Recovering the Evolution of Object Oriented Software Systems Using a Flexible Query Engine

Diplomarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Lukas Steiger

Juni 2001

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Dr. Stéphane Ducasse
Michele Lanza
Institut für Informatik und angewandte Mathematik

The author's address:

Lukas Steiger
Software Composition Group
Institut für Informatik und angewandte Mathematik - Universität Bern
Neubrückstrasse 10
CH-3012 Bern

steiger@iam.unibe.ch
http://www.iam.unibe.ch/~steiger/

Abstract

Since software systems must evolve to cope with changing demands, the investment of time and effort won't cease after first delivery. Developers that join a project later in the development cycle may have a hard time to understand the structure of complex systems. Moreover they may not know about concepts that emerged from earlier implementations. We therefore want to find out what exactly happens during evolution of software systems. We developed a method based on simple metric heuristics to detect changes between different versions of a software system. With our query-based approach we can measure overall changes in terms of removals and additions in the code. We are also able to detect different kinds of refactorings like restructuring in the class hierarchy and moved features between entities. Historical information about code size and changes in the code structure helps us to find interesting patterns and to discover unknown relationships and dependencies among source code entities.

Acknowledgments

I'd like to thank all the people who were involved in this work. Special thanks to *Michele Lanza* who substantially helped me to shape my ideas, supported me in technical and organizational issues and reviewed the drafts of this document. Special thanks also to *Stéphane Ducasse* for his initial motivation, profound support concerning Smalltalk issues and brilliant ideas that helped to improve my work. Thanks to *Oscar Nierstrasz* for being the head of the visionary SCG group to guarantee essential and fruitful research; for the careful reading of this document and for the constructive comments that helped to present my results more concisely.

Special thanks to *Claudio Riva* for organizing the work at Nokia that allowed me to test our tools in an industrial environment; for being my tutor regarding technical, administrative or whatever matters. Thanks also to *Juha Kuusela* for offering me the possibility to work at Nokia during summer 2000. Thanks to all the girls and guys with whom I spent my time in Finland. I had a great time there and realized that the retention towards cold, unpleasant northern weather is partly nothing but lies. Moreover other qualities of live in Finland compensate for occasional bad weather!

Many thanks to all members of the SCG, among them especially to: *Sander* who helped me whenever I had problems with MOOSE or FAMIX, also during the time I spent in Finland; *Matthias* for his tips and tricks regarding Smalltalk, and*espacially* for printing this document remotely and handing it in to the deanery on time! *Pietro* for our discussions about the purpose and possibilities of code analysis tools that helped us to generate many new ideas; *Franz* for his valuable tips regarding LaTeX; *Thomas* for tips regarding MikTeX, useful software and the Vaio; *Daniel* for discussions about traps and pitfalls in software industry and about ups and downs while writing a master's; *Georges* for motivating each other to finish our work after all.

Last but not least special thanks to my parents *Andreas* and *Regina* for supporting me throughout my studies at the university! Thanks also to all my friends with whom I spent my time here during my studies.

Lukas Steiger,
June 2001

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

> *"Changes made by people who do not understand the original design concept almost always cause the structure of the program to degrade. Under those circumstances, changes will be inconsistent with the original concept; in fact, they will invalidate the original concept. Sometimes the damage is small, but often it is quite severe. After those changes, one must know both the original design rules, and the newly introduced exceptions to the rules, to understand the product. After many such changes, the original designers no longer understand the product. Those who made the changes, never did. In other words, nobody understands the modified product."* [PARN 94]

Scenarios like the one described above occur more often than we'd like. There is plenty of code running that has been written years ago. Nobody really understands anymore in detail the behavior of such code, thus developers may well run into problems once they have to change it. The rationale behind design decisions exists only in the minds of developers who programmed in earlier phases. These people however probably found another job or got a more advanced task assigned meanwhile. Software industry is well known for fast changes and rapid employee turnover. Programmers that join a project after the product has been launched often have difficulties to maintain the software. The products are usually complex and their documentation is bad and rarely synchronized. The direct analysis of the source code is frequently the only way to gain knowledge about a system. Reverse engineering tools help us to extract certain design artifacts and detect relationships between source code entities.

Several useful reverse engineering tools have been developed to facilitate the analysis of source code. A comparison of different versions of the same software system

provides additional information about the structure of the code. We believe that information about previous releases helps us to discover more possible shortcomings in the current implementation. We also believe that historical information helps us to understand source code patterns more in detail. We hope to understand more clearly design decisions that emerged from assumptions made in earlier stages of development.

We present in this document an approach that combines *evolution analysis* and *metric* data. We compute metrics for the source code of several versions of the same software. We then analyze the change of the metric values between releases. To focus only on changes allows us to narrow the amount of data we need to analyze. Changed parts tell us a great deal about how a software system got in its current state. We make use of simple metrics that summarize certain properties of source code entities in a single numeric value. Numeric values can be easily compared and thus source code entities matching certain criteria are quickly found. We explicitly use only simple metrics which are more directly related with the code. We avoid using complex metrics which describe source code entities in a more abstract way, hence they are more difficult to interpret.

Once we have defined an adequate method to extract different kinds of change, we establish a catalogue of queries mainly based on change metrics. Each query detects a relevant aspect of change in the source code. Some queries just detect simple changes, others detect different refactorings performed on the source code or dependencies between subsystems of a software product. On top of these queries we define a methodology that helps us to combine the found changes to derive general statements about the analyzed code. Our results also let us make hypotheses about the behavior and skills of the developers.

Based on our ideas we implemented a tool named MOOSEFINDER that helps us to validate our ideas on different case studies. We're able to compare different releases of the source code and to detect added, removed and renamed entities. We investigate our proposed methodology based on a number of composite queries. We present the results of the code analysis for two case studies, a large system developed at Nokia Networks, and our reverse engineering platform MOOSE [DUCA 00]. We detected that the analyzed case studies written in Smalltalk change more during evolution than analyzed systems written in C++ or Java. This finding proposes that refactoring is better supported for Smalltalk and therefore more applied than in other languages. Our methodology worked generally well for all analyzed case studies. For large systems we need to refine some of the queries to narrow the resulting entities.

1.2 Our Goals

We intend to provide a method that helps a developer to explore changes during the evolution of a software system. With historical change information we want to gain an

overview of the system's evolution and to understand more of the current code structure. Changes help us to assess the state and quality of a software system. Additionally we want to provide a way for a retrospective documentation of changes. On the way to achieve such more advanced goals, we set ourselves three concrete goals during the course of this work:

- We intend to detect different kinds of changes in the source code between subsequent versions. Examples of changes are additions, removals, renaming or refactorings.
- We want to be able to qualify the parts of a system in terms of stability over several versions.
- We plan to put up a repository of evolution queries. Each query is supposed to extract a specific kind of change in the source code.

1.3 The Structure of this Document

This document is divided in the following chapters:

- In Chapter 2 we introduce the reader to problems in software development. These problems denote the initial motivation for our analysis of evolving software.
- In Chapter 3 we describe our motivation for an evolution analysis based on source code. We also provide an overview on the state of the art in software evolution.
- In Chapter 4 we present the concepts of our approach based on queries and change metrics for the analysis of evolving software systems.
- Chapter 5 contains a collection of queries that allow us to detect several changes in the source code. We discuss their use and evaluate the results for each query separately.
- Chapter 6 explains an initial methodology towards the analysis of a software system using and combining the queries presented in Chapter 5.
- In Chapter 7 we show some results we obtained during the validation of our tools in industry. We describe lessons learned in a pure reverse engineering experience.
- Chapter 8 describes our conclusions. We summarize and evaluate our approach and the obtained results. We discuss achievements as well as drawbacks. We discuss our planned future work in software evolution and how the presented approach can help us to achieve further goals.

- In Appendix A we introduce the reader to the basic concepts of MOOSE and FAMIX. FAMIX is our standard for source code information exchange. MOOSE is our reverse engineering platform where MOOSEFINDER builds on.
- In Appendix B we present an overview of MOOSEFINDER, the tool developed to validate our approach. We quickly describe the different parts of our current user interface. We also explain the basic concepts regarding the implementation.

Chapter 2

The Implications of Aging Software

> *"An E-type program[1] that is used must be continually adapted else it becomes progressively less satisfactory."* [LEHM 96]

2.1 The Software Crisis

Software has become the key element in electronic data processing. Since decades it has taken over repetitive parts of information processing. The progress in hardware technology gave computers the potential to take over more and more complex tasks. Nowadays software penetrates nearly any other industry and business process. Moreover software has evolved to an important industry itself. Despite the progress in technology, software engineers face a specific problem up to now: the complexity of evolving software systems. The ad hoc and chaotic programming culture established decades ago is still popular today. Rapid prototyping using trial and error techniques is usually the fastest and only way to check whether an implementation works. Continuous change of requirements and code, poor documentation and drifting away from initial proper design have led to substantial problems in later stages of development. A large number of software projects fail. Even most projects which do not fail have major problems. Projects are usually late in schedule and over cost. In general software costs increase constantly over time while hardware costs continually decrease, especially in relation with performance. Initial development does not consume most of the costs, yet maintenance costs increase rapidly due to continuous changes and low quality of the code. The need for a systematic engineering approach to the development of software is evident. History shows that finding such an approach is difficult. Project managers rarely risk testing

[1] E-type program: software systems that solve a problem or implement a computer application in the real world [LEHM 96]

new methods, they like more to rely on established procedures. Experienced developers have developed their own techniques to deal with complexity through practice over long time, but their techniques are not handy enough to be quickly learned by rookies.

The persisting character of the problems in software development led Pressman to use the term *chronic affliction* rather than *software crisis* [PRES 94]. Several inherent facts to software development complicate solving the above described problems: the *complexity* of systems handling complex tasks, the need for *continuous adaptation* of software systems, the problems in *project management* for software development, and the difficulty to find out what a *customer* really *expects*.

Complexity: The structure of a software system, the environment it runs in, the components it works together with, this whole ensemble quickly exceeds the capabilities of one single developer to survey. This is not tragical as long as there are other responsible people taking care that nothing disastrous happens. A human being would not accomplish absurd orders, however a computer will. Intuition would tell a person that there must be a misunderstanding, yet a computer always executes exactly the commands it gets fed. It will never automatically correct logically infeasible commands.

Continuous Adaptation: A software system in use will never stop evolving. The adaptation to new requirements and the elimination of conflicts arising through change propagation force a continuous adaptation of the code. Architects building houses create first a detailed design before the house is constructed. This is not possible in the same way for software systems. The knowledge about constructing material and statics is much more advanced than the knowledge about a new domain in software development. Moreover the environment, in which software is developed, changes more dynamically than the ground a house is built on. It is more difficult to foresee how requirements will have changed once the software is built. Changes that are impossible to predict cannot be taken into account in the initial design.

Project Management: Managers that get behind schedule and therefore add programmers will be astonished that adding people does not necessarily speed up development [BROO 75]. The bigger a team, the more coordination between the members is required. Increased communication between the members consumes additional time and may well compensate the added productivity through added programmers. State-of-the-art hardware and software development tools do not necessarily promise good software. Developers first need to know how to handle new tools in a reasonable way and then really utilize them. The level of programming skills varies a lot between different developers. Each programmer needs to be assigned an appropriate task in order to deploy his potential. An additional problem in project management is the current rapid turnover of programmers in

projects. A rapid employee turnover leads to permanent loss of knowledge about the system and the domain.

Requirements: Not only possible future requirements, but also actual needs and expectations of a customer are hard to seize. The customer won't formulate all his requirements precisely. He won't point out those facts that seem obvious to him. He thinks they are evident for everybody. Yet a developer who isn't familiar with the domain may not know about such implicit requirements. The indications of the customer combined with compulsory domain inherent conditions need to be translated into technical specifications. Whoever implemented software for customers knows about the perfidies of such a transformation.

2.2 The Reverse Engineering Approach

> *"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. Reverse engineering generally involves extracting design artifacts and building or synthesizing abstractions that are less implementation-dependent."* [CHIK 90]

2.2.1 Motivation for Reverse Engineering

Why should we analyze the code of existing software? Is there any immediate need to analyze *interrelationships* in running code which has been built according to a proper design? Indeed there are plenty of reasons to analyze such code.

> *"E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment."* [LEHM 96]

For many reasons the disorder of well structured code *will* increase over time. Invariants of the development process like lack of time and continuous changes are the main power for a transformation of quality code into a *Big Ball of Mud* [FOOT 97]. Quick hacks that fix a problem and still keep the system running are usually the fastest solution in a near term view. They are often preferred to an investment of time in a flexible architecture and implementation. Quick hacks however let the code drift away from proper design and may cause more severe problems in later stages of development. The more quick hacks are performed on a system, the more the structure of that system decays. Counteracting such a decay of an implementation requires an investment of time and energy. Time and

free human resources are usually not available since approaching deadlines absorb both.

Programmers do not document their code automatically. The logic of the code seems self-evident right after a programmer implemented it, but it may not look plausible to him some months later. Furthermore new programmers in a team would certainly appreciate to read documented code. Constant lack of time leads developers to neglect documentation and therefore also leads to poorly documented systems. Rapid employee turnover is widespread in software business, though many fresh developers join a project. They all first need time to explore the structure of the software system they're going to work on. The ideas of certain implementations are not documented, thus the meaning of such code gets lost. A new developer may misinterpret the behavior of these parts and change it in a fateful way.

With increased size and complexity of the system, it may well get in a state where it is very hard to maintain. Once such a situation is reached the project has already gone through a substantial part of the development cycle. A lot of human power and money has already been invested in the project and would mainly get lost. Rewriting the software at this point would be very costly. Therefore a tool that helps programmers to browse and understand existing code can be very valuable in such a situation. Unfortunately the above described scenarios happen more often than we would like to. They provide the motivation to develop reverse engineering tools. These tools are supposed to assist developers finding out how different components of their code interact with each. They may finally help programmers to continue their project, to extend and improve existing code.

> *"Sadly, architecture has been undervalued for so long that many engineers regard life with a* BIG BALL OF MUD *as normal. Indeed some engineers are particularly skilled at learning to navigate these quagmires, and guide others through them."* [FOOT 97]

Code analysis tools developed for reverse engineering also help in almost the same way to look into recently written code. Through the power of the accuracy inherent to electronic data processing, we may find unexpected facts even in healthy code. If we use code analysis tools periodically, we will avoid future navigation through quagmires. A combination of such tools with visualization algorithms allows a developer to generate different views on his own code [DEME 99a]. This helps him to see known structures from different perspectives. Reverse engineering tools show us whether written entities behave in the way we'd like them to. They assist us to enhance the implementation of certain parts, they help us to detect similar parts and possible reuse. We may find deprecated code that only increases the complexity of the system without contributing to the functionality.

Chapter 3

The Analysis of Evolving Software Systems

3.1 Introduction

In this chapter we present some background information about *software evolution*. First we discuss how the term *Software Evolution* is used and how we define it for our work. Second we give an overview of related work that has been carried out on the same topic. At the end of the chapter we formulate goals of an evolution analysis and for what scenarios it can help us during development.

3.2 About the Term Software Evolution

Software Evolution is a general term. Several different interpretations are in use. The most general interpretation takes software itself as the subject. It describes the history of software in general, how it changed over time. In the early fifties already there were machine code programs running on batch oriented systems. Nowadays sophisticated layered component architectures run in distributed client-server environments.

Software Evolution is frequently used as another expression for *Software Maintenance*. A common interpretation of software maintenance used to span the phase after first delivery of a product. The split into a development phase and a maintenance phase is problematic. It derives from the waterfall model (described in [SOMM 92]) where the software lifecycle was divided into several phases: requirements collection, design, implementation, testing, operation and maintenance. In contrary to development processes in other disciplines of engineering, in software development it is unrealistic to

pass through these phases sequentially one after the other. The need to validate requirements and design forces a development team to pass through these phases over and over incrementally.

It has become evident that the development phase and the maintenance phase cannot be clearly separated. Development always incorporates also maintenance since a software system will never be mature after first delivery. A running system on the other hand, will always have to be developed further to cope with changing requirements. Today the software developers use the term *evolutionary development* to incorporate both the development and maintenance process in one expression. We should try to roll out an initial version of a software as early as possible. Only then it can be validated in reality by the customer and only then the developers get useful feedback from the *real world*.

In this document we span the term *Software Evolution* over the whole lifecycle of a software system. We want to find out how a system evolves from early prototypes to a mature system that needs to be maintained. The base for our evolution analysis is frozen source code of several stages in the development process. We want to find out how one single system evolves during its development period and try to gain a more detailed understanding of how and where change processes take place [BURD 00].

The most intuitive and probably only way to understand how real software systems evolve is to study changes in existing systems themselves. Software that has undergone several development phases including restructuring after first delivery is most valuable for our analysis. The approach allows us to directly retrieve changes in the code and evaluate their impact on the system. We can assess whether applied reengineering patterns really brought the estimated improvements for further development.

3.3 State of the Art in Software Evolution

In this section we present a selection of related work realized in the subject of software evolution. First we present two general issues about evolution observations of software systems like the pioneering work carried out by Lehman and Belady. Second we summarize a couple of practical approaches realized to compare different releases of the same software system.

3.3.1 Lehman's Laws of Software Evolution

Lehman and Belady are pioneers in software evolution. Back in the seventies they have been investigating the evolution of the IBM OS/360 operating system. They mainly analyzed the growth rate of different modules over time. Based on their experience about

changes, they formulated general statements about evolution of software systems in several *laws of software evolution*. Readers argued that the observations were coupled to project organization of one specific corporation (IBM) and the results were not statistically significant enough in order to formulate laws. In reality however the generality of the laws has been validated over time [LEHM 96].

The first of Lehman's laws is quoted at the beginning of Chapter 2, the seventh law in Section 2.2.1. We briefly summarize here the content of the remaining six laws: Evolving software inevitably increases in complexity unless restructuring is performed to reduce this complexity. Furthermore functional content of programs must be continually increased over lifetime to maintain user satisfaction. The most controversial law is the fourth one. It predicates that the average global activity rate on an evolving system is invariant over the product life time. The activity is measured rather in achievements concerning the software itself (work output) than in investment of person time (work input). Work output reflects better than work input the impact of many more feedback loops on the total productivity [LEHM 98]. The law relates to the possible counterintuitive effects about adding manpower to a project [BROO 75]. Finally the eight law states that software development stringently bases on an incremental process with user feedback at different stages of development.

3.3.2 Software Aging

David Lorge Parnas has been investigating in the causes and implications of aging software [PARN 94]. He realized that software aging occurs in all *successful* products. In contrast the only programs that don't get changed are really bad ones that nobody wants to use. Parnas distinguishes two distinct types of software aging: *lack of movement* and *ignorant surgery*. The first one results from the failure of users to update or change their software to meet changing needs. The second one is the result of changing software without understanding enough of the system's design concepts. Parnas believes that programmers are too much concerned to get their first version running or to meet a looming deadline. However they should be looking far beyond the first release to the time where the developed product is old. He knows that predicting changes is about as difficult as predicting future. Still he thinks we could classify different kinds of change and then assign a certain probability for each of these change types. We would then have to consider in advance at least the more probable changes.

For Parnas it is not sufficient to take into account possible changes only. He sees the investment of time for good documentation as one of the key factors to avoid major problems in late stages of development. He states that documentation is normally inadequate. Either programmers make a couple of memos that help only themselves to remember some tricks, or they employ a technical writer who does not know the system for the documentation of their product. Such documentation surely won't explain future

programmers the behavior of the system precisely. Parnas believes that investing time in good documentation would pay off substantially in later stages of development.

3.3.3 Software Evolution Based on Product Release History

Gall and Jazayeri examined the structure of a large telecommunication switching system with a size of about 10 MLOC over several releases [GALL 97]. The analysis was based on information stored in a database of product releases, the underlying code was neither available nor considered. They investigated first by measuring the size of components, their growth and change rates. The aim was to find conspicuous changes in the gathered size metrics and to identify candidate subsystems for restructuring and reengineering. A second effort on the same system focused on identifying logical coupling among subsystems in a way that potential structural shortcomings could be identified and examined [GALL 98]. For each subsystem a *change sequence* was extracted. They defined a change sequence as an n-tuple of subsequent system release numbers where the version number of the subsystem changed. They defined two subsystems to be coupled if they have a common subsequence in their change sequence. This indicates that they were changed in the same versions and therefore have a similar change behavior. In a third work Riva developed a tool to visualize changes in 3D space [RIVA 98]. The third dimension allows us to visualize historical information together with the system's structure. [JAZA 99]

The approach based on product release history scales up well to large systems containing a huge amount of code. Considering all change details found in the source code may be rather confusing than practical. An approach that is not based on source code has an additional commercial advantage: developer teams need not to show their source code to external consulting people. On the other hand the whole change analysis based on product release history remains a vague guess about actual changes in the code. It happens that a new version of a subsystem is created without any changes in the source code.

3.3.4 Changes in Calling Structures and Data Usage

Burd and Munro have been analyzing the calling structure of source code [BURD 99]. They transformed calling structures into a graph using dominance relations to indicate call dependencies between functions. Dominance trees were derived from call-directed-acyclic-graphs [BURD 99]. The dominance trees show the complexity of the relationships between functions and potential ripple effects through change propagation. The more ripple effects, the more effort is required to understand the code. More ripple effects signify more side effects after a change. The dominance relations were analyzed

for several versions of the same software. Changes in the graph were tracked over time. Such changes give an indication about the changing complexity of the software and about change impacts. Burd and Munro defined metrics to quantify changes in complexity on the proportion of strongly dominated nodes to direct dominance nodes. The case study to validate the approach was Gnu compiler `gcc` written in C. In total about 9 million lines of code (MLOC) were analyzed.

In another approach Burd and Munro studied the usage of data defined within source code of software systems [BURD 98]. They analyzed how data items change within a program due to evolution. The information was retrieved through the use of data clustering. Procedures using the same data items were grouped together to identify potential candidates for encapsulation during re-modularization. The case study here was a commercial application written in COBOL.

The use of dominance relations is an excellent tool to analyze change propagation within source code. It is questionable how well such a practice could be adapted to analyze object oriented systems as well. A precise identification of the invoked entity is not always possible due to polymorphism. Especially in dynamically typed languages like Smalltalk the invoked method may be any of all implemented methods with the same signature [1]. The list of candidate methods can be rather big. The available information may be too blurry in order to calculate dominance trees. Additionally, a transformation of source code into a graph means also a loss of information.

3.3.5 Efforts in Object Oriented Software Evolution

There has not been much effort done in concrete evolution analysis of object oriented software systems. However the object oriented paradigm itself promises to support development and evolution of software through various techniques:

- **Data Abstraction:** The internal data of an object is encapsulated and accessible only through a public interface. Encapsulation limits the effects of internal changes to the outside and vice versa.
- **Reuse:** Classes help to bundle methods designed to handle similar data structures. Each instantiated object will have access to the methods defined in its respective class.
- **Extensibility:** Inheritance allows one to define common sets of superclasses, also known as frameworks. Domain specific subclasses can be defined for each concrete application, inheriting the functionality defined in the framework.

[1] A signature is composed of the method name and parameters assigned to the method.

- **Decomposition:** The complexity of the whole domain is split into several classes. Classes that interact more closely again form subsystems, an additional layer to reduce overall complexity.

Apparently object oriented techniques themselves are not the promised silver bullet that solves all problems in software development. It seems difficult to really make use of the advantages an object oriented language provides. Software development would be much easier if we had several excellent frameworks at hand, each one covering certain domains [ROBE 96]. Those frameworks would have a structure simple enough to understand the API quickly. Yet they would still provide enough functionality to be easily extended and adapted to specific requirements. In reality unfortunately we rarely find such first class frameworks. Developing good frameworks is difficult and expensive since nobody will write a good framework from scratch. We first have to invest a lot of time and effort to get experience in the domain before we can start building good frameworks.

> *"People develop abstractions by generalizing from concrete examples. Every attempt to determine the correct abstractions on paper without actually developing a running system is doomed to failure. No one is that smart...Domain experts won't understand how to codify the abstractions that they have in their heads, and programmers won't understand the domain well enough to derive the abstractions."* [ROBE 96]

We'll never get it right the first time. Therefore a usual development cycle starts with writing prototypes. We need to check whether our ideas can be implemented the way we thought. Once a prototype runs smoothly, we can extend it by adding more functionality. The program also needs to be adapted to unforeseen shortcomings in the implementation and changes in the requirements. Such operations normally entail a drift away from the initial design. To counteract such forces, we have to insert consolidation phases where we restructure the code and try to find an elegant new design. The new design should still provide the same functionality, but additionally comprise the changed conditions. Our knowledge about the domain will increase the longer we work on it. It will help us to find implementations that map better the required functionality.

Especially class hierarchies usually grow fast in expansion phases. We quickly expand classes to add new functionality. In a consolidation phase we therefore have to factor out common behavior and collect it in common superclasses. Only then we can avoid duplication and keep the structure in shape with the design. We extract common core functionality of different implementations through refactoring and use that code to build new framework parts [FOOT 94].

3.3.6 Evolution Observations of Industrial OO Frameworks

Mattsson has been analyzing historical data of object oriented frameworks over time. One medium sized case study in the telecommunication domain (300-600 classes) consisted of four main releases of a billing gateway. He compared his results with the Microsoft Foundation Classes (MFC) framework. Mattsson collected various evolution observations in his PhD thesis [MATT 00], a collection of previously published papers. Mattsson and Bosch calculated size, change and growth metrics on entity level for the whole system and all the subsystems. Based on these metrics they made assumptions about the structure of the framework. They declared subsystems with different characteristics regarding growth and change rates compared to the whole system as candidates for redesign [MATT 99a]. In another approach Mattsson used a set of architectural metrics, mainly calculations on the structure of class hierarchies. He compared the metric values of different historical versions of the same OO-Framework. Based on the collected metrics he formulated four hypotheses about framework stability, and about how frameworks change during their lifecycle. As an example, the first hypothesis states that stable frameworks tend to have narrow and deeply inherited class hierarchy structures. Besides the structure of the framework he measured the development effort for each version of the framework, normalized to the invested effort for the first release. He also compared between the versions the relative effort spent for different activities like design, implementation, tests, administration. He found out that the main effort for the initial version was the actual development work, while in the last version the testing part consumed most of the time [MATT 99b].

