

A Hybrid Query Engine for the Structural Analysis of Java and AspectJ Programs

Hamoun Ghanbari

**A Thesis
in
The Department
of
Computer Science and Software Engineering**

**Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada
September 2008**

© Hamoun Ghanbari, 2008



Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*
ISBN: 978-0-494-45297-4
Our file *Notre référence*
ISBN: 978-0-494-45297-4

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.


Canada

ABSTRACT

A Hybrid Query Engine for the Structural Analysis of Java and AspectJ Programs

Hamoun Ghanbari

Query-based code browsers can be customized to render many different types of views on-demand. In contrast to single-purpose browsers offered by modern IDEs such as Eclipse (e.g. Call Hierarchy), they release developers from mentally inducting the integral information from several views generated for specific purposes. However, customizing query-based tools, and communicating with their interfaces demands database expertise and understanding of the query language syntax. This puts a burden on maintainers by forcing them to encode their required views using complex queries. In this dissertation we investigate a query engine designed for software structural analysis which 1) provides a visual query interface over the high-level textual query language to eliminate the need for understanding the query language syntax, 2) incorporates the knowledge of programming language constructs into the factbase, query language, and the views, and 3) integrates the query-based and specific-purpose views already provided through the IDE. We are confident that this approach will be beneficial to maintainers during comprehension by allowing to abstract source code to high-level views and to speed up the rummage of the source code.

Acknowledgments

First and foremost I offer my sincerest gratitude to my supervisor, Dr Constantinos Constantinides, who has supported me throughout my thesis with his patience while allowing me to work in my own way. I attribute the level of my Masters degree to his encouragement and effort and without him this thesis, too, would not have been completed or written.

I warmly thank Venera Arnaoudova, Laleh Mousavi Eshkevari, and all other members of Software Maintenance and Evolution Research Group, for providing valuable comments and their friendly help for my research work.

I also would like to thank two of the most influential people in my life, which gave me the motivation toward continuing my studies in academia, Amirhasan Amintabar for teaching me the abc of computer science at secondary school and Professor Mohammad Taghi Rohani Rankoohi, from Shahid Beheshti University for teaching me everything I know about database management systems.

Finally, I owe my loving thanks to my parents for supporting me throughout all my studies at university, and for providing all my needs to complete my thesis.

Contents

| | |
|--|------------|
| List of Figures | vii |
| List of Tables | x |
| 1 Introduction | 1 |
| 1.1 Objective and goals of this dissertation | 2 |
| 1.2 Organization of the dissertation | 3 |
| 2 Background | 4 |
| 2.1 Program comprehension | 4 |
| 2.2 Query methods for supporting program comprehension | 5 |
| 2.3 Aspect-Oriented Programming (AOP) | 8 |
| 3 Problem and motivation | 10 |
| 4 Proposal: Deploying hybrid (visual+textual) queries | 16 |
| 5 Hybrid Query Composition: An Overview | 19 |

| | | |
|----------|--|-----------|
| 5.1 | Textual query composition | 20 |
| 5.1.1 | Predicates: ingredients of analysis | 21 |
| 5.1.2 | Query by example | 23 |
| 5.2 | Visual query compositions | 26 |
| 5.3 | Query result representation | 33 |
| 6 | Deploying query composition to analysis and measurement | 38 |
| 6.1 | Structural analysis queries | 38 |
| 6.1.1 | Identifying virtual method calls | 39 |
| 6.1.2 | Types/methods depending on other types/methods | 39 |
| 6.1.3 | Methods accessing local fields | 40 |
| 6.1.4 | Methods accessing the same field | 43 |
| 6.2 | Measurement queries | 43 |
| 6.2.1 | Queries about cohesion | 44 |
| 6.2.2 | Queries about dependencies and coupling | 49 |
| 6.2.3 | Instability of packages | 53 |
| 6.2.4 | Abstractness of packages | 53 |
| 7 | Automation and tool support | 56 |
| 7.1 | The fact extractor | 57 |
| 7.2 | Implementation of the visual query editor | 59 |
| 7.2.1 | Model package | 60 |

| | | |
|----------|---|-----------|
| 7.2.2 | Actions package | 61 |
| 7.2.3 | Figures package | 62 |
| 7.2.4 | Parts (controller) package | 62 |
| 7.2.5 | Policies (controller helpers) package | 63 |
| 7.2.6 | UI package | 63 |
| 7.3 | Implementation of the textual query editor | 63 |
| 7.3.1 | Textual query parser | 63 |
| 7.3.2 | Textual query editor | 66 |
| 7.4 | Query result evaluation | 69 |
| 7.5 | Query result representation | 71 |
| 7.5.1 | Tabular view | 71 |
| 7.5.2 | Textual tree views | 72 |
| 7.5.3 | UML like view | 73 |
| 7.5.4 | Graph view | 78 |
| 8 | Case Study 1: Graphical Editing Framework (GEF) | 79 |
| 8.1 | Framework overview | 80 |
| 8.2 | Structural analysis queries | 80 |
| 8.2.1 | Finding the correspondence between model, view, and controller classes | 80 |
| 8.2.2 | Finding correspondence between models and Editparts | 82 |
| 8.2.3 | Finding the correspondence between figures and editparts | 83 |

| | | |
|-----------|--|------------|
| 8.2.4 | Finding possible commands for a given request | 86 |
| 8.3 | Measurement queries: Investigating the quality of the GEF code | 90 |
| 9 | Case Study 2: Aspect-oriented implementation of the Observer design pattern | 95 |
| 9.1 | Description of the pattern and its implementation | 95 |
| 9.2 | A comprehension task | 98 |
| 10 | Case Study 3: Spacewar | 103 |
| 10.1 | Deployment of the tool over Spacewar | 105 |
| 11 | Related work and evaluation | 111 |
| 11.1 | Logic-based query approaches | 111 |
| 11.2 | SQL-based approaches | 113 |
| 11.3 | OQL-based approaches | 113 |
| 11.4 | Query-by-example approaches | 114 |
| 11.5 | Visual query approaches | 114 |
| 11.6 | Algebraic query approaches | 115 |
| 11.7 | Limitations | 115 |
| 12 | Conclusion and recommendations | 117 |
| | Bibliography | 119 |

List of Figures

| | | |
|------|--|----|
| 4.1 | Visual query for obtaining the utility classes. | 18 |
| 5.1 | Description of aspect-oriented programs by high-level constructs. . . . | 23 |
| 5.2 | Visualization of a join operation. | 29 |
| 5.3 | Visualization of a quantification operation. | 30 |
| 5.4 | Visualization of a summarization operation. | 31 |
| 5.5 | Visualization of a selection operation. | 32 |
| 5.6 | Visualization of a drill-down operation. | 32 |
| 5.7 | The query result represented in tabular view. | 33 |
| 5.8 | The query result represented in bar chart view. | 34 |
| 5.9 | The query result represented in graph view. | 35 |
| 5.10 | The query result represented in tree view. | 36 |
| 5.11 | The query result represented in UML-like view. | 36 |
| 7.1 | A storage of a sample source code in the codebase, representing nodes and relationships between them. | 58 |

| | | |
|-----|--|----|
| 7.2 | The overall structure of visual query composer plug-in and its dependencies to internal eclipse libraries. | 59 |
| 7.3 | The packages of visual query composer tool. | 60 |
| 7.4 | Data structure of model in terms of classes and their relationships. | 62 |
| 7.5 | The maximal structure of parsed query in terms of its nodes. | 67 |
| 7.6 | Available mechanisms to represent program structures. | 71 |
| 7.7 | Sample mapping of query result to textual tree view. | 74 |
| 7.8 | Formation of boxes in the box-and-line visualization | 76 |
| 7.9 | Structure of classes involved in the box-and-line visualization. | 77 |
| 8.1 | Partial UML class diagram of GEF. | 81 |
| 8.2 | UML sequence diagram for overall workflow of GEF. | 81 |
| 8.3 | The relationship among instances of GEF classes and their role in the MVC architecture. | 82 |
| 8.4 | Investigation of GEF's flow-example using Eclipse views. | 84 |
| 8.5 | The correspondence of the figure classes to controllers, retrieved as a query outcome. | 85 |
| 8.6 | An overall view of modules' of a GEF based applications and their dependencies, divided into user defined and GEF core code. | 88 |
| 8.7 | GEF packages divided into six classes based on measurement attributes. | 93 |
| 8.8 | A 3D bar chart representing the co-occurrence of different categorical attributes of GEF framework. | 94 |

| | | |
|------|---|-----|
| 9.1 | The SubjectObserverProtocolImpl aspect the AspectJ cross-reference view. | 99 |
| 9.2 | The source code of aspect SubjectObserverProtocolImpl in Observer protocol example. | 100 |
| 9.3 | The Button class in the AspectJ cross-reference view. | 100 |
| 9.4 | The source code of class Demo in Observer protocol example. | 101 |
| 9.5 | The query for retrieving occurrences of observer protocol and it's result. | 102 |
| 9.6 | The visual query for finding occurrences of observer protocol. | 102 |
| 10.1 | Partial class diagram of the Spacewar system. | 106 |
| 10.2 | Messages to which an object of a given type can respond to in Spacewar example. | 109 |

List of Tables

| | | |
|------|---|----|
| 4.1 | The textual query for obtaining the utility classes. | 17 |
| 5.1 | List of logic predicates available in the codebase and their corresponding description. | 37 |
| 6.1 | Query to obtain the polymorphic virtual calls from one method to another. | 39 |
| 6.2 | Query to obtain the dependency between methods. | 40 |
| 6.3 | Query to obtain the dependency between types. | 41 |
| 6.4 | Query to obtain the methods accessing a local field from a type. | 42 |
| 6.5 | Query to obtain the fields accessed by a local method. | 42 |
| 6.6 | Query to obtain pairs of methods accessing a common field. | 43 |
| 6.7 | Query to obtain Chidamber and Kemerer Lack of Cohesion. | 46 |
| 6.8 | Query to obtain Henderson-Sellers lack of cohesion metric. | 48 |
| 6.9 | Query to obtain pairs of methods accessing a common field. | 49 |
| 6.10 | Query to obtain the number of incoming dependencies for each package. | 50 |
| 6.11 | Query to obtain the number of outgoing dependencies for each package. | 50 |
| 6.12 | Query to obtain the incoming dependency of a type. | 51 |

| | | |
|------|--|-----|
| 6.13 | Query to obtain the "fan-in" of a method. | 52 |
| 6.14 | Query to obtain the "fan-out" of a method. | 52 |
| 6.15 | Query to obtain the number of incoming dependencies of a field. | 53 |
| 6.16 | Query to obtain the the amount of instability of a package. | 54 |
| 6.17 | Query to obtain the abstractness of a package. | 55 |
| 7.1 | BNF convention used to represent the query language grammar. | 64 |
| 7.2 | The algorithm to evaluate calculus-based conjunctive queries. | 70 |
| 7.3 | The code to represent the program hierarchy as compound graph. | 78 |
| 8.1 | The code to create different editPart for different model objects in the GEF flow-example | 83 |
| 8.2 | Query to obtain the figure classes corresponding to each controller class. | 85 |
| 8.3 | The code to setup the edit policies for a given subclass of Editpart. | 87 |
| 8.4 | Retrieving a Command for a given Request using an EditPolicy. | 87 |
| 8.5 | Retrieving a Command for a given Request. | 90 |
| 10.1 | Partial code of aspects 'Coordinator'. | 107 |
| 10.2 | Obtaining all messages to which an object of a given type can respond. . | 108 |
| 10.3 | Obtaining methods defined by an aspect for a supertype of a given type. | 110 |

Chapter 1

Introduction

Program comprehension is a major activity in most software maintenance task, due to the effort and time required to initially understand the system. The provision of structural analysis and program measurement through flexible reporting during program comprehension is vital to software development and subsequently the maintenance phase. In overall, there are multiple choices of the tool support for structural analysis and measurement. Based on the customizability and ease-of-use tradeoffs, these tools can be placed in the following categories:

1. Specific purpose tools mostly allow low-cost analysis and measurement with a flexibility limited to change of metric parameters rather than allowing the introduction of a new metric.
2. Modern IDEs such as Eclipse [7] allow viewing and navigating source code in an easy way but with low degree of configurability according to the intended software

entities and relationships to be analyzed.

3. Generic tools provide a less easy-to-use but highly configurable environment capable of both structural analysis and measurement by describing the analysis through a high-level query language.

In this dissertation we focus on the third category of tools as very highly customizable solutions. These tools have long been used for transforming the source code into comprehensible diagrams and views (e.g. JQuery [69, 47] and JTL [23]). Such languages can extract, filter, summarize, and combine information from source code, by storing a database representation of the code, allowing maintainers to execute queries using a general purpose query language. These representations are referred to as codebases. By standing on top of general purpose query languages, such models can be beneficial because they provide flexible reporting through user-driven customizations.

1.1 Objective and goals of this dissertation

Our overall objective is to improve the comprehension of programs by providing a generic automated environment under which one can gain static information regarding the software elements and their relationships in a highly-configurable and yet easy-to-use fashion. This is done by allowing maintainers to describe their own analysis and measurement logic through a high-level language, aware of program structure, which does not demand as much database expertise and understanding of query language syntax.

To meet these objectives, we set a number of goals: 1) The creation of a hybrid query engine to parse and execute queries in two different representations, namely visual and textual. 2) To integrate the generic query tool inside an integrated development environment (IDE) to gain the existing benefits of tools offered by such environment. 3) Using more software-design-biased visual constructs to navigate the query result 4) To construct a library of queries to support common structural analysis and measurement tasks currently supported by non-query based tools.

1.2 Organization of the dissertation

The remaining part of this dissertation is organized as follows: In Chapter 2 we provide the necessary theoretical background of program comprehension, query methods, and aspect-oriented programming (AOP). In Chapter 3 we discuss the problem and motivation behind this research, followed by the presentation of our proposal in Chapter 4. We describe our methodology in Chapters 5 and 6. While, the first is discussing the properties of our query language and the query composition process, the second, introduces a set of common queries usable during structural analysis and measurement. In Chapter 7 we discuss the components providing the core functionality of automation and tool support. In Chapters 8, 9, and 10 we present three different case studies to demonstrate how our approach can be deployed for typical comprehension and measurement tasks. In Chapter 11 we discuss related work, and we evaluate our approach. We conclude our discussion in Chapter 12.

Chapter 2

Background

In this Chapter, we discuss the necessary background to this research, starting with an overview of program comprehension. We then discuss query based approach (and its different methods) to gain program comprehension. Finally we give a brief background on AOP as a new generation of paradigms which will be referred to, in the rest of this dissertation.

2.1 Program comprehension

Program comprehension is the process of obtaining knowledge about a program [63]. It has shown to consume a significantly large proportion of resources during the overall maintenance phase [63], particularly when maintainers are not the initial developers of the system. Furthermore, design artifacts, if at all present, cannot provide complete comprehension since they are often incorrect or incomplete. A lot of effort has been

spent on providing approaches to support and facilitate program comprehension and, as a result, a variety of models of human program comprehension process have been proposed [53, 70]. Comprehension methods rely on the study of the dependencies between program (or software) elements.

2.2 Query methods for supporting program comprehension

So far, database theories have been largely applied to software engineering tasks. Aiming at obtaining comprehension, code databases (also generally referred to as code-bases and in [28] called Software Information System (SIS)) have been proposed for the analysis of programs by representing the static structure of a program in a model. They are information systems which are able to help maintainers by storing and querying the software system source code as data. This data are usually populated automatically from the source code (functions, files, data types, etc.).

According to [21], the interfaces provided to communicate with databases can be placed in the following groups: logic based (e.g. FOPL) [37], calculus based and algebraic [57] [50]. Although, all these languages share the same property, declarativeness (as opposed to imperativeness)¹, they have features which makes them different to each other, as follows:

Predicate and description logic: Logic programming languages are based on first-order

¹Declarativeness of a language means that written programs in that language do not state how a result is to be computed; instead they declare how the result should look like.

predicate logic where data are presented by Horn clauses and where a logic inferencing process is used to produce results. They allow the programmers to integrate logical statements with programming constructs. They describe relationships between variables in terms of functions, while the language executor applies some fixed algorithm to these relations to simulate inferencing and to produce a result. Logic statements are made of terms which themselves can be constants, variables or compound terms. A compound term, or functor, is represented as `functor (parameter list)`. A functor is like a predicate in predicate calculus and its parameters can be atoms, variables or other functors. A logical program consists of two kinds of statements which are statically declared: facts and rules. Facts are propositions that are assumed to be true and they constitute the statements used to construct the hypotheses. Rules are implications between propositions. A problem domain is therefore defined in terms of queries, and the goals are addressed by a built-in search mechanism [64].

Relational calculus: Calculus based database sublanguages (e.g. SQL) are adapted versions of predicate logic to relational database models. From user point of view, calculus oriented languages provide a more suitable platform for explaining natural-language-like sentences as queries. However, these sublanguages are designed specifically for query purposes (as opposed to programming tasks) and are less expressive than logic based ones.

As other logic-based expressions, a formula is defined in terms of atomic formulas

through application of connectives and quantifiers. Atomic formulas are made of predicates which are applied to terms and terms are defined inductively as being a constant, variable, or a function applied to a term. From another view, a query is composed of a set of variables which are either free, quantified using existential or universal quantifier, or used in aggregated functions such as *count* and *average*. Each variable is introduced and its type and domain is determined before being used in the query. The result of a query (or semantic of formula in logic sense) is defined in terms of all substitutions of free variables with values which satisfy the query formula (evaluate it to true).

Relational algebra: In contrast to logic based approaches, the theory of algebraic languages is relatively newer, and more dependent on the primary construct of the database model. For example relational algebra is application of relational operators on relations (tables) while multidimensional algebra is the same for multidimensional tables (cubes). An algebraic query could be composed iteratively by having a sequence of algebraic operators repeatedly applied to each other's outcome. This results in decomposition of query writing process in a natural step-by-step parts, making it possible to map each *algebraic operator* to a simple user-system interaction (e.g. UI drag-and-drop operations).

2.3 Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming [49] is a relatively new programming paradigm extending existing ones (such as Object-Oriented), aiming at better separation of concerns, by introducing a new modular unit of decomposition called aspect. Aspects try to localize certain properties of the system that would have been cutting across the inheritance hierarchy, had they been implemented in conventional methodologies. Currently there exist many approaches and technologies to support AOP. One such notable technology is AspectJ [4], a general-purpose aspect-oriented language, which has influenced the design dimensions of several other general-purpose aspect-oriented languages, and has provided the community with a common vocabulary based on its own linguistic constructs. In the AspectJ, an aspect definition is a new unit of modularity providing behavior to be inserted over functional components. This behavior is defined in method-like blocks called *advice blocks*. However, unlike a method, an advice is never explicitly called. Instead, it is only implicitly invoked by an associated construct called a *pointcut expression*. A pointcut expression is a predicate over well-defined points in the execution of the program which are referred to as *join points*. When the program execution reaches a join point captured by a pointcut expression, the associated advice block is executed. Even though the specification and level of granularity of the join point model differ from one language to another, common join points in current language specifications include calls to - and execution of methods and constructors. Most aspect-oriented languages provide a level of granularity which specifies exactly when an advice block

should be executed, such as executing *before*, *after*, or *instead of* the code defined at the associated join point. Furthermore, several advice blocks may apply to the same join point. This granularity introduced by AspectJ has been influenced by the auxiliary functions of CLOS [36]. In cases where advice blocks refer to the same join point, the order of execution is specified by precedence rules defined in the underlying language.

Currently there exist a lot of languages with aspect-oriented programming support. For example Java, Ada, C++, and Ruby each have its own aspect-oriented frameworks: AspectJ [4], AspectAda [58], AspectC [3], and AspectR [5] respectively. However the level that they support aspect orientation is not equal, and, thus, the amount of flexibility achieved in programming and design decisions by their users are different.

Chapter 3

Problem and motivation

The provision of user driven analysis through flexible reporting during program comprehension is vital to software development and maintenance. Modern IDEs such as Eclipse [7] provide source code browsers to view and navigate source code in different ways. For example, in Eclipse the Package Explorer shows program structure in terms of structural inclusion relationships, the Type Hierarchy shows inheritance relationships among Types, and the Call Hierarchy shows the structure of the static call graph. However, there are cases when a maintainer's need does not fall into the set of existing views offered by these browsers. One such example is the extraction of the level of coupling between classes. This would involve obtaining measurement factors such as fan-in and fan-out of packages (see Chapter 6 for their definition) in the system. In order to obtain this information one can use existing browsers (in this example: Call Hierarchy and Type Hierarchy) while performing a number of mental computations, which can end up

being costly in terms of time .

