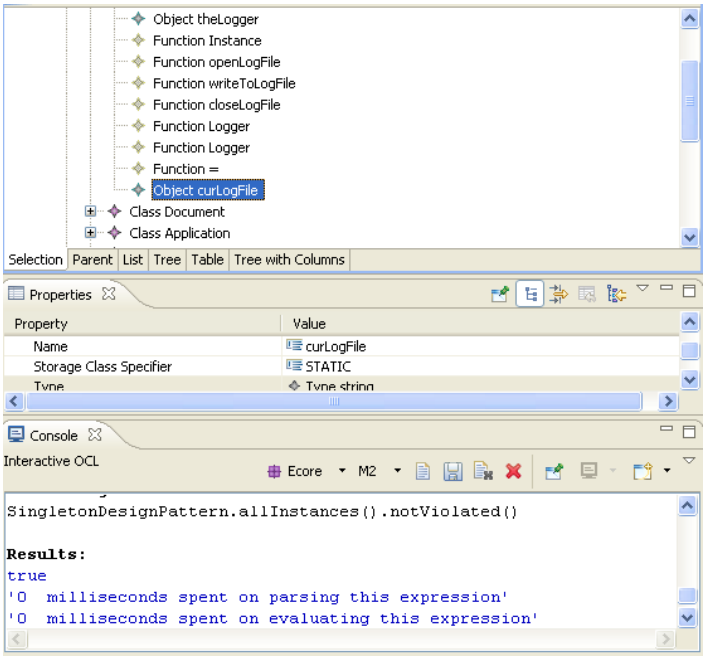


Figure 6.17 Adding Static Data Member *curLogFile* into Class *Logger* Does Not Violate Singleton Pattern

CHAPTER 7 DISCUSSION

In previous chapters, we propose and demonstrate using a model-based approach in support of visible, extensible and adaptable refactoring precondition specification. In this chapter, we discuss some practical and challenging problems that may influence the applicability of this approach. Particularly, we focus on the scalability issue, burden on programmers, and other functional requirements necessary to implement a standard refactoring process using model-based precondition specification and checking.

7.1 Scalability

The model-based precondition specification we propose includes two important steps: extract the program model that conforms to the designed program metamodel, and specify a model-based precondition that uses OCL to query, reason and manipulate OO programs in a straightforward and intuitive way. However, one may argue that in contrast to the use of other programming languages such as Java and C++, the overhead on model generation, OCL parsing and evaluation could slow down the precondition checking, especially when refactoring a large project. That is true. In fact, scalability issue is a key challenge for any model-based software engineering task.

With respect to our task, performance on the entire precondition checking can be managed by using good tools for OCL parsing and validation, and efficient program model generation algorithms. However, successfully scaled performance is not obtainable by simply using efficient

tools and algorithms. In our approach, there are three efforts that contribute to the handling of the scalability issue:

- First is the metamodel design for programs. Programs are modeled to contain language elements that are of interest to most refactorings, rather than every detail about the program (Chapter 3), which reduces the size of the program model and therefore the entire searching space. On the other hand, model elements are organized to make navigation convenient to reduce the number of time-consuming search of entire models (the use of OCL operation *allInstance*()). For example, two frequently used queries are finding the contained members in a scope and finding the scope for a specific member. The program metamodel creates bi-directional associations between element *Scope* and *Member* to make sure that navigation from either end to the other is always simple.
- Second, model-driven code analysis for refactoring gives engineers control over the precondition specification, so that the cost of analysis can be manually reduced to an acceptable level. Programmers can adjust the set of elements to be checked in different situations and at different points in the program. The analysis result may not be as precise as an exhaustive analysis algorithm, but possibly more useful for programmers to achieve their goal, since such an analysis process accommodates their own opinions.
- Lastly, the evaluation time of precondition checking can be improved by writing efficient OCL expressions. The navigational ability of OCL improves its performance on queries that explore the relationships between model elements. Selection of using the navigation path is important in making a query scalable. Other examples of good

practice include: attaching OCL expressions to an appropriate context to avoid unnecessary navigation; using the *let* expression to define a variable that is reusable in the constraint; using the *allInstances* operation as little as possible to reduce lookups of the entire program model; and writing conditions in appropriate order, so that a precondition violation can be detected as early as possible.

We expect these efforts towards scalability can achieve reasonable and acceptable performance for refactoring precondition checking. However, more studies are required to evaluate their effects. This is part of our future work.