3.3.7 Refactorings in Object Oriented Code

> *"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. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written."* [FOWL 99]

This quote of Martin Fowler points out the key attributes of refactoring. Refactoring is a coding technique applied mainly in consolidation phases during development. It has been developed and propagated by Kent Beck and Ward Cunningham in the context of the programming style XP (Extreme Programming, [BECK 99]). Bill Opdyke examined refactoring systematically in his doctoral thesis back in 1992 [OPDY 92]. John Brant and Don Roberts developed the *Refactoring Browser* for Smalltalk [ROBE 97], a tool that assists developers to do basic refactorings. Examples for refactoring are the renaming of a source code entity or the pushing up of methods in a superclass. Refactorings promise the following four advantages:

1. **Improved Design:** Periodical refactoring keeps you cleaning up your own code, it allows you to purge unused elements and to eliminate duplication. Refactoring techniques help a developer to counteract the inevitable decay of code and to keep the structure of source code consistent with a proper design. Clean code will help us to find better implementations during development.

2. **Readability of Code:** In expansion phases we just add new functionality. We first need to check whether a function works before we can think about elegant implementations. If we don't refactor our code periodically, we will keep lots of initial quick hacks that were implemented only to check whether an idea for an implementation works.

3. **Avoid Defects:** The tighter the code correlates with our design, the faster and the better we understand it. The better we understand our code, the less defects we implement either directly or through change propagation. If we avoid duplication, we need not fix a defect over and over in similar parts of the code.

4. **Faster Programming:** By definition we do not change the behavior during refactoring, we also do not add functionality to the software as well. For the time being one might think that we are not productive when refactoring in consolidation phases. To achieve longer term goals however we simply *have to* go through such phases to consolidate our current work. If we don't take us time for consolidation, we will suffer later on. At a certain stage of development we will have hard times to understand how our own code. If we fix a defect in a complex structure, we may well introduce new defects at the same time.

In our evolution analysis of source code we intend to find also changes caused by refactorings. We want to find out if developers of a software system use refactorings to improve their design or whether they only add new source without running through consolidation phases. For the found refactorings we try determine why they were performed and what implication they caused in the development.

3.3.8 Finding Refactorings via Change Metrics

Imagine you would have to find a renamed attribute[2] of a class by textual comparison in a large system! Even if we would find one that looked like a renamed one, we would have to check whether there was not just one attribute removed and another new one added. We would have to analyze manually the job of this attribute. A manual search for refactorings in the source code is really tedious. Demeyer, Ducasse and Nierstrasz investigated how change metric could be used for detecting refactorings [DEME 00]. They

[2] We use *attribute* in this document as a synonym to *instance variables*

examined four heuristics based on change metrics and validated them on three different case studies. The four refactorings cover mainly refactorings for a shift of responsibilities in the class hierarchy.

1. Split into Superclass/Merge with Superclass
2. Split into Subclass/Merge with Subclass, Move to other Class
3. Move to Other Class (Superclass, Subclass or Sibling Class)
4. Split Method/Factor out common Functionality

In our work we're going to present an approach which expands the ideas described in this paper. We're going to explore the capabilities of an change detection based on change metrics.

3.4 Goals of an Evolution Analysis

The main goal of an evolution analysis is to gain a more detailed understanding of how and where change processes take place [BURD 00]. We want to know about the amount and kind of changes performed on code during development. A change analysis may expose repetitive patterns we try to understand and classify. Such an analysis could reveal common problems and lead to new guidelines for software development in general. We explain here a couple of reasons that justify the effort of a source code evolution analysis.

Evolution Patterns: The most general goal of an evolution analysis is to reveal what exactly happens during the development process; to find out what changes are most frequently performed during development; whether these changes could be partly automated to support more efficient development. Another important issue is to find regularities regarding the impact of changes, to find out what effect different changes will have on unchanged parts.

Design Analysis: Another interesting issue is to follow implementations of design patterns through several versions. We would like to know which patterns hold through the whole evolution of the software and which ones get substituted by others because they were not flexible enough to persist changing requirements. The tracing of implemented patterns allows us to assess in reality whether they adapt well to changes. We intend to extract successful implementations and to collect them as patterns for the future. On the contrary we also try to identify inflexible implementations to learn why they do not conform with future requirements.

Effort Estimation: Metrics measure different properties of code in various ways. Size metrics measure the amount of lines or entities defined in the code, other metrics the complexity of a problem and its implementation. Metric values can be used to estimate the amount of time that needs to be invested for future implementations. Especially change metrics, which include information about changes over time, help to estimate time exposure for future work based on previous values.

Automatic Documentation: Code documentation is an everlasting controversial subject for developers. Probably every developer agrees that good documentation of code as a matter of principle is useful. On the other hand documentation is time consuming, especially the periodical synchronization after changes. In reality many programmers just hack their lines first without documenting them. They try to find the best implementation quickly and don't want to lose time for documentation each time they change their code. Documenting after the implementation has become stable is annoying as well. Moreover it is difficult to document code in a reasonable way. The possibility of retracing changes in the code helps us to have the changes documented automatically. This allows us to document only major releases. Based on that we would have an automatically created documentation of the changes for subsequent minor releases. The automatic documentation would contain information about added and removed source code entities, and possibly even list deprecated functions that are not in use anymore.

3.5 Studying Evolution Assists Software Development

We distinguish three target scenarios where we can study the evolution of software. The scenarios differ in size of the analyzed code and in the knowledge about the code. For each scenario we discuss how an evolution analysis can contribute to improve the development process.

Small Project: A small software project covers a manageable amount of source code. An individual developer or a small team work on the same system. The developers are supposed to know about the functionality and the collaboration of their code with other components. Still it may be valuable for them to track removals, additions and changes. With that change data they are able to step back to a previous release in case they reach a dead end. Of course a version control tool provides such basic change functions as well. However it won't provide the detection of deprecated code that is still integrated in the release. Such a code should better be purged since it just makes the API more complex without providing new functionality. A code analysis tool that combines basic change information, entity properties and metric heuristics, may lead developers to interesting parts in their

code. Through a systematic evolution analysis, they may find out facts about their code they would not have expected.

Large Project: Source code of a large development project consists of a huge amount of code that cannot be completely browsed manually. The development team is large and consists of several groups working on different subsystems. Open source projects usually consist of a huge amount of code as well. Because of the size and complexity nobody has a clear view over the whole code and how subsystems interact with each other. Abstraction tools that create more abstract views on the whole system are worthful to overview the structure of the whole system. An evolution analysis tool provides additional information about changes over time. Abstract views may reveal whether a subsystem is stable or whether it changes a lot. It may detect unexpected dependencies and allow a user to track them over time.

Reverse Engineering: Candidates for reverse engineering are usually large, complex systems. People analyzing such systems normally have a broad knowledge about writing quality code, but usually not about the code of the analyzed system. They first need to get an overview over the reverse engineering candidate and try to understand how different components work together. Information about previous implementations and the changes between them give them additional hints about how the system got in its current state. An analysis of the way to the end result always contains lots of additional information compared with a snapshot of the end result. We expect to reveal how a system turned from a healthy state to its actual desolate state. That would be hard to see if we only consider information of the last release.

Chapter 4

A Query-based Approach to Recover Software Evolution

It is hard to analyze code of large software systems just by browsing the files manually, because we come across relevant sections only by chance. Meanwhile we lose quite some time just browsing code, searching for relevant information. The amount of data is so huge that we'd be just lucky if we were able to identify problematic implementations in the unknown code. The evolution analysis of a software system would even force us to browse multiple versions, and therefore a multiple of the amount of data stored in one single release. A key problem is the separation of relevant parts from irrelevant ones, especially from noise. We use the term *noise* for data that is not relevant for our analysis. Source code information that has been misinterpreted by the parser, for instance, belongs to noise. Luckily source code is much more structured than usual data mining information. Relations between objects like *aggregation* or *inheritance* and the identification of the same object in multiple releases help to reduce information and extract desired facts.

In this chapter we're going to describe our approach to reveal evolution data. Our approach is based on filters and simple metrics computed on source code entities. The filters are expressed in queries which select from a collection of entities only those with the correct properties to pass the filter. We provide a method to compose filters in series or in parallel. With such a composition method we can maintain flexibility for the definition of new, more complex conditions for an entity to pass a filter. Filter templates stored in a repository can be adapted and refined to analyze different software.

The computed metrics serve as an additional criteria to filter data. In our approach we intentionally take only simple metrics into account. The higher the abstraction, the less obvious the changes in the abstraction. Complex metrics such as coupling or cohesion [FENT 97] help to state certain properties about the entity they are computed for, yet they are difficult to compose in a meaningful way. The more complex the metric, the harder

it is to trace back and state what exactly happened on source code level. For some metrics we don't see clearly what a change of the according value means on source code level. If we combine conditions of complex metric values, the meaning of the created combined condition gets even more blurry. Summarized, complex computations simply do not necessarily give more information.

We build our approach on an abstraction of the source code called FAMIX (FAMoos Information eXchange model) [DEME 99b]. FAMIX is a format for the exchange of information about object oriented source code entities. The FAMIX meta model maps the basic structure of the underlying source code and contains the entities *Class*, *Attribute* and *Method*. It also maps relations between entities such as *Invocation*, *Access* and *Inheritance Definition*. Appendix A contains an informal description of the FAMIX meta model.

The final intention of our query based approach is to collect a set of evolution queries for a repository. These queries are supposed to be reused for the analysis of various software systems. Each of the evolution queries reveals a set of source code artifacts with certain change characteristics between releases. We then just need several versions of the source code of a software system. We parse the source code for each of these versions and create a model. We then apply the whole set of collected queries on the code. Each query reports facts about particular changes. This basic information serves as a description of the overall change between the releases.

4.1 Extracting Information from Source Code

We use information extracted from source code for the assessment of the evolution of software. Our evolution queries however are defined on the FAMIX meta model, not directly on the source code. Therefore analyzed code needs to be parsed first and transformed into the model format afterwards.

We need the source code of multiple releases as a prerequisite for a change assessment. Therefore we can analyze only systems whose source code of different releases is still available. Thus the code needs to be frozen and stored periodically during development, or the whole development process bases on a version control system. Such a tool ensures that either each step in development is stored separately, or at least the daily work is stored as a new releases in a database.

4.2 Comparing Multiple Releases

Once at least two releases of a software are loaded and stored in FAMIX format, the different models can be compared. At first we extract basic change data. This data

describes us for each source code entity how it changed between two versions, or if it changed at all. We distinguish four basic types of change between two releases:

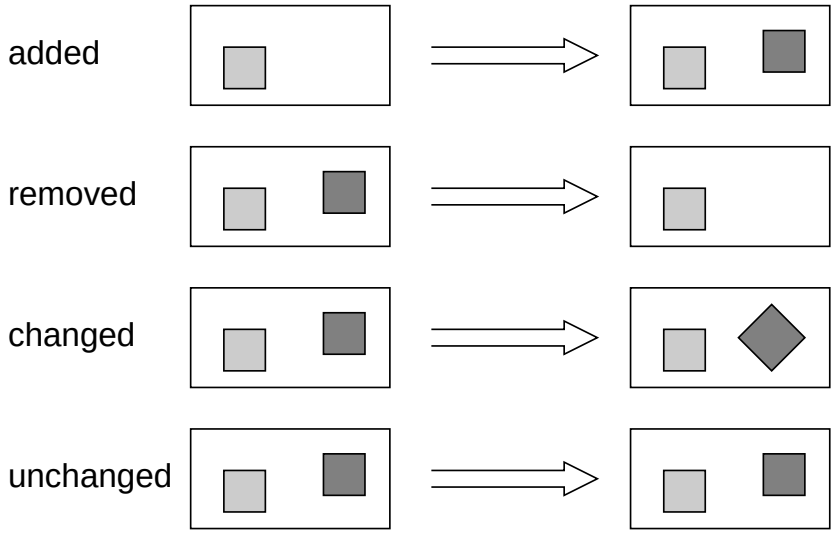

Figure 4.1: The four possible types of change for a source code entity

- **Added:** The entity did not exist in the previous release, but exists in the current release. It has been added between the two releases.
- **Removed:** The entity existed in the previous release, but does not exist in the current release. It has been removed between the two releases.
- **Changed:** The entity exists in both releases, but has changed properties. At least one property of the entity has been changed between the versions.
- **Unchanged:** The entity exists in both versions and did not change. All properties are identical in both releases

Every entity in the system conforms to one of these four types of change. Any information about the evolution of a software must be derived from change data based on these four types. An added entity only tells us that the system has grown, a removed entity that the system has been reduced by one entity. An unchanged entity just states that there have been no changes regarding that specific entity between the versions. This is not much information, yet it can still be valuable for us: we need not analyze these entities further since they are the same as they were in the previous release. The *changed* entities provide the most valuable information about changes between versions.

It is important to choose the right identification method to track entities between releases. They need to be identified over a unique property that matches across versions. A problem arises if the entity persists over two releases, but exactly the identifying property has changed. In such a case we are not able to track the entity further.

In our approach entities are identified between releases over their *unique name*. As the term suggests, a unique name identifies a source code entity uniquely in a single model. The *name* of an entity on the other hand may not be unique and may occur more than once in the same model. In Smalltalk for example, the method named *printOn: aStream* is implemented in several different classes. The unique name of each implementation of printOn: however differs. It is composed of the class name the method belongs to, followed by the name of the method. The printOn: method of the class Object for example has the unique name *Object.printOn:*.

Renaming of entities between versions poses certain identification problems for our identification algorithm. Imagine what happens if an object is renamed and thus its unique name changes: our identification method fails, it seems as if an object disappears in the old version while another object appears in the new version. We won't recognize that the removed and the added entity are the same. In case we take another (unique) identifying property, we would again identify the entity as the same in both versions. Metric values are generally not appropriate to identify entities between versions because they are in most cases not unique and may even change. A good identification technique for classes is the comparison of methods and attributes that belong to the class. They are usually not all renamed as well at the same time. For a detection of renamed methods we may look at invocations defined in the method body. For attributes we may calculate a kind of *fingerprint* that consists of entities accessing the attribute and compare the fingerprints. Once we have identified the renamed entities, we need to combine the information about changes with the one about renaming to track renamed entities further. If we manage that, we will realize that the name of the entity has changed between the versions, yet we will see that it is still the same entity.

As stated above, the most interesting type of change from an evolution point of view are the *changed* entities, because they can be identified as the same object over multiple models. The changed properties clearly indicate what happened with them during evolution of the software. For our evolution analysis we will mainly focus on *changed* entities and build our queries on changed properties.

4.3 The Concept of Query Composition

All queries in our approach have in common that they require a set of objects as input, and also return a set of objects. In our original concept, all queries returned a subset of the input, the objects that fulfill certain conditions defined in the query. Later on we had to extend the concept by some additional queries that return a set of relatives to the input objects. One defined query for example expects entities that belong to a class, like methods or attributes, and returns the according classes instead of the input entities.

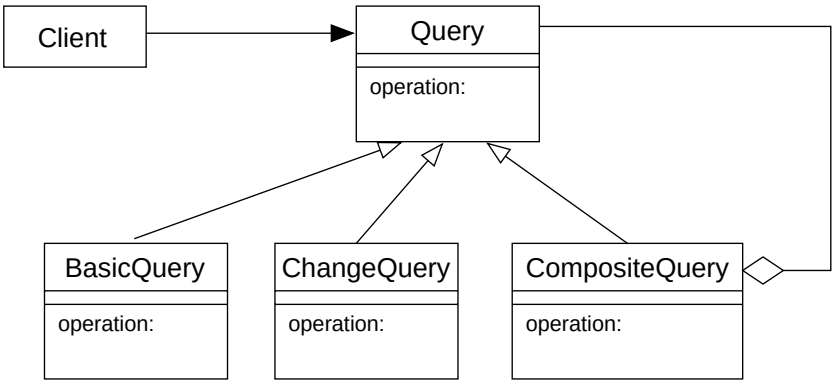

Figure 4.2: The composite pattern applied on queries

All queries follow the structure of a composite pattern [GAMM 95]. An object in a composite structure is either a single object (leaf) or a composite. A composite has a tree structure and consists of branches (other composites) and leaves (single objects). A composite structure lets clients treat individual objects and compositions uniformly. In our query composition concept, basic queries represent all kind of possible leaves while composite queries represent different of branches (see Figure 4.2).

In the following sections we introduce the concepts of our different types of queries. We discuss them in the following order:

- **Basic Queries** are filters on entity properties and metric values.
- **Change Queries** are filters on change metrics.
- **Composite Queries** are compositions of other queries.

4.3.1 Basic Queries

All basic queries expect objects of one single release as input and return a subset of the input. They are used to filter objects according to certain properties. Basic queries represent the leaves in the composite pattern. They are never composed of other queries. We count four different types of basic queries:

1. **Type:** *Type queries* filter objects of one specific FAMIX type only. The core FAMIX types are *Class, Method, Attribute, Invocation, Access* and *Inheritance Definition*.
2. **Name:** Each *Name query* contains a regular expression. The string pattern is compared with the name of each entity sent as input to the query. All entities with a name matching the regular expression are returned.

3. **Metric:** A set of metrics are defined in MOOSE for each specific entity type. The values of these metrics are calculated in advance for each entity in the model. A metric value is always numeric. Each metric query has a metric, a threshold and a comparison operator (less than, less or equal, equal, ...) defined. The threshold is compared with the metric values of the input entities using the operator defined in the query. The query returns all entities with a metric value that holds the condition. Section A.3 in Appendix A explains all metric abbreviations used in this document.

4. **Property:** A property is a value that further characterizes an object. The value of the property is usually a boolean or a string. Examples for class properties are *belongsToSubsystem, isAbstract, isInterface, isStub*.

4.3.2 Change Queries

A change query runs on at least two different models. Instead of comparing one single property of a special entity, the query tests the change of a metric value or a property between the two models. We need to define such additional *Change Queries* since we cannot compose change conditions from basic queries. We are not able to express conditions about the change of a metric value relative to the value it had in the previous release. Therefore we introduce change queries. A change query represents a special type of leaf in the composite pattern. We will frequently use change queries to compose our evolution queries described in Chapter 5. These newly introduced queries help us to identify changes about entities between two versions. This change information is very useful for understanding the evolution of a software system. Furthermore, it is easier only to compare numeric metric values than for example referenced entity attributes.

4.3.3 Composite Queries

Composite queries are all composed of a number of subqueries. We list here the different compositions of query conditions. Each different composite query reflects one of the special kinds of composition defined here. The different compositions differ in two ways. First they differ in the kind of delegating conditions over different relations to other entities. Second they differ in the handling of the output of each subquery, and in the merging of the outputs to an output of composite queries.

Affiliation An affiliation query consists of a query that runs on objects with *belongsTo* relations. The objects can be classes that belong to subsystems, methods or attributes that belong to classes, or also the respective set of classes a collection of methods belong to. For each input entity, the appropriate class or subsystem it belongs to is searched. The condition of the query is not formulated for the input

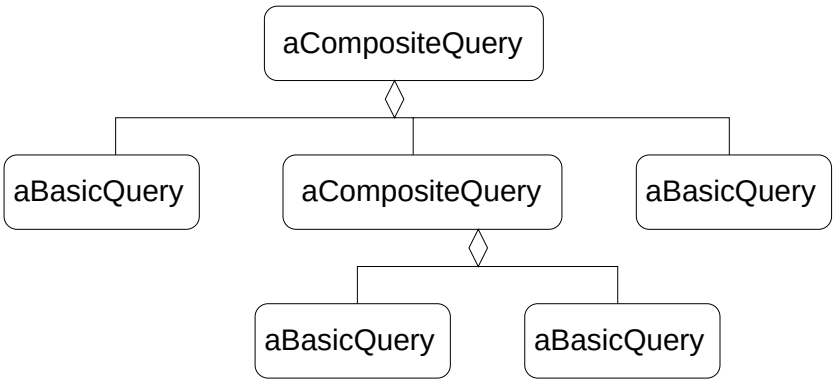

Figure 4.3: A typical structure of a composite query

entity directly. Instead, a condition for the object the entity belongs to is checked. If the related object satisfies the query condition, the input method is returned. The query therefore returns a subset of the input methods.

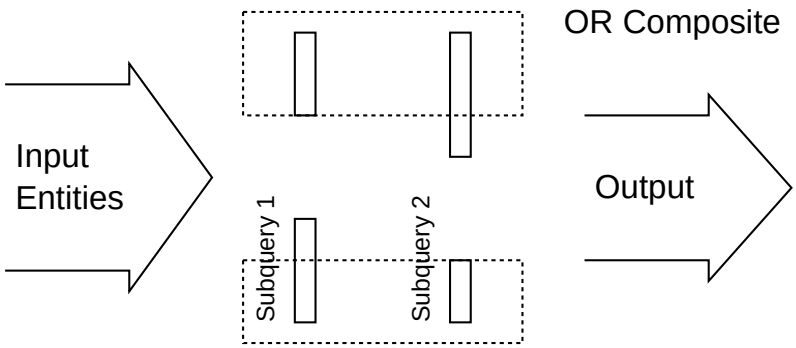

Figure 4.4: OR-Composition of two queries

Composition A composition query contains a collection of subqueries. The output of the subqueries is either merged (OR composition), or an intersection is taken (AND composition). In case of an OR composition, each subquery is fed with all input entities of the whole query. All output entities of the subqueries are then merged and returned as the output of the composed query. Each output entity satisfies the condition of at least one subquery (Figure 4.4). In case of an AND composition, the output of the composed query is the intersection of the output of the subqueries. Each entity in the output of an AND-composed query satisfies all of the conditions defined in the subqueries (Figure 4.5). The subqueries must be defined on the same model for both composition types. If this is not the case, we get a heterogeneous output containing entities of several different models. A heterogenous output is not useful since we may loose the correct references to related entities in a specific model.

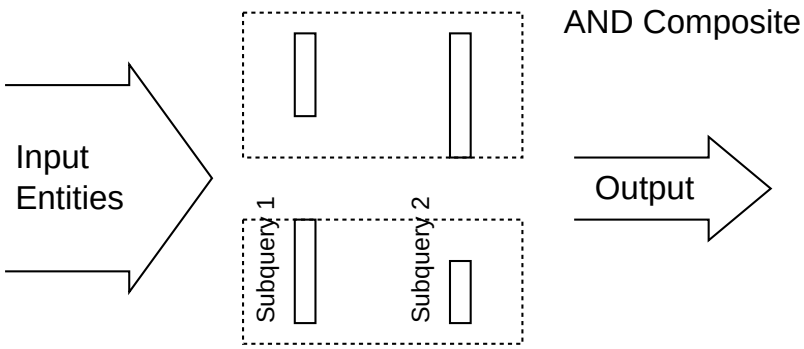

Figure 4.5: AND-Composition of two queries

Dependency We define here dependency as one of the two following relations between objects: invocation and access. A dependency query expects methods or attributes as input. The query condition does not test the actual input entity, the condition is rather forwarded over an invocation definition or an access definition to related entities. The checked entity is an invoked method or an accessed attribute.

Conversion The conversion composite query extends a basic concept of our queries: that always a subset of the input entities is returned. A conversion query returns objects that are in a certain relation with the input entities. For each input entity that satisfies the query condition, an appropriate related object is returned. As an example, a conversion query that expects a set of methods as input entities checks the query conditions for each method. Instead of methods, the query returns all classes that contain a method satisfying the query condition.

Hierarchy A specific predicate to object oriented code is inheritance and thus class hierarchies. Like all other relations between entities, an inheritance relation also helps us to characterize changes more precisely. Movements in the class hierarchy usually entail rather heavy rearrangements in the structure of the source code. Therefore changes in the class hierarchy are a good indicator for considerable restructuring. A *Hierarchy Query* allows us to define conditions on relatives of a class. We can define conditions on superclasses, one single subclass or all subclasses of a input class. A hierarchy composite query forwards the condition to an appropriate relative, if the input entity has such a relative at all.

History A history query consists of a collection of queries that do not all need to be defined on the same model. Like in a *Concatenation Query* described above, the subqueries are performed one after the other composed the logic operators AND or OR. Despite the Concatenation queries, the output entities from one query are not passed directly to the next one. A direct passing is not possible since each subquery needs to receive entities from an appropriate model as input. The respective set of entities is searched in the correct model for the next query (4.6).

A unique property is required for an identification of entities between two models. We currently use the unique entity name for identification, but it may also be a set of metric values or other uniquely identifying properties.

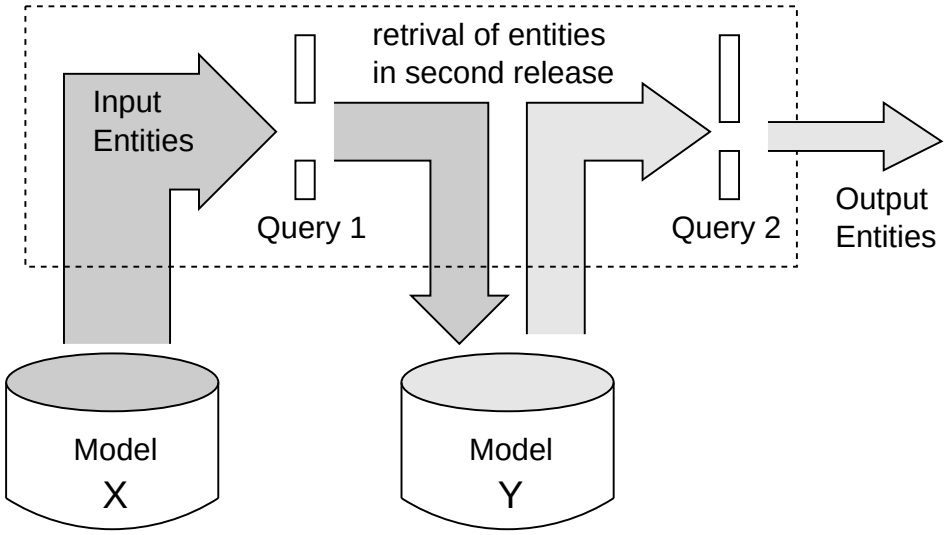

Figure 4.6: Searching the appropriate entities in different models

Chapter 5

Useful Evolution Queries

5.1 Introduction

We present in this chapter a collection of useful queries for a source code evolution analysis. We first introduce five basic queries. They assist us to measure basic information about each release. They also represent the basic modules for the composition of complex queries. In Section 5.6 we list useful queries that are defined on one single model. These queries extract facts about one release only, they do not use change data between two or multiple releases. In Section 5.7 we present a collection of queries defined on multiple models. These queries automatically retrieve special aspects of change between loaded versions.