A possible solution to the above limitations is provided by generic code browsers (e.g. JQuery [69, 47] and JTL [23]). They can be customized to render many different types of views for each need, yet in a more cost-effective way than plug-in extensions. Generic code browsers mainly perform this by storing programs as data; that is they store database representation of the code and allow developers to execute queries using a general purpose query language. They are, therefore, also referred to as query-based code browsers or simply codebases. By standing on top of a general purpose query language, such models are beneficial by provisioning of flexible reporting through user-driven customizations. Based on the high degree of interactivity in these methods, the analysis task is largely user directed. "However, customizing such browsers requires more expertise than just clicking GUI buttons" [69]. Communicating with their interfaces demands databases expertise and an understanding of query language syntax (e.g. SQL for relational systems). This can either put a burden on maintainers and developers to work with query languages, or can place the demand for a person as a customizer to encode the views and navigations required by maintainers into queries. In logic-based code bases, this is intensified by the fact that customizers need to know not only the defined predicates (relations) but also the inference rules used in logic programming constructs (e.g. Prolog). The reason is that in some cases the only way to encode a query is to use multiple inference rules.

A possible solution to this limitation is to allow maintainers to describe their own

measurement logic through a high-level language, aware of program structure, which in-turn will be interpreted to form the intended report. To this end there has been a number of research proposals in the literature. These proposals can be placed in the following three categories based on their underlying query languages:

1. Logic-based query approaches such as SOUL [71], JQuery [69] [47], JTransformer [11], CodeQuest [42] use Prolog-based query engines such as TyRuBa [13] or Datalog [14] to perform query evaluation.
2. SQL-based approaches (e.g. [60]) represent the program structure in the form of relational tables and use the native query engine of a relational database to execute queries.
3. OQL-based approaches such as Semmler [68, 27] form the most recent wave of query tools published in recent years. They provide an intuitive interface through an abstract object-oriented layer above the actual logic predicates.
4. Query-by-example approaches like JTL [23] aim at making queries simpler by making their syntax like the programming language itself.

The motivation behind having the variety of above query approaches is the need to lessen the demands for database expertise by the query interfaces provided for these systems. In this section we present an example illustrating how visual query composition is more advantageous in terms of ease of use and simplicity than its predecessors in a

real world situation. This example is a hypothetical scenario in which a maintainer needs to select and view specific modules having certain measures.

Consider a case in which a maintainer wants to find the unimportant (utility) classes in the system (i.e. those containing no business logic). S/he has a general idea that a utility method could be the one called by a lot by other methods, and, as a result has high fan-in. While it is possible to navigate through classes one-by-one and check the fan-in in a view, this would cost more than it would contribute to the process. As a result, the maintainer customizes a new view using one of the available generic code browsers, formulating the method's importance using the following formula:

$$utility(method) : fan - in(method) \geq threshold$$

in which the threshold is some pre-set amount which can be calculated using statistical methods. S/he also considers that classes having at least two of such methods could be defined as utility classes.

Consider the following primary predicates are available for him to use in the code-base:

1. $Invokes_{method \times method}(m1, m2)$ shows that m1 calls m2.
2. $Includes_{class \times method}(c1, m1)$ shows that the class c1 includes the method m1.
3. $NumberOf_{predicate \times number}$ is a higher order predicate which accepts another predicate as an argument and counts the number of instances satisfying it.

S/he logically finds out that the fan-in of a method could be defined simply as the number of other methods calling this method:

$$fan - in(M) : numberOf(Invokes(X, M))$$

S/he can finally compose a predicate called *utilityClass(C)*, describing the desired view with the following hypothetical formula:

$$utilityClass(C) : numberOf(Includes(C, M) \wedge fan - in(M) \geq threshold_1) > threshold_2$$

This query can be implemented in a logic codebase in a language such as Prolog, similar to the approach used in [32] with multiple rules, as follows:

```

count(Goal, N):-
    flag(counter,Outer, 0),
    (call(Goal),
     flag(counter,Inner,Inner+1),
     fail;
     flag(counter,Count,Outer),
     N =Count).
getMember(X, [X|_]).
getMember(X, [_|T]):-getMember(X,T).
methodWithHighFanIn(Method, ListOfMethods):-
    findall(Method,
            (count(invokes(Amethod,Method),N),N>=12),
            ListOfMethods).
utilityClass(Class):-
    methodWithHighFanIn(Method, ListOfMethods),
    count(includes(
            getMember(X, ListOfMethods),Class),Num),
          (Num>=2).

```

Another approach to implement the query is to use a relational codebase in a language such as SQL, similar to the approach used in [60], as follows¹:

```

SELECT UTILITY_CLASSES._CLASS
FROM

```

¹The HAVING clause was not used in the SQL query in this case, since it is not generally a substitution for nested queries, and here our goal was to show the overall complexity of the approaches.

```

(SELECT
  Includes.CLASS AS _CLASS,
  SUM(Includes.METHOD) UTILITY_METHODS
FROM
  Includes
  , (SELECT CALLEE,SUM(CALLER) AS
      MEASURE FROM Invokes) FAN_IN
WHERE
  INCLUDES.METHOD=FAN_IN.CALLER AND
  (FAN_IN.MEASURE>:threshold1)
)UTILITY_CLASSES
WHERE
  UTILITY_METHODS>:threshold2

```

In both of the above approaches, there is a need for a database expert to encode the queries explaining the new view. The reason is that the query logic is scattered through the query, there is no hint to identify main points of the query, and there is no content assist during query composition because of the distant from software engineering domain.

According to the above limitations, there is a need for a query method which provides a high-level query language to eliminate the need for database expertises which is more developer friendly and closer to software domain. Moreover, it should be an integral part of developer's IDE used for maintaining the system.

Chapter 4

Proposal: Deploying hybrid (visual+textual) queries

As a solution to the limitations imposed on the maintainers during the query composition process, we propose the deployment of hybrid query composition as a low-cost view customization technique suitable for visualizing and navigating codebases. Hybrid query composition deploys a visualization mechanism on top of textual queries. In this approach we use a graphical notation (including text, box and line) similar to the Unified Modeling Language (UML) class diagrams for representing the intention of maintainers as a structure diagram which is, in fact, equivalent to full first-order logic queries. This notational language enables the maintainers to model the query, based on object definitions and object relationships which is more intuitive than text based ones.

This can make it possible to offer simple interactive query transitions instead of hand-coded queries and provide maintainers with an automatic way to compose queries using simple drag-and-drop operations on the graphical user interface. This can be done by mapping of visual drag-and-drop operations to operators of the query language. Instead of invoking queries, end-users will be able to describe query semantics using graphical representations, which will be in turn translated into database commands such as selection, projection, and join. These operations can be interactively invoked by end-users until the intended result (including all elements of interest) is formed.

The proposed textual and visual representation, equivalent to the query discussed in Chapter 3 are shown in Table 4.1 and Figure 4.1 respectively.

```
(Type t
  from (int count by(Method m2) as meth-num
        from (int count by(Method m2) as fan-in
              where m2 invokes m1
            )
        where fan-in>threshold
      )
  where meth-num>2
)
```

Table 4.1: The textual query for obtaining the utility classes.

The main features of the proposed technique (to be discussed and implemented in subsequent sections) are as follows:

1. Instead of relying solely on textual representation of queries the hybrid query composition uses a combination of graphical and text-based interfaces to interact with the query engine.

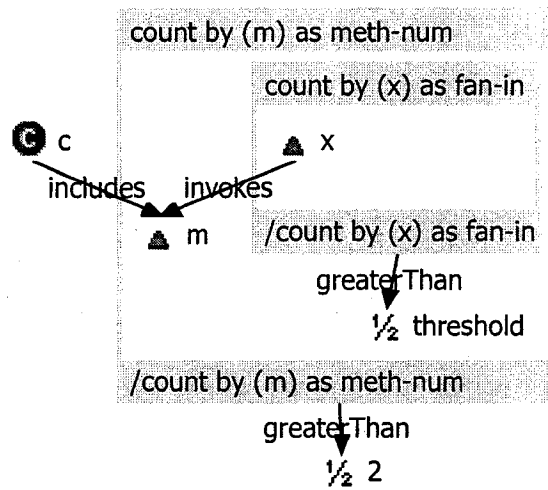


Figure 4.1: Visual query for obtaining the utility classes.

2. Not only query syntax itself, but also could query result be visualized by various visualization techniques such as chart views [2], textual tree views [1], box-and-line visualizations [46] (structure diagrams) and compound graphs. These possibilities could be explored to enable viewing software in the level of granularity.
3. Integrates the query-based and specific-purpose views already provided through the IDE.

Chapter 5

Hybrid Query Composition: An

Overview

In this Chapter we provide a description of the two approaches for the user to encode queries, namely textual (subsection 5.1) and visual (subsection 5.2) and the query language operators in each case. In each subsection, we illustrate how a query language can be deployed to interact with a codebase using examples. The textual definition of a query can be transferred to a visual equivalent and vice versa. This makes it possible to compose queries iteratively using the more appropriate approach at the moment. Using the textual editor a user can benefit from syntax highlighting, auto completion, error recovery, and predicate suggestion. In visual query composition view, a user is able to use the same capabilities as textual editor, however in a more intuitive way. The visual querying serves as a graphical interface mediating the composition and representation

of structural software queries. Influenced by graph-based visualization techniques [41], our graphical mediator is representing the query criteria in patterns of interconnected nodes and arcs. More formally, it is a directed graph consisting of vertices, which represent entities, and edges, which represent semantic relations between these entities. Using declarative graphic representation features of visual queries, one can either represent knowledge by asserting any kind of proposition which can be assumed as true sentence in propositional logic system. Moreover, like conceptual graphs, the graphical notation is augmented by abstraction boxes, representing quantifiers. For example one can assert $exists(x).. : X$ which is a sentence over objects of the type X (x is a variable in this sentence). The visual nature of the interface makes it beneficial for conveying knowledge quickly.

5.1 Textual query composition

The purpose of having a textual query language is to offer the same functions offered by interactive user interface (UI) operations through simple textual commands, as well as being able to express more complicated logics. Thus, textual queries are been provided for users who are more comfortable using conventional textual tools, and for those information which is more difficult to extract through visual queries.

Like other calculus-based query methods (e.g. relational calculus) our query language has an alphabet including *constants* (e.g. "public" and 243453), *variables* ($r1, r2, \dots$),

functions (e.g. $f(\text{int},\text{int})$), *predicates* (monadic like $P1()$ or dyadic like \neq, \leq, \geq), *connectives* (and, or, not), *quantifiers* (forall, exists), and delimiters ($(,), [,], \{, \}$). As other logic-based expressions, a formula in this language is defined in terms of atomic formulas through application of connectives and quantifiers.

We adopted a more user friendly query syntax than the pure calculus, using a grammar with a set of keywords already being used in the widely used query language SQL. In our query language FROM, and WHERE are keywords that denote different parts of the query ¹. Delimiter([..]) introduces elements to be viewed in the query result set along with their data types. The FROM clause is the way to nest sub-queries inside one another. Using FROM clause one can introduce a subquery having new, aggregated, quantified variables and put constraint on those using the predicates. WHERE introduces the set of predicates which is applied to the variables and constants. In the rest of this section we introduce the predicates of our query language in subsection 5.1.1 followed by example of using them in subsection 5.1.2.

5.1.1 Predicates: ingredients of analysis

In this section we are going to introduce the predicates which form the basic building blocks of queries and deploy them to access the source code information in the codebase. We adopt predicates, functions, and connectors to form queries in our system. Predicates are designed to be adaptable to various programming paradigms such as object-oriented,

¹Although the syntax of the query language is designed to be similar to that of SQL, the meaning of keywords are a bit different.

and aspect-oriented. For every language, we constructed a model, conceptualizing the possible static structure formed by programs of such a language. This conceptual model is described in terms of high level programming constructs (e.g. aspect, method, and class) and the relations between them. Figure 5.1 represents such a conceptual model in form of a UML class diagram for aspect-oriented programming². Table 5.1 shows a list of logic predicates, available in our codebase for supporting this conceptual model, along with their meaning. We can divide the predicates of the logical model into the following three categories:

Static inclusion predicates Binary relations modeling modular containment of various program elements (e.g. between packages and files).

Object-oriented predicates Constructs which are originally supported by object-oriented languages (also adapted to an aspect-oriented context).

Aspect-oriented predicates Constructs which are exclusively supported by aspect-oriented languages.

A program element from the programming language domain is a node that corresponds to a logical or physical elements introduced by the programming language:

Project, Package, File, Class, Interface, Aspect, Enum, Constructor, Method, Field,

²Though not bound to object-orientation, aspect-oriented programming can be thought of as a superset of object-oriented programming.

Pointcut, and Advice. Program elements are related using program relationships: *Extends, Implements, AdvisesFieldAccess, AdvisesMethod, DeclaresOn, MatchedBy, DeclaredParentOn, Declare, DeclareInterType, UsesPointcut, and Has-a.* We then translate Abstract Structure Model (ASM) to construct our codebase. Figure 5.1 represents such a model in the form of UML class diagram.

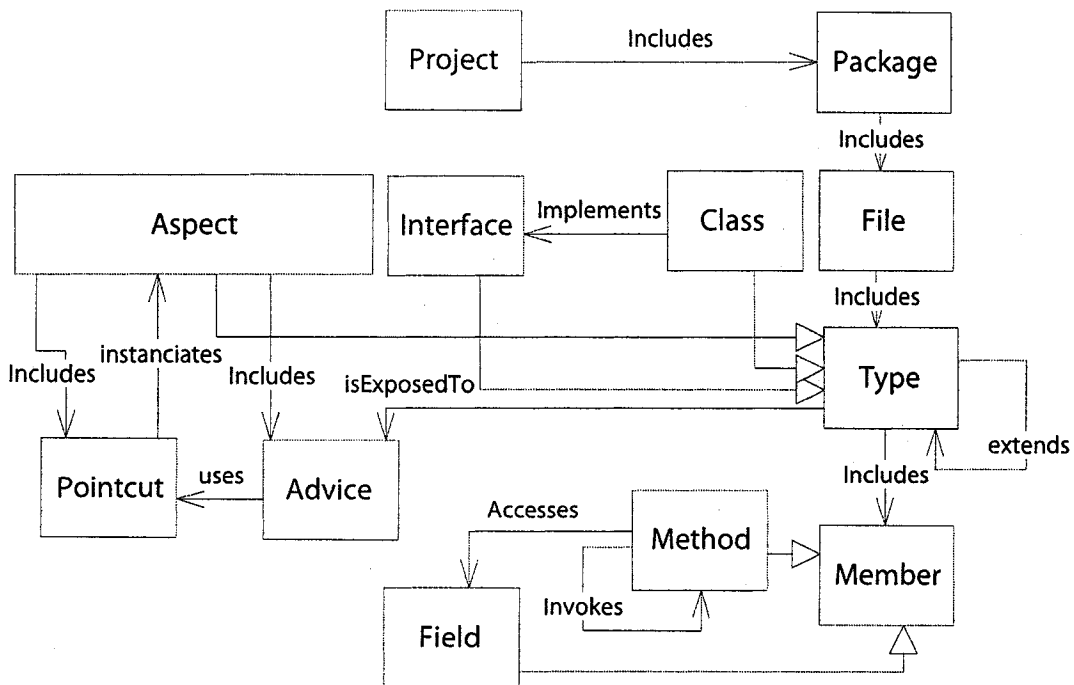


Figure 5.1: Description of aspect-oriented programs by high-level constructs.

5.1.2 Query by example

In the following, the query elements of our query language (e.g. quantifiers, aggregate functions, scalar functions, named-nested queries) are discussed together with examples to show how they could be applied to code-databases domain. Here we are going to use

two predicates, one from the Table 5.1, namely *AdvisesInvocation* and the other called *Line-Of-code* determines how many lines of code each method has.

Quantifiers quantify free variables in formulas. For example, the following query is using `exists` quantifier to define the table, representing all of the advised methods:

```
Is-Advised=  
[Advice advice, Method callee] from  
exists[Method caller] where  
AdvisesInvocation(advice, caller, callee)
```

Aggregate functions are the application of a specific set based function which operates on a multi-valued field, and could belong to `sum`, `avg`, `count`, `min`, `rank(n)`. For example, the following query represents the code coverage for each advice and caller in terms of number of times a method invocation from the *method(it is the caller)* is *advised* by the *advice*. A measure for a certain pair of advice-method is obtained by adding up the number of times the *method* invokes another method, while this invocation is advised by the *advice*.

```
Advice-Caller-Code-Coverage=  
[Advice advice, Method caller, count(Method callee)] where  
AdvisesInvocation(advice, caller, callee)
```

A complex example, combining quantifiers, aggregate functions, and connectives, is retrieving the number of classes in *package1* having at least one inter-type declared field by *aspect1* which is defined by the following query:

```
[count (Class class)] from
  exists [Field field ,Aspect aspect] from
    package1 includes class and
    DeclaresMemberOn("aspect1", class , field)
```

Scalar Functions are predefined functions, such as standard mathematical operators (+, -, *, /), and can be used to map atomic inputs to outputs in the query formula. For example the following query represents the code coverage, for each aspect in a class. A measure for a certain pair of advice-method is obtained by multiplying the number of methods invoked from the *class* by their associated line-of-code (LOC), while this invocation is advised by an advice from the *aspect*.

```
Advice-Callee-Code-Coverage=
  [Aspect aspect , Class class ,sum line-Of-code by (aspect , class)]
  from
    exists [Advice advice , Method caller ,Method callee] where
      callee has-number-of-lines line-Of-code
      class2 includes callee and
      aspect includes advice and
      AdvisesInvocation(advice , caller , callee)
```

Named-Nested queries are used to encode complicated query logics by breaking it into modular units; that is, predicates which themselves, could be nested inside other predicates in a way that both queries (outer and nested) can share some variables. For example, in order to know the classes which have fan-in of more than 5, instead of writing a new query one can easily break it into the following two queries:

```
Class-Fan-In-Greater-Than-5=
  [Class class]
  from class-Fan-In(callee , fan-in)
```

```
where fan-in>5
```

while the query to obtain the fan-in of classes could be written as:

```
Class-Fan-In=  
[int count(Method callee) as fan-in] from  
exists[Method caller] where  
Invokes(caller, callee) and  
class includes callee
```

In the first query, the variable 'class' in the nested *Class-Fan-In* is not anymore free; that is, it is bound to its value in the outer query. In another example one could derive a query from the *Class-Fan-In*, to obtain packages in which only classes with fan-in of at least 5 exist:

```
[Package p] from  
forall [Class class] from  
class-Fan-In(callee, fan-in) where  
fan-in>5 and  
package includes class
```

In this query, the variable *class* in the nested *Class-Fan-In* is bound to its value in the enclosing quantifier (*forall x1*).

5.2 Visual query compositions

Visual queries are constructed by dragging the elements from toolbar in the query composition view. More detail regarding applying the tool in the Chapter 7. Each figure in the rest of this subsection represents applying one operation at a time, while subfigure (a) represents the visual query and (b) represents the result in tree format. In

each subfigure (a) and (b) the large solid arrow represents the transition before and after applying the visual operation (i.e. drag-and-drop).

In the beginning, there is no element dragged into the query composition view. Since no element is requested to be shown in the query result and the formula is not explicitly mentioned, this view is equivalent to the query:

```
from [] where true
```

meaning that select nothing from a query result set which every combination of tuples is an answer to the query. Dragging an icon of a programming language element from the toolbar into the view, the result view will be equivalent to the query:

```
from [ProgrammingElement pe] where true
```

In this case the query answer will be all program elements of that type. The following is a brief description of actions available besides graphical model based on their equivalence to algebraic query operator:

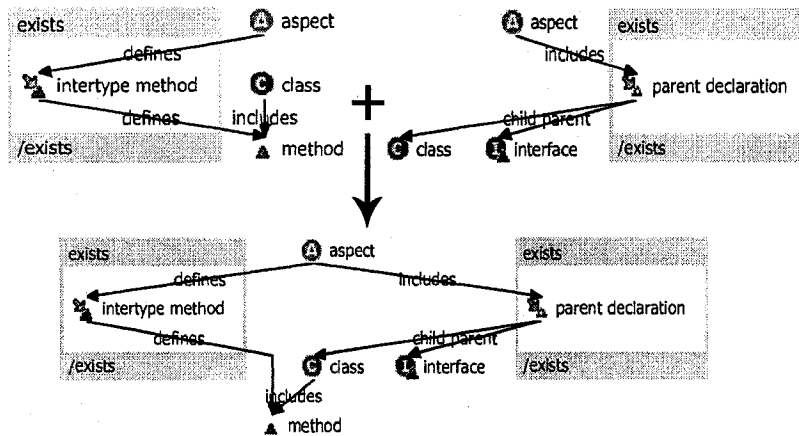
A cartesian product of instances of two programming element types is obtained by dragging their icons from the toolbar into the view. One can define a cartesian product of types interactively by dragging icons into the area while not connecting them with a relationship. All newly created disconnected subgraphs are elements of the cartesian product and the query result corresponds to all combinations of program elements of those types. This view is equivalent to the query:

```
from [ProgrammingElement pe1, pe2, . . . .] where true
```

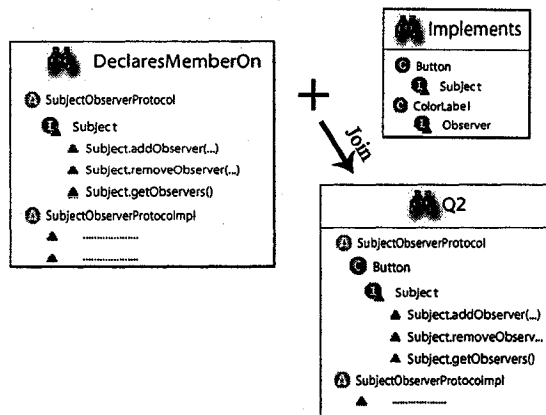

A join (combination) combines information of two types (e.g classes and aspects) based on a relationship between instances of those types. It is graphically performed by dragging the icon of two programming language elements from the toolbar into the view (like cartesian product), and dragging a specific arrow from one end to another which represents the semantic relationship between those. For example in Figure 5.2 the developer combines the information from the query representing the intertype methods (see Section 2.3) declared by an aspect into a class to the query which represents an aspect which introduces a parent for the class. This is done by dragging and dropping the corresponding icons on each other and results in finding the classes which have a parent and a method declared in them by the same aspect. This view is equivalent to the query:

```
from [ProgrammingElement pe1,pe2 ,....] where
    pe1 relationship pe2 and
    .....
```

A quantification operation is performed by dragging a programming element type which is currently free, into a quantifier box. This is also called constraint since it selects instances of those programming element types staying out of the quantifier box while limiting them to those satisfying the applied quantifier. Precisely speaking, it selects those elements having at least one instance (for existential box) or having all instances (for universal box) of quantified types participating in the original relation with them. A sample of applying quantification operation is demonstrated in Figure 5.3. In this figure the query selects the methods and aspects for which,



(a)



(b)

Figure 5.2: Visualization of a join operation.