7.2 Burden on Programmers

When programmers are able to specify their own rules for refactorings, which is not allowed with traditional refactoring tools, they need to learn how to use this advantage. Associated with our implementation, programmers need to learn OCL skills to write efficient OCL expressions. They also should be familiar with the program metamodel. On the other hand, code refactoring is conducted under certain software constraints that should be satisfied before and after refactoring. In Chapter 4 and 5, we have considered constraints that endeavor to preserve the behaviors of code, and retain the structural and behavioral features of design patterns. More constraints could be considered such as naming rules, metrics and others that may not be built into the programming language itself. Handling different constraints appropriately is the other part of programmers' responsibility to get their code refactored effectively and efficiently. They need to learn when and how to adjust specific precondition at different situations.

Although evaluations are required to answer the question of whether the extra duty is reasonable and acceptable, our approach to precondition specification does own some features that can alleviate the workload on programmers. First is the use of OCL. OCL is designed to

closely couple with standard UML to reduce the learning curve. It is powerful enough to express in a concise and elegant way very complex conditions in an object-oriented paradigm. OCL does have drawbacks. Many studies have been conducted to overcome these drawbacks. For example, a template-based approach and an algorithm for automatic disambiguation of OCL expressions have been proposed to improve the usability of OCL (Wahler 2008) (Cabot 2006). In the meantime, since our metamodel is designed by considering programming language grammar, programmers who gain a thorough understanding of the source program grammar should be able to learn its structure and the relationship between model elements and specify constraints with little problem. For instance, the C++ concept *Scope* determines the lifetime of a name that does not denote an object of static extent. It also determines the visibility of a name. We use UML composition to represent the containment relationship between a name and the scope where the name is declared, which makes the scope lookup straightforward and conceptually close to what programmers have in their mind.

7.3 Toward Effective and Complete Refactorings

We treat refactoring as a type of code change under constraints occurring frequently during the process of software development. These constraints aim to guarantee not only behavior preservation but also other features of the program such as maintaining the implementations of design patterns. Therefore, constraint specification constitutes our primary focus for this dissertation. To improve its effectiveness, other tools need to be integrated into the software development environment and collaborate with the constraint specification tool. Among them are:

- Code smell detection tool to find refactoring opportunities. For example, a clone detection tool can find identical or very similar code scattered across the program. This issue could be solved by implementing *pullUpMethod* or *extractMethod* refactoring.

- Reverse engineering tools to provide programmers with better visualizations of the metamodels against which different types of constraints are specified. The metamodel instances can also be visualized to improve the comprehensibility of current programs.
- Design pattern detection tools to find useful patterns that should be taken care of but of which the programmer may not be aware. As we address in Chapter 5, formalization of the detected patterns can enhance the understanding of their semantics and automatically detected refactorings violating their implementation intents.
- Tools to manage constraints. Functions include browsing, dynamically loading and unloading, adding, deleting and updating the constraints for different purposes.

The constraint specification using OCL is essentially a code investigation and analysis process that has no side effect on the existing program. A standard refactoring activity should also include a code transformation stage in which actual change is applied to the code by using the results of analysis. This dissertation does not cover code transformation. Compared to most refactoring tools where code transformation is bound to hard-coded preconditions, we think the biggest challenge for code transformation based on dynamic constraints is how to represent different kinds of constraint checking results to the code transformation engine. We leave it as part of our future research.

CHAPTER 8 CONCLUSION AND FUTURE WORK

Refactoring is an important activity during software development and evolution. Before the actual changes on code are implemented, precondition checking is first conducted to prevent syntax errors, preserve semantics of the program, retain the implementation essentials of design patterns and protect other features of the program from being broken. This observation drives us to reconsider the format of precondition checking of current refactoring tools, pointing out that preconditions of refactorings should be visible to programmers, conveniently extensible to new refactorings and adaptable to different refactoring concerns. We propose a model-based approach to overcome the deficiency of current refactoring tools on refactoring precondition specification and checking. With this approach, a programmer will be not just a refactoring executor but also a precondition designer.

We choose standard C++ as our objective language for this research. To support the model-based and program-directed precondition specification, we design a metamodel for C++ programs based on the analysis of C++ features and 19 primitive refactorings. Using this metamodel we specify preconditions as OCL expressions for 18 primitive refactorings. These preconditions describe verifications on name collision, name hiding, reachability and references for syntactical correctness and behavior preservation. The implementations of design patterns impose additional constraints on the program that developers should be aware of during refactoring. Therefore, we extend the scope and concept of the precondition to make sure the integrity of design patterns. We present the specifications of the template method and singleton

design patterns to specify this type of constraints. To demonstrate the effects of all defined constraints, we set up an experiment where we can create metamodels and their instances, specify OCL expressions and execute constraint verifications.