5.2 Structure of the Query Descriptions

We summarize basic properties in a header table for each query described in this chapter:

- **Query Type:** Here we state the type of basic queries. For composite queries we describe what kind of composition relation we use, according to the different compositions described in Section 4.3.3.
- **Model Scope:** We state here on how many models the query is defined.
- **Information used:** Here we list what information of an entity is used to apply the query. This information may be the entity name, special metrics or properties.
- **Entity Scope:** Here we describe which type of source code entities the query expects as input.

In the body of each query description we discuss the following aspects:

- **Definition:** Here we give a formal description of the query condition.
- **Idea:** Here we describe for each query what entities we expect to reveal, and which kind of information about the system the query provides. We also describe how the query extracts the correct entities.
- **Example:** Here we give an example of the syntax for the query in the MOOSEFINDER tool described in Appendix B. This syntax is used to load a query with all defined parameters. The queries are stored in XML format since it is human readable and supported by various open source parsers.
- **Evaluation:** The evaluation section illustrates in which cases the approximation may fail. Most of the presented queries in this chapter do not provide precise results in all situations, they rather provide good approximations. The queries select a subset of objects which can be further assessed, also by manual comparison with the output of other queries.
- **Variations:** Here we list variations of the query. Many queries enable a user to choose among a couple of similar representations. For complex queries usually a couple of parameters of subqueries can be specified in various ways.
- **Combinations:** Here we list possible synergies with the output of other queries. Since many queries provide approximations, a combination of the results from different queries often helps to eliminate uncertainty. It is sometimes hard to find a way to compose the results of two queries into one single query.
- **Results:** Here we illustrate and discuss what we found out running the query over our case studies.

5.3 The Case Study

We took seven releases of Moose for a validation of our evolution queries. We chose Moose as case study for several reasons:

- The source code of all (more than 100) releases from July 1999 up to now is still available.
- The software has been written by people of our research group and is therefore familiar to us. Analyzing known code opens the possibility to validate observations before applying them on unknown systems. The fact that we have most of the developers in-house allows us to ask them about the purpose of found changes.

- The system has undergone several refactorings and redesigns from the first running version 1.01 to the last available version 3.49.

Table 5.1 contains basic size metrics about the different releases of our case study. The values were counted after the generation of the FAMIX models. The classes are all counted twice since there exists for each Smalltalk class a metaclass with an appropriate anonymous name. The uneven number of classes in the last two results from stub classes which were added to the model without their corresponding meta class.

Moose Release		Number of		
Number	Date	Classes	Methods	Attributes
1.01	1999/07/16	182	1621	248
1.09	1999/09/15	190	1724	262
2.02	1999/12/03	208	1813	275
2.35	2000/05/18	184	1925	264
2.55	2000/10/06	200	2015	280
3.31	2000/12/13	231	2031	294
3.49	2001/02/20	209	2101	280

Table 5.1: Basic size metrics of all Moose releases

5.4 Syntax Declarations

We provide for each query a formal description of the condition. In the table below we list all definitions we use for a the description of the query condition. In contrast, the syntax of the respective query implementation in our tool MOOSEFINDER is much more verbose than the description we provide here. The syntax of the implementations is described in the example paragraph for basic queries.

We frequently use abbreviations of metrics. Section A.3 in Appendix A at the end of this document contains a list of all metrics used in this document with their according abbreviation.

E_x	All source code entities defined in model x
C_x	All classes defined in model x
M_x	All methods defined in model x
A_x	All attributes defined in model x
$e \in E_x$	An arbitrary source code entity of model x
$c \in C_x$	An arbitrary class of model x
$m \in M_x$	An arbitrary method of model x
$a \in A_x$	An arbitrary attribute of model x
$Names(E)$	All unique names of entities in set E
$metric(e)$	Any kind of metric defined for entity e
$NOM(e)$	Number of methods of entity e (example for a concrete metric)
$\delta NOM(e)$	Difference in NOM between two versions for the same entity
$subclass(c)$	subclass of class c
$super(c)$	superclass of class c
$class(a)$	the class where attribute a belongs to
$name(e)$	unique name of entity e
$signature(m)$	signature of method m
$attributename(a)$	name of attribute a
$op.$	a comparison operator, $op \in <, <=, =, >=, >$

5.5 Basic Queries

The query framework is built up from a couple of basic queries. Each query expects a collection of source code entities as input and returns a subset of the input. The output contains entities which fulfill the condition defined in the query.

5.5.1 Entity Name Query

Query Type	Basic
Information used	Name
Model Scope	Single Model
Entity Scope	Entities

Definition:

$$\forall e \in E_x \quad | \quad name(e) \subseteq aString \tag{5.1}$$

Idea:

Code of the same application frequently follows naming conventions like for:

- **Classes:** Class names may start with the abbreviation letters of the subsystem they belong to.
- **Methods:** Accessor methods may be implemented as *getX()* and *setX()*
- **Attributes:** private attributes may contain only lowercase letters, while public attributes may also contain upper case letters.

Such conventions can be used to extract entities that have some common characteristics. A name query selects entities with names matching a given string pattern formulated in a regular expression. The query either selects all these matching entities or it rejects exactly those if the negation flag is set.

Parameters

comparison	name, uniqueName
negation	true, false
case sensitive	true, false

The design of the query leaves the choice to match a string pattern with the *name* or the *uniqueName* of an entity. The query condition can be negated by setting the *negation* flag to false. The comparison of the name is either case sensitive or just an order of subsequent letters.

Example:

```
<MooseQuery type="MSEMatchNameQuery">
<negateFlag value="false" />
<matchingPattern value="*Abstract*"/>
<considerCase value="true"/>
<compareUniqueName value="false"/>
</MooseQuery>
```

This example query reveals all entities with a name matching the pattern `'*Abstract*'`. The regular expression is compared with the entity name instead of the uniqueName. The query condition is not negated and the comparison is performed case sensitively.

Results with Moose:

Each class of Moose is supposed to start with the letters `'MSE'`. Additionally the name of abstract classes is supposed to match the expression `'*Abstract*'` and vice versa.

As we see in Table 5.2, the total number of classes is always bigger than classes starting with the key letters *MSE*. When a model in Moose is generated, a number of classes outside the loaded application are also taken into account. Instances of classes such as *Object, String and Behavior* are frequently used somewhere within loaded code, but not defined there.

The collection of classes annotated as *isAbstract* does not exactly match with the classes whose name matches the string pattern `*Abstract*`. This is due to the definition for abstract classes in Smalltalk. There is no general definition of an abstract class in the language itself. Therefore Moose interprets a class as abstract if at least one method of the class sends the message *subclassResponsibility*.

Moose Release	Number of Classes			
	Total	Name matching		Property
		MSE*	*Abstract*	isAbstract = true
1.01	91	67	17	11
1.09	105	83	17	15
2.02	104	77	20	13
2.35	92	80	19	15
2.55	100	92	20	14
3.31	116	93	21	14
3.49	105	96	21	15

Table 5.2: Class names matching expressions

5.5.2 Metric Value Query

Query Type	Basic
Information used	*aMetric*
Model Scope	Single Model
Entity Scope	Entity

Definition:

$$\forall e \in E_x \quad | \quad metric(e) \quad op. \quad aThreshold \tag{5.2}$$

Idea:

For each entity in Moose a couple of metrics are computed. These metrics characterize the entity. A *Metric Value Query* uses these values to select entities in the model that satisfy a certain metric criteria. The query compares the metric values against a defined threshold using a predefined comparison operator. Only entities with a metric value below, equal or above the threshold are selected. Frequently used metric values are HNL, NOM, NIV, NOC, WNI (abbreviations see Appendix A.3).

Parameters

metricName	*aMetricName*
metricValue	*aValue*
comparisonOperator	$<, <=, =, >=, >$
negation	true, false

For each query a metric and a threshold called *metricValue* need to be specified. For the comparison with the entity's metric values any of the five comparison operators can be chosen. The metric condition can be negated setting the negate flag to true.

Example:

```
<MooseQuery type="MSEMetricValueQuery">
<negateFlag value="false" />
<metricName value="WNOC" />
<comparisonOperator value=">" />
<metricValue value="20" />
</MooseQuery>
```

Results with Moose:

This query is suited to find out about reasonable thresholds for metric values in different case studies. Once we know how many entities in total are above a certain metric value, we can start adapting our predefined queries. We ensure by an adaptation of the thresholds that we never get too many entities returned. The two classes in Moose v3.49 with more than 100 methods are *MSEClass* and *MSEModel*.

Moose	NOM			NOC	
Release	>20	>50	>100	>5	>10
1.01	20	7	1	10	2
1.09	24	8	2	10	4
2.02	23	6	2	10	2
2.35	25	5	2	8	2
2.55	27	5	2	8	2
3.31	25	5	2	6	4
3.49	27	5	2	6	4

Table 5.3: Thresholds in Moose for the metrics NOM and NOC

5.5.3 Type Query

Query Type	Basic
Information used	*aType*
Model Scope	Single Model
Entity Scope	Entity

Definition:

$$\forall e \in E \quad | \quad type(e) = aType \tag{5.3}$$

Idea:

A Type Query selects all objects of a certain FAMIX type in a model. We use the query to extract all objects of the same type in a model. We can then for example compute size metrics of the system. We can also use the query to filter only objects of one type. Only then we can run queries afterwards that expect all input entities to responds to a certain interface. In such a case the only task a type query has is to select only these types of entities that understand a certain message. The message sent to each input entity is one of those that check for a core FAMIX type: *Class*, *Method*, *Attribute*, *Invocation*, *Access* and *InheritanceDefinition*.

Parameters

inputString	[:anObject | anObject aMessage]
negation	true, false

We implemented in MOOSEFINDER a query that contains its condition in a *Smalltalk Block*. The Smalltalk block contains a parameter (anObject) and a message (aMessage) sent to the object. The concept of blocks allows one to easily send a user specified message to each object in a collection. We use this concept for sending a message defined in the block to each input entity. The message is one of the following: *isClass*, *isMethod*, *isAttribute*, *isInvocation*, *isAccess*, *isInheritanceDefinition* ... Each of these messages checks if an entity conforms to a certain FAMIX type.

Example:

```
<MooseQuery type="MSEBlockQuery">
<inputString: value="[ :anObject | anObject isClass]" />
</MooseQuery>
```

Results with Moose

We used type queries to generate the values in Table 5.1 of the case study description Section 5.3. To count the number of classes, methods and attributes, we extracted entities of the same type in one model with type queries.

Since *Type Queries* are basic queries, they are frequently used in various composed queries to select only a defined type of entities as input for other queries.

5.5.4 Property Query

Query Type	Basic
Information used	*aProperty*
Model Scope	Single Model
Entity Scope	Entity

Definition:

$$\forall e \in E \quad | \quad property(e) = aPropertyValue \tag{5.4}$$

Idea:

A *Property Query* selects only objects matching a given property criteria. The checked values for the properties *isAbstract*, *isStub*, *isInterface* are of the type boolean. For the property *SubsystemName*, the checked value is a string.

Parameters

inputString	[:anObject | anObject aMessageSentToTheObject]
negation	true, false

To maintain flexibility, the property criteria is defined in a Smalltalk block like the type query (see query 5.5.3). This block is passed to each of the input objects.

Example:

```
<MooseQuery type="MSEBlockQuery">
<inputString: value="[ :aClass | aClass isAbstract]" />
</MooseQuery>
```

Results with Moose:

Table 5.2 presents results of the example query above and compares the numbers with a similar query matching the pattern `'*Abstract*'` in the class names.

5.5.5 Metric Change Query

Query Type	Change
Information used	*aMetric*
Model Scope	Two Models
Entity Scope	Entity

Definition:

$$\forall e \in E_{old}, e \in E_{new} \quad | \quad \delta metric(e) \quad op. \quad aValue \tag{5.5}$$

Idea:

When we analyze two different releases of the same source code, we expect most source code entities to exist in both versions. Some of these entities however may have changed metric values. There may have been some methods added to a class, thus the metric NOM is supposed to be increased. Especially the change of a metric values provides useful information about what happened to the code in between. A *Metric Change Query* calculates the change value and allows constraints to be defined on this value. We allow a user to find out three main types of change through appropriate specification of a *Metric Change Query*:

- *Unchanged:* We want to reveal entities with an identical value for a certain metric in both versions. We therefore set the threshold to zero.
- *Difference:* We want to reveal entities with a certain absolute change for a metric between the two versions. We set the threshold to a certain value of change.
- *Percentage:* We want to reveal entities with a certain percental change for a metric between the two versions. We set the threshold to a certain value between 0 and 1 for an increment, and a value between -1 and 0 for a decrement.

Parameters

metricName	*aMetricName*
changeValue	*aValue*
changeOperator	$<, <=, =, >=, >$
negation	true, false
outputModel	*aModelName*
modelList	*aModelList*

We need to specify for each query a certain metric and a threshold. Additionally we need to choose a change operator to find out about increment, decrement or equality. The models which are considered for the metric changes, are defined as a collection in the parameter *modelList*. Since several instances are defined for each entity, *outputModel* defines from which model the resulting entities are returned.

Example:

```
<MooseQuery type="MSEMetricChangeQuery">
<metricName value="HNL" />
<changeOperator value=">" />
<changeValue value="0" />
<changeMode value="difference" />
<outputModel value="1" />
<modelList value="#( 1 2 )" />
</MooseQuery>
```

Results with Moose

For an evolution analysis of a software system, it is always useful to collect at first a list of basic changes between releases. Such a basic overview in Table 5.4 [1] shows how many classes have increased or decreased metric values, and how many have equal values between the versions. We compare here each Moose release with the successive one. Changes in the HNL metric report changes in the hierarchy. That is in most cases either the movement of an existing class or an insertion of a new class. Changes in NIV often indicate refactorings through a split. Changes in NOM may also denote refactorings like push-ups, but NOM is generally more vulnerable for any kind of restructuring than NIV. WNI is even more fragile since no movement of methods is required to change it. The change of a method implementation suffices to change WNI. A class with unchanged values for HNL, NIV, NOM and WNI has likely been taken over from the previous version without changes.

[1] $<$: value has decreased; $=$ unchanged; $>$ increased

Moose Release		Number of Classes											
		HNL			NIV			NOM			WNI		
old	new	<	=	>	<	=	>	<	=	>	<	=	>
1.01	1.09	2	174	2	-	176	2	2	154	22	3	152	23
1.09	2.02	-	146	44	3	184	3	4	170	16	10	169	11
2.02	2.35	-	164	-	6	148	10	8	107	49	10	104	50
2.35	2.55	-	154	4	2	153	3	13	104	41	23	94	41
2.55	3.31	4	166	-	3	159	8	39	111	20	33	117	20
3.31	3.49	-	185	12	1	191	5	9	167	21	12	159	26

Table 5.4: Changes in metric values between versions

Regarding HNL, the 44 classes with increased HNL values between Moose v1.09 and v2.02 strike the eye. Besides there are 12 classes with changes between v3.31 and v3.49 and some minor changes in the remaining releases. The 44 classes with changes between v1.09 and v2.02 will be classified further by queries presented later on (see section 5.7). The query described in 5.7.2 reveals that a new class named *MSEAbstractModelRoot* has been inserted relatively high in the hierarchy, moving down all subclasses. The big number of classes with increased HNL values results only from this insertion. We therefore defined more specific queries to better identify the different causes for a change in the class hierarchy. We present such queries later on in Section 5.7.

5.6 Useful Single Model Queries

5.6.1 Subsystem Affiliation

QueryType	Dependency
Model Scope	Single Model
Information used	subsystem
Entity Scope	Entity

Definition:

$$\forall e \in E_x \quad | \quad subsystem(e) = aSubsystemname \tag{5.6}$$

Idea:

We usually compare the number of entities in the code between versions to estimate the growth rate of a software system. If we count the number of entities for big systems and make a comparison between versions, they normally do not state a lot about the growth for a trivial reason: Some subsystems that were just add-ons in earlier versions may be integrated in later versions. We need to know more precisely about changes in different subsystems in order to see the real changes in number of entities. Therefore we better compare the number of entities in different subsystems separately. The query introduced here returns only entities that belong to a list of defined subsystems. The query helps us to extract entities of one single subsystem only, or entities of a collection of subsystems.

Evaluation:

Before the query can be run over a model, the respective subsystem needs to be assigned for each entity. Currently the subsystem affiliation is stored for each entity as a property. For the moment there is no grouping entity defined in Moose. We would be able to collapse a couple of entities belonging to the same subsystem in such a group entity. An iterator runs over each entity of a model and extracts the appropriate subsystem affiliation from another property. Different iterators derive the affiliation from different sources such as *source anchor*, a property that contains the path and the filename a source code entity is defined in. Smalltalk code is not stored in separate files, but can be loaded directly from the image. The subsystem affiliation for Smalltalk entities can be assigned directly since Smalltalk classes are defined in a category or an application.

The category name or the application name are then just the appropriate subsystem for the entity.

Example:

```
<MooseQuery type="MSESubsystemListQuery">
<mooseModel value="Model1" />
<negateFlag value="false" />
<subsystemPropertyName value="SubsystemName" />
<subsystemNamesList value="( Subsystem1 Subsystem2 )" />
</MooseQuery>
```

The example query reveals all source code entities in model named *Model1* that belong either to *Subsystem1* or *Subsystem2*. The query expects the subsystem affiliation of an entity to be stored in the property *SubsystemName*.

Results with Moose:

Table 5.5 shows the number of classes for selected subsystems in all analyzed versions of Moose. We see that the number of classes in each subsystem generally declines. This is due to the fact that several classes have been defined in one of the core subsystems even though they do not belong to the core. Such classes have been moved out occasionally to an extension subsystem. The number of classes in subsystem *MooseModel* on the other hand increases because numerous additional entity types were defined and added. Subsystem *MooseImporters* has been renamed to *MooseImporting* between v2.55 and v3.31.

	Moose 1.01	Moose 1.09	Moose 2.02	Moose 2.35	Moose 2.55	Moose 3.31	Moose 3.49
AbstractBase	10	10	10	8	8	14	14
CDIFReader	8	8	8	6	6	6	6
Importers	18	18	18	18	16	-	-
Importing	-	-	-	-	-	10	10
Model	44	44	56	54	64	64	64
Operators	28	28	34	20	22	8	8
ParseTree	14	14	14	10	10	10	10
Storage	24	24	24	22	24	18	18

Table 5.5: Number of classes in each Moose subsystem

5.6.2 Invocations between Subsystems

QueryType	Dependency
Model Scope	Single Model
Information used	A invokes B
Entity Scope	Invocation

Idea:

We want to find out how subsystems interact with each other. Most of the interaction between subsystems goes over the invocation of methods defined in another subsystem. We want to analyze how many times methods of a foreign subsystem are invoked. We also want to find out if the invocations across subsystems are unidirectional or bidirectional. It is also interesting to see which subsystems do not interact at all. Subsystem A gets dependent from subsystem B in case a method defined in subsystem A invokes a method defined in subsystem B. A change in an invoked method of subsystem B may propagate to subsystem A. We use here the invocations defined in FAMIX. They base on static information defined in the source code. We do not have any runtime information available.

Example:

```
<MooseQuery>
<queryType value="MSESubsystemInvocationQuery" />
<mooseModel value="Model1" />
<negateFlag value="false" />
<subsystemPropertyName value="SubsystemName" />
<sourceSubsystemName value="SubsystemA" />
<targetSubsystemName value="SubsystemB" />
```

The example query retrieves all invocations from behavioral entities of subsystem A to behavioral entities of subsystem B. Behavioral entities are an abstract type in FAMIX and comprise all types of entities that implement behavior. In object oriented code such entities are usually methods. Functions are another kind of behavioral entities.

Evaluation:

The assignment about subsystem affiliation is not automatically performed when loading a model. This is simply because we apply different assignment methods according to

where in the code the subsystem affiliation is defined. Before we apply the query on a model, we need to run an iterator that assigns each source code entity the correct affiliation. In Smalltalk it is usually the application or category the entity is stored in. In C++ code the subsystem affiliation either appears from the directory structure, or we need to rely on the classification by a developer.

Unfortunately we can not identify the invoked method precisely because in FAMIX we have only static information available. In dynamically typed languages, the object on which the method is invoked can be of any type that implements a method with the same name. In statically typed languages we can reduce the candidate methods on a subtree in the class hierarchy. We know at least the type of the object except for polymorphic variations. In such cases we assume an instance of the base class in the subtree to be invoked.

Results with Moose:

Table 5.6 and 5.7 show all found invocations between the listed subsystems. We see that there have been a couple of cross dependencies removed in the release 3.49. In MOOSE release 2.02 every subsystem accesses some methods of subsystem *Storage*. In MOOSE release 3.49 these invocations have been removed. The subsystems *CDIFReader*, *ParseTree* and *Storage* all have no invocations from outside. The decoupling of these subsystems is clearly an improvement compared to the earlier release 2.02. The invocation of the methods defined in the subsystems CDIFReader, ParseTree and Storage are only accessed from the graphical user interface in MOOSE v3.34. The user interface subsystem is not considered in tables 5.6 and 5.7.

	AbstractBase	CDIFReader	Importers	Model	Operators	ParseTree	Storage
AbstractBase	9	-	-	-	-	-	1
CDIFReader	-	75	5	4	-	-	5
Importers	13	-	116	66	-	4	7
Model	-	-	-	213	-	-	8
Operators	40	-	-	136	249	5	11
ParseTree	-	-	15	24	2	36	5
Storage	3	-	-	4	8	-	420

Table 5.6: Invocations between subsystems, Moose v2.02

	AbstractBase	CDIFReader	Importing	Model	Operators	ParseTree	Storage
AbstractBase	46	-	-	2	2	-	-
CDIFReader	-	88	7	5	-	-	-
Importing	6	-	98	21	-	-	-
Model	2	-	8	555	-	-	-
Operators	2	-	-	44	19	-	-
ParseTree	2	-	-	22	-	34	-
Storage	3	-	-	41	-	-	80

Table 5.7: Invocations between subsystems, Moose v3.49

5.6.3 Accesses between Subsystems

QueryType	Dependency
Model Scope	Single Model
Information used	A accesses B
Entity Scope	Access

Idea:

We want to find out how data is accessed and exchanged between subsystems. We therefore analyze from where attributes are accessed. It is problematic to let methods access attributes defined in foreign subsystems. Such accesses violate the concept of information hiding. If a method defined in subsystem A accesses an attribute of a class in subsystem B, a change in the class of subsystem B may imply an adaptation of the accessing method in subsystem A.

Example:

```
<MooseQuery>
<queryType value="MSESubsystemAccessesQuery" />
<mooseModel value="Model1" />
<negateFlag value="false" />
<subsystemPropertyName value="SubsystemName" />
<sourceSubsystemName value="SubsystemA" />
<targetSubsystemName value="SubsystemB" />
```

The query defined here returns all accesses of structural entities defined in subsystem A through behavioral entities of subsystem B. The term structural entity derives from the FAMIX meta model and comprises all types of variables like attributes or local variables.

Evaluation:

For attribute accesses we do not have the same problem regarding dynamically typed languages as we have for invocations. We can determine the class implementing an attribute uniquely. However we cannot determine precisely the type of an instance at runtime containing the attribute. The instantiated type can be of one of the subclasses inheriting the attribute.

The query reveals only direct attribute accesses between subsystems. It does not detect indirect manipulation and accesses of attributes over accessor methods. In Smalltalk

all attributes are protected. Thus the attributes can only be accessed by another class directly over accessor methods. We would have to count the invocation of pure accessor methods instead of attribute accesses.

Results with Moose:

Since all attributes in Smalltalk are protected, they are only accessible by methods within the same class or downwards the inheritance tree. Accesses across subsystems are therefore only found in case a class inherits form a class defined in another subsystem. In order to conform to the concept of data abstraction, in Smalltalk code there are frequently accessor methods used instead of direct attribute access. An invocation of the accessor method in the respective class returns the value of or a reference to the attribute. In Moose we are able to find out whether an invoked method simply returns the value of an attribute (*pure accessor method*) for Smalltalk code.

	AbstractBase	CDIFReader	Importing	Model	Operators	ParseTree	Storage
AbstractBase	62	-	-	-	-	-	-
CDIFReader	-	37	-	-	-	-	-
Importing	-	-	80	-	-	-	-
Model	-	-	-	730	-	-	-
Operators	2	-	-	-	17	-	-
ParseTree	-	-	-	-	-	140	-
Storage	-	-	-	-	-	-	68

Table 5.8: Accesses between subsystems, Moose v3.49

The query provides more useful results for code written in languages with public attributes. Section 7.3.5 contains results regarding attribute accesses between subsystems extracted from an industrial case study written in C++.