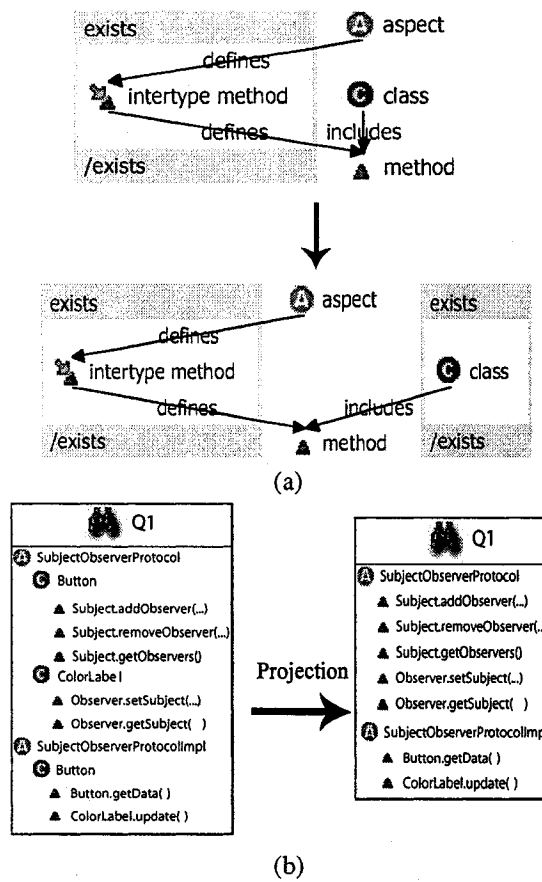


Figure 5.3: Visualization of a quantification operation.

there exists a class whose method is declared by the aspect.

A summarization (aggregation) is done in two steps. First, it takes the table from a specific state, cuts out aggregated columns, which results in a bag (a relation with multiple equal tuples). Then, it packs the resulted bag and makes a new non-normalized relation with multi-value aggregated fields. Second, it applies the aggregation function (set based function) on the instances of each multi-valued field separately. Set based functions could be one of {sum, avg, count, min, and

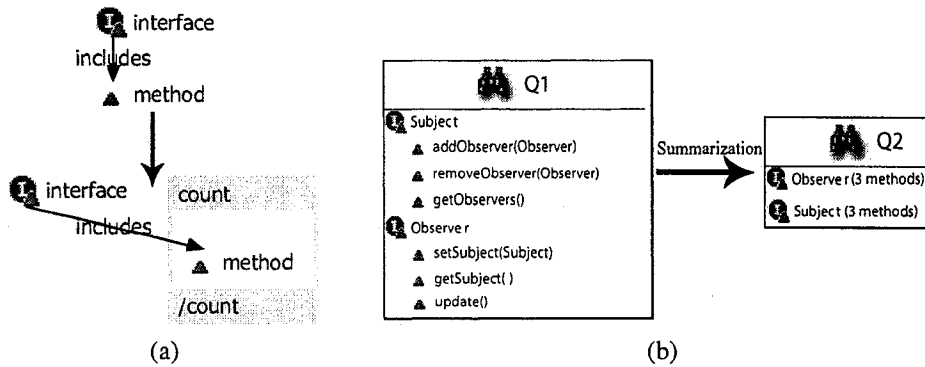


Figure 5.4: Visualization of a summarization operation.

rank(n)}. A sample of applying summarization operation is demonstrated in Figure 5.4. In this figure, after summarizing multiple method instances in each cell in the Method column using the 'count' aggregation function, the number of methods per interface is obtained in the result.

A selection selects a subset of tuples satisfying a user defined formula composed of unary or binary predicates, Boolean algebraic operators, and numeric algebraic ones (e.g. $<$ and $=$). These algebraic operators are applied to atomic values of instances to test if they exist in the new selection or not. This operation is graphically done by assigning the boolean expression as the alias of the dragged type. A sample of applying selection operation is demonstrated in Figure 5.5. In this figure, selection is applied based on the name attribute of the 'interface' using 'hasName' predicate, and resulted in a new table limited to 'Observer' interface.

A drill-down is a special case of *join* operation which is done on a special type of

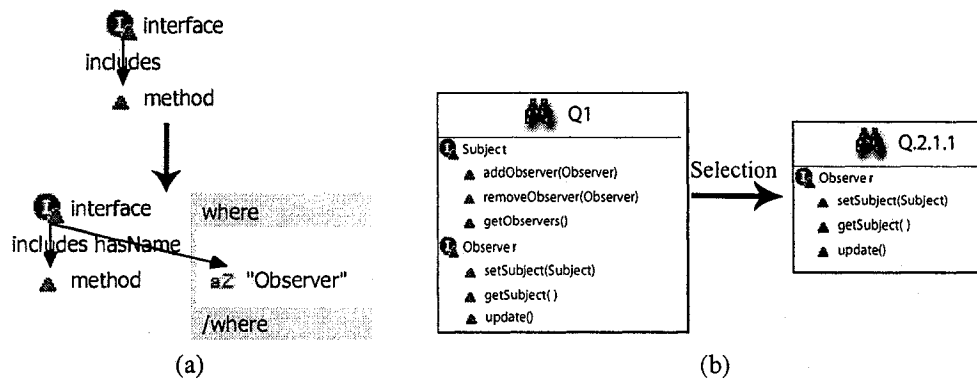


Figure 5.5: Visualization of a selection operation.

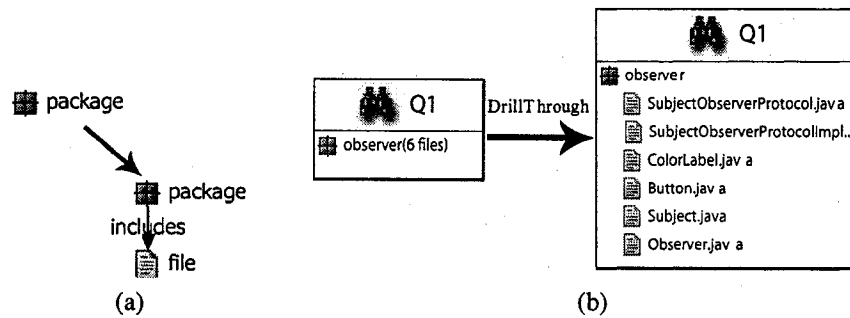


Figure 5.6: Visualization of a drill-down operation.

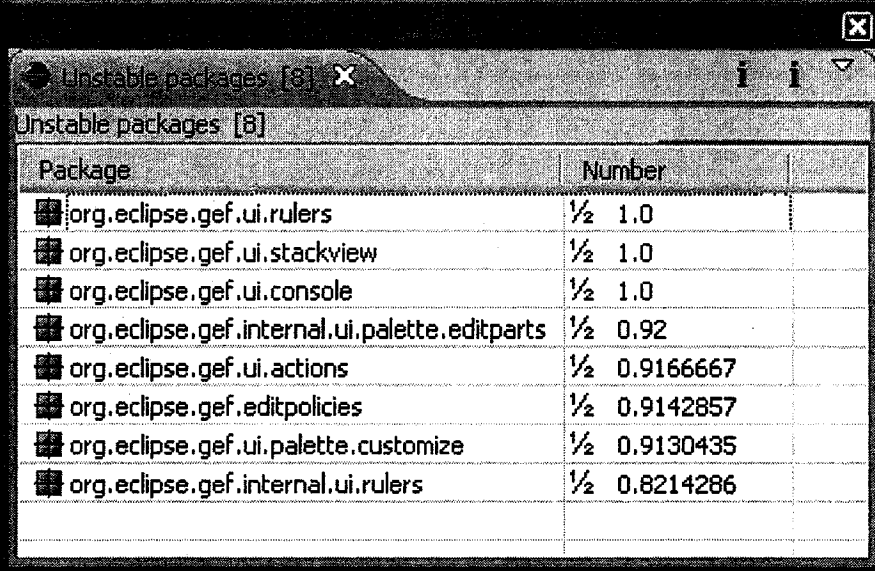
relationship called 'includes' which states the containment relation between programming elements.³ This operation joins the higher level of hierarchy with values of the lower hierarchy level, based on containment (or inclusion) relations. A sample of applying drill-down operation is demonstrated in Figure 5.6. In this figure, developer drills-down the packages in the system to see the types inside each package.

³Modular containment relationships (or inclusion relations) can be expressed in terms as a hierarchy with the level sequence, composed of first participants of 'includes' binary relationships (e.g. file1→aspect1→pointcut1). For example a member rolls up to Interface, Aspect and Class.

5.3 Query result representation

There is a set of visualization methods which we used in order to show up the result data. Their variety will optimize the amount of effort needed to find specific information by the user. These representation forms are table, bar chart, graph, and tree:

Tabular view is the most basic view which represents the query result as set of rows without applying any special formatting. Each tuple is basically shown as a row in a table. This view is beneficial while exploring an overall measurement of the program. Figure 5.7 shows an example of tabular view.



| Package | Number |
|---|---------------|
| org.eclipse.gef.ui.rulers | 1/2 1.0 |
| org.eclipse.gef.ui.stackview | 1/2 1.0 |
| org.eclipse.gef.ui.console | 1/2 1.0 |
| org.eclipse.gef.internal.ui.palette.editparts | 1/2 0.92 |
| org.eclipse.gef.ui.actions | 1/2 0.9166667 |
| org.eclipse.gef.editpolicies | 1/2 0.9142857 |
| org.eclipse.gef.ui.palette.customize | 1/2 0.9130435 |
| org.eclipse.gef.internal.ui.rulers | 1/2 0.8214286 |

Figure 5.7: The query result represented in tabular view.

Bar chart view lets us have an overall view of numerical based query (A query which has a numerical value in a column of its result). It represents a numerical value in the query result in terms of a other values including programming language

elements. For example it can show the amount of effect each aspect has on each class in the system by showing a bar whose height is proportional to the lines of code affected in the class. Figure 5.8 shows an example of bar-chart view.

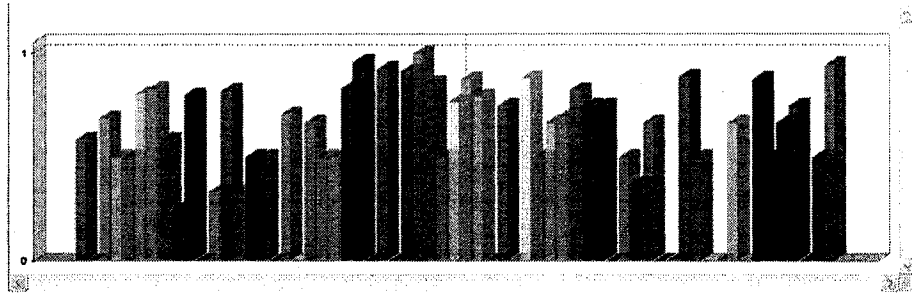


Figure 5.8: The query result represented in bar chart view.

Graph view provides a more natural way for a user to navigate to the query result, because not only information about the components themselves is represented, but also the relationships between those are also captured. For example, the hierarchical relationship between packages, classes, methods could be shown in a graph viewer. Figure 5.9 shows an example of graph view.

Tree view is able to represent the result in a hierarchical fashion. Here the user is able to expand and collapse each node. Figure 5.10 shows an example of tree view.

UML like view uses a combination of labeled graphs and textual annotations, and could be used to display elements, details, and relationships in software engineering documents. This approach is inspired by Unified Modeling Language (UML) and it is also adopted in the ActiveAspect [22] tool. This view is good when users want detailed information about the result. Figure 5.11 shows an example of UML-like

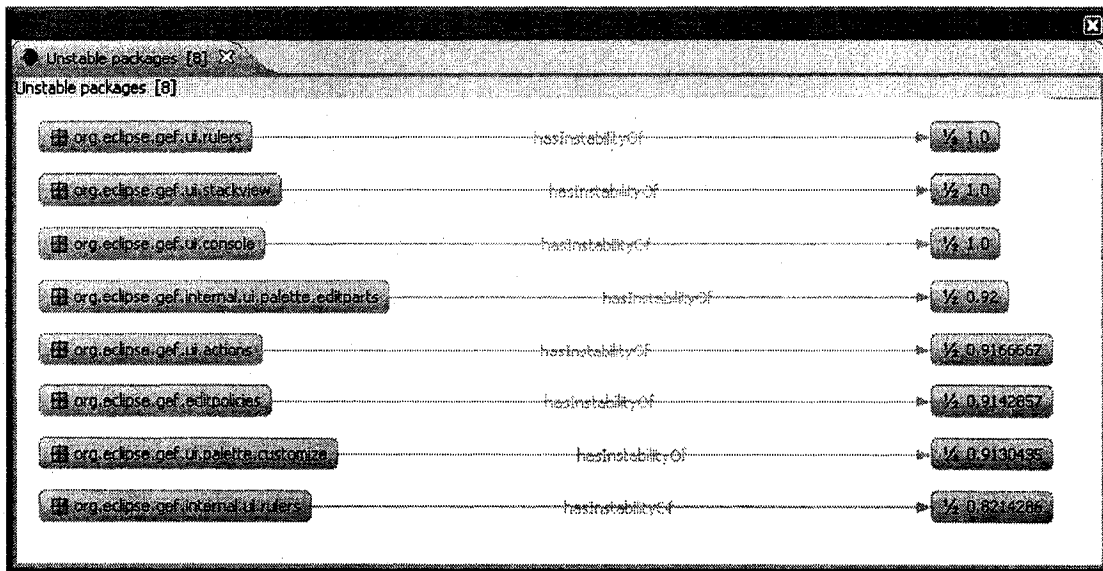


Figure 5.9: The query result represented in graph view.

view.

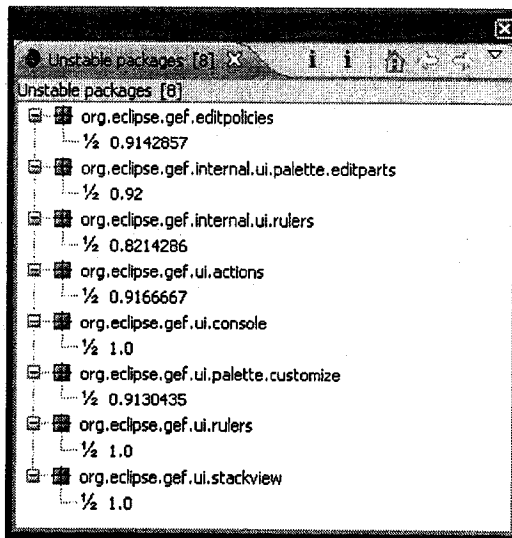
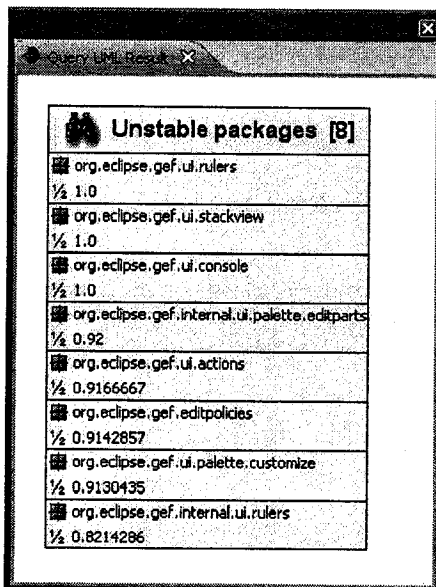
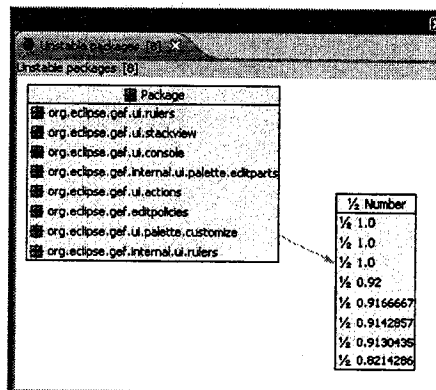


Figure 5.10: The query result represented in tree view.



(a) Attached view.



(b) Detached view.

Figure 5.11: The query result represented in UML-like view.

| | Predicate | Arguments | Meaning |
|-----------|-------------------------|-----------------------------------|--|
| Inclusion | Includes | (project, package) | A <i>program element</i> (e.g. a method) is structurally included inside another <i>program element</i> (e.g. a class) |
| | | (package, file) | |
| | | (file, interface/-class/aspect) | |
| | | (type, method/field) | |
| OOP | Extends | (type, type) | <i>class/aspect/interface</i> extends another <i>class/aspect/interface</i> |
| | Implements | (class/aspect, interface) | <i>class/aspect</i> implements an <i>interface</i> |
| | Accesses | (method, field) | A <i>method</i> gets or sets the value of a <i>field</i> |
| | Invokes | (method, method) | A <i>method</i> invokes (i.e. send a message to) another <i>method</i> |
| AOP | AdvisesInvocation | (advice, method, method) | An <i>advice</i> advises an invocation of a <i>callee method</i> by a <i>caller method</i> |
| | AdvisesFieldAccess | (aspect, method, field) | An <i>advice</i> advises a <i>method</i> which is accessing a <i>field</i> |
| | DeclaresMemberOn | (aspect, type, member) | An <i>aspect</i> declares intertype member (i.e. method or field) into another <i>type</i> |
| | DeclaresParentOn | (aspect, type, type) | A <i>supertype</i> is defined for another <i>type</i> by an <i>aspect</i> through intertype declaration |
| | DeclaresImplementsOn | (aspect, class/aspect, interface) | A <i>type</i> is defined to implement <i>an interface</i> by an <i>aspect</i> through intertype declaration |
| | MatchedBy | (pointcut, method, method) | A call site, identifying a <i>caller method</i> and a <i>callee method</i> is matched by the definition of a <i>pointcut</i> |
| | UsesPointcut | (advice, pointcut) | An <i>advice</i> uses a <i>pointcut</i> to define a set of join points |
| | ControlsInstantiationOf | (class, class/aspect) | A <i>pointcut</i> is used to define control the instantiation of an <i>aspect</i> |
| | IsExposedTo | (type, advice) | An instance of a <i>type</i> is passed through context exposure to an <i>advice</i> block |

Table 5.1: List of logic predicates available in the codebase and their corresponding description.

Chapter 6

Deploying query composition to analysis and measurement

In this Chapter, we discuss the application of our query approach to perform two tasks: structural analysis and measurement of software.

6.1 Structural analysis queries

Structural analysis queries are built in order to extract knowledge about the static structure of a program, like inheritance relationships, dependencies between modules, etc. This category of queries also constitutes the basis for measurement queries discussed later. Examples of these rules are: types depending on other types, methods accessing local fields, the depth of an inheritance tree, methods advised by a point-cut, and virtual method calls (i.e. calls made on behalf of the overriding method by an

overridden one in the super class). We illustrate some detailed examples of structural analysis queries below.

6.1.1 Identifying virtual method calls

One of the most important features of OO methodology is polymorphism. In many cases a behavior is executed from a class which is not lexically defined in the class itself. This behavior could have been defined in any ancestor of the class¹. In this case it is desirable to know the calls made on behalf of the overriding method by an overridden one in the super class. Table 6.1 shows a query to retrieve the source and target methods the virtual calls.