We also discuss in this dissertation several issues relevant to this study, which promote the following tasks as our future work:

(1) Empower our approach by developing a user-friendly constraint editing, validation and management tool. This tool and different assisting tools such as code smell detection tools, reverse engineering tools and design pattern detection tool should be integrated into an IDE.

(2) Develop a code transformation engine that utilizes the precondition checking results to change the program.

(3) Investigate possible solutions for the conflicts of different kinds of constraints, for instance, the conflicts between semantic constraints and constraints imposed by design pattern, as we have discussed in Chapter 5.

(4) Conduct empirical studies to evaluate the applicability of the model-based precondition specification for refactoring purposes. Are developers comfortable with specifying their refactoring principles? Or do they prefer fixed preconditions embedded into a refactoring tool to free them from the extra burden

REFERENCES

Programming Language C++, ISO/IEC 14882:2003. (2003).

http://www.eclipse.org. (2011).

http://www.moosetechnology.org. (2011).

http://www.omg.org/spec/OCL. (2011).

http://www.refactoring.com/catalog/. (2011).

Antoniol, G., Di Penta,M. and Merl, E. (May 10-11, 2003). YAAB (Yet Another AST Browser): Using OCL to Navigate ASTs. *in Proc. 11th IEEE International Workshop on Program Comprehension.*

Atkinson, D.C. and Griswold, W.G. (1996). The Design of Whole-Program Analysis Tools. *in Proc. 18th ACM/IEEE International Conference on Software Engineering (ICSE'96)*, (pp. 16-27).

Brichau, J., Kellens, A.,Castro, S. and D'Hondt, T. (2008). Enforcing Structural Regularities in Software Using IntensiVE. *in Proc. 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1).*

Cabot, J. (2006). Ambiguity Issues in OCL Postconditions. *in Proc. OCL for (Meta-) Models in Multiple Application Domain - MODELS'06, Technical Report.*

Cali, A., Gottlob, G. and Lukasiewicz, T. (2009). Datalog: a Unified Approach to Ontologies and Integrity Constraints. *In Proc. 12th International Conference on Database Theory ICDT '09.*

Campbell, D. and Miller, M. (2008). Designing Refactoring Tools for Developers. *in Proc. 2nd ACM Workshop on Refactoring Tools.* Nashville, Tennessee.

Canfora, G. and Penta, M.D. (2007). New Penta Frontiers of Reverse Engineering. *In Proc. 2007 Future of Software Engineering(FOSE'07).*

Chen, Y.F., Gansner, E. R. and Koutsofios, E. (1998). A C++ Data Model to Support Reachability Analysis and Dead Code Detection. *IEEE Transactions on Software Engineering, vol. 24, no.9,* pp. 682-693.

Chikofsky, E.J. and Cross, J.H. (January, 1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, pp. 13-17.

Dig, D. and Johnson, R. (2005). The Role of Refactorings in API Evolution. *In Proc. International Conference on Software Maintenance*, (pp. 389-398). Washington, DC, USA.

Dong, J., Sun, Y. and Zhao, Y. (March, 2008). Design Pattern Detection by Template Matching. *the Proc. 23rd Annual ACM Symposium on Applied Computing (SAC)*, (pp. 765-769). Ceará, Brazil.

Ferenc, R. and Beszédes, A. (2002). Data Exchange with the Columbus Schema for C++. *In Proc. 6th European Conference on Software Maintenance and Reengineering (CSMR)*, (pp. 59-66). Budapest, Hungary.

Ferenc, R., Sim, S.E., Holt, R.C., Koschke, R. and Gyimóthy, T. (2001). Towards A Standard Schema for C/C++. *In Proc. Working Conference on Reverse Engineering (WCRE)*, (pp. 49-58). Stuttgart, Germany.

Fontana, F. A., Maggioni, S. and Raibulet, C. (June 2011). Design Patterns: A Survey on Their Micro-structures. *Journal of Software Maintenance and Evolution: Research and Practice*.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley. ISBN 0-201-48567-2.