5.6.4 Subsystem Inheritance Query

QueryType	Dependency
Model Scope	Single Model
Information used	A inherits from B
Entity Scope	InheritanceDefiniton

Idea:

Apart from invocations and accesses, inheritance denotes a third aspect of dependency between two subsystems. If a class defined in subsystem A inherits from a class in subsystem B, a change of the superclass defined in subsystem B may imply an adaptation of the class defined in subsystem A. A class defined in an application subsystem should inherit from a class in the respective framework subsystem and not vice versa. In combination with a dependency analysis based on invocations between subsystems a part of the uncertainty because of inherited polymorphic methods can be eliminated.

Example:

```
<MooseQuery>
<queryType value="MSESubsystemInheritanceQuery" />
<mooseModel value="Model1" />
<negateFlag value="false" />
<subsystemPropertyName value="SubsystemName" />
<sourceSubsystemName value="SubsystemA" />
<targetSubsystemName value="SubsystemB" />
```

The query defined here returns all inheritances of classes defined in subsystem A from classes defined in subsystem B.

Evaluation:

The query shows up unwanted inheritance dependencies between subsystems, for example from a framework part to the application part. We use this query to ensure that there is no unwanted inheritance between classes defined in different subsystems. A class defined in a framework should not inherit from a class defined in an application. Such an inheritance relation would entail a change propagation from an application to the framework. It would make the whole framework dependent from one single application built on top of the framework. With the *Subsystem Inheritance Query* we can assess if a system conforms to the design guideline stated above.

Results with Moose:

Table 5.9 shows the inheritance across subsystems. All subsystems except *ParseTree* inherit from a class defined in *AbstractBase*. We expected that since *AbstractBase* contains abstract superclasses of Moose. Other core subsystems then inherit from classes defined in AbstractBase. All subsystems except *CDIFReader* inherit only either from classes of the same subsystem or *AbstractBase*. Two classes in *CDIFReader* inherit from *Importing*: *MSECDIFImporter* and its respective meta class defined in subsystem *CDIFReader* inherit from *MSEImporter* defined in *Importing*.

	AbstractBase	CDIFReader	Importing	Model	Operators	ParseTree	Storage
AbstractBase	8	-	-	-	-	-	-
CDIFReader	2	-	2	-	-	-	-
Importing	8	-	2	-	-	-	-
Model	8	-	-	56	-	-	-
Operators	4	-	-	-	2	-	-
ParseTree	-	-	-	-	-	8	-
Storage	6	-	-	-	-	-	10

Table 5.9: Inheritance across subsystems, Moose v3.49

5.7 Useful Multiple Models Queries

5.7.1 Added, Removed Entities

Composition	Concatenation
Information used	Name
Model Scope	Two Models
Entity Scope	Entity

DefinitionAdded Classes:

$$\forall e \in E_n, m < n \quad | \quad e \notin E_m \tag{5.7}$$

DefinitionRemoved Classes:

$$\forall e \in E_m, m < n \quad | \quad e \notin E_n \tag{5.8}$$

Idea:

The *Added Entities* query extracts the unique names of all entities that are defined only in the *new* version. For each collected unique name, the respective entity is searched in the new model. Added entities of a certain FAMIX type are collected as follows: All entities of a defined type are extracted from both versions. From both resulting sets of entities, the unique entity names are extracted and stored in another two sets. The unique names that are defined also in the old model are rejected from the collection of entity names of the new model. The respective entities of these names are the added entities.

The *Removed Entities* query extracts the unique names of all entities that are defined only in the *old* version. For each collected unique name, the respective entity is searched in the old model. The removed entities of a certain FAMIX type are collected just complementary to the extraction of added entities described above. The unique names defined in the new model are rejected from the unique entity names defined in the old model.

Evaluation:

Since entities are matched over their entity name, the query only holds as long as the entity has not been renamed between the two versions. If an entity still exists in the new

version but has been renamed, it looks as if it was removed in the old version and a new entity added in the new version.

Variations:

To find out which methods or attributes of one defined class were added or removed, we can extend our query condition by a *belongsToClass* condition. The query then returns only removed or added entities of one specific class.

Results with Moose:

Table 5.10 shows quantitative summary of the changes among the classes of the system between the subsequent Moose releases. For each class the respective meta class is considered as a separate class. Therefore all values in Table 5.10 are multiples of 2. The only exception are the uneven 61 classes added between Moose v2.55 and v3.31. This is due to the fact that stub class *Behavior* has been added to the model without its respective metaclass.

Changes between releases		Number of Classes		
old release	new release	added	remaining	removed
1.01	1.09	12	178	4
1.09	2.02	18	190	-
2.02	2.35	20	164	44
2.35	2.55	42	158	26
2.55	3.31	61	170	30
3.31	3.49	12	197	34

Table 5.10: Changes in Moose regarding classes

5.7.2 Class Inserted in Hierarchy

Composition	Hierarchy
Information used	HNL, Name
Model Scope	Two Models
Entity Scope	Class

Definition:

$$\exists subclass(c), \quad c \in C_n, c \notin C_m, m < n \quad | \quad \delta HNL(subclass(c)) > 0 \tag{5.9}$$

Idea:

In certain situations it makes sense to split a class into two to have a new abstraction level. If for example a class gets a new sibling with a lot of common behavior, we can define a new superclass and move up common behavior. At the same time as a new super class gets inserted, also a sibling of the split class is inserted. There are also other reasons for splitting a class: We may split a class in two if it simply got overloaded with functionality. In both cases we add a new superclass somewhere in the middle of a hierarchy tree. The original class moves down, and the new superclass takes position of the split class in the hierarchy tree. The query described here is supposed to detect to such inserted classes. The inserted class must not exist in the old version. In order to detect the inserted class, at least one subclass of the newly inserted class needs to be moved down. Otherwise HNL of the original class does not increase and the detection algorithm fails.

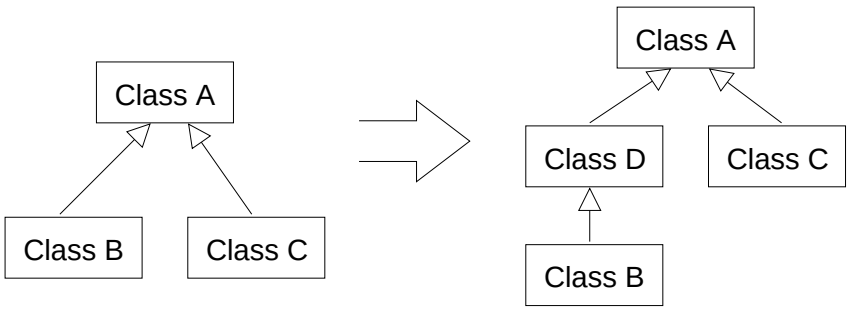

Figure 5.1: Class inserted in the class hierarchy

Evaluation:

In case a class in the upward hierarchy of the inserted class is removed at the same time, HNL of the moved down class won't increase. As consequence the inserted class won't be detected. Since classes are identified over their name between versions, a split class won't be recognized as the same if it has been renamed between two versions.

Results with Moose:

We found in total five inserted classes in the hierarchy, three between v2.02 and v2.35 and two between v2.35 and v2.55 (see Table 5.11). The detected classes have been introduced to collect common behavior of an existing and a newly inserted class. Class *AbstractPackagable* for example contains the common behavior of the existing class *MSEClass* and the added class *MSEPackage* between the two releases.

Moose releases		Found Inserted Classes
old	new	
1.09	2.02	AbstractPackagable MSEAbstractMetricOperator MSEAbstractModelRoot
2.35	2.55	MSEAbstractLocalEntity MSESingleValueConverter

Table 5.11: Classes inserted in the class hierarchy

5.7.3 Removed Superclass

QueryType	Hierarchy
Information used	HNL, Name
Model Scope	Two Models
Entity Scope	Class

Definition:

$$\exists superclass(c), \quad c \in C_{old}, c \notin C_{new} \quad | \quad \delta HNL(subclass(c)) < 0 \tag{5.10}$$

Idea:

We want to detect classes that have been removed in the middle of a hierarchy tree. We search for classes that have been removed between the two versions. Removed classes that have subclasses...

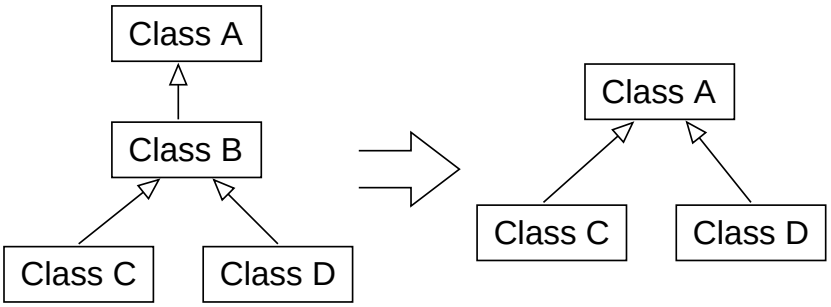

Figure 5.2: Class B is removed between the versions

Evaluation:

The query holds as long as there are no other changes at the same time in the upward hierarchy of the removed class. A renaming of subclasses also confuses the condition of the query.

Results with Moose:

We found *MSEModelDescriptor* that has been removed between Moose v2.55 and v3.31

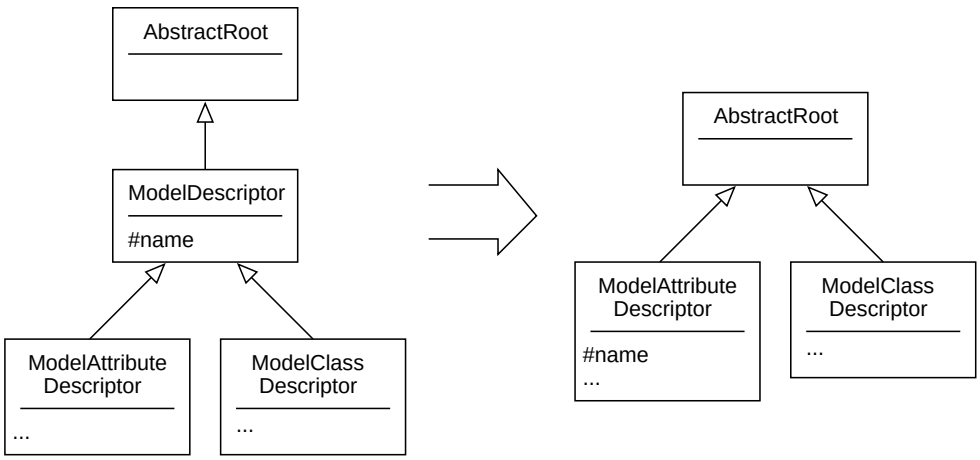

Figure 5.3: Class ModelDescriptor is removed in Moose v3.49

5.7.4 Subclass Becomes Sibling

Composition	Hierarchy
Information used	HNL, NOC
Model Scope	Two Models
Entity Scope	Class

Definition:

$$c \in C_{new} \quad | \quad \delta HNL(c) = -1 \quad \wedge \quad \delta NOC(super(c_{new})) < 0 \tag{5.11}$$

Idea:

A *Subclass Becomes Sibling* query helps to classify different kinds of moving in the class hierarchy. It detects child classes that move one level up and become a sibling of their previous superclass. HNL of the class and NOC of the superclass are expected to decrease.

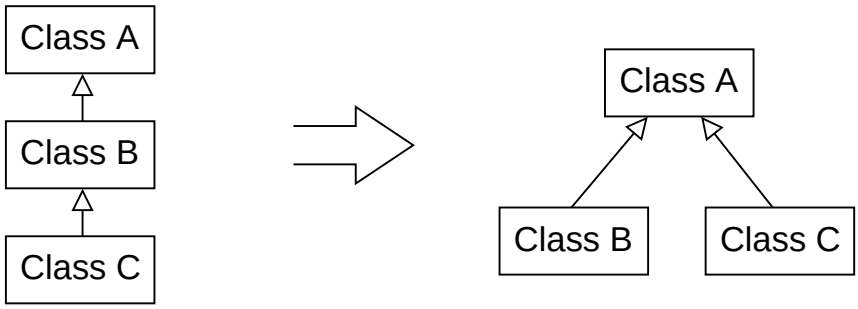

Figure 5.4: Class C becomes a sibling of former superclass B

Evaluation:

This query holds as long as the moved class is not renamed, else it is no longer recognized as the same class in the new release. If there are too many other changes in the hierarchy, the query also fails. If class B in the figure above gets new children, NOC won't decrease. If a new class is inserted in the superclass line of A, HNL won't decrease. In fact there are possible causes to make a detection fail. Yet from experience analyzing existing systems the query generally extracts the targeted classes well. There are rarely many different changes in the hierarchy of a system at the same time. And if there are, one solution is to decrease the interval between two analyzed version to get more fine grained data about changes step by step.

Results with Moose:

We found class *MSEGlobalVariable* that became a sibling of *MSEImplicitVariable* in release 1.09. In release 1.01 the class was a subclass of the future sibling class *MSEImplicitVariable*. Although the two classes have some functionality in common, a *Global Variable* is conceptually not an *Implicit Variable*. The inheritance relation was therefore removed, and the classes moved on the same level in the class hierarchy.

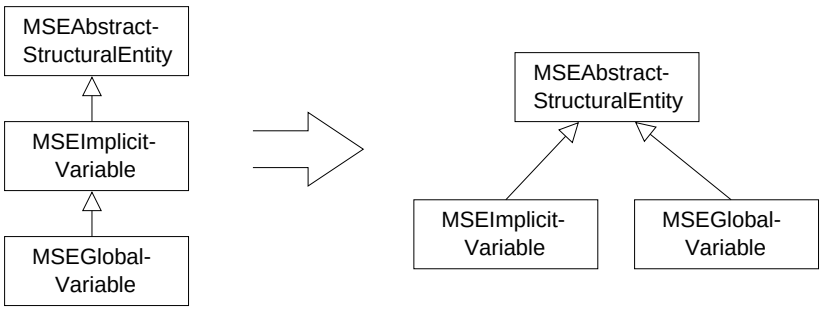

Figure 5.5: MSEGlobalVariable get sibling of MSEImplicitVariable

5.7.5 Sibling Becomes Subclass

Composition	Hierarchy
Model Scope	Two Models
Information used	HNL, NOC
Entity Scope	Class

Definition:

$$c \in C_{new} \quad | \quad \delta HNL(c) = 1 \quad \wedge \quad \delta NOC(super(c_{new})) > 0 \tag{5.12}$$

Idea:

The query detects a special change in the class hierarchy: classes that move down in the hierarchy and get a subclass of a previous sibling class (class B). The hierarchy nesting level (HNL) of the moved class C increases by one. At the same time the number of children (NOC) of the new superclass B increases by one.

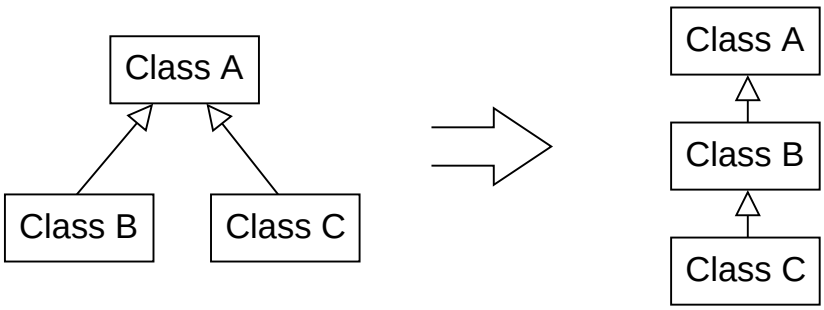

Figure 5.6: Class C becomes a subclass of former sibling class B

Evaluation:

In case there are many other changes in the hierarchy at the same time, the query may fail. NOC won't necessarily increase if a child of class B in Figure 5.6 is removed at the same time. On the other hand if a superclass of class B is removed, HNL of class C won't increase. These are possible causes to make a detection of the discussed hierarchy change fail. Our experience gained from an analysis of different systems tell us that there are rarely other changes at the same time that make the query fail. The query generally extracts the targeted classes well. Since we identify classes over their unique name between models, the query only holds as long as the moved class is not renamed. Else it is no longer recognized as the same class in the new release.

Variations:

NOC may be replaced by weighted number of children (WNOC). WNOC counts all children of a class, not only direct subclasses. Therefore WNOC changes more significantly than NOC if the moved class has a whole hierarchy tree of subclasses.

Results with Moose:

Class *MSEAbstractMetricOperator* is a sibling class of *MSEPropertyOperator* in Moose release 3.31. In release 3.49 MSEPropertyOperator is moved down one level in the class hierarchy, it got a subclass of *MSEAbstractMetricOperator* in release 3.49. This change turned out to be a refactoring by mistake. The weird inheritance has been removed again in the newest release. Astonishingly, class MSEAbstractMetricOperator of release 3.49 was still running correctly.

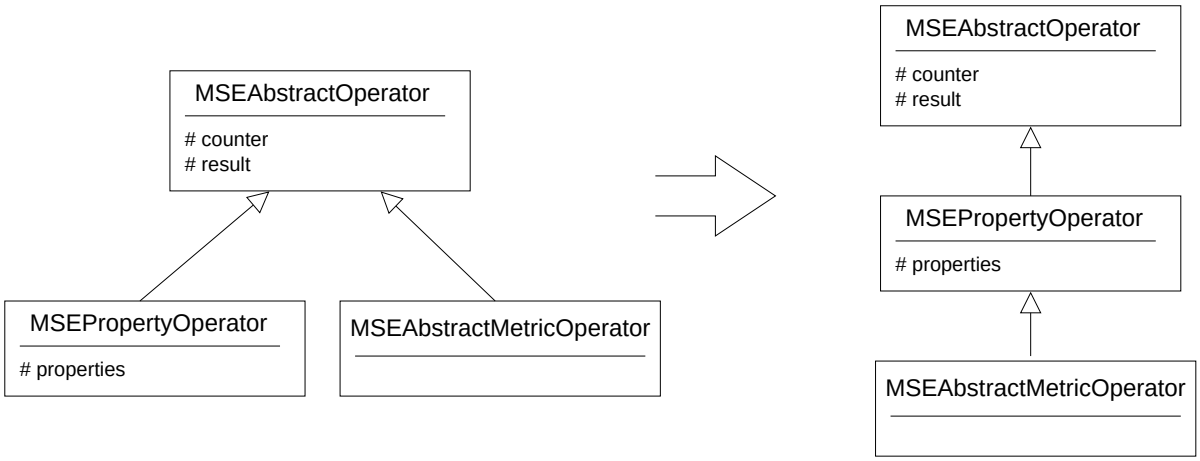

Figure 5.7: MSEAbstractMetricOperator gets a subclass of MSEPropertyOperator

5.7.6 Heavy Change in Hierarchy

Composition	Hierarchy
Information used	HNL, WNOC, Name
Model Scope	Two Models
Entity Scope	Class

Definition:

$$\forall c \in C_{new} \quad | \quad \delta HNL(c) > 0 \quad \wedge \quad WNOC(c) > 20 \tag{5.13}$$

Idea:

This query detects moved classes with relatively heavy impact on the whole hierarchy structure. The classes have several children and get pushed down in the hierarchy. A possible scenario for such a change is a split of an abstract class where one part gets declared in a newly inserted class.

Evaluation:

The query holds if there is no superclass removed from the system at the same time since in such a case HNL would not change. To specify a hard-coded threshold of 20 for changes in NOC is a risk since there might also be an interesting split candidate with only 19 children in total. This change value is supposed to be adapted to the size of a software system.

Results with Moose:

The class *MSEAbstractObject* has been moved down in the class hierarchy between release 1.09 and 2.02. Class *MSEAbstractModelRoot* has been inserted as the new superclass. Class MSEAbstractObject has been split into two classes. Part of the behavior has been moved up in the new class *MSEAbstractModelRoot*.

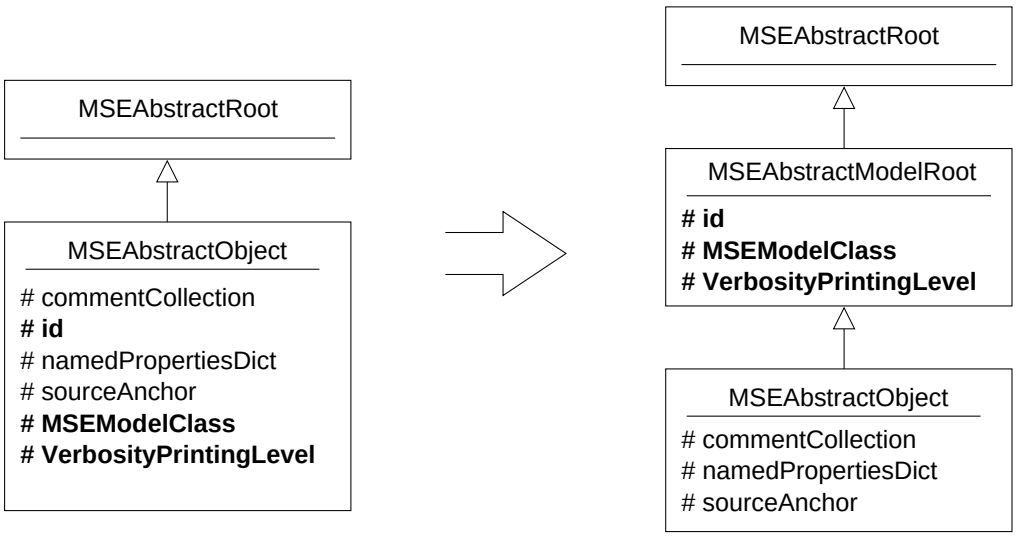

Figure 5.8: MSEAbstractObject gets split into two classes

5.7.7 Attribute Push Up Classes

Composition	Hierarchy
Information used	NIV
Model Scope	Two Models
Entity Scope	Class

Definition:

$$\forall c \in C_{new} \quad | \quad \delta NIV(c) < 0 \quad \wedge \quad \delta NIV(super(c)) > 0 \tag{5.14}$$

Idea:

If a variable for some reason is pushed up in its superclass, NIV decreases by one. NIV of the superclass instead increases by one at the same time. This combination of change is not likely to happen for another reason, therefore it is an indication for variable push-ups. The query detects candidate classes for pushed up attributes from release M_{old} to release M_{new}.

Evaluation:

The algorithm supposes to have at least one attribute removed from a class and added in the respective superclass of the next version. If between the versions other attributes were added to the class, $\delta NIV(x)$ may not be negative. This implies that the class is not be detected as push up candidate anymore.

Results with Moose:

Between Moose v2.02 and v2.35 we found class *MSESTAbstractParseTreeModelAnnotator* as candidate using the query for pushed up attributes. A verification shows that indeed two attributes have been pushed up to superclass *MSESTMetricParseTreeEnumerator*. This push up would have been hard to detect over entity names since the variables are renamed in the superclass.

We found another push up candidate between Moose v2.55 and v.3.31. Attribute *stream* has moved from class *MSECDIFSaver* to superclass *MSEAbstractSchemaSaver*. There were no false positives detected among the analyzed versions.

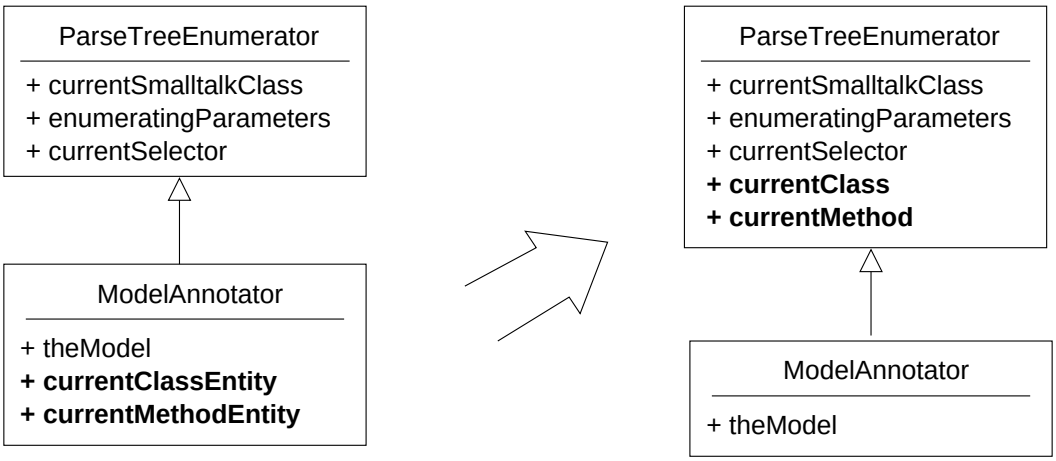

Figure 5.9: Two attributes are renamed and pushed up

5.7.8 Method Push Up Classes

Composition	Hierarchy
Information used	NOM
Model Scope	Two Models
Entity Scope	Class

Definition:

$$\forall c \in C_{new} \quad | \quad \delta NOM(c) < 0 \quad \wedge \quad \delta NOM(super(c)) > 0 \tag{5.15}$$

Idea:

In case methods are pushed up to a superclass between releases M_{old} and M_{new}, NOM of the receiving superclass increases. Simultaneously the class pushing up the methods looses them and has NOM decreased. The query checks changes in NOM for each input class and the respective superclass. If δNOM is negative for the analyzed class and positive for the superclass, the class is returned as a candidate class for pushed up methods.

Evaluation:

We detect pure method push-ups without problems with the above presented metric change conditions. However in case other changes are performed between versions at the same time, the query may fail. The NOM value is rather fragile, there are many more methods added between versions than attributes. Therefore to be sure whether the changes really originate from a push up it is necessary to compare the names of removed methods in the analyzed class with added methods in the superclass.

False Negatives: $\delta NOM(x) < 0$ may not hold in case that methods were added to the push-up candidate at the same time.

False Positives: There may have been some methods removed in the analyzed class and some methods added in the superclass. In such a case the class fulfills the conditions regarding changes in NOM even though no method has been pushed up.

Combinations:

If we compare the results of this query with the ones of query 5.7.10 (Moved Methods), we find out whether there were really some methods moved up to the superclass class. A combination of both queries provides more precise results.