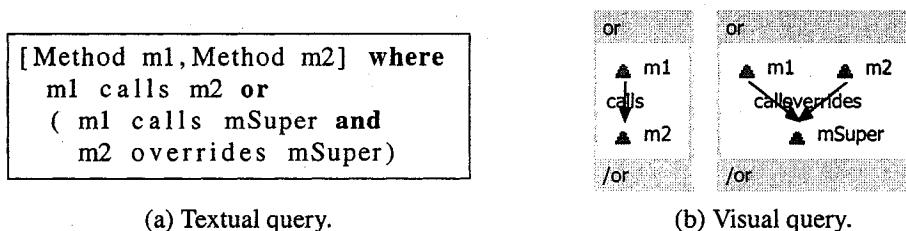


Table 6.1: Query to obtain the polymorphic virtual calls from one method to another.

6.1.2 Types/methods depending on other types/methods

There are situations where we want to identify the dependency between methods or modules (for example in retrieving metrics such as fan-in and fan-out). In case of method dependency, we consider a method is depending on the other one if it actually or virtually (see Table in subsection 6.1.1) calls it. This is encoded as a query in

¹Being an ancestor in this context means any type in the inheritance hierarchy which is in the transitive closure of parent relationship with the type

Table 6.2.

```
[Method m1, Method m2] where
polyCalls (m1, m2)
```

(a) Textual query.

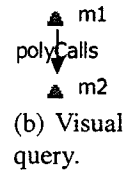


Table 6.2: Query to obtain the dependency between methods.

In case of type dependency, we considered a type dependent on the other one, if one of the following conditions holds between them:

1. type1 inherits from type2 (type2 is super type of type1).
2. type1 has a field of type2.
3. type1 has a method whose return type is type2.
4. type1 has a method which has a parameter of type2.
5. type1 has a method which throws exception of type2.
6. type2 has a method called by a method in type1.
7. type1 reads or writes a field of type2.

The type dependency is encoded as a query in Table 6.3.

6.1.3 Methods accessing local fields

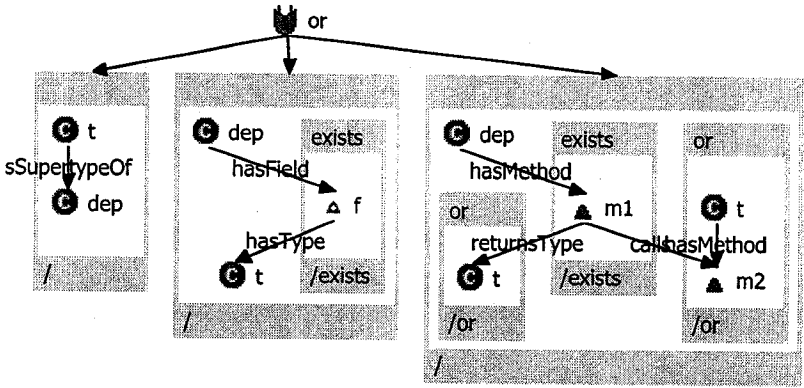
In many cases, it is important to know the methods accessing a field. For example in computing incoming dependencies of a field, the number of accessing methods is

```

[Type dep, Type t] where t isSupertypeOf dep
or exists[Field f] where
  dep hasField f and f hasType t
or exists[Method m1] where dep hasMethod m1 and
  ( m1 returnsType t
  or exists[Method m2] where
    t hasMethod m2 and m1 calls m2)

```

(a) Textual query.



(b) Visual query.

Table 6.3: Query to obtain the dependency between types.

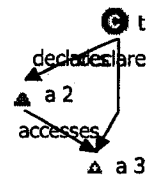
important. Also sometimes, it is important to know if there is any write access to a field while another method reads the same field. Although *accesses* is defined as a primary predicate in our system, we construct two other important compound predicates based on it:

1. *accessesLocalField*: Table 6.4 identifies methods accessing field 'f' defined in the same type as the methods.
2. *accessedField*: Table 6.5 returns all fields accessed by a local method

```

accessesLocalField (t,m,f) ::=
[Type t, Method m, Field f]
where
  m accesses f and
  t declares m and
  t declares f
  
```

(a) Textual query.



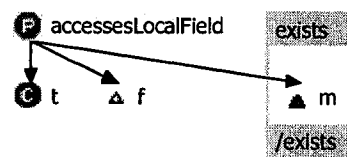
(b) Visual query.

Table 6.4: Query to obtain the methods accessing a local field from a type.

```

[Type t, Field f] from
exists [Method m] where
  accessesLocalField (t,m,f)
  
```

(a) Textual query.



(b) Visual query.

Table 6.5: Query to obtain the fields accessed by a local method.

6.1.4 Methods accessing the same field

In some cases we need to know the distinct methods within a class, which access the same field. The query *sharingFieldMethods* in Table 6.6 identifies methods in the same type and accessing a common field 'f'.

| | |
|---|--|
| <pre>sharingFieldMethods(t,m1,m2)::= [Type t,Method m1, Method m2] from exists[Field f] from m1 accesses f and t declares m1 and m2 accesses f and t declares m2 and m1 != m2</pre> | |
|---|--|

(a) Textual query.

(b) Visual query.

Table 6.6: Query to obtain pairs of methods accessing a common field.

6.2 Measurement queries

The provision of measurements is vital to software development and maintenance. There exists many scenarios in which developers and maintainers need to select and view specific modules having certain measures. One common example is extracting information on the quality and the complexity of the program. Often the complexity of a system depends on a number of measurable attributes such as inheritance, coupling, cohesion, polymorphism, and application size. In this section we are going to investigate the applicability of the proposed query language to perform program measurement. This is done through providing a library of OO measurements encoded in our query language.

The following measurement queries are based on some of the metrics presented in [6, 44, 65].

6.2.1 Queries about cohesion

One of the most important metrics in OO context is obtaining the cohesion of the modules in the system. Informally a cohesive type would be one whose methods are strongly related to perform a specific task leading to a more focused module. In other words, the class is said to have high cohesion if its methods are similar in many aspects. This is usually a good indication that the class's methods perform a variety of unrelated activities. Currently, there exist more than one formal definition for cohesion, however all approaches consider non-cohesive methods, those which often using unrelated sets of data.

Lack of cohesion, Chidamber and Kemerer method

One way in which cohesion can be measured is using 'lack of cohesion metric' (LCOM1) introduced by Chidamber & Kemerer [6]. LCOM1 is calculated as follows:

Step 1) Consider P and Q as two numeric variables equal to zero.

Step 2) Take each pair of methods in the class. If they access disjoint sets of instance variables, increase P by one. If they share at least one variable access, increase Q by one.

Step 3) LCOM1 is computed as follows:

$$LCOM1 = \begin{cases} P - Q & \text{if } P \geq Q, \\ 0 & \text{otherwise.} \end{cases}$$

Step 4)

$$\text{A class is } \begin{cases} \textit{cohesive} & \text{if } LCOM1 = 0, \\ \textit{incohesive} & \text{if } LCOM1 \geq 0. \end{cases}$$

Thus, classes with higher LCOM1 values are attempting to achieve many different objectives, and behave in less predictable ways than classes that have lower value. We formulated the LCOM1 metric already defined in the literature in our query language as shown in Table 6.7. The reader should notice that 'distinctMembers' and 'shareField' queries are reused from section 6.1.

Lack of cohesion, Henderson-Sellers method

Another method to retrieve the cohesion of a type is discussed in [44]. The underlying intuition of this method is the same as the one discussed earlier, which means a class is cohesive if many methods accessing the same fields in it. However the formulation is a bit different, as follows:

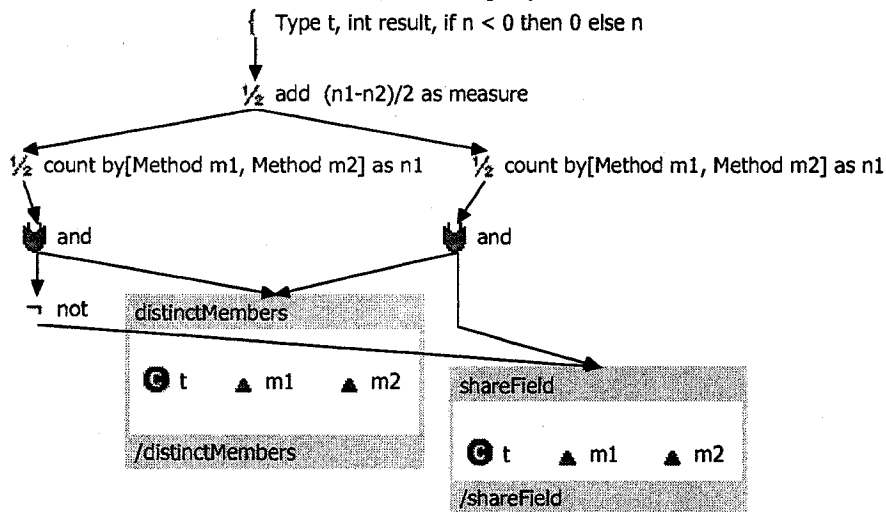
Consider, function *accessNum(f)* which computes the number of methods that access

```

let int n be(
  (count by(Method m1, Method m2) where
    distinctMembers(t,m1,m2) and
    not(shareField(t,m1,m2))
  )
  -
  (count by(Method m1, Method m2) where
    distinctMembers(t,m1,m2) and
    shareField(t,m1,m2)
  )
)/2;
[Type t, int result, if n < 0 then 0 else n]

```

(a) Textual query.



(b) Visual query.

Table 6.7: Query to obtain Chidamber and Kemerer Lack of Cohesion.

each field (f). Then take this function's mean over field f in set of fields in class (call it `averageAccessNum()`). Also compute the total number of methods in the class as `methodNum`.

This is formally denoted as:

1. `methodNum` = number of methods in class
2. `accessNum(f)` = number of methods that access field f
3. `averageAccessNum` = mean of `accessNum(f)` over f

We then define LCOM of the class under consideration to be:

$$LCOM_{class} = \frac{(averageAccessNum - methodNum)}{(1 - methodNum)}$$

We formulated this metric in our query language as shown in Table 6.8. Readers should keep in mind that 'accessesLocalField' and 'accessedField' queries are reused from section 6.1.

Lack of cohesion in packages

Another important case of applying cohesion metric is in calculating the package cohesion.

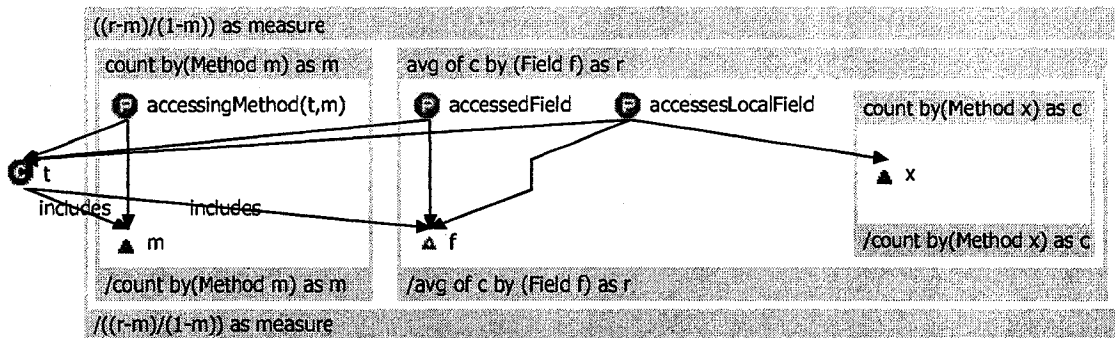
Package cohesion is defined as the average number of outgoing dependency of the types in that package. The query 'packageRelationalCohesion' shown in Table 6.9 is encoding this logic. This query is composed of an inner and an outer query. The inner

```

[Type t,
 float ((r-m)/(1-m) as measure)] from
 [float avg m by(Field f) as r] from
 [int count by(Method me) as m] where
   accessesLocalField(t,x,f) and
   accessedField(t,f) and
   accessingMethod(t,me) and
   t hasMethod m and
   t hasField f

```

(a) Textual query.



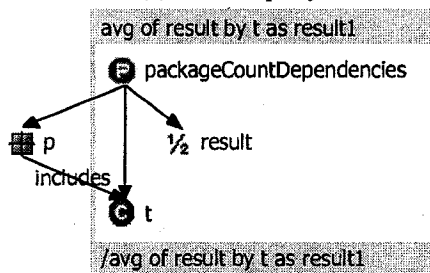
(b) Visual query.

Table 6.8: Query to obtain Henderson-Sellers lack of cohesion metric.

query calculates the outgoing dependency of each type in the package, which is the number of types that the type is depending on. The outer query computes the average of those outgoing dependencies over all types in the package.

```
[Package p,
 (avg of result by (Type t))] from
 [int count by (Type s)
 as result] where
 p includes s and
 t dependsOn s and
 p includes t;
```

(a) Textual query.



(b) Visual query.

Table 6.9: Query to obtain pairs of methods accessing a common field.

6.2.2 Queries about dependencies and coupling

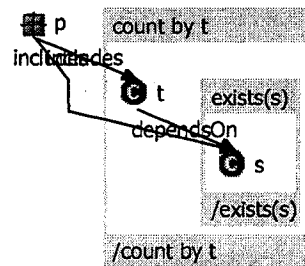
One of the most important metrics involved in software components is the degree of mutual dependence between them, known as incoming/outgoing dependency or afferent/efferent coupling. In this subsection we focus on queries to retrieve the amount of coupling between different parts of the system. Since each program/system element can depend on other ones in many different ways, there could also be different types

of coupling metrics defined on various programming elements and different relationships among them. For each type of programming language elements we considered a separate coupling metric formula. For example we defined a coupling metric formula for the following types of elements: package, type, method, field. Each type of element depends on other types in a different way, however we extracted the definition of 'dependency' using 'depends' predicate (see subsection 6.1.2).

As an example the queries for package incoming dependency and package outgoing dependency are represented in Tables 6.10 and 6.11 respectively. Other types of dependency are discussed in the subsequent subsections.

```
[Package p,
count by(Type t)] where
p not includes t and
exists[Type s] where
p includes s and
t dependsOn s
```

(a) Textual query.

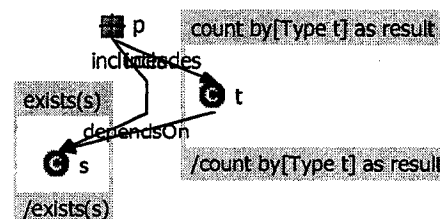


(b) Visual query.

Table 6.10: Query to obtain the number of incoming dependencies for each package.

```
[Package p,int result] from
[int count by(Type t)
as result]
where
p includes t and
exists[Type s] where
p includes s and
t dependsOn s
```

(a) Textual query.



(b) Visual query.

Table 6.11: Query to obtain the number of outgoing dependencies for each package.

Type incoming dependency

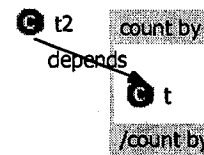
The incoming dependency of a type is the number of types that depend on it. Recall from subsection 6.1.2 that the 'dependency' relationship, in case of types, can originate from one of the following cases:

1. Content coupling: one type accessing local data of another type.
2. Common coupling: types share the same global data (e.g. a global variable).
3. External coupling: types share an externally imposed data format.
4. Control coupling: a method from one type is calling another one in the other type.
5. Subclass coupling: the class is connected to its parent (not vice versa).

High afferent coupling of type indicates that the type has many responsibilities, and, thus changing it, relatively affects more parts of the system. We formulated this metric in our query language as shown in Table 6.12.

```
[Type t, int count by(Type t2)]  
  where t2 depends t;
```

(a) Textual query.



(b) Visual query.

Table 6.12: Query to obtain the incoming dependency of a type.

Method incoming dependency

The method incoming dependency metric (also called as the "fan-in" of a method) is the number of methods that depend on this method. While the query for calculating this

metric is very similar 'type incoming dependency' the definition of 'depends' predicate in this case is different. Methods depend on each other if there is an actual or virtual call between the two methods. We formulated this metric in our query language as shown in Table 6.13.

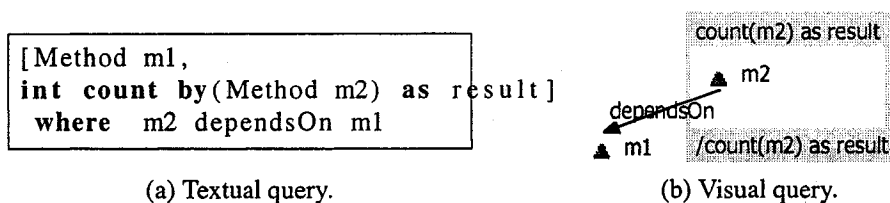


Table 6.13: Query to obtain the "fan-in" of a method.

Method outgoing dependency

The method outgoing dependency metric (also called the "fan-out" of a method) is the number of methods that this method depends on. The 'depends' predicate is the same as the one in 'method incoming dependency'. We formulated this metric in our query language as shown in Table 6.14.

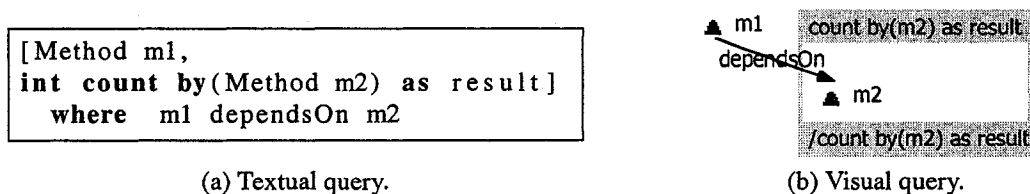
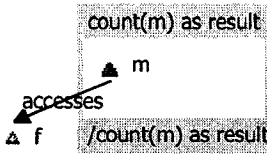


Table 6.14: Query to obtain the "fan-out" of a method.

Field incoming dependency

Similar to other types of incoming dependency metric, the incoming dependency of a field is defined as the number of methods that access it. We formulated this metric in our query language as shown in Table 6.15.

| | |
|---|---|
| <pre>[Field f, int count (Method m) as result] where m accesses f</pre> |  |
|---|---|

(a) Textual query. (b) Visual query.

Table 6.15: Query to obtain the number of incoming dependencies of a field.

6.2.3 Instability of packages

Instability of a package is defined as the number of outgoing dependencies relative to the total number of dependencies. As a result there is an inverse-proportional relationship between 'incoming dependency' and 'package instability'. For example when 'incoming dependency' value is 0, so the instability is 1 (maximum instability). When the instability of a package is high, it will be easy to change it because there are less packages in the system using it, and, therefore less packages should be modified accordingly. We formulated this metric in our query language as shown in Table 6.16.

6.2.4 Abstractness of packages

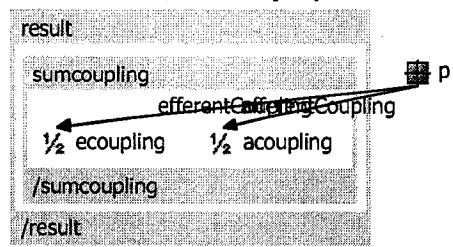
Abstractness of a package is a measure of what portion of the package is composed of abstract types. It is formally defined as the number of the package's abstract types

```

[Package p,
 float ecoupling/(ecoupling + acoupling)
 as instability] from
    efferentCoupling(p, ecoupling) and
    afferentCoupling(p, acoupling)
where ecoupling + acoupling > 0

```

(a) Textual query.



(b) Visual query.

Table 6.16: Query to obtain the the amount of instability of a package.

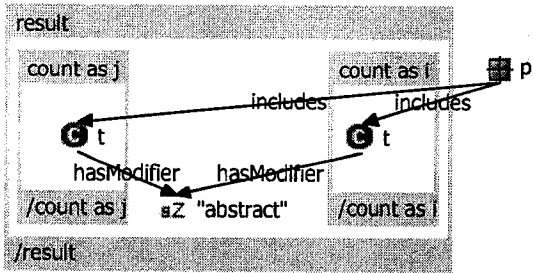
relative to the total number of types in it. A good design is likely to produce a highly abstract and stable package. The reason is that a good abstract package is likely to be used by many concrete ones, resulting in a high incoming dependency and stability. In contrast a low abstract package would contain many concrete classes with a high external dependency and, therefore becomes more unstable. We formulated package abstractness metric in our query language as shown in Table 6.17.

```

[Package p,
 float if i > 0 then j / i else 0 as result]
from
  [p,count by(Type t) as i] where p includes t
  ,[p,count by(Type t) as j] where p includes t
where
  t hasModifier "abstract"

```

(a) Textual query.



(b) Visual query.

Table 6.17: Query to obtain the abstractness of a package.