France, R., Kim, D., Ghosh, S. and Song, E. (2004). A UML-Based Pattern Specification Technique. *IEEE Transaction Software Engineering, vol. 30, no. 3.*

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Softwar.* Addison-Wesley. ISBN 0-201-63361-2.

Gorp, P.V., Stenten, H., Mens, T. and Demeyer, S. (2003). Towards Automating Source Consistent UML Refactorings. *UML - the Unified Modeling Language: Modeling Languages and Applications*, (pp. 144-158). Springer-Verlag.

Gupta, M., Rao, R. S., Pande, A. and Tripathi, A. K. (2011). Design Pattern Mining Using State Space Representation of Graph Matching. *Communications in Computer and Information Science, vol. 131.*

Heuzeroth, D., Holl, T., Högström, G. and Löwe, W. (May 2003). Automatic Design Pattern Detection. *in Proc. 11th IEEE International Workshop Program Comprehension (IWPC).*

Holt, R. C., Schürr, A., Sim, S. E. and Winter, A. (April 2006). GXL: A Graph-Based Standard Exchange Format for Reengineering. *Science of Computer Programming, vol. 60, no. 2*, pp. 149-170.

Hou, D. and Hoover, J. (2006). Using SCL to Specify and Check Design Intent in Source Code. *IEEE Transactions on Software Engineering, vol. 32*, pp. 404-423.

Kataoka, Y., Ernst, M. D., Griswold, W. G. and Notkin, D. (2001). Automated support for program refactoring using invariants. *in Proc. International Conference of Software Maintenance*, (pp. 736-743).

Kienle, H.M. and Muller, H.A. (2010). the Tools Perspective on Software Reverse Engineering: Requirements, Construction and Evaluation. *Advances in Computers, vol. 79*, pp. 189-290.

Kniesel, G. and Koch, H. (August 2004). Static Composition of Refactorings. *Science of Computer Programming - Special Issue on Program Transformation, vol. 52*, pp. 9-51.

Kraft, N. A., Malloy, B. A. and Power, J. F. (March 2007). An Infrastructure to Support Interoperability among Reverse Engineering Tools. *Information and Software Technology, vol. 49*, pp. 292-307.

Lee, H., Youn, H. and Lee, E. (2008, Journal). A Design Pattern Detection Technique that Aids Reverse Engineering. *International Journal of Security and Its Applications, vol. 2*, pp. 1-12.

Lethbridge, T. C. (2003). The Dagstuhl Middle Model: An Overview. *in Proc. 1st International Workshop on Metamodels and Schemas for Reverse Engineering—ateM.*

Li, H., Reinke, C. and Thompson, S. (August 2003). Tool Support for Refactoring Functional Programs. *in Proc. ACM SIGPLAN Haskell Workshop.* Uppsala, Sweden.

Liang, Y. (2009). Automating Matching Artifacts for Autonomous Tool Evaluation. *in Proc. 47th ACM Southeast Conference.*

Liang, Y. (2011). On the Exploration of Lightweight Reverse Engineering Tool Development for C++ Programs. *in Proc. International Conference on Software Engineering Research and Practice.*

Liang, Y., Kraft, N. A., and Smith, R. K. (2009). Automatic Class Matching to Compare Extracted Class Diagrams: Approach and Case Study. *in Proc. 21st International Conference on Software Engineering and Knowledge Engineering (SEKE).*

Lin, Y. H. (2003). Completeness of A Fact Extractor. *in Proc. 10th Working Conference on Reverse Engineering (WCRE).*

Mens, T. and Tourwe, T. (2004). A Survey of Software Refactoring. *IEEE Transactions on Software Engineering, vol. 30, no. 2*, pp. 126-139.

Mens, T., Van Eetvelde, N., Demeyer, S. and Janssens, D. (2005). Formalizing Refactorings With Graph Transformations. *Journal on Software Maintenance and Evolution: Research and Practice*.

Moise, D.L. and Wong, K. (2003). Issues in Integrating Schemas for Reverse Engineering. *in Proc. International Workshop on Metamodels and Schemas for Reverse Engineering (ateM).*

Muller, H.A., Jahnke, J.H., Smith, D.B., Storey, M. -A., Tilley, S.R. and Wong, K. (2000). Reverse Engineering: A Roadmap. *in Proc. 22nd International Conference on Software Engineering (ICSE)*, (pp. 47-60).