Results with Moose:

We found a couple of false positives between various Moose versions. Usually we found a couple of removed deprecated methods, and at the same time some added methods in the superclass. Nevertheless we found a pushed up method between Moose v2.02 and v2.35: Method *import()* moved up from subclass *MSEVisualWorksParsingImporter* to class *MSEVisualWorksAbstractImporter*.

5.7.9 Moved Attributes

Composition	History
Model Scope	Two Models
Information used	name, unique name
Entity Scope	Attribute

Definition:

$$\begin{aligned} a_{new} \in A_{add} \quad | \quad \exists a_{old} \in A_{rem}, \quad attributename(a_{new}) = attributename(a_{old}) \\ A_{add} : \forall a \in A_{new} \quad | \quad name(a) \notin Names(A_{old}) \\ A_{rem} : \forall a \in A_{old} \quad | \quad name(a) \notin Names(A_{new}) \end{aligned} \tag{5.16}$$

Idea:

A *Moved Attributes* query helps us to find attributes that were moved from one class to another one. If an attribute is moved to another class, its unique name changes. The unique name [2] is a concatenation of class name and attribute name whereof the class name part changes. The attribute name (in contrast to the unique name) however does not change. We use the unique name as identifier of entities between versions. Since the unique name of a moved attribute changes, it seems for us as if the moved attribute is removed in the new version. At the same time it seems as if an attribute with the same name appears in another class. To detect moved attributes, we make use of the fact that the unique name changes, however the attribute name does not. We therefore extract two sets containing attributes: one with all attributes that disappear in the old version, and one with all added attributes in the new version. We compare these two sets and use the attribute name as identifier instead of the unique name as usual. Pairs with matching name are candidate moved attributes.

Evaluation:

We find all moved attributes that are not renamed at the same time. In case of a renaming, the attribute name would change as well as the unique name. If there are classes in the system that were renamed, their attributes are listed in the result as well. Attributes of renamed classes look as if they were moved from one class to another one. In systems with lots of renamed classes, a considerable part of the result are attributes of renamed classes. In a system without renamed classes, the resulting candidate moved attributes are push ups, push downs or another kind of moved attributes.

[2] attribute unique name: 'classname.attributename'

Variations:

We'd like to compare the set of moved attributes in both releases to see from which classes they were moved to which ones. However our query concept does not allow us to return both matching attributes from two different versions. It only allows a result collection to contain only entities of one single model. Therefore we need to choose either the new version or the old version as output model. To identify from where to where an attribute has been moved, we have to compare the output of both versions. Therefore we define two queries, one returning attributes of the new model and one returning attributes of the old model.

Combinations:

In case we are interested only in the classes the moved attributes belong to, we can extend the query by an additional conversion composite (see Section 4.3.3). The new query returns classes instead of the moved attributes.

Results with Moose:

We found many moved attributes between the different versions. In we list an informal classification of different reasons for the movement of attributes. The different reasons are mainly renaming of the class they belong to, attribute push ups and push downs. As we see in Table 5.12, many of the resulting attributes were not really moved, instead the class they belong to was renamed. We have always two classes defined in Moose for each class defined in Smalltalk code (instance and class side). If a Smalltalk class is renamed, both the instance side and the class sides get renamed. We therefore expect even numbers for detected renamed classes. However we have also uneven numbers in Table 5.12. If one of the class sides has no methods defined, we do not detect that class side as renamed. The numbers in brackets stand for false positives. False positives occur if two distinct attributes with the same name are defined in two different classes: one in a class that is removed and the other one in an class that is added between the versions.

We present as an example the moved attributes between Moose v2.55 and v3.31 more in detail. In total we found eleven moved attributes. We list the resulting attributes in Table 5.13 and additionally state for each attribute the reason of the movement.

Class *MooseLoader*, which is an implementation of a graphical user interface, has been renamed to *MSESmalltalkLoaderUI*, taking over 6 of 9 attributes. A part of the graphical user interface defined in class MooseLoader got separately defined in class

Release		Total	Class	Attribute pushed		misc.
old	new	moved	renamed	up	down	
1.01	1.09	2	1	-	-	(1)
1.09	2.02	6	-	5	-	(1)
2.02	2.35	14	4	-	-	10
2.35	2.55	29	28	-	1	-
2.55	3.31	11	7	2	1	1
3.31	3.49	6	6	-	-	-

Table 5.12: Moved attributes between subsequent Moose releases

Attribute	Belongs to Class	
Name	in Version 2.55	in Version 3.31
checkedIntermediates	MooseLoader	MSESmalltalkLoaderUI
cmbImportLevel	”	”
leftPanel	”	”
reificationLevels	”	”
selectedClasses	”	”
selectedClassList	”	”
fileName	MooseLoader	MSEFileLoaderUI
converter	MSECDIFSaver	MSEAbstractSchemaSaver
stream	”	”
name	MSEModelDescriptor	MSEModelAttributeDescriptor
schemaDictionary	MSESchemaSaveToStream	MSESchema

Table 5.13: Moved attributes, Moose v2.55 and v3.31

MSEFileLoaderUI in version 3.31. The attributes *converter* and *stream* of class *MSECDIFSaver* have been moved up to superclass *MSEAbstractSchemaSaver*. Class *MSEModelDescriptor* is removed in version 3.31, attribute *name* has been moved down to subclass *MSEModelAttributeDescriptor*. Class *MSESchemaSaveToStream* has been renamed to *MSESchema*.

The 10 moved attributes in column *miscellaneous* of Table 5.12 between Moose v2.02 and v2.35 result from the extracted class *MSEImportingContext* (see Figure 5.10). The importing context contains information about how to load a model. It has been extracted from the importer class in order to be able to instantiate a couple of default importing contexts without changing the importer.

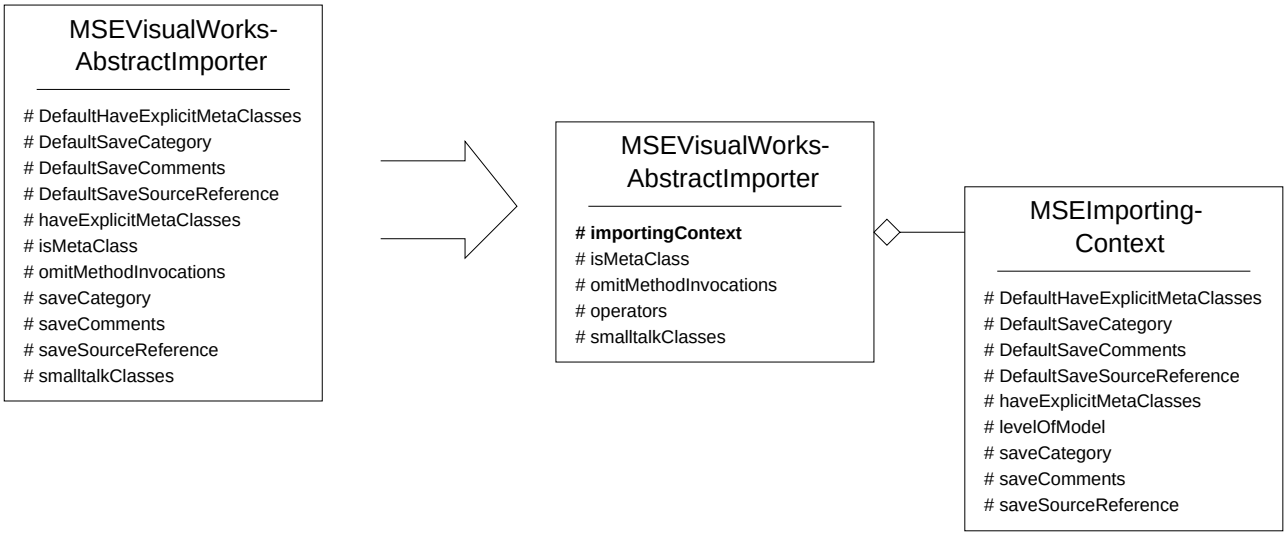

Figure 5.10: ImportingContext has been extracted from VisualWorksAbstractImporter

5.7.10 Moved Methods

Composition	History
Model Scope	Two Models
Information used	signature, unique name
Entity Scope	Method

Definition:

$$m_{new} \in M_{add} \quad | \quad \exists m_{old} \in M_{rem}, \quad signature(m_{new}) = signature(m_{old}) \tag{5.17}$$
$$M_{add} : \forall m \in M_{new} \quad | \quad name(m) \notin Names(M_{old})$$
$$M_{rem} : \forall m \in M_{old} \quad | \quad name(m) \notin Names(M_{new})$$

Idea:

A *Moved Method* query helps us to find methods that were moved from one class to another one. In case a method is moved to another class, its unique name changes. The method signature however does not. Since the unique name of a moved method changes, it seems for us as if the method is removed in the new version. It seems also as if a method with the same signature appears in another class. For the detection of moved methods, we make use of the fact that the unique name changes, however the method signature does not. We extract two sets containing methods: one with all methods that disappear in the old version, and one with all added methods in the new version. We compare these two sets and use the method signature as identifier instead of the unique name as usual. Pairs with matching signature are candidate moved methods.

Evaluation:

We find all moved methods that are not renamed at the same time. In case of a renaming, the method signature would change as well as the unique name. If there are classes in the system that were renamed, its methods are listed in the result as well. Methods of renamed classes look as if they were moved from one class to another one. In systems with lots of renamed classes, a considerable part of the result are methods of renamed classes. The query therefore also allows us to detect renamed classes. In a system without renamed classes, the resulting candidate moved methods are push ups, push downs or another kind of moving methods.

The *Moved Methods* query generally contains more noise than a *Moved Attributes* query 5.7.9. In Smalltalk code nearly every class implements a method called *initialize*. If

between the versions a class is removed and another one added, both implementing a method named initialize, the two methods are identified as a pair. They are therefore listed in the result even though there was no method moved between the two versions.

Variations:

We'd like to compare the set of moved methods in both releases to see from which classes the methods were moved to which ones. However our query concept does not allow us to return both matching methods from two different versions. It only allows a result collection to contain only entities of one single model. Therefore we need to choose either the new version, or the old version as output model. To identify from where to where a method has been moved, we have to compare the output of both versions. Therefore we define two queries, one returning entities of the new model and one returning entities of the old model.

Combinations:

The query can be combined with a *belongs to class* relation query to retrieve only the classes with moved methods. An intersection with all classes that have an increased NOM metric value reveals candidates that received the moved methods.

Results with Moose:

Since there are methods than attributes with the same name defined in several classes, the amount of false positives is bigger. Table 5.14 shows the amount of moved methods for all analyzed releases. The number of moved methods in the old version does not match with the number for the new version. These numbers should theoretically be identical since a moved method exists in both versions. The result therefore contains also noise. In the last two columns we state the amount of methods with the signature *initialize*. We see that for example between the first two versions that one method with signature *initialize* has been removed, and 16 classes implementing the same signature added. Only one of these 16 initialize methods is the moved method one.

Analyzing moved methods also helps to detect renamed classes. A method that belongs to a renamed class likely kept its name, but has a changed unique name since the according class name has changed. If a class is renamed, all its methods appear as alleged moved methods (it looks as if they were moved from the old class name to the new one). Classes with a big number of moved methods are potential candidates for renaming (Table 5.15).

Moose Release		Total Methods		*initialize*	
		old	new	old	new
1.01	1.09	5	21	1	16
1.09	2.02	133	127	16	7
2.02	2.35	103	104	21	11
2.35	2.55	230	236	15	14
2.55	3.31	78	75	6	25
3.31	3.49	49	50	13	15

Table 5.14: Moved methods between subsequent Moose releases

Moose Release		Number of Moved Methods			
		1	>1	>5	>10
1.01	1.09	-	1	-	-
1.09	2.02	5	3	-	3
2.02	2.35	18	8	1	1
2.35	2.55	10	14	6	9
2.55	3.31	27	10	-	1
3.31	3.49	9	2	3	1

Table 5.15: Number of classes containing a number of moved methods

5.7.11 Method Extracted

Composition	Affiliation
Model Scope	Two Models
Information used	NOM, NI
Entity Scope	Method

Definition:

$$m \in M \quad | \quad \delta NI(m) < 1 \quad \wedge \quad \delta NOM(class(m)) > 0 \tag{5.18}$$

Idea:

Too many invocations in one method indicate split candidates. A part of the algorithm defined in complex methods may be used for other methods as well. Furthermore methods containing lots of invocations are hard to understand. A *Split Methods* query tries to find methods that have been split. What happens if a method is split in two? First the number of invocations (NI) of the method decreases. The removed invocations need to be implemented somewhere else, usually in a newly created method of the same class. Therefore we expect that the number of methods (NOM), of the class the method belongs to, increases.

Evaluation:

There are usually a couple of classes with increased number of methods. Also methods with a more elegant implementation in the new version are frequently found. Therefore NOM and NI are both volatile metrics. As a consequence the query also returns some false positives. In case we get too many false positives as result, we can adapt the thresholds. However if we choose too restrictive thresholds, we risk to increase the number of false negatives.

Variations:

The missing invocations in the method may have been refactored and pushed up in a superclass. We could also check the condition NOM(super(class(x))) instead to see whether there were any methods extracted in the superclass.

Results with Moose:

A part of the functionality of method *preClass* in class *MSESTParseTreeBuildingEnumerator* has been extracted between Moose release 1.01 and 1.09. The extracted functionality is implemented in two methods *reifyClassAndSuperClass* and *reifyAttributes.* in release 1.09.

```
preClass
    "Create a MSE class if not already created, then create
     its superclass and the inheritance relationships between
     the two."

   super preClass.
   self reifyClassAndSuperClass.
   self reifyAttributes.
   selfVarDefinition := nil.
   superVarDefinition := nil
```

Astonishingly we only found a couple of false positives besides the above mentioned method. A lot of methods are usually extracted in the prototyping phase of a development cycle. The analyzed core of Moose is rather stable now. This may be a reason why we did not find method extractions. Frequently when we extract functionality from a method, we give the original method a new appropriate name. Unfortunately we don't find extracted renamed methods with our query. We have to combine the information about renamed classes with the information about extracted methods to tackle this problem. We are then able to track renamed methods beyond a renaming. We would also detect extracted parts of a renamed method.

5.8 Summary

In the previous sections we described our findings about MOOSE for each query separately. Here we summarize our results to show what we found out about the analyzed case study using our queries. Table 5.16 summarizes the findings of Table 5.10 and Table 5.12. We take into account that each Smalltalk class is stored twice in a MOOSE model (instance and class side) and count each class only once.

Release	Number of Classes			
	Total	Added	Removed	Renamed
1.01	91	-	-	
1.09	95	5	1	1
2.02	104	9	-	-
2.35	92	8	20	2
2.55	100	8	-	14
3.31	116	27	11	4
3.49	105	3	14	3

Table 5.16: Summary of the changes in Moose regarding classes

We see that MOOSE is a rather vivid system with changes between two releases up to about 25% of the total amount of classes. Compared to other case studies we detected many movements in the class hierarchy and many renamed classes. The changes in metric values (see Table 5.4) further suggest that there has been continuous refactoring applied on the source code. The developers seem to care about where they implement new functionality and to adapt existing parts if necessary. Moose has been implemented in VisualWorks Smalltalk. This software development environment supports well different refactorings through the use of the *Refactoring Browser* [ROBE 97].

Chapter 6

Towards a Methodology for an Evolution Analysis

6.1 Introduction

In Chapter 5 we presented a number of different queries, each one extracting entities with specific characteristics regarding change, dependency or other criteria. Based on these findings we try to define a methodology in order to retrieve essential information about an analyzed software system. We suggest which queries we best apply in which situation. We also suggest a certain order in which we best apply our queries. The idea is to apply more general queries first to get an overview of the system. Once we have determined interesting aspects to follow more in-depth, we apply more specific queries on particular subsystems to analyze them more in detail.

6.2 An Initial Methodology

We group our queries described in Chapter 5 into different categories. Each category of queries enables us to investigate a part of the code structure or to filter the model from irrelevant data. The functional categories are the following:

- **Filtering:** Filtering of Source Code Entities to Create a Clean Model
- **Change:** Calculation of Size Metrics and Change Metrics of the System
- **Subsystem:** Grouping of Entities into Subsystems, Subsystem Dependency Analysis
- **Hierarchy:** Detection of Changes in the Class Hierarchy

- **Move:** Moving of Features between Entities
- **Renaming:** Detection of Renaming

Table 6.1 shows an overview of all queries described in Chapter 5. There we classified the presented composite queries into different *query types* according to the composition mechanism used. Here we assign each query to one of the above described *functional* categories. These categories are targeted for practical use to retrieve information about specific criteria.

Query Name	Information Used	Scope	Category
Entity Name	name	Entities	Filtering
Metric Value	metric values	Entities	Filtering
Subsystem Affiliation	change metrics	Entities	Filtering
Metric Change	change metrics	Entities	Change
Added Entities	name	Entities	Change
Removed Entities	name	Entities	Change
Subsystem Invocations	invocation	Invocations	Subsystem
Subsystem Accesses	accesses	Accesses	Subsystem
Subsystem Inheritance	inheritance	Inheritance Definitions	Subsystem
Class Inserted	HNL, name	Classes	Hierarchy
Removed Superclass	HNL, name	Classes	Hierarchy
Subclass Becomes Sibling	HNL, NOC	Classes	Hierarchy
Sibling Becomes Subclass	HNL, NOC	Classes	Hierarchy
Heavy Change in Hierarchy	HNL, WNOC, name	Classes	Hierarchy
Attribute Push-up Classes	NIV	Classes	Move, Hierarchy
Method Push-up Classes	NIV	Classes	Move, Hierarchy
Moved Attributes	name	Attributes	Move, Renaming
Moved Methods	name	Methods	Move, Renaming
Method Extracted	NOM, NI	Methods	Move

Table 6.1: An overview of all presented queries in Chapter 5

Figure 6.1 suggests a procedure for an analysis of unknown case studies. We describe here some rationale for each category separately:

Filtering: We start on top with the category *Filtering*. We apply these queries first on a case study to remove unimportant source code information.

Change Overview: The queries in the next category *Change Overview* support us to get to know about overall changes between different source code releases. We

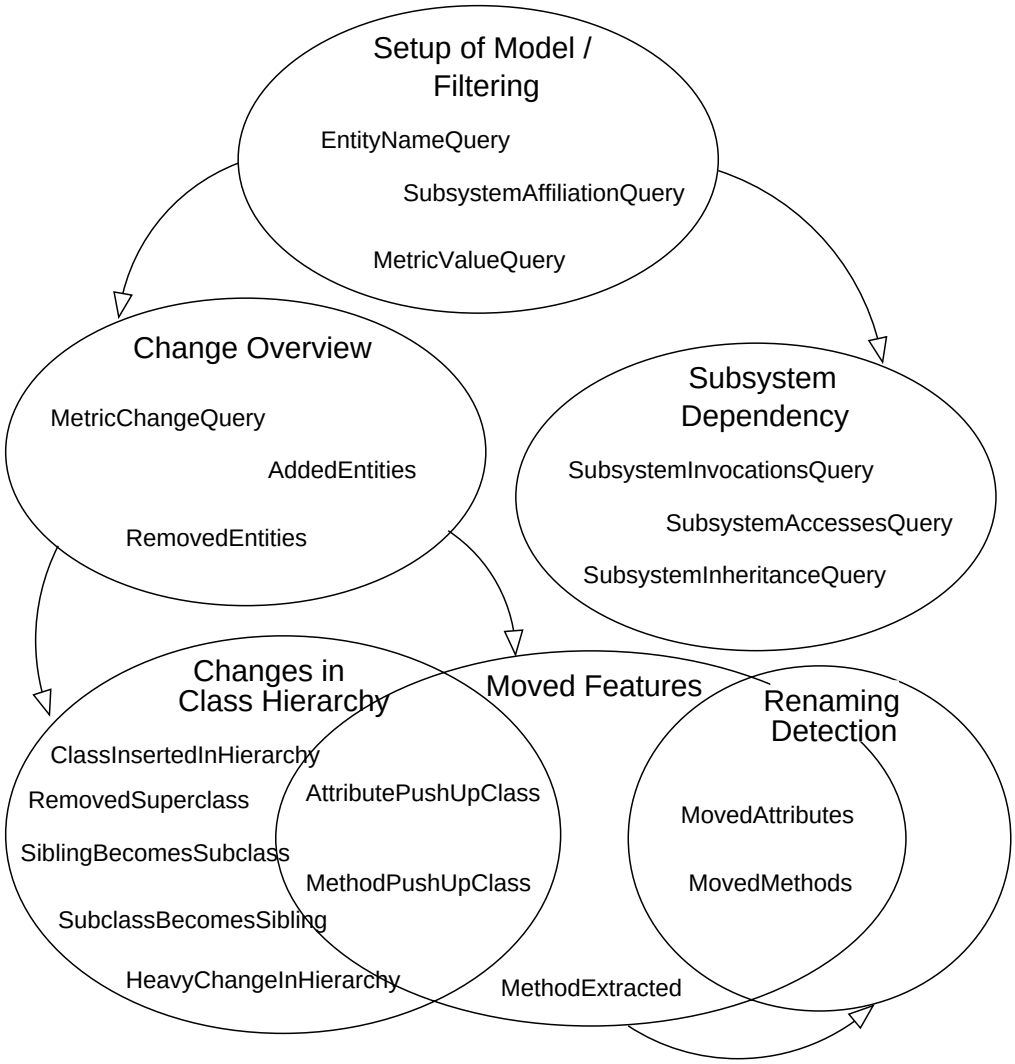

Figure 6.1: Towards a methodology to identify changes and dependencies

find out about additions and removals of entities, and about changes of persisting entities.

Subsystem Dependency: In a next step we can apply *Subsystem Dependency* queries on each of the releases separately. It is advisable to group the functionality of a large system into functionally related parts. This makes a system more understandable and encapsulates closely related functionality like for example I/O handling in subsystems. With our subsystem dependency queries we want to find out how the different parts interact with each other. We want to see whether the functionality of one subsystem is only accessible over a well defined interface to the outside, or if external classes have direct access to data defined in other subsystems.

Class Hierarchy: Once we know about overall changes in the history of a system, we may analyze changes in the class hierarchy more in detail. We search for classes which have been implemented quickly somewhere in the system, but would rather belong to another place. We find them in case they have been moved during redesign. We may also find classes that have been added to generalize common behavior that would otherwise be implemented twice.

Moved Features: We want to analyze movements of objects between classes or within the same class (method extraction). Such movements express a redistribution of responsibility. We want to find out where features turned out to be implemented at the wrong place, for example after an initial design has been extended. We may also detect moved features as a consequence of refactoring.

Renaming Detection: Since we heavily rely on the identification over unique entity names between versions, we are interested to see where such an identification fails. Therefore we need a to find renamed classes, methods and attributes. A *Renaming Detection* query finds candidate entities that have likely been renamed.

6.3 Conclusion

We applied the described methodology or parts of it on five case studies. The results of two case studies are presented in this work: the result of MOOSE for each query separately in Chapter 5, and the analysis of a large C++ system developed at Nokia Networks in Chapter 7. Additionally we analyzed the visualization tool CodeCrawler written in Smalltalk ($\sim$ 100 classes) [LANZ 99], a smaller C++ case study from industry ($\sim$ 150 classes) and the Java Swing framework. Presenting the results of all these case studies would go beyond the scope of this work. We rather evaluate for each category separately the results we were able to extract:

- **Filtering:** We use the filtering mainly for non-Smalltalk code. We cannot control the extracted data in external parsers and therefore need to filter it before we analyze the model. We observed drastic differences regarding system metrics (number of classes etc..) between filtered and unfiltered model information. We were able to exclude irrelevant stub classes by taking only classes containing at least one method, or by filtering according to naming conventions. We were able to select entities of desired subsystems for Smalltalk code as well as for non-Smalltalk code.
- **Change:** We got a good overview of additions and removals regarding entities for all analyzed case studies. We also got clear statements about changes regarding metric values. We especially focused on changes in HNL, NOM, NIV, NOC

and WNI. Since we did not encounter heavy renaming between two versions of the analyzed case studies, we got our results rather confirmed in a manual code analysis.

- **Subsystem:** We were able to detect dependencies between static information defined in the source code. The queries regarding inheritance and invocation worked well for all case studies. We were able to detect weird accesses, invocations and inheritance from classes defined in a framework to attributes defined in the source code of an application. We could not find accesses across subsystems in Smalltalk code since the language does not support direct access of attributes from outside a class.
- **Hierarchy:** In four of the five case studies we detected only few changes in the class hierarchy, still we found some in each analyzed case study. Since a change in the class hierarchy entails rather heavy restructuring in the code, we still regard the queries of this category to be useful. Moose contains the most interesting movements in the class hierarchy.
- **Move:** We analyzed moved methods and attributes in the two Smalltalk case studies only. We found out that the result contained many methods defined in renamed classes. This showed us that we need to split these queries, one extracting entities of renamed classes, and one extracting the actually moved entities.
- **Renaming:** We found an efficient way to detect renamed classes with the analysis of moved methods and attributes between releases. However we did not integrate the information about renamed classes into our change analysis. This would allow us to track changes of a class beyond the release where it got renamed.

Chapter 7

Experience and Validation in Industry

> *"We end up writing papers that are read by our fellow researchers but not many others. We also spend too little time finding out what practitioners know, think and need."* [PARN 94]

7.1 Introduction

During the now concluded FAMOOS project (see Section A.1) we initiated a collaboration on reengineering topics with partners in industry. This collaboration allowed us to validate and extend our reengineering platform *Moose* and tools based on Moose in a non-academic environment. Our research group has been able to analyze twice a software system during a one week workshop at Nokia, a leading telecommunications equipment manufacturer. During a seven month internship at Nokia in summer 2000, we were able to test and adapt the research tool for an analysis of code written in industry. The internship has been funded by the ESAPS project, a project within the Eureka $\Sigma!$ 2023 Programme (ITEA project 99005).