Chapter 7

Automation and tool support

For a proof of concept, we have implemented a prototypical tool as an Eclipse plug-in to allow maintainers to examine their own software systems. This tool supports storing programs from two programming paradigms, namely object-oriented (Java) and aspect-oriented (AspectJ [7]). Our tool deploys a small in-memory database engine with volatile storage to execute the queries. There are a certain number of tasks, which need to be accomplished via the implementation of the suggested mechanism. This includes determination of source code to be analyzed, mapping graphical notation to queries and vice versa, processing the queries and obtaining the result, illustration of the result using desired visual construct. This implementation could be done in different ways for example as stand alone application, IDE plug-in, or using existing CASE tools. We chose the second approach for two reasons: 1) Eclipse is highly used by developers and choosing this approach will allow them to use our tool without having to switch between

different IDEs. 2) We were able to use existing UI components and frameworks, such as tree views, graph layouts, and graphical editing framework globally available. Thus, we used Eclipse [7] as our platform which has a rich set of plug-ins and frameworks available for open source development.

The tool is architecturally composed of three separate components, each deployed as a separate eclipse plug-in listed below, each will be described in the subsequent sections of this Chapter:

The fact extractor reads the source code and assembles a model (creates a database) composed of a collection of nodes and relations (facts).

The textual query interpreter parses the queries according to the query syntax.

The visual query editor is used to assemble queries by drag-and-drop operations.

The result viewer is capable of representing the query result in different formats, such as graph, tree, and table.

7.1 The fact extractor

The fact extractor reads the source code and assembles a model (creates a database) composed of a collection of nodes and relations (facts). The transformation process from source code to facts is transparent to the users and is done in three steps. First, the AST corresponding to each compilation unit of the program (.java and .aj files) is

retrieved and traversed. Second, the abstract structure model (ASM) of the program¹ is retrieved and traversed. Third, the extracted information from these steps is translated to facts. These facts are then added to the model and used during the query execution process. Traversing the AST is performed by depth-first traversal deploying the Visitor design pattern.

Figure 7.1 depicts the storage of a sample program, by placing nodes based on inclusion relationship (direct lines) and depicting AO and OO relationships among them afterwards (dashed lines).

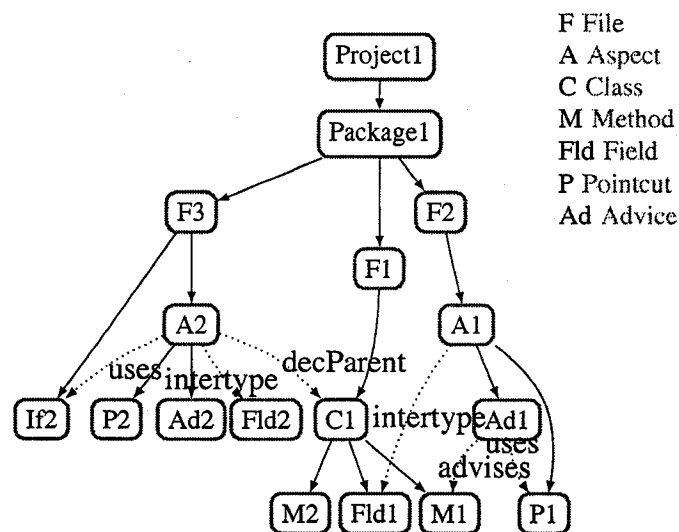


Figure 7.1: A storage of a sample source code in the codebase, representing nodes and relationships between them.

¹The abstract structure model (ASM) of a program, currently part of AspectJ compiler [4], conceptualizes programs in terms of high level programming constructs (e.g. aspect, method, and class) and the relations between them. For example, in aspect-oriented program it contains structural information obtained after parsing joinpoints.

7.2 Implementation of the visual query editor

Figure 7.2 Shows the overall structure of visual query composer plug-in and its dependencies to internal eclipse libraries (mainly GEF).

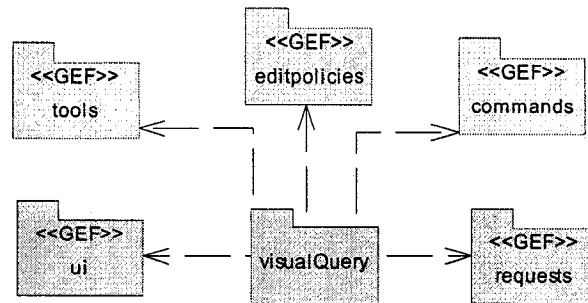


Figure 7.2: The overall structure of visual query composer plug-in and its dependencies to internal eclipse libraries.

Figure 7.3 shows the internal structure of visual query composer. Most of the packages extend classes from corresponding GEF packages with the same name. The overall interaction of our query composition plug-in is similar to any other GEF based application as follows:

1. Asking the current figure which the mouse is operating on and finding its corresponding controller object (also called EditPart in GEF terminology).
2. Asking the Editpart to show feedback and hints based on their role in the interaction.
3. Generation of a request as a result of user interaction and sending it to Editparts. Requests are generated by either tools in the palette (e.g selection tool) or actions in the popup or toolbar (e.g delete).

4. Asking the Editpart to return a command for the given request. This command is encapsulating changes to the model.
5. Putting the command on the application command stack.
6. Picking from the command stack and executing it in a way that can be undone and redone by the user.

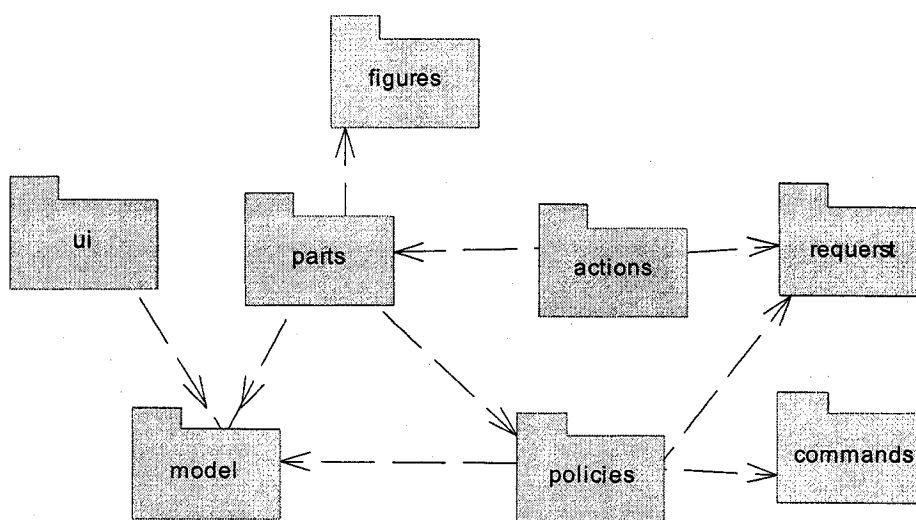


Figure 7.3: The packages of visual query composer tool.

7.2.1 Model package

The model is a set of data structures that support diagramming needs and are used as a backbone of graphical notation. The model is responsible for keeping different views such as the query tree view, calculus based editor, and graph based query composer synchronized using a notification mechanism. It also helps in persisting the graphical queries. The structure of model package is shown in Figure 7.4. QueryElement,

Subquery, Reference, and QueryDiagram are main domain model classes.

QueryElement is visual representation of a programming language element that could be used in query's visual representation. QueryElement instances are added to the model by dragging and dropping their corresponding graphical symbol from the palette into the query diagram. A Reference is the connector between two QueryElements which is shown in the diagram using an arrow between two symbols and is conceptually representing a predicate applied to two terms. A Subquery is representative of an abstract query and is a tool to reuse the already defined queries as predicates. Its graphical representation is a box, containing the QueryElements (free variables of subquery) introduced in its query. Each of its sub-elements is reusable in the outer query. QueryDiagram class represents the whole model by using a single instance (singleton pattern) as the parent of every QueryElement instance in the diagram. Object of type QueryDiagram is not visualizable, however it is used to obtain the rest of diagram when visualization, execution and transformation of visual query to textual representation. Figure 7.4 shows the data structure of the model in terms of classes and their relationships.

7.2.2 Actions package

This package contains the actions usually displayed on the toolbar, menubar, or popup context menu. This package is responsible for assigning the general actions such as redo, undo, and delete their corresponding icons in toolbar and context menu.

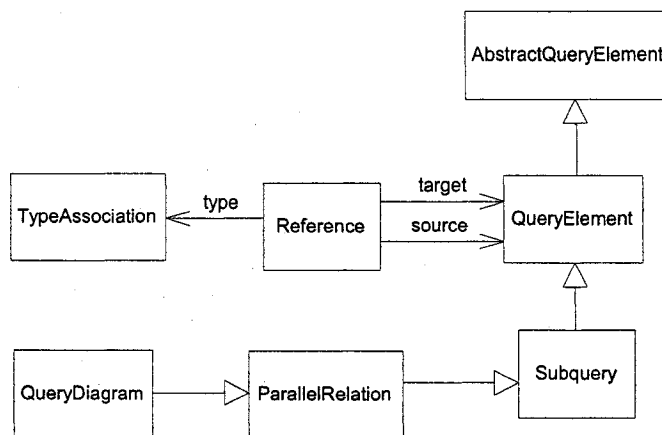


Figure 7.4: Data structure of model in terms of classes and their relationships.

7.2.3 Figures package

Classes of this package are used to make a view of the model by extending and using 'draw2d' figures. Each figure class is corresponding to a model class and encapsulates the way the model objects should be drawn. These figures are either composite (e.g a box) or simple (e.g. labeled icon). For example each Subquery instance in the model is associated with a SubqueryFigure and this figure may have children of type QueryElementFigure.

7.2.4 Parts (controller) package

The controller in GEF terminology is called `EditPart`. Classes in parts package are controllers responsible for editing the model based on user interactions and update the view accordingly. Usually there exists one controller per visualized model object, meaning that there is a reference between each controller object and a model object from one side and a figure object from the other side. Moreover `EditParts` could

reuse some logics by instantiating helper classes from policies package.

7.2.5 Policies (controller helpers) package

Editparts contain helpers called `EditPolicies`, which are common editing logics reused for the editing task. These policies determine how the requests originated from user actions are handled. Such request could be dragging and dropping, selecting, direct editing, deleting, moving, resizing, copying and pasting a component or connection or changing connection endpoints. These policies will generate model-change logics encapsulated in commands which is then returned to the user and stacked to be processed by the UI engine.

7.2.6 UI package

This package is the main point of interaction of user interface with Eclipse. It contains the code for graphical editor, its palette, and query composition wizards.

7.3 Implementation of the textual query editor

7.3.1 Textual query parser

Java Compiler Compiler (JavaCC [10]) is the parser generator we used for encoding the query language grammar. Because JavaCC is a top-down parser, left-recursion is not

allowed in the grammar specification. It accepts a LL1 grammar² and token specification using regular expressions as an input and generates the tokenizer and parser in Java language. The reason we used JavaCC was the language compatibility with the rest of the frameworks used. While parsing the query the AST of query is made using JJTree, which is an additional tool for making user defined parse trees. The resulting tree is mapped to operations on model according to the semantics defined in the interpreter. The maximal structure of parsed query in terms of the nodes of its AST is demonstrated in Figure 7.5.

In order to show the Backus-Naur form (BNF) of the grammar in our implementation, we follow the convention of the BNF definition shown in table 7.1.

| Form | Meaning |
|---------------|--|
| <SYMBOL> | SYMBOL is a definition of token and must be substituted |
| SYMBOL | SYMBOL is reserved word or symbol and must be typed as it is |
| S1 S2 | either S1 or S2 can be used |
| (SYMBOL)? | SYMBOL is optional |
| (SYMBOL)* | SYMBOL may appear zero or more times |
| (SYMBOLS) | grouping SYMBOLS as one unit for high precedence |

Table 7.1: BNF convention used to represent the query language grammar.

There are five token definitions: <EOF> for end-of-file; <IDENTIFIER> for identifier; <INTEGER_LITERAL> for integer constants; <FLOAT_LITERAL> for floating constants; and <STRING_LITERAL> for string constants.

$\langle \text{QueryExpression} \rangle \rightarrow \langle \text{Projection} \rangle ;$

²An LL parser is a top-down parser which parses the input from Left to right, and constructs a Leftmost derivation of the sentence (as opposed to LR parser). LL1 parsers use one token of look-ahead when parsing a sentence and, thus, only need to look at the next token to make their parsing decisions (although JavaCC is not limited to this pure definition).

⟨Projection⟩ → ⟨Projector⟩ (**from** ⟨Projection⟩ | ⟨Selection⟩) | ⟨Selection⟩
 ⟨Selection⟩ → **where** ⟨Term⟩ [**group by** ⟨ExpressionList⟩ ⟨Packing⟩]
 ⟨Projector⟩ → (⟨QuantifierOperator⟩)? [(⟨VarDeclarationList⟩)?]
 ⟨QuantifierOperator⟩ → **forall** | **exists**
 ⟨VarDeclarationList⟩ → ⟨VarDeclaration⟩ (, ⟨VarDeclaration⟩)*
 ⟨VarDeclaration⟩ → ⟨UIDENTIFIER⟩ ⟨IDENTIFIER⟩ ³
 ⟨Term⟩ → ⟨ConditionalAndTerm⟩ (**or** ⟨ConditionalAndTerm⟩)*
 ⟨ConditionalAndTerm⟩ → ⟨EqualityExpression⟩ (**and** ⟨EqualityExpression⟩)*
 ⟨EqualityExpression⟩ → ⟨RelationalExpression⟩ ((**==** | **!=**)⟨RelationalExpression⟩)*
 ⟨RelationalExpression⟩ → ⟨Expression⟩ ((< | > | <= | >=) ⟨Expression⟩)*
 ⟨Expression⟩ → ⟨MultiplicativeExpression⟩ ((+ | -)⟨MultiplicativeExpression⟩)*
 ⟨MultiplicativeExpression⟩ → ⟨UnaryExpression⟩ ((* | / | %)⟨UnaryExpression⟩)*
 ⟨UnaryExpression⟩ → ! ⟨UnaryExpression⟩ | ⟨PrimaryExpression⟩
 ⟨PrimaryExpression⟩ → ⟨Literal⟩ | ⟨PredicateCall⟩ | ⟨Identifier⟩ | (⟨Expression⟩)
 | ⟨Function⟩ | ⟨IfThenElseExpression⟩ | ⟨AggregateExpression⟩
 ⟨AggregateExpression⟩ → (**count** | **sum** | **avg** | **max** | **min**) (⟨Expression⟩)
 ⟨Function⟩ → (**sqrt** | **sin** | **asin** | **cos** | **acos** | **tan** | **atan** | **sinh** | **cosh** | **abs**)
 (⟨Expression⟩)

³In the non-terminal VarDeclaration the first identifier is the type of the variable and the second is its name

$\langle \text{PredicateCall} \rangle \rightarrow = \langle \text{Identifier} \rangle \langle \text{Predicate} \rangle \langle \text{Identifier} \rangle$

$\langle \text{Predicate} \rangle \rightarrow \langle \text{IDENTIFIER} \rangle$

$\langle \text{Literal} \rangle \rightarrow \langle \text{INTEGER_LITERAL} \rangle \mid \langle \text{FLOATING_POINT_LITERAL} \rangle \mid$

$\langle \text{STRING_LITERAL} \rangle \mid \text{null} \mid \text{true} \mid \text{false}$

$\langle \text{IfThenElseExpression} \rangle \rightarrow \text{if } \langle \text{Expression} \rangle \text{ then } \langle \text{Expression} \rangle \text{ else } \langle \text{Expression} \rangle$

7.3.2 Textual query editor

Building a textual editor for a new programming language is a difficult task. Such an editor should usually support capabilities such as syntax highlighting, content assistance, error markers, and an integration with outline view. To this end there is already a few available tool support to automate the text editor construction [18]. From this set we used the IDE Meta-tooling Platform (IMP [9]) which is specifically implemented as an Eclipse tool and is able to support developers in implementing the editor services according to the defined language grammar and AST representation. It also handles static program analysis (pointer analysis, type analysis, etc.) in support of the editor services.

We customized the IMP provided editor to match it to the requirement of our environment. This first important capability provided by the editor is syntax highlighting according to which each piece of text in the editor is rewritten with special color according to type of its underlying token. In our case, tokens are divided into keywords, identifiers and literals; Moreover, literals are divided into numbers and string. Each of these token types is written with a different color. The collaboration with IMP in this

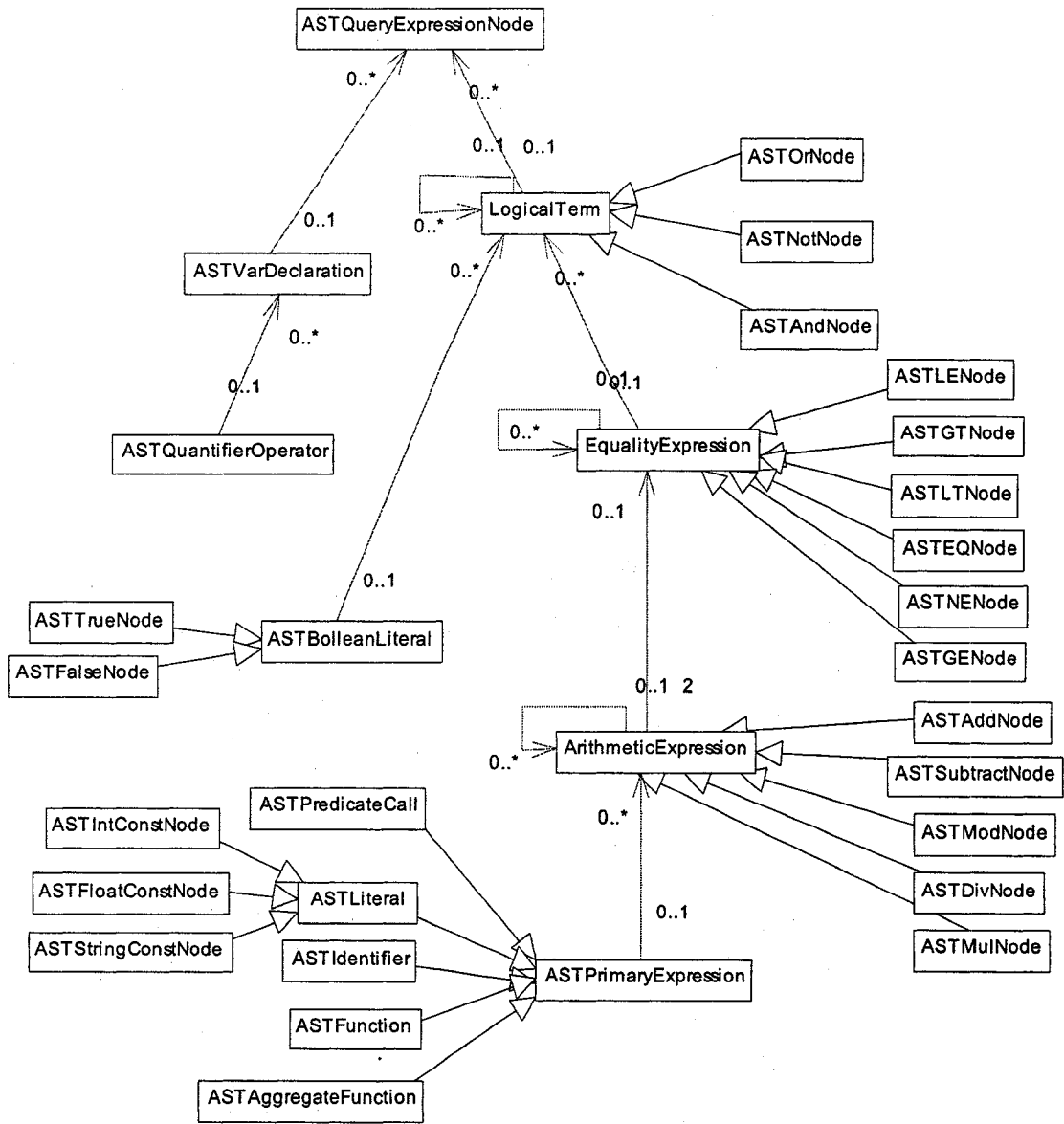


Figure 7.5: The maximal structure of parsed query in terms of its nodes.

case is done by providing the implementation for the 'getColoring' method as follows:

```
1 public TextAttribute getColoring(IParseController  
2 controller, Object o)  
3 {  
4     switch (token.getKind()) {  
5         case TK_IDENTIFIER:   
6             return identifierAttribute;  
7         case TK_NUMBER:   
8             return numberAttribute;  
9         case TK_DoubleLiteral:   
10            return doubleAttribute;  
11         default:   
12            }  
13 }
```

another important feature of the textual query editor is the integration with Outline View of the IDE to show the hierarchical structure of the query. This is useful when a developer wants to see the overall structure of the query specifically in the case of nested queries. This feature is provided through the IMP Outlining Service.