Murphy-Hill, E. and Black, A. (2006). Tools for a successful refactoring. *in Proc. OOPSLA Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications.*

Murphy-Hill, E. and Black, A. (2007). Why Don't People Use Refactoring Tools? *in Proc. 1st Workshop on Refactoring Tools.*

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks.* PhD thesis, Urbana-Champaign, IL, USA.

Prete, K., Rachatasumrit, N., Sudan, N. and Kim, M. (2010). Template-Based Reconstruction of Complex Refactorings. *in Proc. 26th IEEE International Conference on Software Maintenance.*

Roberts, D. (1999). *Practical Analysis for Refactoring.* Ph.D. thesis, University of Illinois at Urbana-Champaign.

Roy, C. C. (2009). Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Journal of Science of Computer Programming*.

Schorn, M. (2009). Using CDT APIs to Programmatically Introspect C/C++ code. *Eclipsecon.*

Seifert, M. and Samlaus, R. (2008). Static Source Code Analysis Using OCL. *in Proc. Workshop OCL Tools: From Implementation to Evaluation and Comparison (OCL'08).*

Sim, S. H. (2002). On Using A Benchmark to Evaluate C++ Extractors. *in Proc. 10th International Workshop on Program Comprehension (IWPC'02).*

Simon,F., Steinbr ücker, F. and Lewerentz, C. (2001). Metrics Based Refactoring. *in Proc. 5th European Conference on Software Maintenance and Reengineering*, (pp. 30-38).

Steinberg, D., Budinsky, F., Paternostro, M. and Merks, E. (2008). *EMF: Eclipse Modeling Framework (2nd Edition).* Addison-Wesley.

Stencel, K. and Wegrzynowicz, P. (2009). Implementation Variants of the Singleton Design Pattern. *in Proc. 24th IEEE/ACM International Conference on Automated Software Engineering.*

Sterritt, A., Clarke, S. and Cahill, V. (2010). Precise Specification of Design Pattern Structure and Behaviour. *in Proc. ECMFA*, (pp. 277-292).

Strein, D., Lincke,R. D., Lundberg, J. and Lo ¨we, W. (Septempber 2007). An Extensible Metamodel for Program Analysis. *IEEE Transactions on Software Engineering, vol. 33.*

Sutton,A. and Maletic,J.I. (2007, January). Recovering UML Class Models from C++: A Detailed Explanation. *Journal of Information and Software Technology, vol. 49, no. 3*, pp. 212-229.

Tahvildari, L. and Kontogiannis, K. (2003). A Metric-Based Approach to Enhance Design Quality Through Meta-Pattern Transformations. *in Proc. 7th European Conference on Software Maintenance and Reengineering*, (pp. 183-192).

Tichelaar, S., Ducasse, S., Demeyer, S. and Nierstrasz, O. (2000). A Metamodel for Language-Independent Refactoring. *in Proc. International Symposium on Principles of Software Evalution.*

Tip,F., Kiezun, A. and Baumer, D. (2003). Refactoring for Generalization Using Type Constraints. *in Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications.*

Tokuda, L. and Batory, D. S. (2001). Evolving Object-Oriented Designs With Refactorings. *in Proc. 14th IEEE International Conference on Automated Software Engineering*, *vol. 8*, pp. 89-120.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. and Halkidis, S. T. (2006). Design Pattern Detection Using Similarity Scoring. *IEEE Transactions on Software Engineering*, *vol. 32*, pp. 896-909.

Verbaere, M., Ettinger, R. and O, Ettinger. (2006). JunGL: A Scripting Language for Refactoring. *in Proc. 28 International Conference on Software Engineering(ICSE).*

Vidacs, L., Gogolla, M. and Ferenc, R. (2005). From C++ Refactoring to Graph Transformations. *Journal of Software Maintenance and Evolution: Research and Practice, vol. 17*, 127-141.

Wahler, M. (2008). *Patterns to Develop Consistent Design Constraints.* PhD Thesis, ETH Zurich, Switzerland.

Xing, Z. and Stroulia, E. (2006). Refactoring Practice: How It Is And How It Should Be Supported – An Eclipse Case Study. *in Proc. 22nd International Conference on Software Maintenance.*

Zhu, H. a. (2008). Refactoring Formal Specifications in Object-Z. *in Proc. international conference on Computer science and software engineering*, *vol. 2*, pp. 342-245.