All results presented in this chapter are extracted from the same case study, a large embedded system written in C++ and partly in C ($\sim$ 600 KLOC). The system is a network node management software developed at Nokia Networks. The system consists of a whole family of software that manages the access between various wireless and cable networks. The software has been developed over several years. Customized versions of the system have been delivered to various network service providers in the telecommunication domain. We have extracted a core part called *Network Access Node* for an evolution analysis.

First we describe how we extracted the source code from the system and loaded the data for an analysis in our reengineering tool. In a second part we present an informal overview of the results we gathered while we analyzed the software. The results should

present what information we are able to extract from the source code and how the data can be interpreted. We did not intend to present an in-depth analysis of the software system.

7.2 From the Source Code to the Moose Model

7.2.1 Code Extraction and Metric Calculation

The source code of the Network Access Node software was stored in the version control tool ClearCase [RATI 00]. We extracted the information directly from the source code. First we parsed the C and C++ files using the software analysis tool Sniff [SNIF 00]. We extracted the information from Sniff using a tool that directly accesses the Sniff API and stores relevant entities information in the exchange format CDIF [DEME 99b]. Then we loaded the CDIF file containing the whole model information of one single release and stored it as a model in the Moose environment. Furthermore we ran several operators over the entities, each operator calculated some basic metrics of the entities. For a comparison between different versions it was necessary to load several models at the same time. We stored each version in a separate model.

7.2.2 Cleaning the Model

When information is extracted directly from the source code of a large framework, there are always a lot of details extracted that are not important for an analysis of the structure. Our parser of the code was rather tolerant, therefore we had as a consequence some data stored in the model that may be misinterpreted. Such noise in a model can significantly falsify metrics calculations. As an example, our parser interpreted each defined STRUCT as a class with attributes but no methods. Therefore we had a lot more classes stored in the model. A major part of the stored classes were just data structures which were not classes with responsibilities defined in methods and attributes. Using different queries we were able to detect such struct classes. These queries contained conditions about name conventions, metric values, source anchor etc.

7.2.3 Size Metrics on System Level

The queries defined in the tool MOOSEFINDER are helpful for counting entities satisfying specific conditions. A query always requires a set of meta objects as input and then returns a selected set of them as output. New metrics can easily be set up running

a query over one version, simply counting the number of output objects satisfying the query conditions.

7.2.4 Extracting Subsystem Information

The grouping of entities is not yet implemented in Moose. However each stored entity in the CDIF file has a source anchor attribute. The source anchor contains information about the path and the file where the entity is defined in the code. If classes are stored in different directories, one directory for each subsystem, then the subsystem information can be extracted from the source anchor. In the analyzed software the entities are stored in different subfolders according to the subsystems structure. Before running queries using subsystem information, a subsystem operator has to run over the model and assign each entity its according subsystem. The subsystem is currently stored as a property of the entity.

7.3 Results of the Code Analysis

7.3.1 How we apply query-based approach

An important issue of our query based-approach is scalability. It is helpful to follow a systematic sequence applying queries on a case study. We first apply a set of general queries on the code to get an overview of the system and the changes between versions. We collect the results for different releases and subsystems in a spreadsheet. The spreadsheet shows us which parts of the system have changed and what kind of changes have been performed. After we have identified some general types of changes, we apply more specific queries on the respective system parts. These queries reveal more precise information about detected changes in the code. We avoid analyzing more in detail subsystems that did not change. We just state the parts of the code that did not change during evolution. In case we detect subsystems with for example no changes in the class hierarchy, we do not further analyze and classify the movements in the class hierarchy.

7.3.2 System Level Metrics

Table 7.1 gives an overview of the system size and the changes between the subsequent versions. The system grows slightly in size from version to version. The time from the release of the first version to the last version is about 18 months.

Release	Number of Classes	Methods	Attributes	Invocations	Accesses
1	2305	24240	28237	55638	62703
2	2348	24936	28248	56587	64190
3	2475	26227	30183	61514	68969
4	2478	26306	30325	61707	69349
5	2742	29125	34683	69438	77670
6	2822	29650	37106	71067	83328

Table 7.1: Basic size metrics of the 6 extracted releases

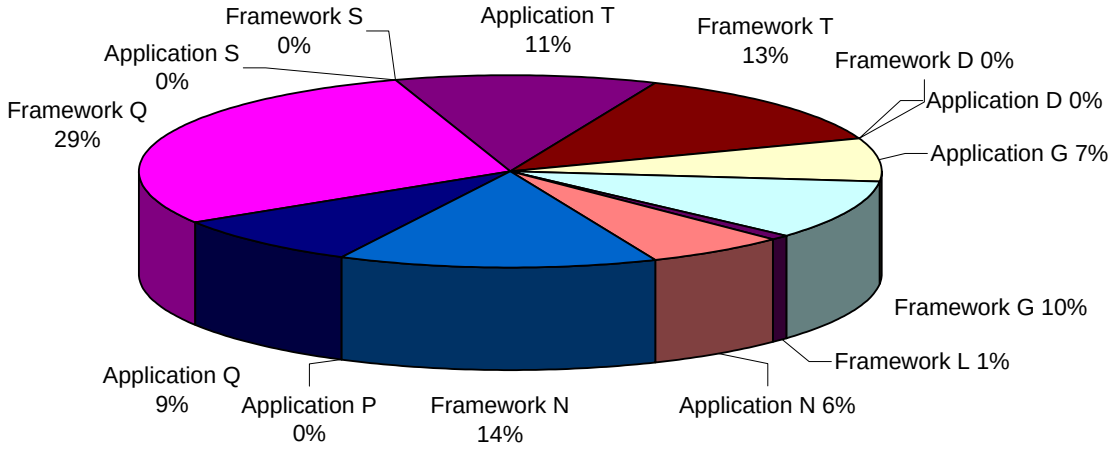

Figure 7.1: Relative size of the analyzed subsystems

7.3.3 Change Analysis between Versions

Changes in Class Names

We first analyzed the changes regarding the names of classes defined in each of the six analyzed versions. A new class name indicates that either the class has been added to the system or an existing class has been renamed. Figure 7.2 shows the changes regarding class names over the lifecycle of the system. Each pile is split in existing class names and added class names in the respective release. The remaining classes are at each case less than the previous total number of classes. This shows that between each release some classes have been removed. These numbers were calculated using queries matching a specific entity type in the same release and counting the output.

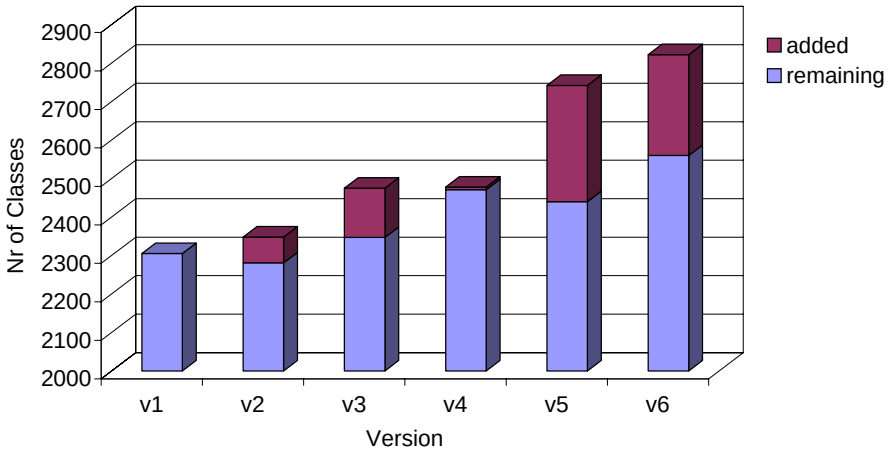

Figure 7.2: Changes regarding class names

Changes in Class Level Metrics

Comparing the metric values of the same entity defined in two subsequent versions shows where and what changes have been performed between two versions. Figure 7.3 shows the total number of classes in for each version and the amount of classes with changed NOM value. A change in NOM of a class shows where functionality has been added or removed. Towards version 5 and 6 the change rate has increased more than the total number of classes. This indicates that there has been more restructuring on method level than in earlier versions. Nevertheless a major part of the classes in the system has no methods added or removed, this assumes that the interface of the framework remains quite stable.

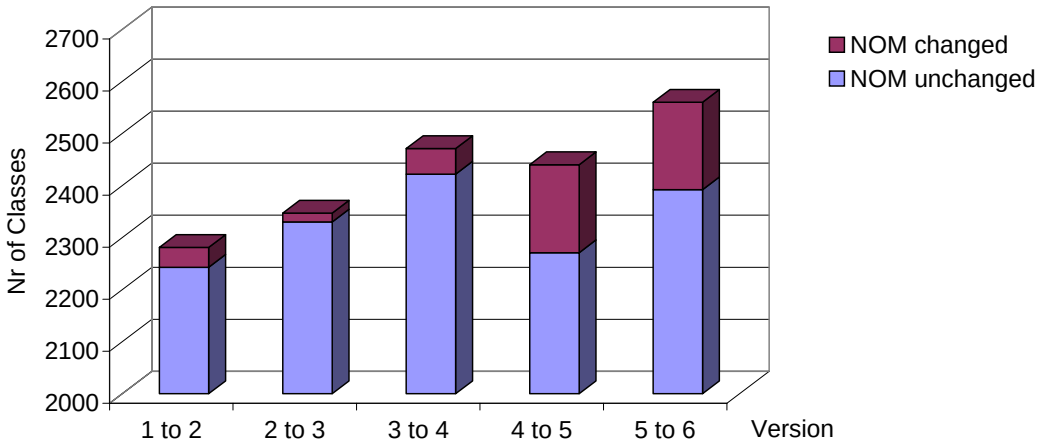

Figure 7.3: Changes in NOM among all classes

Versions	Number of Classes									
	Common Classes	HNL			NIV			NOM		
		$<$	=	$>$	$<$	=	$>$	$<$	=	$>$
V1 $->$ V2	2281	2	2271	8	6	2257	18	2	2243	36
V2 $->$ V3	2347	0	2347	0	4	2332	11	8	2330	9
V3 $->$ V4	2471	0	2470	1	5	2434	32	5	2422	44
V4 $->$ V5	2440	5	2425	10	38	2327	75	46	2271	123
V5 $->$ V6	2560	6	2552	2	33	2428	99	22	2392	146

Table 7.2: Changes in metric values between versions between V5 and V6

7.3.4 Subsystem Level Metrics

One goal when analyzing a system is to be able to detect hot spots in the system with significant changes. These hot spots should consist of a reasonable amount of code entities which can be further analyzed manually. The system may be so big that also hot spots contain too many entities to analyze all of them in depth. In such a case its easier to classify closer related entities into subsystems. Each subsystem may be a reusable component in other applications. In case we can reuse only one specific subsystem of the whole, we need to analyze a single subsystem only anyway. In the Network Access Node case study, each stored entity of the model has a source anchor attribute. This attribute contains the path and name of the file where the entity is defined. Classes that belong to the same subsystem are stored in the same subdirectory. That offers an easy way to determine subsystem affiliation for each entity.

Metrics on Class Level

The change of metric values for classes that persist over several versions indicates various changes between versions. Table 7.3 presents for each subsystem the change of the metric values *HNL*, *NOM* and *WNI* between the latest two releases. We see that a major part of the common classes do not have changed values. There are only few classes with increased *NOM* and even less having methods removed. We also see that there are hardly any changes in the hierarchy nesting level (HNL). Yet there are a couple of changes in the weighted number of invocations. WNI seems to be the most fragile metric value, indicating also slight changes in invocations inside a method. Based on these extracted change metrics we can conclude that the main changes between version 5 and 6 are some added classes (Figure 7.2) and adaptations in existing methods.

A significant amount of classes with changed HNL value indicates either that a whole

Subsystem	Number of Classes									
	Total	HNL			NOM			WNI		
		<	=	>	<	=	>	<	=	>
Application D	31	-	31	-	-	31	-	-	31	-
Framework D	1	-	1	-	-	1	-	-	1	-
Application G	269	-	269	-	-	261	8	16	241	12
Framework G	338	-	338	-	-	335	3	54	271	13
Framework L	28	-	28	-	-	27	1	3	23	2
Application N	132	-	132	-	7	102	23	25	72	35
Framework N	446	-	446	-	2	432	12	38	377	31
Application P	45	-	45	-	-	45	-	-	45	-
Application Q	128	6	122	-	4	115	9	30	86	12
Framework Q	836	-	834	2	3	787	46	168	618	50
Application S	5	-	5	-	-	5	-	-	5	-
Framework S	165	-	165	-	-	165	-	2	163	-
Application T	253	-	253	-	3	218	32	68	139	46
Framework T	316	-	316	-	3	301	12	37	254	25

Table 7.3: Change metrics for each subsystem separately

leaf has been moved with the root inheriting from a new class, or a class at a high level in the hierarchy has been inserted (HNL increased) or removed (HNL removed). We analyzed further the decrease in HNL of the 6 classes in application Q. The decrease mainly originates from a shift of a whole leaf containing one superclass and 4 inheriting classes. The superclass inherits in the new release from a stub class outside the scope of the analyzed subsystems 7.4. Classes *SUserPort*[1] and *VObserver* have a HNL value of 0 because their superclasses are not defined in the analyzed code.

Changes between Releases on Subsystem Level

The analysis of each individual subsystem separately gives more fine grained information about the evolution of the different parts of the software system. For an analysis of a single subsystem we only consider entities that belong to the chosen subsystem. We may detect rather stable, autonomous subsystems that do not change over several versions. Other subsystems may grow fast and change a lot between each release.

[1]The class names in the diagram were renamed, preserving part of the meaning they have in the software system.

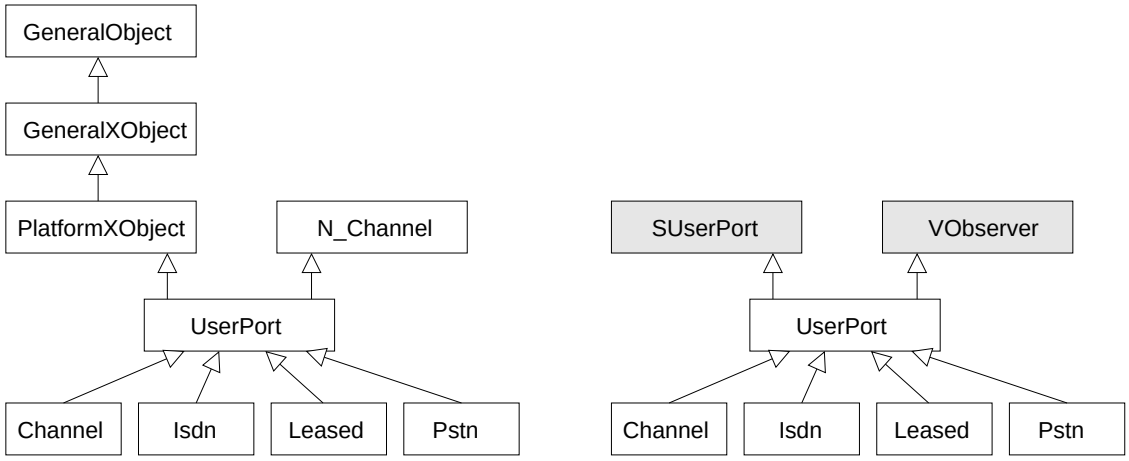

Figure 7.4: Changed class hierarchy from versions 5 to 6

Besides the growth and change rate of subsystems another interesting aspect is the coupling between different subsystems. Do they change synchronously because of close interaction and therefore change propagation? Figure 7.5 shows the changes in WNI of the application side of subsystem Q.

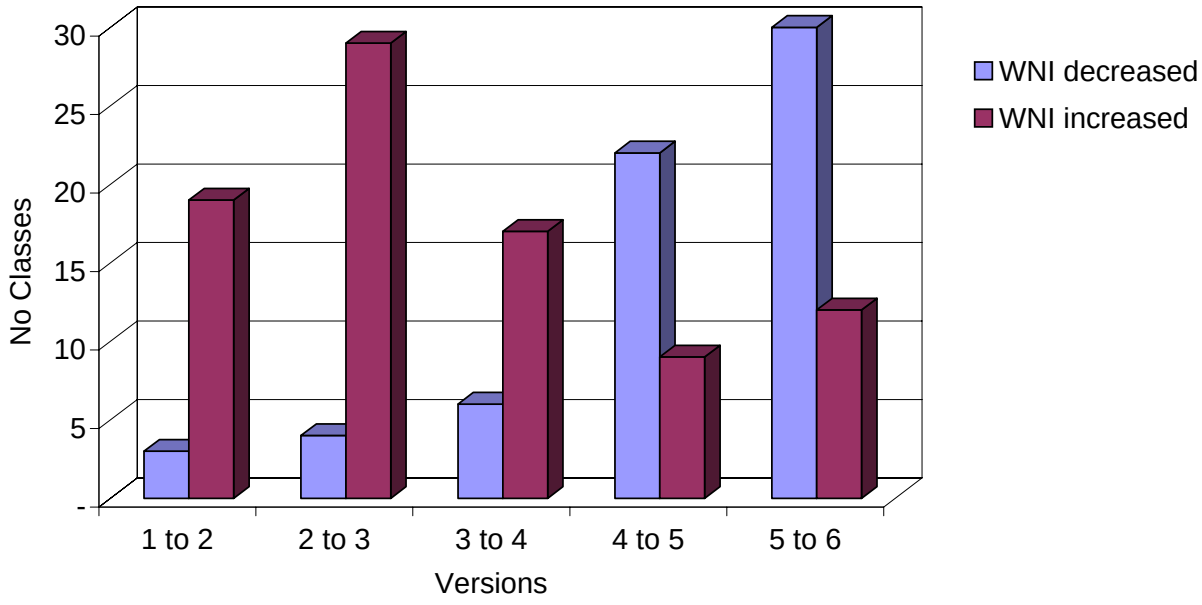

Figure 7.5: Changes in weighted number of invocations WNI

7.3.5 Subsystem Dependency

Besides changes in metrics of entities, dependency changes among subsystems are another interesting aspect in evolution of a software system. Subsystem dependency is interpreted here counting invocations, accesses and inheritance across subsystems. A high number of invocations and accesses between two subsystems means that they interact closely. In this case changes in one subsystem may affect also the other subsystem. If a clean initial design separates the subsystems rather strictly, evolution analysis may document that the code gets more dependent through quick adaptations, ignoring the concepts of a clean design. Inheritance from a class defined in another subsystem means that each time the subclass is changed, also the subsystem containing the superclass has to be tested and recompiled.

Invocations across Subsystems

Version 6 Invocations		Application						
		D	G	N	P	Q	S	T
Framework	D	-	-	-	-	-	-	-
	G	-	26	-	-	-	-	-
	L	-	2	-	-	-	-	-
	N	-	17	7	-	-	-	-
	Q	-	73	3	-	46	-	-
	S	-	2	-	-	-	-	-
	T	-	64	-	-	-	-	-

Table 7.4: Invocations from framework to application

Version 6 Invocations		Framework						
		D	G	L	N	Q	S	T
Application	D	-	-	3	-	-	-	-
	G	-	201	1	-	-	-	87
	N	-	147	3	428	15	-	407
	P	-	-	-	-	-	-	-
	Q	-	61	10	16	1805	-	-
	S	-	4	-	-	-	-	-
	T	-	57	12	-	-	-	1072

Table 7.5: Invocations from application to framework

An interesting dependency in the analyzed case study is the one from the application to the framework part. From an architectural point of view it would be preferable that the framework part does only invoke methods in the application part that have been implemented as hooks in the framework part already. In other words, the framework part should invoke in the application overridden methods defined in a framework class only. Only then the framework provides the common basic functionality for different applications built on top of it. Table 7.4 shows the result of queries counting the number of invocations across subsystems. Comparison over several versions indicates whether the subsystems get more dependent from each other when adding complexity.

There are very few invocations from the framework to the application compared with invocations from the application part to the framework part. However the invocations in the direction Framework -> Application are critical, especially those invoking between different subsystems. There are several framework methods invoking methods of the application G (General Services). We need to say here that we are not able to derive an invoked method precisely since we do not know exactly the type of an object due to polymorphism. Yet in case of invocations, a dependency problem remains anyway, since either classes of the framework part invoke or inherit from classes of the application part. Polymorphism is only possible up the inheritance branch.

Accesses across subsystems

Version 6 Accesses		Application D	G	N	P	Q	S	T
Framework	D	-	-	-	-	-	-	-
	G	2	347	-	-	-	-	20
	L	-	-	-	-	-	-	-
	N	-	-	109	-	-	-	-
	Q	-	-	-	-	102	-	-
	S	-	-	-	-	-	-	-
	T	-	-	-	-	-	-	116

Table 7.6: Accesses from framework to application

Accesses unlike invocations can be determined uniquely in a static analysis of C++ code. We do not have the information about the exact type of an object in our model. Protected and public attributes may belong to a variety of object types at runtime due to polymorphism. The possible types are reduced to the set of subclasses of the class the accessed attribute is defined in after all. We were astonished to detect various direct attribute accesses across subsystem boundaries. To be sure, we really checked a couple of these accesses manually and found our findings confirmed.

Version 6 Accesses		Framework D	G	L	N	Q	S	T
Application	D	-	-	-	-	-	-	-
	G	-	26	-	-	-	-	-
	N	-	-	-	-	-	-	-
	P	-	-	-	-	-	-	-
	Q	-	-	-	-	-	-	-
	S	-	-	-	-	-	-	-
	T	-	-	-	-	-	-	-

Table 7.7: Accesses from application to framework

Inheritance across Subsystems

Version 6 Inheritance		Application D	G	L	N	Q	S	T
Framework	D	-	-	-	-	-	-	-
	G	-	4	-	-	-	-	-
	L	-	-	-	-	-	-	-
	N	-	6	-	-	-	-	-
	Q	-	2	-	-	-	-	-
	S	-	1	-	-	-	-	-
	T	-	3	-	-	-	-	-

Table 7.8: Inheritance from framework to application

Inheritance relations state another kind of dependency between subsystems. Changes in a superclass may have an influence on the subclasses. If the implementation of an inherited method changes, it changes also for all subclasses that do not override the method. Also changes in protected or public attributes are propagated down the hierarchy tree. Inheritance between classes defined in different subsystems therefore makes the subsystems dependent from each other. In statically compiled languages, all dependent subsystems have to be recompiled after changes in one subsystem.

In a clean framework-application design classes defined in a framework should not inherit form classes defined in an application. Table 7.8 shows that classes defined in framework G inherits from a class defined in the application part G. We checked these classes manually in the source code and found out that several classes inherit from an obsolete class defined in the application subsystem G. We suggest to either remove the obsolete class, or to move it at leas in the framework part.

A closer analysis in CodeCrawler showed a rather heterogeneous hierarchy tree. Figure

Version 6 Inheritance		Framework D	G	L	N	Q	S	T
Application	D	-	-	-	-	-	-	-
	G	-	18	-	-	-	-	1
	N	-	-	-	51	2	-	18
	P	-	-	-	-	-	-	-
	Q	-	-	-	-	77	-	-
	S	-	-	-	-	-	-	-
	T	-	-	-	-	-	-	194

Table 7.9: Inheritance from application to framework

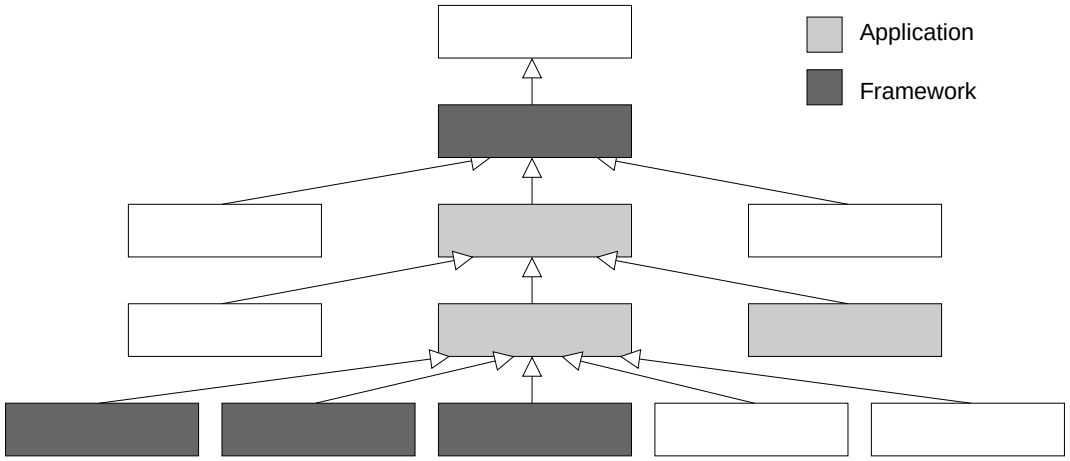

Figure 7.6: Heterogeneous class hierarchy

7.6 shows an abstract schema of the hierarchy tree structure. We see that in subsystem G a framework class inherits from classes defined the application. Further down in the hierarchy are again framework classes defined, inheriting from an application class. The uncolored classes are defined outside subsystem G.

7.4 Conclusion

We were able to detect different types of change such as additions or removals of entities. We also found out changes in metric values of persisting classes. These changes are not big compared to the total size of the system. There were never more than 15% of new classes added between two releases, and classes with changed number of methods (NOM) were generally less than 5% of the total amount of classes. We did not detect many movements in the class hierarchy and not many changes of NIV values

either. We therefore conclude that the main changes between the analyzed releases are classes added due to a system extension. The analyzed system presumably was pretty mature since the first analyzed version already. Yet the sum of the changes for the whole system equalizes the changes we detected in separate subsystems. An analysis of each subsystem separately revealed subsystems with quite substantial changes regarding size. Yet we did not find many changes in consequence of refactoring in separate subsystems either.

The analysis of subsystem dependencies revealed that the subsystems do not satisfy the criteria we would expect for object oriented code. The concept of encapsulation and hiding of internal data was not followed strictly. In contrast we even found classes defined in a framework part that were dependent from classes defined in the application.