The editor also gives developers the ability to collapse regions of query which are marked by little + or - annotations to the left of the text. This feature is implemented using IMP Text-Folding Service by defining the nested queries as foldable regions, by associating the appropriate query AST element which the editor should consider as foldable.

A developer is also able to navigate from an identifier within the query to the declaration of that identifier in the query variable declaration section. With CTRL-mouseover the definition of the variable is revealed and the developer can navigate it later. This is provided through IMP Hyperlinking Service which supports hyperlinking between regions of text in editors for the language.

The editor is also able to suggest text completion when provided with a prefix string, depending on the context in which the keyword is given ('content-assist' feature). For

example, a developer can obtain a list of visible query variables by typing the first few characters of that variable and then type CTRL-Space followed by selecting one of those choices and hit return to apply that choice. The same could be done with keywords or other syntactic constructs. The implementation of this feature is done with help of IMP Content-Proposer service.

When the cursor is positioned over an appropriate piece of source code in the editor a pop-up will show up providing the developer with additional help text. For example it provides information for variables, showing their declaration. This facility was implemented through the IMP Hover-Helper service.

Text formatting and automatic indentation is also provided so that a developer can choose a region and have it formatted based on the formatting specification defined. The formatter has used the IMP formatting specification language called "Box" which translates the AST to a new textual representation by replacing and modifying non-important tokens.

7.4 Query result evaluation

We implemented our own libraries for evaluation of calculus-based queries and algebraic operators (i.e. selection, join, etc). The query is evaluated by one of the variable substitution strategies; that is, strategies which assign values to the variables and evaluate the final result of the expression based on predicates, quantifiers and connectives. These strategies have different performances in time and memory and each demand

certain query format. For example we used the evaluation approach which stresses in memory efficiency by pipelining of query result. Listing 7.2 represents the simplified version of this algorithm adapted for simple conjunctive queries.

```

1 public Boolean evaluate ( List<Map<Variable ,Object>> varValues )
2 {
3   Boolean value;
4   Variable var=pickNextUnboundVariable ();
5
6   if (quantifier of var is free){
7     //for (each value in the domain of the variable) {
8     for (value=var.type.getDomain()){
9       substitute all occurrences of the variable with the value in the formula
10      boolean result= evaluate(varValues);
11      if (result is true) {
12        put the variables values in the answer set
13      }
14    }
15  }
16  else if (quantifier of var is existential){
17    {
18      value = false;
19      Iterator i=TypeDomainMapper.getDomain(var.type);
20      for (each value in the domain of the variable
21        && value != true ) {
22        substitute all occurrences of the variable with the value in the formula
23        Boolean result=evaluate(varValues);
24        if (result.equals(true)) { value = true; };
25      }
26      return value;
27    }
28  else if (quantifier of var is universal){
29    value = true;
30    Iterator i=TypeDomainMapper.getDomain(var.type);
31    for (each value in the domain of the variable
32      && value != false ) {
33      substitute all occurrences of the variable with the value in the formula
34      Boolean result=evaluate(varValues);
35      if (result.equals(false)) { value = false; };
36    }
37    return value;
38  } else {
39    //formula is now conventional logic composed of only true , false , and connectives
40    return queryStructure .interpret(variableValues);
41  }
42 }
43 }

```

Table 7.2: The algorithm to evaluate calculus-based conjunctive queries.

7.5 Query result representation

In this section we discuss the visualization component of the proposal. We discuss different visualizations and show how they could be adapted to represent the query result. In each approach we mention a reference to an algorithm to faithfully map a table resulted from a query to the specified view. These approaches could be explored to enable viewing software in different levels of granularity and from different perspectives. However, according to the similarities found between them, they can be placed into either of following categories: Tree view, Graph view, Table View, UML like view.

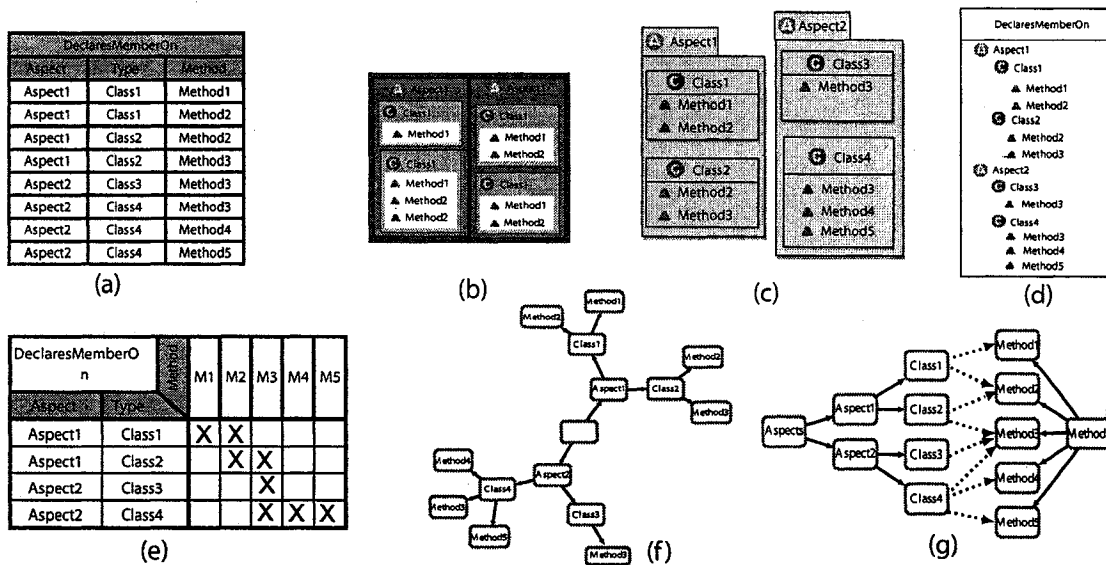


Figure 7.6: Available mechanisms to represent program structures.

7.5.1 Tabular view

The simplest form of 1-dimensional visualization adaptable to represent a query result is tabular format (Figure 7.6(a)). In this method, the query result is shown by a set of

tuples placed horizontally along a vertical axis. Semantically in logic sense, each tuple in the table is a satisfactory substitution of variables involved in the query formula. More advanced, the 2-dimensional adaptation of tabular format is pivot table (Figure 7.6(e)). Pivot tables are showing the data set by splitting it into two axes instead of one (as opposed to tabular format). The same information as simple table could be conveyed in the following manner: at each intersection of each two values from the two axes, there is a sign representing the existence of a tuple(value) formed by concatenation of the two values in the total relationship. As a result, the space provided by Cartesian product of two axes (represented in the rectangular area between the two) is divided into two sets: existent (extant) and non-existent tuples.

7.5.2 Textual tree views

Textual tree views (Figure 7.6(d)), used by [1, 69], are another alternative to represent query results. Transformation to textual tree views is done by mapping between a query result in tabular format to a hierarchical form. In this approach each column in the tabular form is associated with a level in the hierarchy. The computation of parent-child relations is performed according to the distribution of values in columns using an algorithm offered in [21]. It offers a way to convert a normalized relation to a non-normalized form using a recursive application of A-factoring operation. In there, applying A-factoring operation (denoted by a $\lambda(R, A)$) with successively smaller lists A can convert R to a multi-level hierarchical relation. Tree visualizations are able to

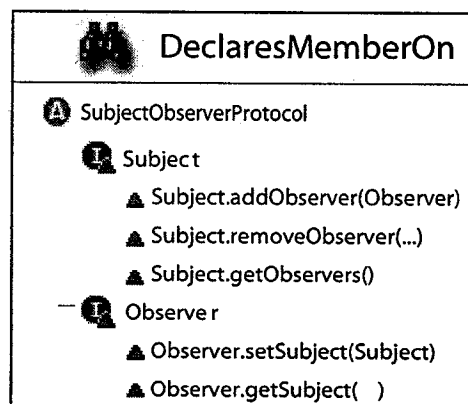
express the query result in hierarchical format, using a set of nested nodes. For example treemap layout (Figure 7.6(b)), used by [59], uses nested rectangles as nodes. The mapping from a hierarchical format to this representation is as follow: each element in the hierarchy is mapped to a rectangle in the view. Children of such element are represented as rectangles, bounded inside it. An algorithmic solution to calculate the positioning and border of the rectangles is provided in [17]. An important feature resulted from the algorithm is that elements having greater number of children occupy more space in the view, and, thus are more eye catching for an observer. In [59] treemap layout has been used to represent modular containment between program elements as well as "Advises-Invocation" relationship between aspects and classes. Another possible tree visualizations layout is radial (Figure 7.6(f)) which offers a space efficient representation of a hierarchy.

7.5.3 UML like view

Box-and-line (UML like) visualizations [46] (Figure 7.6(c)) used by ActiveAspect [22] display elements and relationships grouped in the boxes. Each box is representing an aggregation of original program elements using icons and labels. For example, a single box in this view can contain a package and its classes. The benefit of this approach is the possibility of reducing the visual clutter by 1) showing the containment relationships implicitly by representing the parent of containment relation (e.g. a package) as a box and the child (e.g. a class) as a node inside the box. 2) summarizing actual relations between inner elements of the boxes to abstract relations between boxes

| DeclaresMemberOn (aspect × method × class) | | |
|--|-----------------------------------|------------|
| SOP | Subject.addObserver(Observer) | Button |
| SOP | Subject.removeObserver (Observer) | Button |
| SOP | Subject.getObservers() | Button |
| SOP | Observer.setSubject(Subject) | ColorLabel |
| SOP | Observer.getSubject() | ColorLabel |
| SOPImpl | Button.getData() | Button |
| SOPImpl | ColorLabel.update() | ColorLabel |

(a)

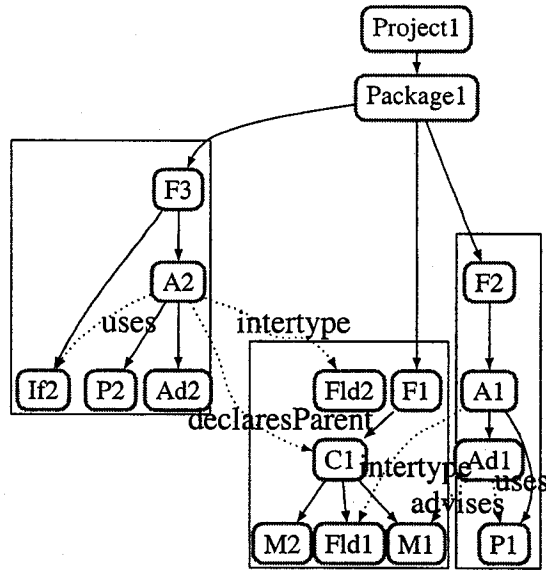


(b)

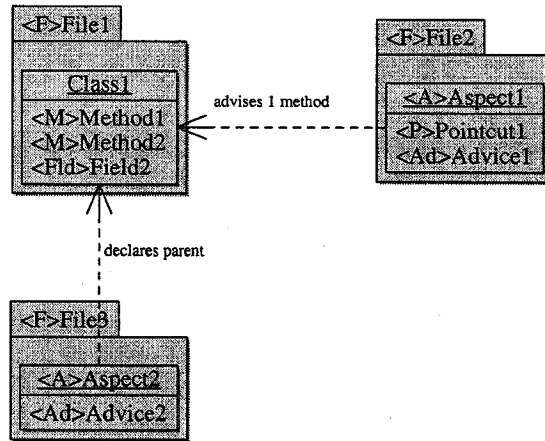
Figure 7.7: Sample mapping of query result to textual tree view.

themselves.

Algorithm: To map a query result into Box-and-line visualization, one could traverse the hierarchy obtained by applying factoring operation in a bottom-up fashion, and use stack-based store-and-flush mechanism to obtain the elements belonging to each box. For the rest of this section we refer to the programming elements in the view represented as a box containing other nodes as `AggregatorNodeKind`, those contained within a box as `AggregatedNodeKind`, and those hidden elements not represented in the view as `InvisibleNodeKind`. This algorithm consists of the following steps: Each `AggregatorNodeKind` is represented in the visual model as an identical box, by the condition that, any two descendants of a `AggregatedNodeKind` node could not co-exist in the same box while their root is represented using a different box; meaning that `AggregatedNodeKind` nodes which belong to the same `AggregatedNodeKind`, can not share a exclusive box from root's. The `AggregatorNodeKind` of a hierarchy is represented as a main caption for the box. The algorithm used here, does the reverse breadth first traversal of the hierarchy from its bottom to the top, while aggregating information about the nodes and relationships. As a result, whenever a `AggregatedNodeKind` node is visited the information about that node, including its type and its relationships, is stored. Whenever a `AggregatorNodeKind` node is visited, it will exhibit all the information gathered from `AggregatedNodeKind` nodes beneath it, which was previously stored. Figure 7.8b represents a UML-like view (b) constructed from a sample model (a).



(a) Intermediate representation.



(b) Actual representation.

Figure 7.8: Formation of boxes in the box-and-line visualization

Implementation: As represented in Figure 7.9 the interface `NodeKindVisitor` provides the facade for every processor which tends to traverse the model based on the node types. The class `PrintingVisitor` implements `NodeKindVisitor` in order to encapsulate core box-and-line visualization code and its model traversal logic. The interface `PrettyPrinter` is a common base for classes tend to provide export mechanisms to exchangeable formats readable by other graph drawing tools. Classes overriding `PrettyPrinter` in order to export the view to another exchangeable format readable by other graph drawing tools. For example `GraphPrettyPrinter` and `MetaUMLPrettyPrinter`, both extend `PrettyPrinter` to provide transformation to Eclipse graph and MetaUML tool formats.

Table 7.3 represents a client code which uses the `GraphPrettyPrinter` class in order to translate the model into a graph representation.

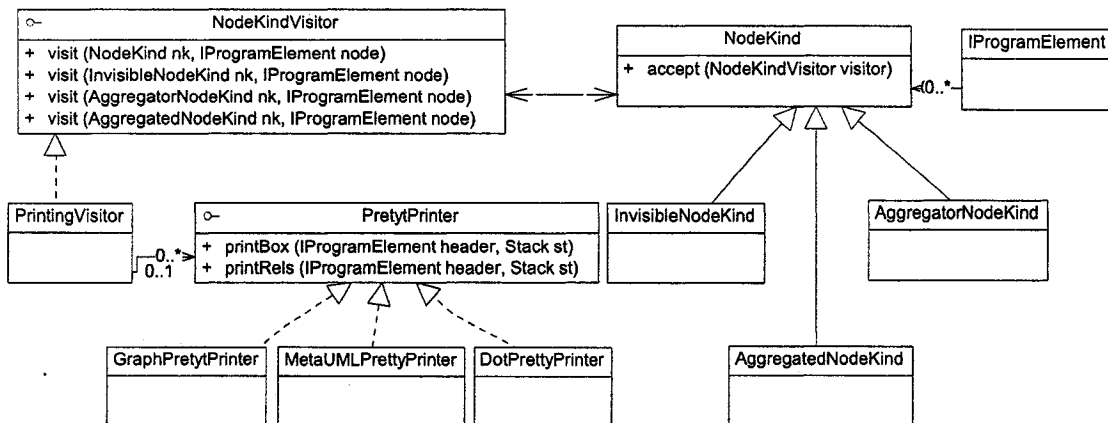


Figure 7.9: Structure of classes involved in the box-and-line visualization.

```

1 private void printModelIntoGraph(Graph graph){
2     PrintingVisitor visitor=new PrintingVisitor(new GraphPrettyPrinter(graph));
3     IProgramElement root=ModelBase.getInstance().getRootElem();
4     HierarchicalModel.getNodeKindForNode(root).accept(visitor, root);
5 }

```

Table 7.3: The code to represent the program hierarchy as compound graph.

7.5.4 Graph view

Compound graphs (Figure 7.6(g)) show the view by separating it into hierarchically related set of elements using parent-child relations, as well as non-hierarchically related ones related by adjacency relations. The mapping between compound graphs and a tabular result could be done by intermediate transformation of data to a pivot table. The resulting compound graph could be constructed by representing each axis of the pivot table in a separate tree. In this approach the data in each axis is transferred to the hierarchical form. Then the hierarchy is used to construct a visual tree. The two resulting trees take the role of axes. Afterward, it is possible to represent the existence of relation between each two element (value) from the trees by an adjacency edge. This approach tends to be very space efficient and a possible visual clutter resulting from adjacency relation, could also be lessened using smart positioning of nodes or bundling of adjacency edges [45].

Chapter 8

Case Study 1: Graphical Editing

Framework (GEF)

As a proof of concept, we deploy our approach over a widely known object-oriented framework, called Graphical Editing Framework (GEF) [8] which is used for building rich, interactive user interfaces within Eclipse IDE specifically aimed at modeling purposes. It uses the Eclipse Rich Client Platform (RCP), and is separated into two plug-ins:

1. Draw2d (org.eclipse.draw2d) - the lightweight toolkit for painting and layout on an SWT Canvas.
2. GEF (org.eclipse.gef) - an interactive MVC framework built on top of Draw2d.

In this case study we focus on the second plug-in while still referring to it as “GEF”.

8.1 Framework overview

The GEF framework provides the link between an application's model and view using a set of controller classes or Editparts. It provides input handlers, such as tools and actions, that wrap actual SWT widgets and turn SWT events originated from these widgets into requests. Requests, encapsulating user interactions and abstracting away the source of interaction, are in turn sent to the controller (or Editpart) responsible for maintaining the view, interpreting requests and turning them into operations on the model (commands). Editparts are asked for a command for a given request. Commands are used to encapsulate model change logic, and returned to the user, stored in the command stack, in a way that can be undone and redone by the user. The model is a plain data structure, possible to be persisted, and able to notify the controller if changed. Figure 8.1 shows a high-level view of GEF by representing its top-level packages in a class diagram.

In the following subsections, we illustrate examples of structural analysis queries and measurement queries applied on the GEF framework.

8.2 Structural analysis queries

8.2.1 Finding the correspondence between model, view, and controller classes

In GEF, Editparts (controllers) associate their view and model, meaning that there is usually a one-to-one association between each controller object and a view and model object. Usually a figure may be compound and composed of several figures. In this case

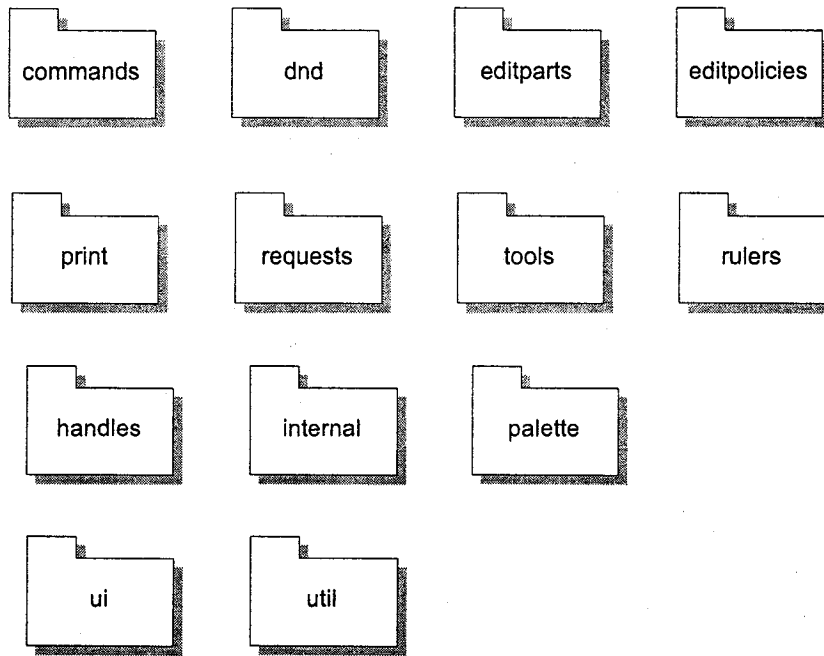


Figure 8.1: Partial UML class diagram of GEF.