APPENDIX A

Table A.1 Portion of OCL Feature Description

OCL Symbol	Purpose
allInstances()	Access all instances of a class
exists(v|Boolean-expression-with-v)	Specify a Boolean expression that must hold for at least one object in a collection
forAll(v|Boolean-expression-with-v)	Boolean-expression must hold for any object v in a collection
let v in	Define a variable v that can be reused in the constraint expression
oclAsType(t:OclType)	convert the type of an object into type t
oclIsKindOf(t: OclType)	verify if an object is an instance of type t or one of the supertypes of t
oclIsTypeof (t: OclType)	verify if an object is an instance of t
select	select a sub collection from a collection
self	a contextual instance
Boolean-expression-1 implies Boolean-expression-2	Boolean-expression-1 infers Boolean-expression-2

APPENDIX B: OCL Analysis Operations

B.1 Operations in the Context of *Class*

context Class::getDirectBaseClasses():Set(Class)

```
body:  if self.inheritances->isEmpty()
         then Set{}
         else self.inheritances.base->asSet()
         endif
```

context Class::getDirectDerivedClasses():Set(Class)

```
body: let curClass:Class =self in
        Class.allInstances()->select(c|c.inheritances.base->exists(bc|bc=curClass))->asSet()
```

context Class::getAllBaseClasses():Set(Class)

```
body:        let cSet:Set(Class)=self.getDirectBaseClasses() in
              if cSet->isEmpty()
              then Set{}
              else let curSet:Set(Class) = cSet in
                   cSet->iterate(c:Class;s:Set(Class)=curSet|s->union(c.getAllBaseClasses()) )
              endif
```

context Class::getAllDerivedClasses():Set(Class)

```
body:        let cSet:Set(Class)=self.getDirectDerivedClasses() in
              if cSet->isEmpty()
             then Set{}
             else   let curSet:Set(Class) = cSet in
                    cSet->iterate(c:Class;s:Set(Class)=curSet| s->union(c.getAllDerivedClasses()) )
             endif
```

context Class::getDataMembers():Sequence(Object)

body: self.members->select(m|m.oclIsTypeOf(Object)).oclAsType(Object)->asSequence()

context Class::getFunctionMembers():Sequence(Function)

body: self.members->select(m|m.oclIsTypeOf(Function)).oclAsType(Function)->asSequence()

B2. Operations in the Context of *Function*

context Function::hasSameSignature(f:Function):Boolean

body: self.name=f.name and self.parameters.type.name = f.parameters.type.name

context Function::getOverridingMethods():Set(Function)

```
body: if ((self.isVirtual=false and self.isPureVirtual=false) or self.scope.oclIsTypeOf(Class)=false)
        then Set{}
        else
            let curClass:Class = self.scope.oclAsType(Class) in
            curClass.getAllDrivedClasses().getFunctionMembers->select(m|m.isVirtual
                                                    and m.hasSameSignature(self))
        endif
```

context Function::getPolymorphicClasses(): Set(Class)

```
--get all classes that declare overriding/overridden methods of current method.
body: if ((self.isVirtual=false and self.isPureVirtual=false) or self.scope.oclIsTypeOf(Class)=false)
        then Set{}
        else
          let curClass:Class = self.scope.oclAsType(Class) in
          curClass.getAllBaseClasses()->union(curClass.getAllDerivedClasses())->asSet()->select(c|
            c.getFunctionMembers()->exists(f|(f.isVirtual or f.isPureVirtual) and f.hasSameSignature(self)))
        endif
```

B3. Operations in the Context of *Member*

context Member::getFullQualifiedName(): String

--get full qualified name of current member

```
body:   if self.scope=null
        then self.name
        else self.scope.getFullQualifiedName().concat('::').concat(self.name)
        endif
```

B4. Operations for Design Pattern Specification

context ClassRole::getClass(): Souce::Class

--get a *Class* object in the program model whose fully-qualified name equals to that of the current *ClassRole* object

```
body: let classes:Sequence(Source::Class)=Source::Class.allInstances()->
                                    select(c|c.getFullQualifiedName() = self.name)->asSequence() in
      if classes->size() = 1
      then classes->first()
      else null
      endif
```

context MethodRole::getMethod(): Source::Function

--get a *Function* object whose fully-qualified name equals to that of the current *MethodRole* object

```
body: let methods:Sequence(Function)=Function.allInstances()->select(m|m.getFullQualifiedName()=self.name
                        and m.parameters.type.name=self.paraSeq
                        and m.functionKind=FunctionKind::NORMAL)->asSequence() in
      if methods->size()=1
      then methods->first()
      else null
      endif
```