Based on the changes of metric values we can assume that the developers did not refactor their code a lot. We detected hardly any changes in the class hierarchy (HNL values). We need to state that we did not extract moved methods and attributes for this case study in depth. We thought we would not detect many movements anyway. As a consequence we did not search for renamed classes either.

Summarized we can state that we were able to detect the changes we were looking for. We were able to assess the stability of different subsystems in terms of changes between releases. However some metric change queries returned too many entities to analyze all of them manually. We only checked whether metric values changed at all, and if they changed, whether they increased or decreased. We did not evaluate reasonable thresholds for the metric changes. The subsystem dependency queries generally returned too many invocations or accesses to browse them further manually. We used the results mainly to get to know between which subsystems there were dependencies at all, not for an analysis of each dependency separately. Up to now we need to combine the results manually to characterize a software system. We plan to investigate more combinations of the proposed queries to retrieve precise information more automatically.

7.5 Lessons Learned

Academic research is often accused of creating good ideas, but not considering the realization of the ideas in industry. We believe that our efforts finding new methodologies to improve code quality could be of great use in industry. We know that many software development teams in industry are too tightly focused just on achieving the next step in their project plan. They don’t have time to test new and maybe immature implementation techniques. Our aim is to develop techniques an tools that can be used in industry. This was the driving force for us not only to invest time in searching new methodologies, but

also to validate and adapt our own tools in an industrial context. We believe that this is essential especially in a reverse engineering context.

7.5.1 Our Tools

Our tools are prototypes, we change them constantly to implement new ideas. Yet the core functionality of MOOSE runs rather stable already. A couple of industrial software systems have been analyzed with MOOSE during the FAMOOS project. Nevertheless our tools are not used regularly in *"real world"* software development projects. We normally validate new functionality just on example case studies written in Smalltalk. Despite our aim to provide language independent analysis techniques, the adaptation for the analysis of C++ code in industry still consumed a considerable amount of time of this whole work. We list here a couple of problems we had to solve before we could carry out an effective code analysis:

- In our tool that converts C++ code into a FAMIX model we made the assumption that entity names would not exceed a certain length. The upper limit for the length of entity names was never reached before, probably because we just never analyzed such large systems before.
- In FAMIX it is not possible to have in the same model an attribute that has the same unique name as a class. The unique name of an attribute is a concatenation of the attribute name and the class name it belongs to, separated by a dot in between (classname.attributename). In our C++ case study we had *structs* that had the same name as attributes defined in the same model. There is no struct entity defined in FAMIX. Our C++ parser and conversion tool interpreted structs as classes. In MOOSE we rely on the unique entity names as a unique identifier of the stored objects. The fact that we had *two* entities with the *same* unique name in the model, and not even entities of the same type (attribute and class), confused our whole model concept. It was impossible to reference entities and calculate metrics or dependencies. Therefore we had to exclude the interfering structs from the model. It turned out that this was not a big loss for our analysis anyway.
- When we detected weird facts about the analyzed code, it was not always clear whether these facts really existed in the code. After checking the source code manually we sometimes found out that the mapping of the source was not correct. Some abnormalities in the code led us to a bug in the transformation from the source code into a model representation.

Such problems show how difficult the development of language independent reverse engineering tools can be. We really need to test all our tools constantly on software

written in different programming languages before we can claim to support language independent proceedings. The quality of the results heavily relies on the quality of the used parsers and source code conversion tools, in fact on the whole chain of tools our reverse engineering platform is based on.

7.5.2 The Case Study

Dealing with the amount of data of a huge black box framework is a challenging task. It is a delicate work to filter out unused information and to keep only meaningful material. We risk to count information that falsifies our results if we don't cut the noise. Yet if we cut too much information, we corrupt our results more than ever. There was some documentation about the system around, but it was not really useful for an understanding of the whole system. A manual code analysis was unimaginable because of the system's size, therefore we had to rely mainly on our tools.

After having created our first models we found all kind of classes with strange properties in that model. We first had to clean the model and decide which information to keep. For the cleaning we had to set up several criteria: Do we consider classes without any method definition? How do we exclude C++ structs that are not real classes in the object oriented sense? Is it reasonable to only consider classes stored in a file with a name matching the class? Although we had our reverse engineering tools available, we still had to make a couple of decisions for each case study manually. On the other hand, without our tools we would probably have been simply lost in the sheer amount of data.

For the analysis of our case studies we mainly used a PC with an Intel Pentium III running at 600MHz and 256MB of memory. Since so far we do not use an underlying database to store model information, Smalltalk loads the whole data in memory. As consequence we had to deal with hardware limitations when loading multiple large models. We were able to load two models easily without swapping memory on the hard disk, yet it was not possible to load all six analyzed versions at once without heavy swapping. Luckily for most of our change analysis we did not have to load more than two models at once. With two models loaded we were able to compare all possible combinations of models. We loaded two models, calculated changes, then purged one model, loaded a third one and compared those. To avoid losing time loading models we usually ran some scripts overnight which handled the loading of the models automatically.

7.5.3 The Developers

People conducting research on reverse engineering subjects have a difficult task to communicate their efforts to project managers and programmers. Developers generally agree that tools that support the understanding of source code are useful. However they

are sceptical whether it is worth investing time for learning and applying concrete new methods. Each programmer has his own technique to browse code and won't easily change it. The risk of just losing time for testing a new tools that might not be of much use after all keeps them working with familiar techniques. The programmers have a much more in depth knowledge of their own software systems. Reverse engineers don't want to present results that seem trivial to them. To get within their reach regarding code understanding means an investment of a considerable amount of time. Yet our task goes even beyond code understanding since we want to help them improving their code after all.

Project managers basically don't like critiques on their running code. They would like much more to have the quality of their code approved by experts. Such an approval helps them to sell their products, yet the knowledge about all kind of problems in the code simply means additional work (at first). Project managers also don't like to make available their source for a code analysis, unless they see a real advantage for the project. There is a risk that we might just spy their future products in development and make use of the gained knowledge.

It is difficult to explain unconventional research efforts and techniques to experienced developers in industry. They learned in their daily work the objectives for finding immediate solutions and for accelerating their implementation phases. They therefore have doubts about revolutionary approaches like for example extreme programming. Revolutionary approaches normally have also many previously undiscovered drawbacks when they get applied in practice.

Chapter 8

Conclusion

8.1 Summary

In the context of this work we investigated the benefits of historic change data for reverse engineering with our environment MOOSE [DUCA 00]. We extended MOOSE to be able to load the source code of several different versions of a software. We developed a query engine on top of the new multi-model MOOSE to extract changes between versions. We provided the query engine with composition facilities to query the system in a flexible way. The query engine got integrated in a research prototype named MOOSEFINDER. We put up a repository of queries, each one extracting a specific aspect of change. We proposed a methodology based on queries defined in MOOSEFINDER to

- provide techniques to clean a model from unnecessary information before we analyze it.
- compare different releases of the source code and detect added, removed and renamed entities.
- extract a number of refactorings performed on the source code such as changes in the class hierarchy, moved entities or renaming.
- locate different dependencies such as invocations, accesses and inheritance between subsystems of a software system.
- guess the behavior and skills of the developers, for example in which extent they do restructuring or apply refactoring techniques.

We validated our approach on five case studies: two written in C++, two in Smalltalk and the Java Swing framework. The results of two case studies we analyzed in depth are presented in this document: a large system written in C++ and our research tool

MOOSE written in Smalltalk. Our methodology worked well and helped us to find different aspects of change. The amount of added and removed entities states something about the current state of a software product in the development phase. There are generally more additions and removals of whole classes in early phases of development. Changes in entities that persist over several versions state whether developers apply refactoring techniques or just add and remove functionality. The change data helped us to make assumptions about the stability of different parts, and to some extent also about the quality of source code. We detected that the two Smalltalk case studies changed much more during their evolution compared to the analyzed software written in C++ or Java. The results confirmed our assumptions that refactoring is better supported and therefore more frequently applied in VisualWorks Smalltalk than in C++ development environments.

We do not provide a systematic interpretation of the different findings. The interpretation of our change facts still depends on each analyzed system. This is mainly due to the fact that our evolution queries do not all provide precise information. Especially queries based on changes in metric values return only candidates for a certain type of change. Yet a query output at least substantially narrows the amount of data we need to analyze further manually to sort out false positives. A main goal of our flexible query engine targets the quick adaptation to different systems. The query composition possibilities allow us to reduce the amount of false positives by switching multiple queries serially to filter noise.

We still need to combine the results of different queries manually to derive general statements abut a software system. In order to formulate *general statements* about the evolution of object oriented software systems, we have to analyze more case studies. Only then we find significant thresholds for metric change values and can measure the efficiency of different queries.

8.2 Main Contributions

The main contributions of this work are the following:

- **Scalability:** We established a query engine with composition functionality on top of our reverse engineering tool MOOSE. This enables us to interactively create and change queries. The ability to compose queries scales well to different case studies.
- **Historical Data:** We investigated the use of historical source code information for reverse engineering. We detected that focusing on changes only is an excellent way to reduce the amount of analyzed data and concentrate on relevant parts only.

- **Language Independence:** Our reverse engineering platform MOOSE allows us to analyze source code of different programming languages. In the context of this work we validated our approach on systems written in C++, Smalltalk and Java. Furthermore we were able to prove platform independence of our tools. We successfully analyzed systems on various operating systems (Windows, UNIX, MacOS).

- **Validation in Industry:** We were able to validate our tools and the proposed methodology on software systems developed in industry. We found out desired information about analyzed case studies and detected drawbacks we're going to work on.

8.3 Limitations of the Approach

We were able to extract interesting facts, yet we also discovered a couple limitations inherent to our approach. We list here three general limitations of our reverse engineering approach:

Static Information: Our approach bases solely on information extracted from source code. We therefore have only static information available for a code analysis. We lack information about the dynamic behavior of the analyzed system. Dynamic information would let us identify the invoked methods precisely. We would be able to eliminate uncertainty due to polymorphism. Dynamic information allows us to trace possible impacts of changes more precisely. The deeper we trace a change over invocations and accesses using static information only, the more blurry our results get. Despite the above mentioned advantages we still believe that static code analysis is superior to dynamic code analysis in many respects. Dynamic code analysis leads to a much larger amount of data we have to process. We also have problems to validate the full functionality of a system instead of assessing specific code sequences only.

Multiple Layers: We obtain our results at the end of a chain of conversions. First we need to have a good parser to extract information from underlying source code. Second we need a tool to convert the parsed code into a FAMIX meta model. Third we need a tool that stores the model information and allows us to query the stored data. In the end we apply our MOOSEFINDER tool to detect changes. The result is at best as good as the weakest part in the chain. For Smalltalk code we're able to do the whole conversion from the code parsing to the model creation in MOOSE, thus we are able to perform necessary corrections in MOOSE directly. For C++ code however we have to rely on a commercial product for the

source code parsing [SNIF 00]. Commercial tools don't allow a user to calibrate the source code conversion at will. We therefore intend to move to *open source* parsers where possible.

Difficulty of Portability: Our approach claims to be language independent based on the general FAMIX meta model. Language independent approaches always face a tradeoff between portability and specialization. We lose language specific information in case we consider only general observations. Yet we jeopardize language independence if we take into account language specific details. If for example we define an evolution query that assumes the language to be statically typed, we may run into problems applying the same query on dynamically typed Smalltalk code. From the analysis of source code written in different languages we learned that we cannot avoid adaptations to different languages. That underlines the importance of keeping the composition facilities for query conditions as flexible as possible.

8.4 Future Work

We plan to refine and expand our analysis methodology based on combinations of different query conditions to better classify different types of change. We will improve our query engine to provide more predefined composition possibilities. We also plan to validate our approach on more case studies to get to know which queries fit best to which type of system. We intend to refine our metric heuristics to reduce the amount of false positives in the output. We also plan to create sets of evolution queries that we apply automatically on an analyzed software system and store obtained results to a file.

So far our query-based approach provides only collections of entities as results. These collections need to be browsed further manually. A visualization of changes between versions would provide a clear overview of the system and promote a faster understanding of code structures. A graphical representation of the results would allow us also to study multiple types of change in parallel.

Appendix A

Moose

A.1 The Famix Meta Model

FAMIX has been introduced in the context of the FAMOOS research project. FAMOOS is an acronym for *Framework-based Approach for Mastering Object-Oriented Software Evolution*. FAMOOS has been a project in the context of ESPRIT, a R&D programme of the European Union on information technology. The three year project FAMOOS ended in September 1999. Six partners were involved in the project, among them the leading European companies Nokia and Daimler-Benz.

The FAMOOS partners have built a number of tool prototypes to support object oriented reengineering. These prototypes were validated during experiments on various case studies. The source code of the available case studies was written in different implementation languages (C++, Ada, Java and Smalltalk). To avoid equipping the tool prototypes with parsing technology for all those programming languages, a common information exchange model with language specific extensions was specified (see Figure A.1). This model has been named FAMIX (**FAM**OOS **I**nformation E**x**change Model).

The core model consists of the basic entities in object oriented languages, namely Class, Method, Attribute and InheritanceDefinition. For reengineering we additionally need to know about relations between the basic entities. Invocations and accesses provide information about such relations. An Invocation represents the definition of a method calling another method. An access represents a method accessing an attribute. These abstractions are needed for reengineering tasks such as dependency analysis, metrics computation and reengineering operations.

To satisfy the need for information exchange between tools, the CDIF standard was chosen in the FAMOOS project as the basis for transferring information. CDIF is an extensible format supported by industry standards. The plain text encoding facilities of CDIF have ben adopted to support information exchange between tools. The chosen

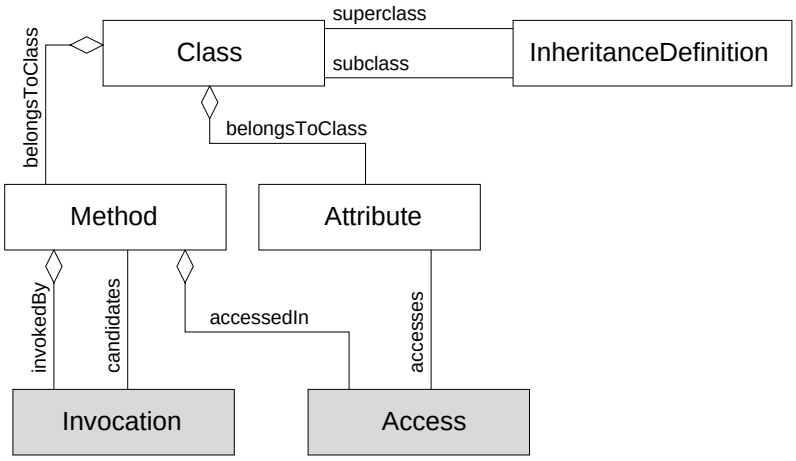

Figure A.1: Core of the FAMIX meta model

format is human readable and simple to process. The need for data exchange has increased rapidly in the last years through the wide use of the internet. XMI has been accepted in industry as a new standard for information exchange. We plan to shift from CDIF to XMI as exchange format to keep compatibility with industry standards.

A.2 The Structure of Moose

MOOSE is our reengineering research platform implemented in Smalltalk [DUCA 00]. It has been developed during the FAMOOS project to reverse engineer and re-engineer object-oriented systems. It consists of a repository to store models of source code. The models are stored based on the entities defined in FAMIX. The software analysis functionality of MOOSE is language independent. The FAMIX models can be loaded from and stored to files. Besides the repository there are other features implemented to support reverse engineering activities:

- a parser for Smalltalk code
- an interface to load and store information exchange files
- a software metrics calculation engine
- an interface for additional tools to browse and visualize stored entities

Figure A.2 shows the architecture of MOOSE. Various tools are implemented on top of MOOSE, using the interface to the above described repository functionality of MOOSE:

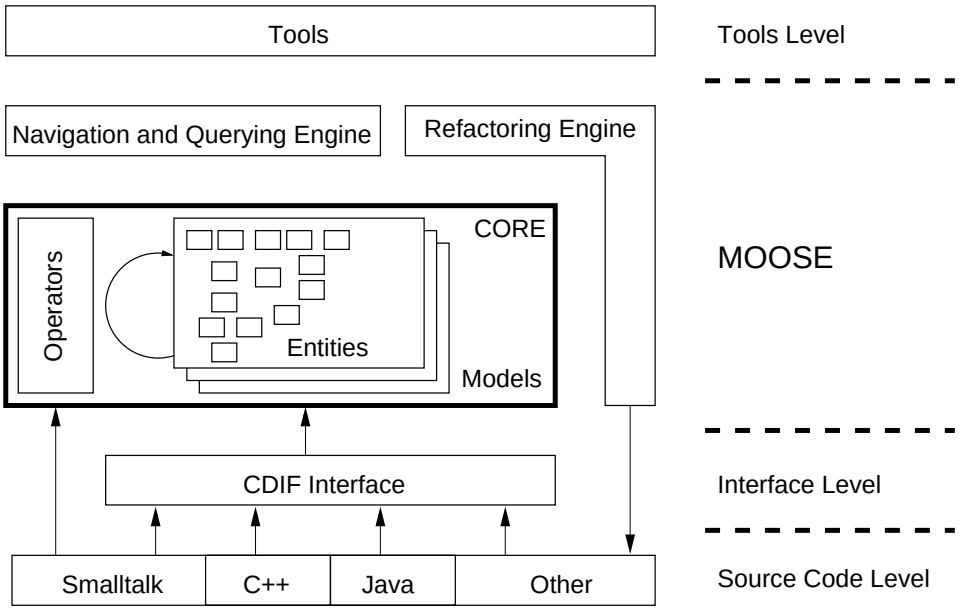

Figure A.2: The architecture of Moose

CodeCrawler: CodeCrawler is a visualization tool that supports different views on a model [LANZ 99]. The tool visualizes entities with shape and color according to metric values combined with different graph layouts. It enables a user to gain insights in large systems in a short time. Furthermore the graphs help to quickly identify source code entities with special combinations of metric values.

MooseExplorer: This tool [DUCA 00] provides a uniform way to represent model information. It addresses the problems of navigating large amounts of closely related information. MOOSEEXPLORER allows a user to browse different entity types in a consistent way. MOOSEEXPLORER shows for each entity its properties and related entities. A user can click through the entities and thereby further explore related entities.

MooseFinder: MOOSEFINDER is a query tool that helps to compose queries to retrieve source code entities matching special criteria [LANZ 01]. Such queries can also be defined on multiple models defining certain change criteria. This tool helped us to gain the evolution facts presented in this work. Appendix B provides a more detailed description of MOOSEFINDER.

A.3 Metrics defined in Moose

A.3.1 Class Metrics

In Table A.1 we list the currently defined class metrics in Moose. Classes are the core entities of every object oriented language. They provide implementations of methods and define attributes. Class metrics measure the complexity of classes and how they interact with other source code entities.

Name	Description
HNL	*Hierarchy nesting level*, also called *depth of inheritance tree*. The number of classes in superclass chain of class. In case of multiple inheritance, count the number of classes in the longest chain.
NA	*Number of accessors*, the number of get/set - methods in a class.
NAM	*Number of abstract methods*.
NC	*Number of constructors*.
NCV	*Number of class variables*.
NIA	*Number of inherited attributes*, the number of attributes defined in all superclasses of the subject class.
NIV	*Number of instance variables*.
NMA	*Number of methods added*, the number of methods defined in the subject class but not in its superclass.
NME	*Number of methods extended*, the number of methods redefined in subject class by invoking the same method on a superclass.
NMI	*Number of methods inherited*, i.e. defined in superclass and inherited unmodified.
NMO	*Number of methods overridden*, i.e. redefined in subject class.
NOC	*Number of immediate children of a class*.
NOM	*Number of methods*, each method counts as 1. NOM = NMA + NME + NMO.
NOMP	*Number of method protocols*. This is Smalltalk - specific: methods can be grouped into method protocols.
PriA	*Number of private attributes*.
PriM	*Number of private methods*.
ProA	*Number of protected attributes.*
ProM	*Number of protected methods*.
PubA	*Number of public attributes*.
PubM	*Number of public methods*.
WLOC	*Lines of code*, sum of all lines of code in all method bodies of the class.
WMSG	*Number of message sends*, sum of number of message sends in all method bodies of class.
WMCX	*Sum of method complexities*.
WNAA	*Number of times all attributes defined in the class are accessed*.
WNI	*Number of method invocations for a class*, i.e. the sum of the invocations of all methods defined in a class
WNMAA	*Number of all accesses on attributes*.
WNOC	*Number of all descendants*, i.e. sum of all direct and indirect children of a class.
WNOS	*Number of statements*, sum of statements in all method bodies of class.

Table A.1: Additional class metrics defined in Moose

A.3.2 Method Metrics

In Table A.2 we list every method metric currently defined in Moose. Methods can be seen as a flow of instructions which take input through parameters and which produce output. Methods can invoke other methods or access attributes. The method metrics are defined in this context.

Name	Description
LOC	*Lines of code* in method body.
MHNL	*Hierarchy nesting level* of class in which method is implemented.
MSG	*Number of message sends* in method body.
NI	*Number of invocations* of other methods in method body.
NMAA	*Number of accesses on attributes* in method body.
NOP	*Number of parameters* which the method takes.
NOS	*Number of statements* in method body.
NTIG	*Number of times invoked by methods non-local to its class*, i.e. from methods implemented in other classes.
NTIL	*Number of times invoked by methods local to its class*, i.e. from methods implemented in the same class.

Table A.2: The method metrics defined in Moose

A.3.3 Attribute Metrics

In Table A.3 we list every attribute metric currently defined in Moose. Attributes represent properties of classes. Their main function is to return their value when accessed by methods. The attribute metrics are defined in this context.

Name	Description
AHNL	*Hierarchy nesting level* of class in which attribute is defined.
NAA	*Number of times accessed.* NAA = NGA + NLA.
NCM	*Number of classes having methods that access it.*
NGA	*Number of times accessed by methods non-local* to its class.
NLA	*Number of times accessed by methods local* to its class.
NM	*Number of methods accessing it.*

Table A.3: The attribute metrics defined in Moose

Appendix B

Moose Finder

B.1 Introduction

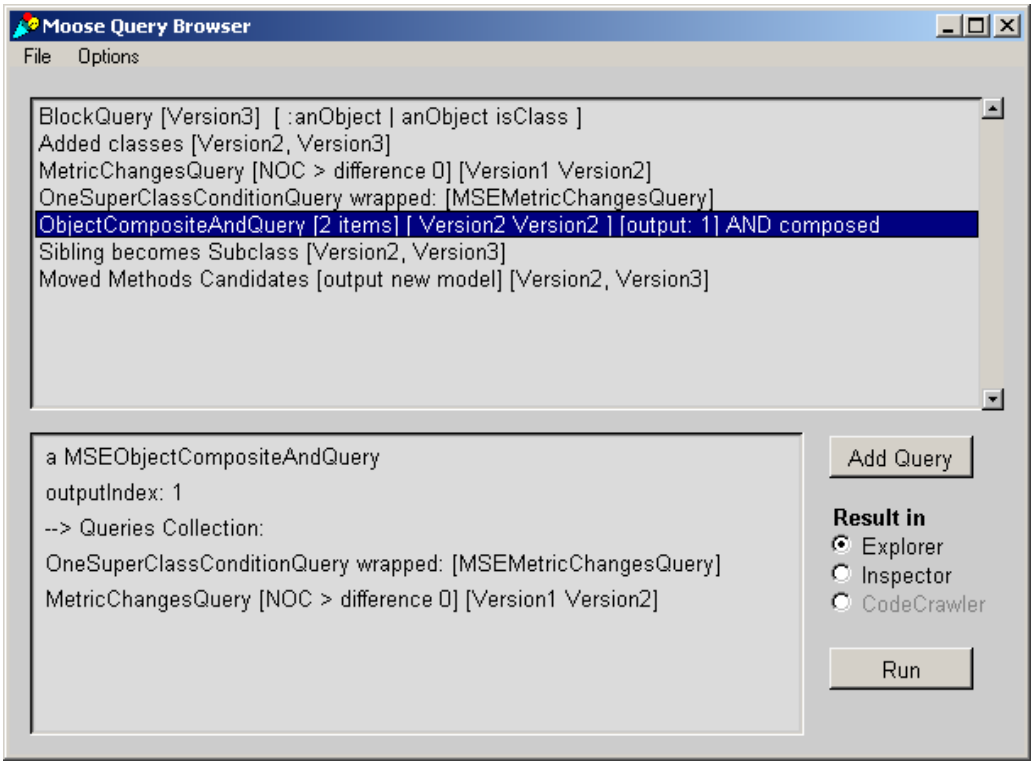

Figure B.1: MooseFinder main window containing the query list

MOOSEFINDER is the query tool used for the evolution analysis described in this work (see also [LANZ 01]). It has been implemented to validate the presented ideas. MOOSEFINDER is built on top of MOOSE, a reengineering tool described in Appendix A.