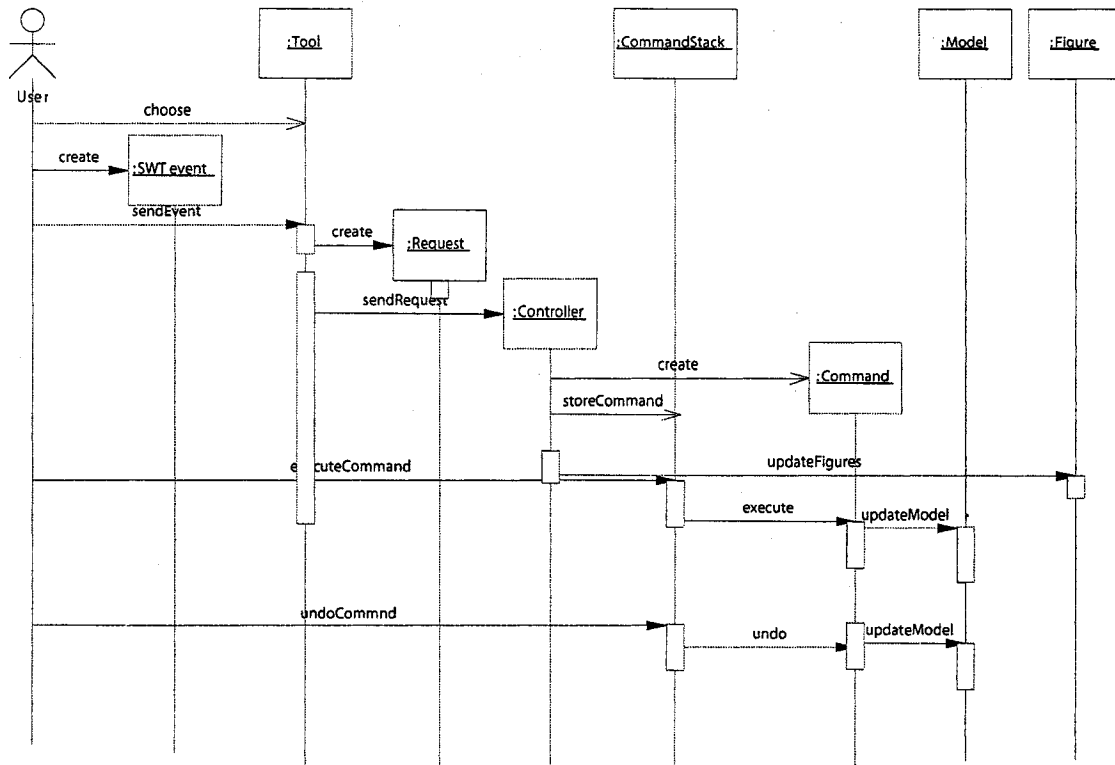


Figure 8.2: UML sequence diagram for overall workflow of GEF.

there is a corresponding Editpart class which contains multiple Editpart children, each corresponding to a figure. Moreover, this corresponds to a similar containment found in the model (See Figure 8.3).

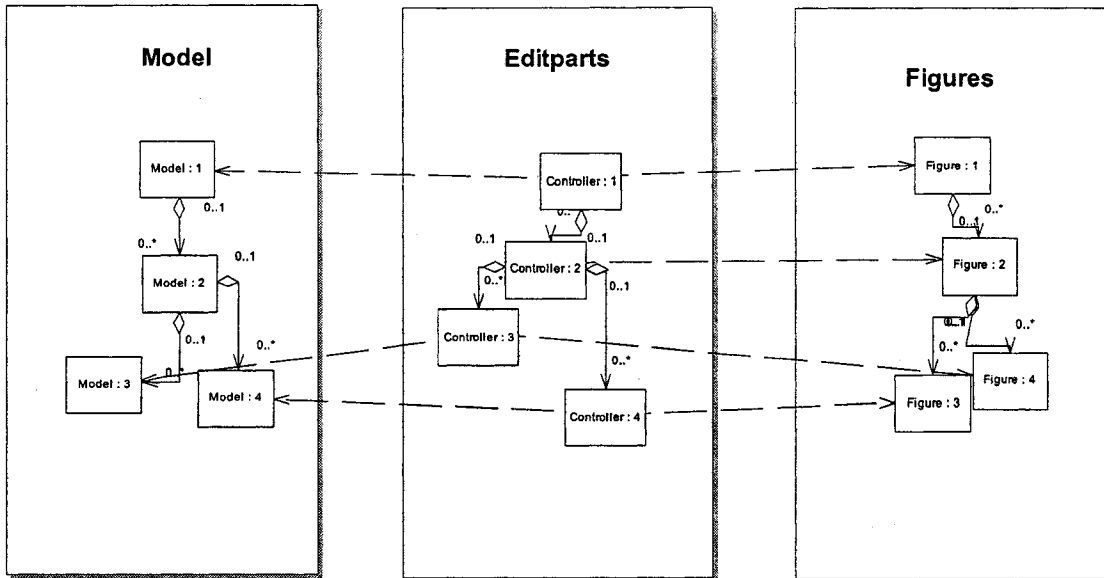


Figure 8.3: The relationship among instances of GEF classes and their role in the MVC architecture.

8.2.2 Finding correspondence between models and Editparts

In order to find the correspondence between models and Editparts, one should find where each instance of a controller is made and its corresponding model object is passed to it. In GEF framework this is done in a class implementing `EditPartFactory` interface. A user can define the classes implementing `EditPartFactory` by simply executing the following query :

```
[Type t] from [Interface i] where
  t implements i and
  i hasName "EditPartFactory"
```

In the flow-example this query has only one result: ActivityPartFactory. A closer look at this class shows how a particular EditPart instance (the argument "context") is created for a model (see Table 8.1)

```
1 public class ActivityPartFactory implements EditPartFactory {
2     public EditPart createEditPart(EditPart context, Object model) {
3         EditPart part = null;
4         if (model instanceof ActivityDiagram)
5             part = new ActivityDiagramPart();
6         else if (model instanceof ParallelActivity)
7             part = new ParallelActivityPart();
8         else if (model instanceof SequentialActivity)
9             part = new SequentialActivityPart();
10        else if (model instanceof Activity)
11            part = new SimpleActivityPart();
12        else if (model instanceof Transition)
13            part = new TransitionPart();
14        part.setModel(model);
15        return part;
16    }
17 }
```

Table 8.1: The code to create different editPart for different model objects in the GEF flow-example .

8.2.3 Finding the correspondence between figures and editparts

In order to find the figure and part correspondence a maintainer needs multiple interactions with different views of the IDE. 1) First s/he needs to find all editparts. This could be done through Hierarchy view of Eclipse IDE by finding all classes extending the AbstractEditPart (see Figure 8.4-1). Second, s/he lists the methods of each of this classes (see Figure 8.4-2). Third, s/he obtains the calls made from the method createFigure () to check which figure class has been created during the execution of this method (createFigure()). This is done by opening the 'Call Hierarchy' view for this method in 'callee hierarchy' mode. As demonstrated in Figure 8.4-3 the

createFigure() in ParallelActivityPart class is making an instance of ParallelActivityFigure. This fact is also possible track in the source code (see Figure 8.4-4).

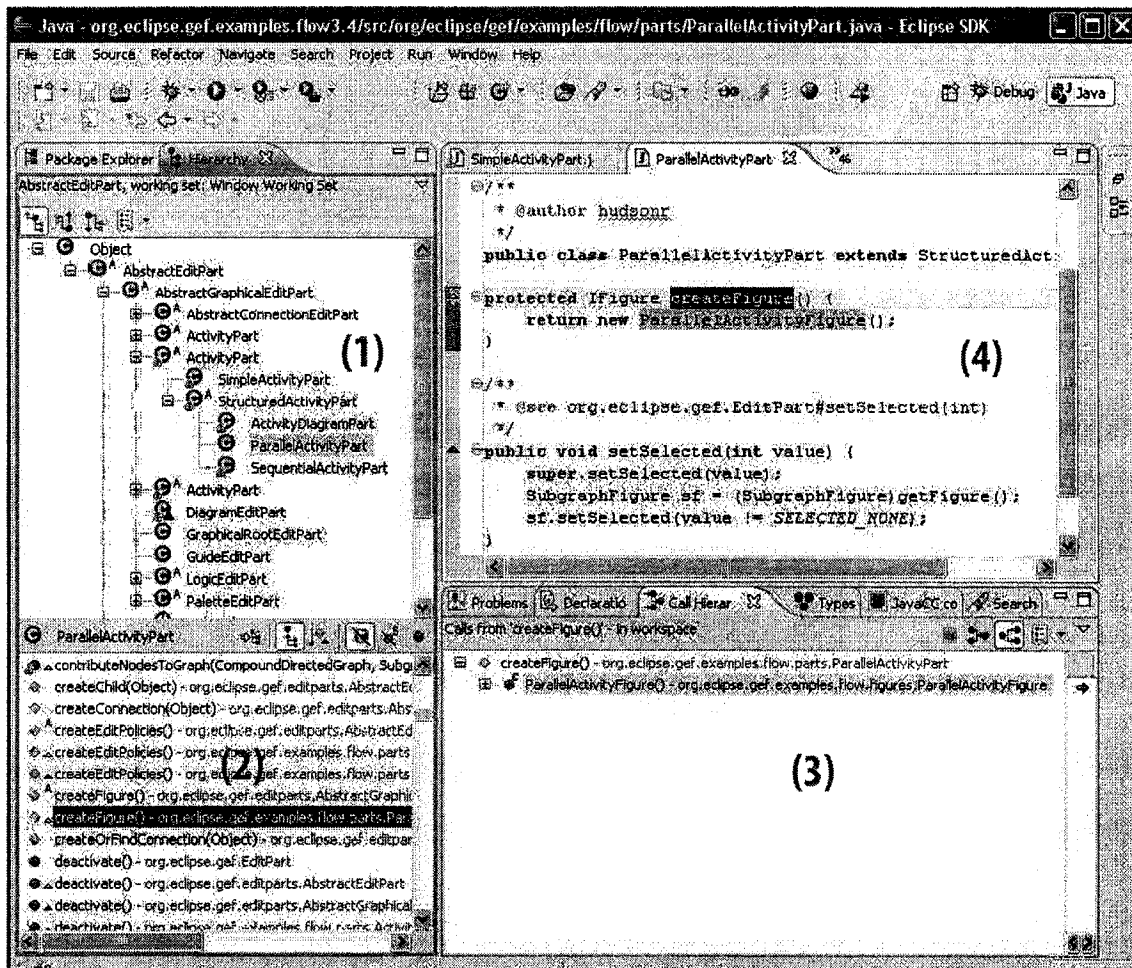


Figure 8.4: Investigation of GEF's flow-example using Eclipse views.

It is obvious from this example that relying only on standard views, the developer has to perform multiple interactions to constitute a single view. In this example the user has to check each Editpart subclass one-by-one, leading to an excessive overhead, proportional to the size of the system. In contrast, the developer can obtain the same

| Type | Type |
|----------------------------|----------------------------|
| SimpleActivityLabel | SimpleActivityLabel |
| Figure | Figure |
| ParallelActivityFigure() | ParallelActivityFigure() |
| SequentialActivityFigure() | SequentialActivityFigure() |

Figure 8.5: The correspondence of the figure classes to controllers, retrieved as a query outcome.

information using only one query represented in the Table 8.2. This query obtains the type of every instantiated object within the body of method `createFigure()` in each subclass of `AbstractEditPart`, if the type of the instantiated object is subtype of `Figure`.

```
[Type controller ,Type figure] from
exists[Method m,Type t ] where
  controller isSubclassOf "AbstractEditPart" and
  figure isSubclassOf "Figure" and
  controller includes m and
  m hasName "createFigure" and
  m instantiates t
```

Table 8.2: Query to obtain the figure classes corresponding to each controller class.

In the GEF-flow example running this query returns the Figure 8.5

8.2.4 Finding possible commands for a given request

As shown in the sequence diagram of the Figure 8.2 Editparts are asked for a command for a given request. As a result the Editparts will decide which command instance to return to what kind of requests. However in GEF, Editparts do not handle editing directly. Instead, developers should define a set of common behaviors which encapsulate the request-command-mapping logic and could be arbitrarily installed on different Editparts. This is to allow editing behavior to be selectively reused across different Editpart implementations. By installing each Editpolicy, the Editpart delegates a given request to its policies and allows them to contribute in handling it. The GEF based code is required to install the appropriate policies in the method `createEditPolicies()` which will be called during the Editpart's creation. In the Table 8.3 the `ActivityPart` associates each editpolicy with an identifier to determine for which request the policy will contribute. For example, it associates the `REQ_DELETE` request to `ActivityEditPolicy` by installing it on the role `COMPONENT_ROLE`¹.

In the Table 8.4 it is shown how the installed policy (`ActivityNodeEditPolicy`) is responding to the request `CreateConnectionRequest` by creating a new instance of the command `ConnectionCreateCommand`². The overall view of the described scenario is represented in Figure 8.6.

¹The `COMPONENT_ROLE` key is used when installing an editpolicy on a component Editpart to fill in the commands for deletion.

²GEF manual asserts that the type of `CreateConnectionRequest` could be `REQ_RECONNECT_TARGET`.

```

1 public abstract class ActivityPart
2   extends AbstractGraphicalEditPart
3   implements PropertyChangeListener , NodeEditPart
4 {
5   protected void createEditPolicies () {
6     installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE, new ActivityNodeEditPolicy ());
7     installEditPolicy(EditPolicy.CONTAINER_ROLE, new ActivitySourceEditPolicy ());
8     installEditPolicy(EditPolicy.COMPONENT_ROLE, new ActivityEditPolicy ());
9     installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new ActivityDirectEditPolicy ());
10  }
11  ....
12 }

```

Table 8.3: The code to setup the edit policies for a given subclass of Editpart.

```

1 public class ActivityNodeEditPolicy extends GraphicalNodeEditPolicy {
2   ....
3   protected Command getConnectionCreateCommand(CreateConnectionRequest request) {
4     ConnectionCreateCommand cmd = new ConnectionCreateCommand();
5     cmd.setSource(getActivity ());
6     request.setStartCommand(cmd);
7     return cmd;
8   }
9   ....
10 }

```

Table 8.4: Retrieving a Command for a given Request using an EditPolicy.

It is clear from the simple example that it takes a lot of effort to finally find the correspondence between a given request (e.g. REQ_RECONNECT_TARGET) and the result command (e.g. ReconnectSourceCommand) for a given user action. Here are the steps to find the answer to this problem for a given GEF based application by a developer:

1. Navigate through all Editparts one by one and look into the policy classes instantiated in the `createEditPolicies()` method, and manually match each policy class with its corresponding role

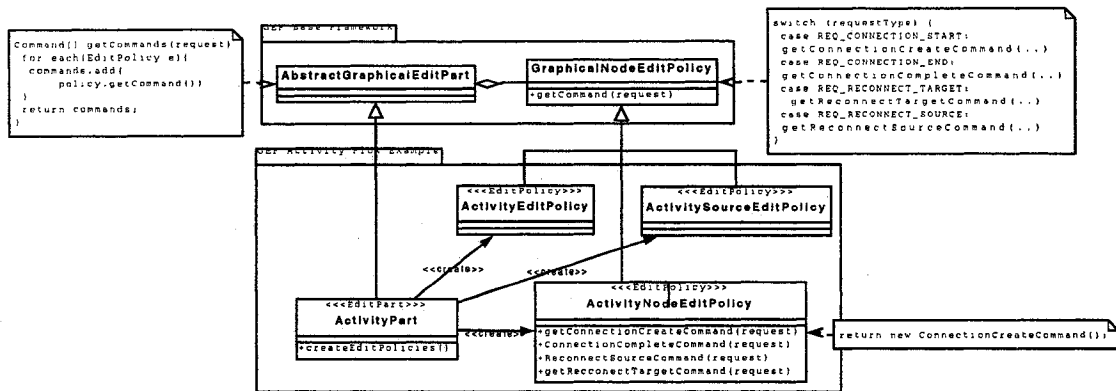


Figure 8.6: An overall view of modules' of a GEF based applications and their dependencies, divided into user defined and GEF core code.

2. Manually check the method `Command getCommand(Request)` in the abstract base class of the Editpolicy (part of GEF framework) to see for which request type which method of the concrete EditPolicy (the one implemented in the application) the request is delegated to.
3. In the found method, check for the instantiation of the Command classes.

Instead of performing these steps using manual navigation through the code the developer can configure the following queries to be reused for any GEF based application.

1. Look into the policy classes instantiated in the `createEditPolicies()` method of each Editpart:

```

[Type editpart,Type policy] from
exists [Method m] where
  editpart isSubclassOf AbstractGraphicalEditPart and
  editpart includes m and
  m hasName createEditPolicies and
  m instansiates policy and
  policy isSubclassOf AbstractEditPolicy
  
```

2. For each request type and a given policy determine which method of the policy will handle the request. Since this fact is known from GEF framework itself, it

can be hard-coded in the query language and reused instead of being dynamically asked from GEF application.

```
[int reqType,  
Type policy,  
Method  
if (policy isSubtypeOf GraphicalNodeEditPolicy) then  
begin  
  if (reqType==REQ.CONNECTION_START)  
    then getConnectionCreateCommand  
  if (reqType==REQ.CONNECTION_END)  
    then getConnectionCompleteCommand  
  if (reqType==REQ.RECONNECT_TARGET)  
    then getReconnectTargetCommand  
  if (reqType==REQ.RECONNECT_SOURCE)  
    then getReconnectSourceCommand  
end else if (policy isSubtypeOf DirectEditPolicy) then  
.....
```

3. instantiation of the Command classes within a given Editpolicy's handler method

```
[Method m, Type command] where  
  command isSubtypeOf Command and  
  m instansiates command
```

Finally, the developer can combine the tree queries as:

$[editpart,policy] \times [policy,reqType,method] \times [method,command]$

and obtain $[editpart,reqType,command]$ relation, representing the command issued for each request type when applied to a particular Editpart (representing the diagram element type).

```

1 public Command getCommand(Request request) {
2     .....
3     if (REQ_CONNECTION_END.equals(request.getType()))
4         return getConnectionCompleteCommand((CreateConnectionRequest)request);
5     if (REQ_RECONNECT_TARGET.equals(request.getType()))
6         return getReconnectTargetCommand((ReconnectRequest)request);
7     .....
8 }

```

Table 8.5: Retrieving a Command for a given Request.

8.3 Measurement queries: Investigating the quality of the GEF code

In this section we use our query tool to investigate the code quality of the specified framework (GEF) by analyzing how code attributes such as abstractness, stability, and incoming dependency are distributed among different parts of the system.

As mentioned in subsection 6.2.4 a good design is likely to produce a highly abstract and stable package. The reason is that a good abstract package is likely to be used by many concrete ones resulting in a high incoming dependency and stability. In contrast a low abstract package would contain many concrete classes with a high external dependency and, therefore becomes more unstable. In this subsection we investigate this fact to measure how much the GEF implementation follows a good design. To perform this, we try to find the amount of correlation among abstract, stable, and with-high-fan-in packages on one side and concrete, unstable, and with-low-fan-in on the other side. To realize this we use the queries discussed in section 6.2 with specific arguments. For example we use the query for obtaining package abstractness and augment it with an additional 'where clause' to obtain abstract and concrete packages separately. The same technique is used to differentiate between stable and unstable, and high fan-in and high

fan-out packages as follows:

First, based on package abstractness formula we can find abstract and concrete packages as follows:

```
[Package p] from
  [float (if i > 0 then j / i else 0) as k ] from
    [p, count by(Type t) as i] where
      p includes t and t hasModifier "abstract"
    ,[p, count by(Type t) as j] where
      p includes t and t hasModifier "abstract"
where :clause
order by k desc
```

$$\text{clause is } \begin{cases} k > 0.20 & \text{Abstract packages} \\ k = 0 & \text{Concrete packages} \end{cases}$$

Second, based on package stability formula we can find abstract and concrete packages as follows:

```
[Package p] from
  [float ecoupling/(ecoupling + acoupling)
  as instability] from
    efferentCoupling(p, ecoupling) and
    afferentCoupling(p, acoupling)
  where ecoupling + acoupling > 0
where :clause
```

$$\text{:clause is } \begin{cases} \textit{instability} < 0.2 & \text{Stable packages} \\ \textit{instability} > 0.8 & \text{Unstable packages} \end{cases}$$

And, third, based on package's dependency formula we can find abstract and concrete packages as follows:


```

[Package p] from
[int count by(Type t) as n]
where
  p not includes t and
  exists[Type s] where
    p includes s and
    : clause1
where : clause2

```

clause 1,2 are $\left\{ \begin{array}{l} \text{'t dependsOn s', } n > 20 \text{ high Incoming dependency packages} \\ \text{'s dependsOn t', } n > 22 \text{ high Outgoing dependency packages} \end{array} \right.$

The categorization of GEF packages based on the mentioned three attributes (stability, abstractness, and dependency) will result in six non-distinct sets of packages each sharing some members with others (represented in Figure 8.7).