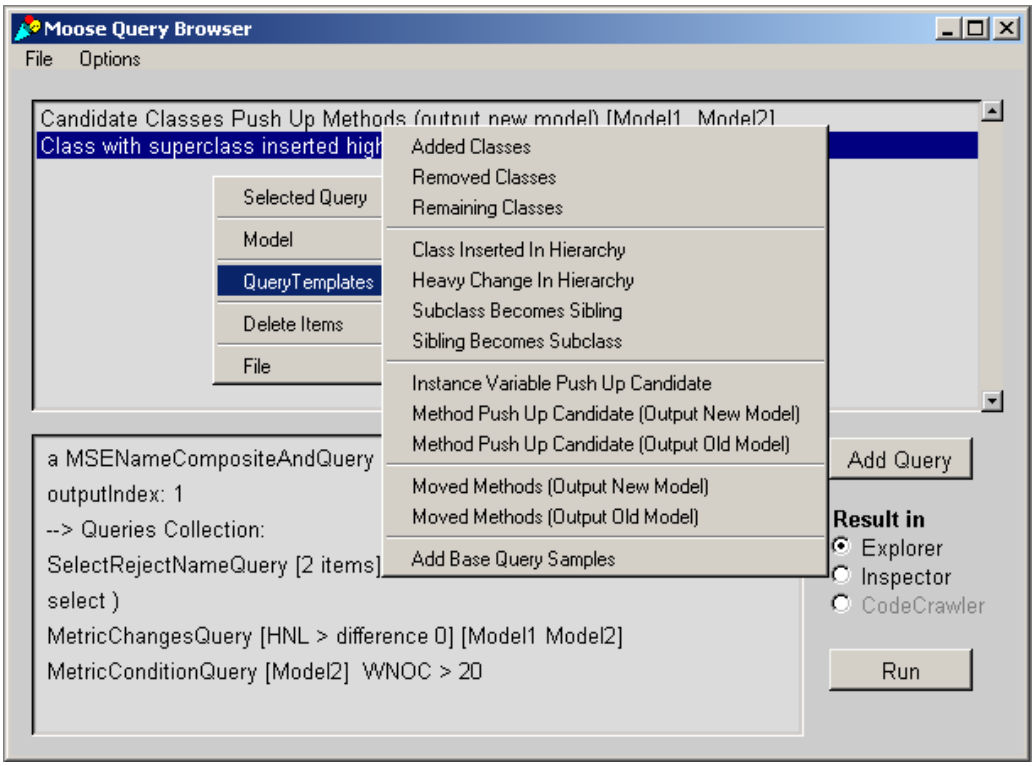

Figure B.2: Template Queries in the popup menu

Both programs are implemented in VisualWorks Smalltalk 3.0. MOOSE serves as the database to store models of source code. Queries composed in MOOSEFINDER then run on these models. The first description contains a section about the GUI of MOOSEFINDER, then follows a section where the query composition mechanism is described. The common API of all queries is presented at the end of this chapter.

B.2 How to use MooseFinder

The main window of MOOSEFINDER contains a list with the currently loaded queries. These queries can be applied on a set of entities by pressing the *run* button. The query is applied on entities of the default model defined in the query. The query list contains a basic description of each query. Below in a text field a more detailed description of a query is shown if for the query that is selected. A popup window in the query list offers several manipulations on the query list and selected queries (see Figure B.2):

- **Query Manipulations:** We can edit a selected query to view and change its attributes. We can also duplicate the query, change the model a query is defined on

or add a description for the query. We can also inspect the actual query instance in a Smalltalk inspector window.

- **List Manipulation:** We can delete items from the list of queries or just delete the whole list. We can change the order in which the queries in the list appear using drag and drop.

- **Template Queries:** We can load a predefined template query from the popup menu (Figure B.2). All template queries have default models assigned and therefore need to have the right models assigned first before we can apply them on loaded code.

- **Input/Output:** We can file out a selected query or a list of queries. In such a case, all attributes of a query are stored in an ASCII file. The queries, once stored in a file, can always be reloaded in the query list.

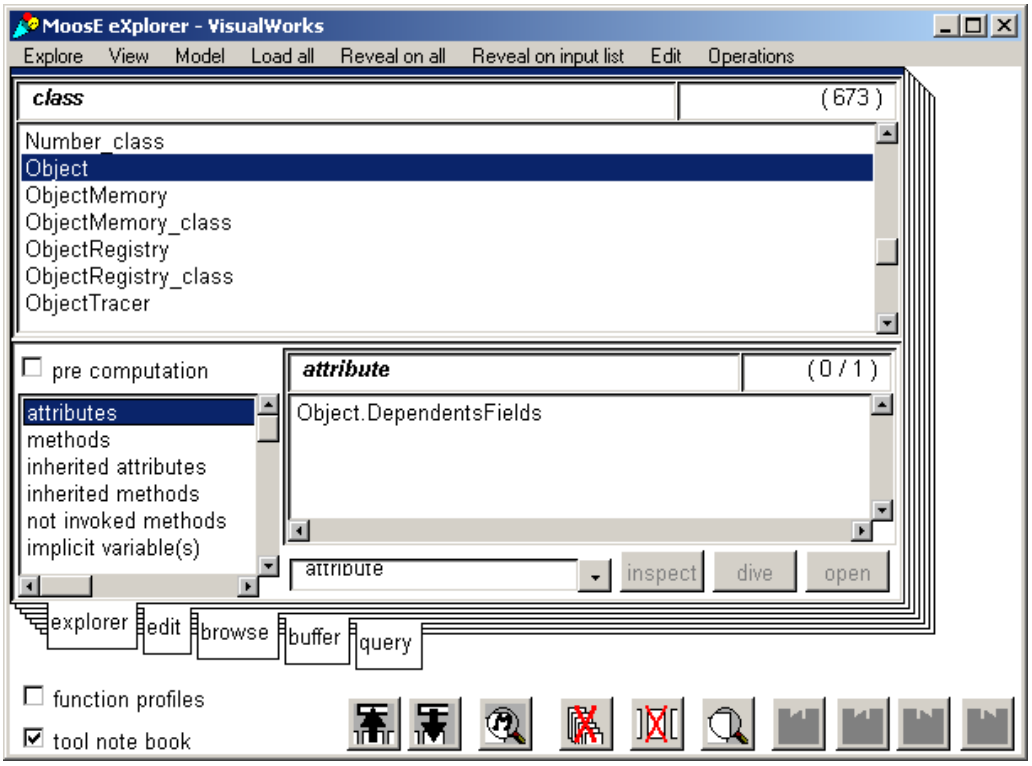

Figure B.3: Moose Explorer showing some loaded VisualWorks core classes

Each query returns a collection of source code entities. We can choose to which tool this collection is passed to show the output. The default tool is MOOSEEXPLORER, a navigation tool to browse the models loaded in MOOSE (Figure B.3). MOOSEEXPLORER

shows the basic properties of entities and additionally allows a user to apply different views on a list of entities. An instance of MOOSEFINDER is always integrated in MOOSEEXPLORER as a separate panel. This allows us to use MOOSEFINDER also in MOOSEEXPLORER and to define or load queries from there directly. MOOSEEXPLORER allows a user to store different sets of entities in a buffer. Thanks to the buffering it is possible to run queries not only on a default collection of entities, but also on a buffered collection of entities. This allows us to apply queries serially, each query then gets the output of the previous query as input.

The output collection of entities can be passed to any other application. The output collection can also be passed to the Smalltalk inspector to analyze and manipulate the actual instances of the resulting entities. Another option allows the user to pass output entities directly to the visualization tool CodeCrawler.

MOOSEFINDER has a query composer window integrated. The composer user interface helps a user to create new queries and to compose complex queries using the queries defined in the list. The query composition window consists of several subpanels, each one covers the configuration of a special type of query. We quickly explain each of the different subpanels:

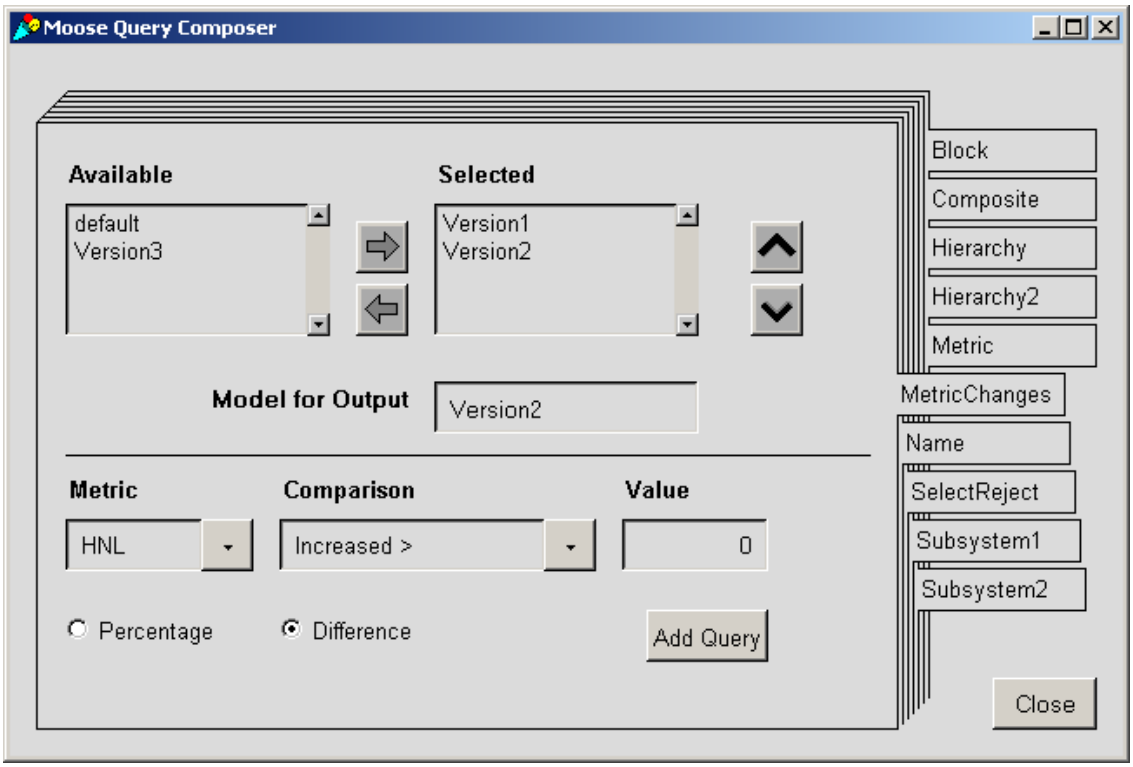

Figure B.4: The Query Composition Window

- **Block:** The block panel allows a user to define a query that contains a condition expressed in a Smalltalk block. This block is passed to each input entity of the query. All entities satisfying the block condition are collected for the output. A block query should be used with care since it may happen that not all input entities understand the messages defined in the block.

- **Composite:** In the composite panel we can compose more complex queries of existing ones defined in the query list. We therefore drag queries from the query list and drop them in the panel's composite list of the new composite query. The user can choose several composition options. In case we want the entities to fulfill all conditions defined in the subqueries, we choose AND composition. In case the output entities need to fulfill at least one of the conditions defined in subqueries, we choose OR composition. We can also specify in the panel how to pass entities from one subquery to the next one. A *NameCompositeQuery* identifies the entities over their unique. We can therefore have subqueries defined on different models in the same composite query. *ObjectCompositeQueries* just pass the resulting output entities of one query as input of the next one.

- **Hierarchy:** The hierarchy panel provides an interface to compose queries with constraints on entities that are related over their class hierarchy tree. Hierarchy queries contain other queries and delegate the condition to superclasses or subclasses of the actual input entities. The hierarchy interface also allows a user to compose queries that return all subclasses of a set of classes defined by the output of a subquery.

- **Metric:** The metric panel allows a user to define conditions on metric values. Either a single metric value is checked against a threshold or a set of metric conditions can be consolidated in one single query. A metric condition is understood as the comparison of a metric value with a threshold.

- **Metric Change:** The metric changes panel helps a user to define a query containing a condition about the change of a specific metric between several versions. The desired models can be chosen among the defined ones in the model.

- **Name:** The name panel provides an interface to define a query with a condition on entity names. The user can choose whether a string pattern should also match the case and if name or unique name are compared.

- **SelectReject:** The select/reject panel allows a user to compose a query using set operations. In all cases a query defined as base provides a basic set of entities. Entities that satisfy all conditions of the queries dragged in the *select* list are kept in the basic set of entities. Entities that do not satisfy all conditions are rejected from the basic set. All entities that satisfy a condition of a query dragged in the *reject*

list are rejected from the basic set of entities. The remaining set of basic entities is then returned as output. If the subqueries are defined on different models, entities are identified over their unique name.

- **Subsystem:** The subsystem panel helps a user to create queries with conditions on subsystem affiliation. Queries to select only entities defined in a single subsystem or a set of subsystems can be composed here. Also queries that select invocations, accesses or inheritance definitions between two defined subsystems can be instantiated over this panel.

We can edit a string representation of a query and in an editor window (Figure B.5). The editor window shows the same string representation created for storing a query in a file. Subqueries are separated by a <SubQuery> tag from the attributes of the main query. We introduced identification numbers for a correct referencing among the subqueries. The id's of the subqueries of a composite query are listed in *subQueryIdList:*.

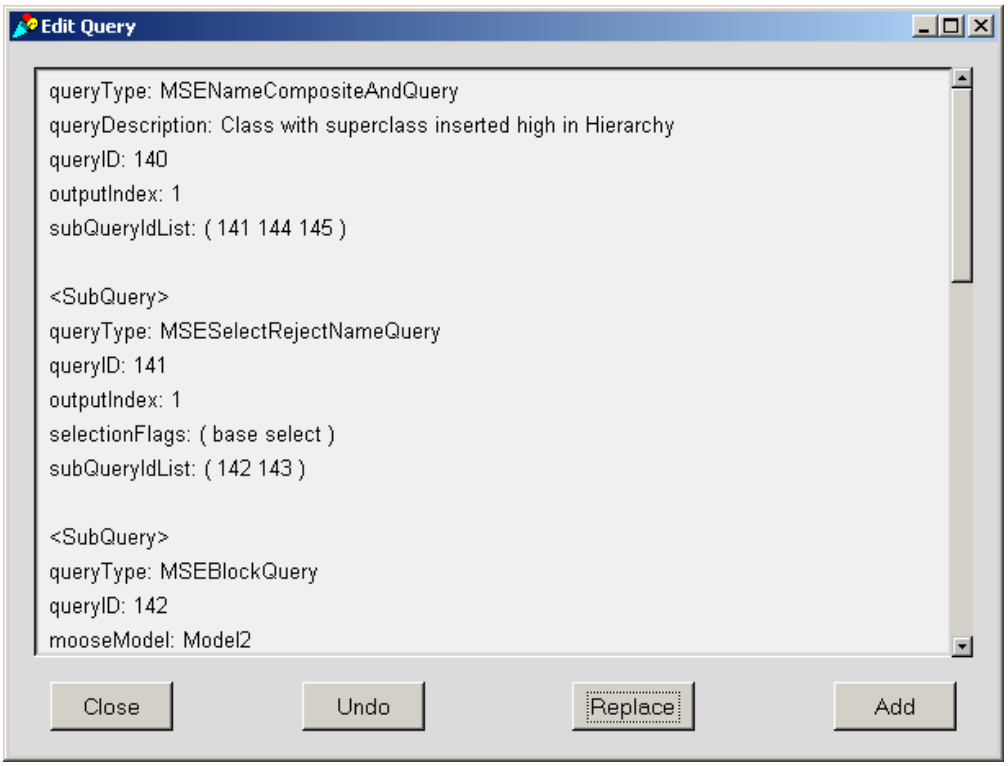

Figure B.5: The Query Editor Window

B.3 Implementation of the Queries

B.3.1 Conceptual issues

We built the all queries on the concept of a composite pattern described in Section 4.3. The *Basic Queries* represent the leaves that contain only a condition to filter source code entities. The *Composite Queries* represent branches in the composite pattern. They do not contain a query condition directly, though they contain subclasses containing conditions. Figure B.6 shows the implemented class hierarchy of the queries.

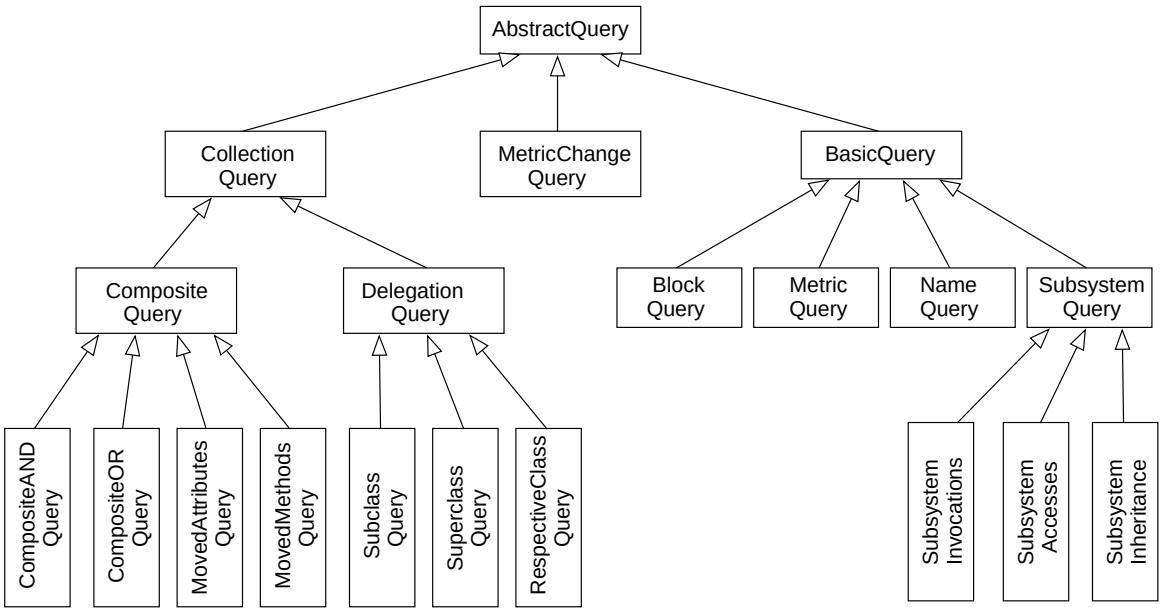

Figure B.6: Class hierarchy of the defined queries

B.3.2 Implementation

Each query has three distinct kinds of responsibility. These responsibilities must be implemented or inherited for each defined query.

Condition: Since each query operates as a filter the query has a special condition defined. implementation of the condition runOn: aCollection and fulfills: anObject

Load/Store: A query needs to know how to store its attributes as a string. Each query also has the responsibility to know which of its attributes need to be assigned in order to know create a running query instance. The reading of a stored query from an ASCII file is delegated to an I/O Handler. We chose a format based on XMI to

store the queries because it is human readable and supported by various open source parsers.

Representation: Since we want to list and browse query instances in a graphical user interface, we need different string representations for each query: a string that fits on one line to represent the query in a list widget; a compact representation in a couple of lines to display the most relevant attributes in a textfield widget; a full string representation to load and store a query, and to edit all its attributes.

B.3.3 The Common Query API

This section lists the common application programming interface for all predefined queries. The common API of all queries provides the key structure for a composition of complex queries in a flexible way. The fact that all queries know how to handle a defined set of messages ensures the flexibility in composing complex queries. The subqueries will handle invocation messages correctly as long as the communication between the main query its subqueries bases on the common API,

runOn: aCollectionOfEntities expects a set of entities as input and returns a subset of the input entities that satisfy the condition defined in the query.

defaultInputObjects returns all default entities the query runs on. If a query expects classes as input, *defaultInputObjects* of that query returns all classes of the model the query is defined on.

runOnDefault gets as input the entities returned by *defaultInputObjects*, runs on the input entities and returns all entities that satisfy the query condition.

```
runOnDefault
^self runOn: self defaultInputObjects
```

setMooseModel: aModelname sets the a new model the query runs on. This method is mainly used for basic queries that are defined on one single model only. If a query is defined on more than one model, the model defined as output model is changed.

outputModelName returns model name of the output entities. This information may be needed because the collection of output entities does not provide information about model affiliation of the entities. For further navigation in the correct model, MOOSEEXPLORER needs to know to which model the output entities belong to.

listOfModelsUsed returns all models defined in the query or in any subquery. We need this method to get to know on how many different models especially a composite query is defined.

replaceCurrentModels: aListOfCurrentModels with: aListOfNewModels replaces the models a query is defined on by other ones. The first model of list *aListOfCurrentModels* is replaced by the first element of *aListOfNewModels*. The same replacement applies for the rest of the elements. Therefore both lists need to contain the same amount of model names.

storeInstance returns a string representation of the query containing a list of all current attribute values.

displayProperties returns a string listing all attribute values of a query. For a complex query also the types of assigned subqueries are listed. The information is more compact than in the string returned by *storeInstance*. It is for a presentation of the query attributes in a textfield widget.

compactStringRepresentation returns a short string with the most important information about a query instance. The information is kept on one line and is meant to be displayed in a list widget.

requiredKeys returns for each query type a set of keywords for attributes that need to be defined to create the query.

Bibliography

[BECK 99] K. Beck. Kent Beck's Guide to Better Smalltalk. Sigs Books, 1999.

[BROO 75] F. P. Brooks. The Mythical Man-Month. Addison-Wesley, Reading, Mass, 1975.

[BURD 98] E. Burd and M. Munro. *Investigating Component-Based Maintenance and the Effect of Software Evolution: A Reengineering Approach Using Data Clustering*. In Proceedings of the International Conference on Software Maintenance (ICSM'98), 1998.

[BURD 99] E. Burd and M. Munro. *An Initial Approach towards Measuring and Characterizing Software Evolution*. In Proceedings of the Working Conference on Reverse Engineering (WCRE'99), pages 168–174, 1999.

[BURD 00] E. Burd and M. Munro. *Using evolution to evaluate reverse engineering technologies: mapping the process of software change*. The Journal of Systems and Software, Elsevier, no. 53, 2000.

[CHIK 90] E. J. Chikofsky and J. H. Cross, II. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, pages 13–17, January 1990.

[DEME 99a] S. Demeyer, S. Ducasse, and M. Lanza. *A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization*. In F. Balmas, M. Blaha, and S. Rugaber, editors, Proceedings WCRE'99 (6th Working Conference on Reverse Engineering), IEEE, October 1999.

[DEME 99b] S. Demeyer, S. Tichelaar, and P. Steyaert. *FAMIX 2.0 - The FAMOOS Information Exchange Model*. Research report, University of Berne, August 1999.

[DEME 00] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Finding Refactorings via Change Metrics*. In Proceedings of OOPSLA'2000, ACM SIGPLAN Notices, pages 166–178, 2000.

[DUCA 00] S. Ducasse, M. Lanza, and S. Tichelaar. *Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*. In Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET'00), June 2000.

[FENT 97] N. Fenton and S. L. Pfleeger. Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press, London, UK, Second edition, 1997.

[FOOT 94] B. Foote and W. F. Opdyke. *Lifecycle and Refactoring Patterns that Support Evolution and Reuse*. In Proceedings of the First Conference on Patterns Languages of Programs (PLoP'94), 1994.

[FOOT 97] B. Foote and J. W. Yoder. *Big Ball of Mud*. In Proceedings of PLop'97, 1997.

[FOWL 99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[GALL 97] H. Gall, M. Jazayeri, R. R. Klösch, and G. Trausmuth. *Software Evolution Observations Based on Product Release History*. In Proceedings of the International Conference on Software Maintenance (ICSM'97), pages 160–166, 1997.

[GALL 98] H. Gall, K. Hajek, and M. Jazayeri. *Detection of Logical Coupling Based on Product Release History*. In Proceedings of the International Conference on Software Maintenance (ICSM'98), pages 190–198, 1998.

[GAMM 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. Addison Wesley, Reading, MA, 1995.

[JAZA 99] M. Jazayeri, H. Gall, and C. Riva. *Visualizing Software Release Histories: The Use of Color and Third Dimension*. In ICSM'99 Proceedings (International Conference on Software Maintenance), IEEE Computer Society, 1999.

[LANZ 99] M. Lanza. Combining Metrics and Graphs for Object Oriented Reverse Engineering. Diploma thesis, University of Bern, October 1999.

[LANZ 01] M. Lanza, S. Ducasse, and L. Steiger. *Understanding Software Evolution using a Flexible Query Engine*. In Proceedings of the Workshop on Formal Foundations of Software Evolution, 2001.

[LEHM 96] M. M. Lehman. *Laws of Software Evolution Revisited*. In European Workshop on Software Process Technology, pages 108–124, 1996.

[LEHM 98] M. M. Lehman, D. E. Perry, and J. F. Ramil. *On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution*. In Proceedings of the Fifth International Symposium on Software Metrics (METRICS'98), pages 84–88, 1998.

[MATT 99a] M. Mattsson and J. Bosch. *Observations on the Evolution of an Industrial OO Framework*. In Proceedings of the International Conference on Software Maintenance, Oxford, England (ICSM'99), pages 139–145, 1999.

[MATT 99b] M. Mattsson. *Evolution Characteristics of an Industrial Application Framework*. In Workshop on Object-Oriented Architectural Evolution at the 13th European Conference on Object-Oriented Programming (ECOOP'99), 1999.

[MATT 00] M. Mattsson. *Evolution and Composition of Object-Oriented Frameworks*. Ph.D. thesis, University of Karskrona/Ronneby, Sweden, 2000.

[OPDY 92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[PARN 94] D. L. Parnas. *Software Aging*. In Proceedings of the 16th International Conference on Software Engineering (ICSM'94), Sorrento, Italy, 1994.

[PRES 94] R. S. Pressman. Software Engineering: A Practitioner's Approach. McGraw-Hill, 1994.

[RATI 00] Rational. *Rational ClearCase, Rational Software Inc.* Rational ClearCase offers essential, comprehensive software configuration management functions such as version control, workspace management, process configurability and build management. See http://www.rational.com/products/clearcase/ for further information, 2000.

[RIVA 98] C. Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. Masters thesis, Politecnico di Milano, carried out at Technical University of Vienna, 1998.

[ROBE 96] D. Roberts and R. Johnson. *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*. In Proceedings of Pattern Languages of Programs (PLOP'96), Allerton Park, Illinois, 1996.

[ROBE 97] D. Roberts, J. Brant, and R. E. Johnson. *A Refactoring Tool for Smalltalk*. Theory and Practice of Object Systems (TAPOS), vol. 3, no. 4, pages 253–263, 1997.

[SNIF 00] SNiFF+. *SNiFF+,Wind River Systems, Inc.* A source code analysis environment for large applications supporting C/C++, Java, CORBA IDL, Assembly and Fortran. See http://www.windriver.com/products/html/sniff.html for further information, 2000.

[SOMM 92] I. Sommerville. Software Engineering. Addison-Wesley, 1992.