By obtaining the correlation between the identified sets, one can conclude about the correlation among possible different categories. We considered the correlation to be the number of shared members between the two sets representing instances of those two categories. The final result represented in Figure 8.8 demonstrates that in GEF framework 1) usually abstract packages are stable and with high incoming dependency. 2) usually concrete packages are unstable and with more outgoing dependency. Thus we can conclude that GEF has a rather overall good design in terms of following the design guideline mentioned in the beginning of the section.

| Package | Number |
|------------------------------|--------------|
| org.eclipse.gef.editpolicies | ½ 0.5151515 |
| org.eclipse.gef.rulers | ½ 0.33333334 |
| org.eclipse.gef.dnd | ½ 0.27272728 |

(a) Abstract packages.

| Package | Number |
|-------------------------------------|--------|
| org.eclipse.gef.ui.console | ½ 0.0 |
| org.eclipse.gef.requests | ½ 0.0 |
| org.eclipse.gef.util | ½ 0.0 |
| org.eclipse.gef.internal.ui.palette | ½ 0.0 |
| org.eclipse.gef.ui.rulers | ½ 0.0 |
| org.eclipse.gef.print | ½ 0.0 |
| org.eclipse.gef.ui.views.palette | ½ 0.0 |
| org.eclipse.gef.ui.properties | ½ 0.0 |
| org.eclipse.gef.ui.stackview | ½ 0.0 |
| org.eclipse.gef.internal | ½ 0.0 |
| org.eclipse.gef.internal.ui.rulers | ½ 0.0 |

(b) Concrete packages.

| Package | Number |
|--------------------------|--------------|
| org.eclipse.gef.rulers | ½ 0.16666667 |
| org.eclipse.gef.commands | ½ 0.11290322 |
| org.eclipse.gef.internal | ½ 0.10204082 |
| org.eclipse.gef.util | ½ 0.09090909 |

(c) Stable packages.

| Package | Number |
|---|--------------|
| org.eclipse.gef.ui.rulers | ½ 1.0 |
| org.eclipse.gef.ui.stackview | ½ 1.0 |
| org.eclipse.gef.ui.console | ½ 1.0 |
| org.eclipse.gef.internal.ui.palette.editparts | ½ 0.92 |
| org.eclipse.gef.ui.actions | ½ 0.91666667 |
| org.eclipse.gef.editpolicies | ½ 0.9142857 |
| org.eclipse.gef.ui.palette.customize | ½ 0.9130435 |
| org.eclipse.gef.internal.ui.rulers | ½ 0.8214286 |

(d) Unstable packages.

| Package | Number |
|--------------------------------------|--------|
| org.eclipse.gef | ½ 174 |
| org.eclipse.gef.commands | ½ 55 |
| org.eclipse.gef.ui.palette | ½ 45 |
| org.eclipse.gef.editparts | ½ 45 |
| org.eclipse.gef.internal | ½ 44 |
| org.eclipse.gef.palette | ½ 43 |
| org.eclipse.gef.requests | ½ 40 |
| org.eclipse.gef.tools | ½ 25 |
| org.eclipse.gef.ui.palette.editparts | ½ 23 |

(e) High fan-in packages.

| Package | Number |
|------------------|--------|
| org.eclipse.g... | ½ 69 |
| org.eclipse.g... | ½ 57 |
| org.eclipse.gef | ½ 46 |
| org.eclipse.g... | ½ 44 |
| org.eclipse.g... | ½ 42 |
| org.eclipse.g... | ½ 39 |
| org.eclipse.g... | ½ 33 |
| org.eclipse.g... | ½ 32 |
| org.eclipse.g... | ½ 32 |
| org.eclipse.g... | ½ 23 |

(f) High fan-out packages.

Figure 8.7: GEF packages divided into six classes based on measurement attributes.

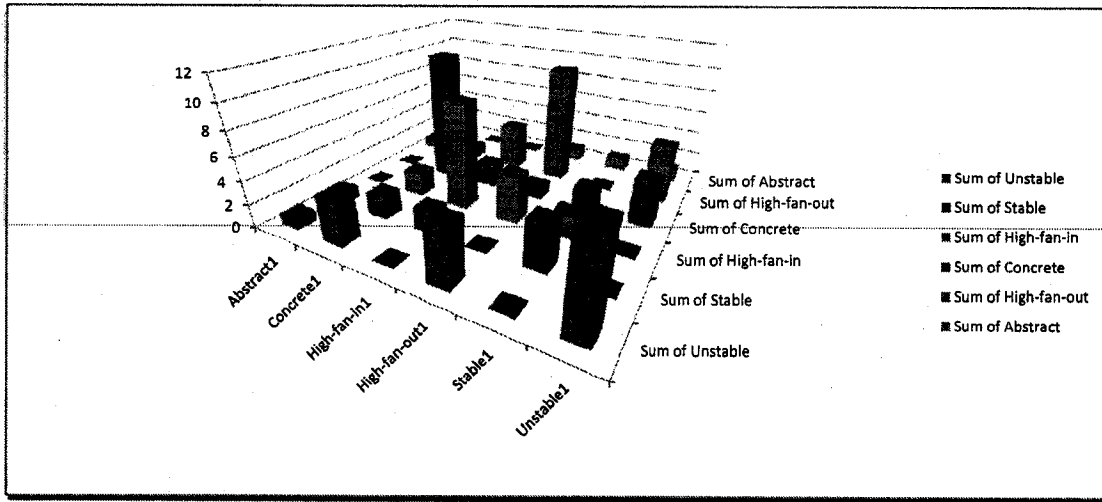


Figure 8.8: A 3D bar chart representing the co-occurrence of different categorical attributes of GEF framework.

Chapter 9

Case Study 2: Aspect-oriented implementation of the Observer design pattern

In this section, we deploy our approach over an aspect-oriented implementation of *Observer design pattern* introduced in [52] and deployed as a part of AJDT package's examples. The main implementation is accompanied by some extra classes, for the showcase of the pattern.

9.1 Description of the pattern and its implementation

"The Observer protocol defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

To achieve this one object (subject) should know about its dependents. Subject maintains list of its dependents. Each dependent who wants to get notification on subject state change, should register with subject.” [38]

The implementation of *Observer protocol* is as follows: after triggering the 'stateChanges()' event in a Subject the 'update()' method in each associated Observer is called. Each class is implementing Subject or Observer interfaces. These classes are oblivious to the fact that they are being subjected to the Observer protocol since the strategy is applied through aspect SubjectObserverProtocolImpl through inter-type parent declaration. For example class Button implements Subject and ColorLabel implements the Observer.

This includes the SubjectObserverProtocol aspect which modularizes the core observer. Following is its code:

```
1 abstract aspect SubjectObserverProtocol {
2
3     abstract pointcut stateChanges(Subject s);
4
5     after(Subject s): stateChanges(s) {
6         for (int i = 0; i < s.getObservers().size(); i++) {
7             ((Observer)s.getObservers().elementAt(i)).update();
8         }
9     }
10
11     private Vector Subject.observers = new Vector();
12     public void Subject.addObserver(Observer obs) {
13         observers.addElement(obs);
14         obs.setSubject(this);
15     }
16     public void Subject.removeObserver(Observer obs) {
17         observers.removeElement(obs);
18         obs.setSubject(null);
19     }
20     public Vector Subject.getObservers() { return observers; }
21
22     private Subject Observer.subject = null;
23     public void Observer.setSubject(Subject s) { subject = s; }
24     public Subject Observer.getSubject() { return subject; }
25 }
```

Afterward a concrete aspect SubjectObserverProtocolImpl associates the roles (Subject and Observer) with the appropriate classes and defines the observed

pattern by a pointcut(stateChanges). In this example the subject is a Button with click() event and the observers are instances of ColorLable class. Clicking the button should have an effect on the color of the labels. The following source code demonstrates the implementation of SubjectObserverProtocolImpl aspect:

```
1 aspect SubjectObserverProtocolImpl extends SubjectObserverProtocol {
2
3     declare parents: Button implements Subject;
4     public Object Button.getData() { return this; }
5
6     declare parents: ColorLabel implements Observer;
7     public void ColorLabel.update() {
8         colorCycle();
9     }
10
11     pointcut stateChanges(Subject s):
12         target(s) &&
13         call(void Button.click());
14
15 }
```

The following code represents the actual client application which instantiates the Button and ColorLable instances and is now oblivious of what happens if the buttons are clicked.

```
1 public class Demo {
2     public static void main(String[] args) {
3
4         Display display = new Display();
5         Button b1 = new Button(display);
6         Button b2 = new Button(display);
7         ColorLabel c1 = new ColorLabel(display);
8         ColorLabel c2 = new ColorLabel(display);
9         ColorLabel c3 = new ColorLabel(display);
10
11         b1.addObserver(c1);
12         b1.addObserver(c2);
13         b2.addObserver(c3);
14     }
15 }
```

9.2 A comprehension task

Assume that a maintainer needs to find out the intention of applying this pattern to a sample program. This needs finding all possible occurrences of the Observer protocol in the program; that is, to determine the hidden dependency between methods in Observers and Subjects classes introduced by Observer Protocol (i.e. which method in subject leads to execution of which methods in observers). More deterministic, this means finding methods in classes implementing Subject which trigger the `update()`, and associating them with methods called by overridden `update()` in classes implementing Observer. This information could be obtained by a multiple interactions with an IDE such as Eclipse as follows:

Step 1) User starts by checking the `ObserverProtocolImpl` aspect (see Figure 9.1),

trying to find components affected by the aspect; That is components having eaten an intertype declared member, declared parent, or advised method by the aspect.

Then the user find out that the classes `Button` and `ColorLabel` implement the interfaces `Observer` and `Subject` respectively through aspectual declare parent construct by `ObserverProtocolImpl`. He also finds the inherited method from `Observer` which is `ColorLabel.update`.

Step 2) Consequently, using the source code view (see Figure 9.2) he can obtain meth-

ods invoked by `ColorLabel.update`, which is `ColorLabel.colorCycle()`.

Step 3) Now the user should search through `Button`'s methods which trigger the 'state

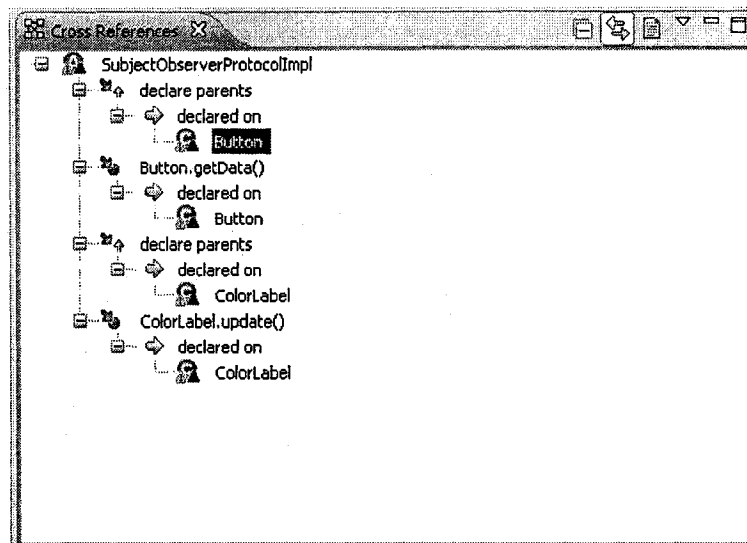


Figure 9.1: The SubjectObserverProtocolImpl aspect the AspectJ cross-reference view.

change' event. The user already knows that 'state change' event is implemented by the pointcut `stateChanges` in `ObserverProtocol` and matched by the advice `after() : stateChanges` in `ObserverProtocolImpl`. Viewing the cross-reference view (see Figure 9.3) he finds out that the method `Button.click()` triggers the execution of the advice `after() : stateChanges`.

Step 4) Finally by comparing what found in step 2 and 3 the user is able to match two methods `ColorLabel.colorCycle()` and `Button.click()` and find out that observer protocol is used to notify `ColorLabel` to update it's color each time the `Button` is clicked.

Step 5) Observing the actual `Demo` class the user is able to view the actual instances and thus the final behavior of the program (see Figure 9.4).


```
SubjectObserverProto SubjectObserverProto Demo.java Button.java »
import java.util.Vector;

aspect SubjectObserverProtocolImpl extends SubjectObserverProtocol {

    declare parents: Button implements Subject;
    public Object Button.getData() { return this; }

    declare parents: ColorLabel implements Observer;
    public void ColorLabel.update() {
        colorCycle();
    }

    pointcut stateChanges(Subject s):
        target(s) &&
        call(void Button.click());
}
```

Figure 9.2: The source code of aspect SubjectObserverProtocolImpl in Observer protocol example.

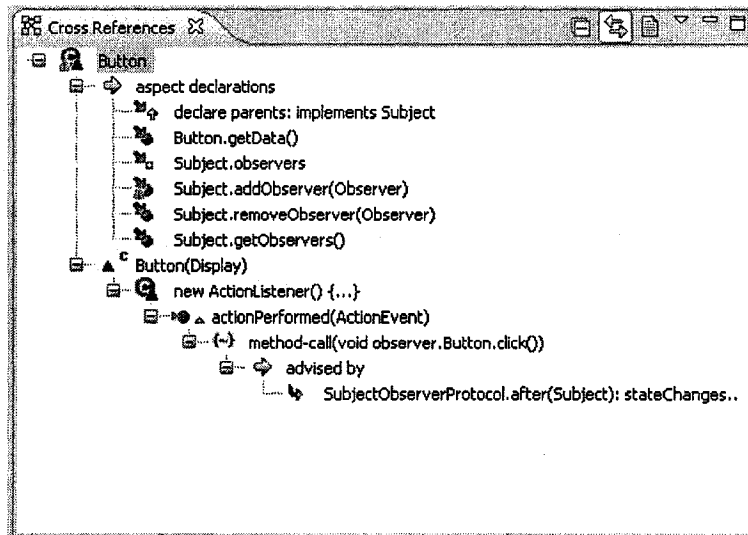
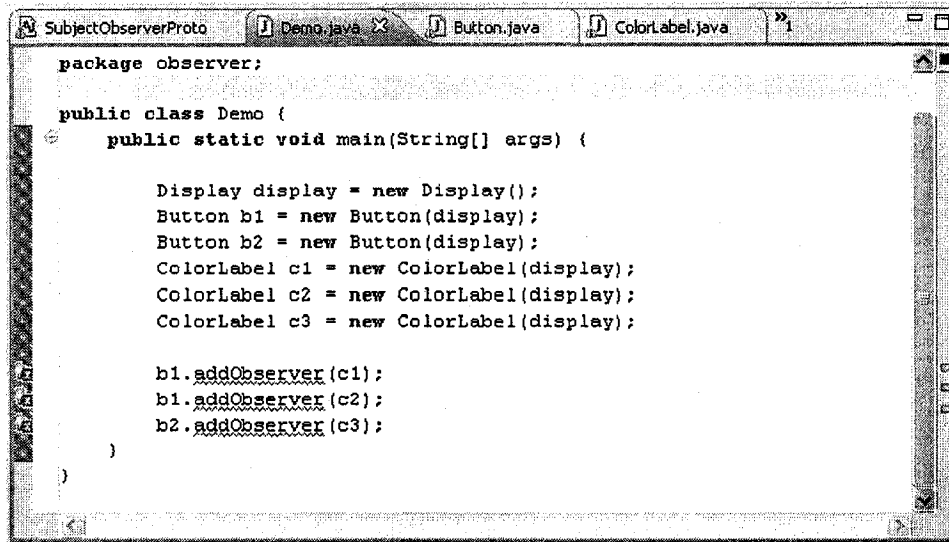


Figure 9.3: The Button class in the AspectJ cross-reference view.

A screenshot of a Java IDE window titled "SubjectObserverProto". The window contains several tabs: "SubjectObserverProto", "Demo.java", "Button.java", and "ColorLabel.java". The "Demo.java" tab is active, showing the following source code:

```
package observer;

public class Demo {
    public static void main(String[] args) {

        Display display = new Display();
        Button b1 = new Button(display);
        Button b2 = new Button(display);
        ColorLabel c1 = new ColorLabel(display);
        ColorLabel c2 = new ColorLabel(display);
        ColorLabel c3 = new ColorLabel(display);

        b1.addObserver(c1);
        b1.addObserver(c2);
        b2.addObserver(c3);
    }
}
```

Figure 9.4: The source code of class Demo in Observer protocol example.

Instead of performing these steps using manual navigation through the code the developer can intuitively obtain the same result using the textual query displayed in Figure 9.5 or its equivalent visual representation in Figure 9.6.

```

[Type subject ,
 Method m1,
 Type observer ,
 Method m3
]
exists[Method update]
where
  aspect extends 'SubjectObserverProtocol' and
  aspect defines stateChanges and
  m1 advisedby 'after():stateChanges' and
  update overrides 'Observer.update' and
  update calls m3 and
  observer implements 'Observer' and
  subject implements 'Subject' and
  subject defines m1
  observer defines m3

```

(a) Textual query.

| subject | m1 | observer | m3 |
|---------|--------------|------------|-----------------------|
| Button | Button.click | Colorlabel | Colorlabel.colorcycle |

(b) Query result.

Figure 9.5: The query for retrieving occurrences of observer protocol and its result.

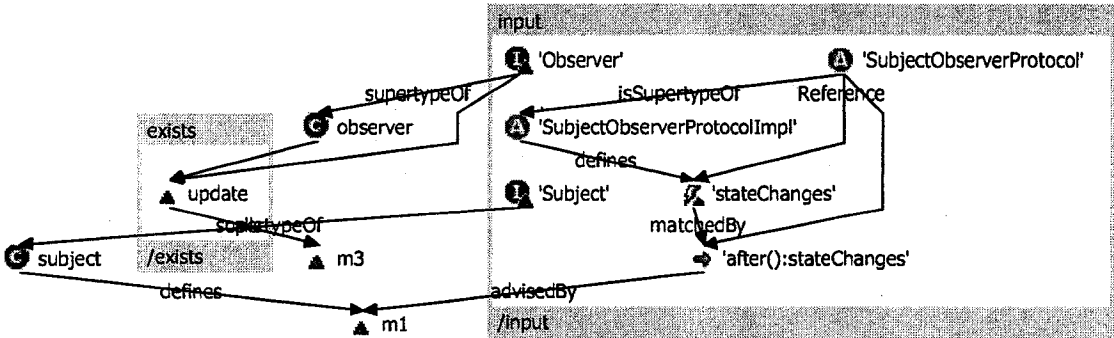


Figure 9.6: The visual query for finding occurrences of observer protocol.

Chapter 10

Case Study 3: Spacewar

We have deployed our approach over a medium-scale (3053 SLOC) AspectJ source code, Spacewar (can be found in AspectJ compiler example package). This system has been deployed in the literature as a benchmark [62, 15, 72]. Spacewar simulates an arcade asteroids game where a user flies a spaceship, represented by a movable triangle form, and attempts to eliminate other spaceships (same triangle form with different color). The aspects defined in this system provide the synchronization required to enforce the mutual exclusion of several methods. Figure 10.1 shows a partial class diagram of the system. In order to represent aspects in a UML class diagram we have deployed an `<< Aspect >>` stereotype. Spacewar consists of two packages, `coordination` and `spacewar`. The first package contains five classes, three interfaces and one aspect, namely `Coordinator`. This is an abstract aspect that provides basic functionality for synchronizing and coordinating different threads upon entering

and exiting methods. Methods which must be executed by only one thread at a time are marked as self-exclusive. Execution of a method which is marked as mutually exclusive will block temporarily the execution of the methods which are also marked as mutually exclusive with this method in other threads. Methods `guardedEntry(...)` and `guardedExit(...)` are invoked before and after any call to methods which need coordination and synchronization. Package `spacewar` contains 10 classes and five aspects. Moreover this package contains three units which are defined as Java classes but they are considered as AspectJ compilation units as the extension of the files is ".aj" and not ".java". For example `Ship.aj` is defined to be a class and it contains a pointcut definition. Another example is `Display.aj` which is also defined to be a Java class and it contains an inner aspect `DisplayAspect`. Class `Game` is the root of the `spacewar` game. Class `SpaceObject` which is an abstract class simulates the objects that float around in space. Subtypes of class `SpaceObject` (`Ship`, `Bullet`, and `EnergyPacket`) are created when class `Game` is created. Upon creation of any `SpaceObject`, it adds itself to the registry and when it dies, it removes itself from the registry. Class `Registry` is responsible for tracking all the space objects that are floating around. The synchronization is done by the inner aspect `RegistrySynchronization` in this class. Aspect `DisplayAspect` which is an inner aspect of class `Display` draws the `SpaceObject` on the screen and indicates the space it occupies.

Aspects `GameSynchronization`, and `RegistrySynchronization` are subtypes of aspect `Coordinator`. Aspect `GameSynchronization` ensures synchronized access to methods of class `Game` when several threads exist.

Aspect `RegistrySynchronization` ensures synchronized access to methods of class `Registry` when several threads exist. Aspect `EnsureShipIsAlive` ensures that the `Ship` is alive before performing any console commands. Table 10.1 shows a code segment of aspect `Coordinator`.

10.1 Deployment of the tool over Spacewar

In this subsection, we illustrate examples of our query tool applied on the Spacewar system.

Messages to which an object of a given type responds: There are situations where one wants to identify the messages to which an instance can respond. In an object-oriented system an object can respond to a message if it has or inherits a method (with access modifier `public` or `protected`) with type signature corresponding to this message. However, in an aspect-oriented system inter-type declaration allows to introduce methods or attributes for a class or an interface. This implies that an object o of type T , can also respond to messages introduced by aspects for T , or for supertypes of T . Table 10.2-(a) shows the query corresponding to this definition. If one decides to find all messages to which an object of type `Bullet` can respond to, s/he needs to modify the query as in Table 10.2-(b). The result of this query is shown in Figure 10.2.

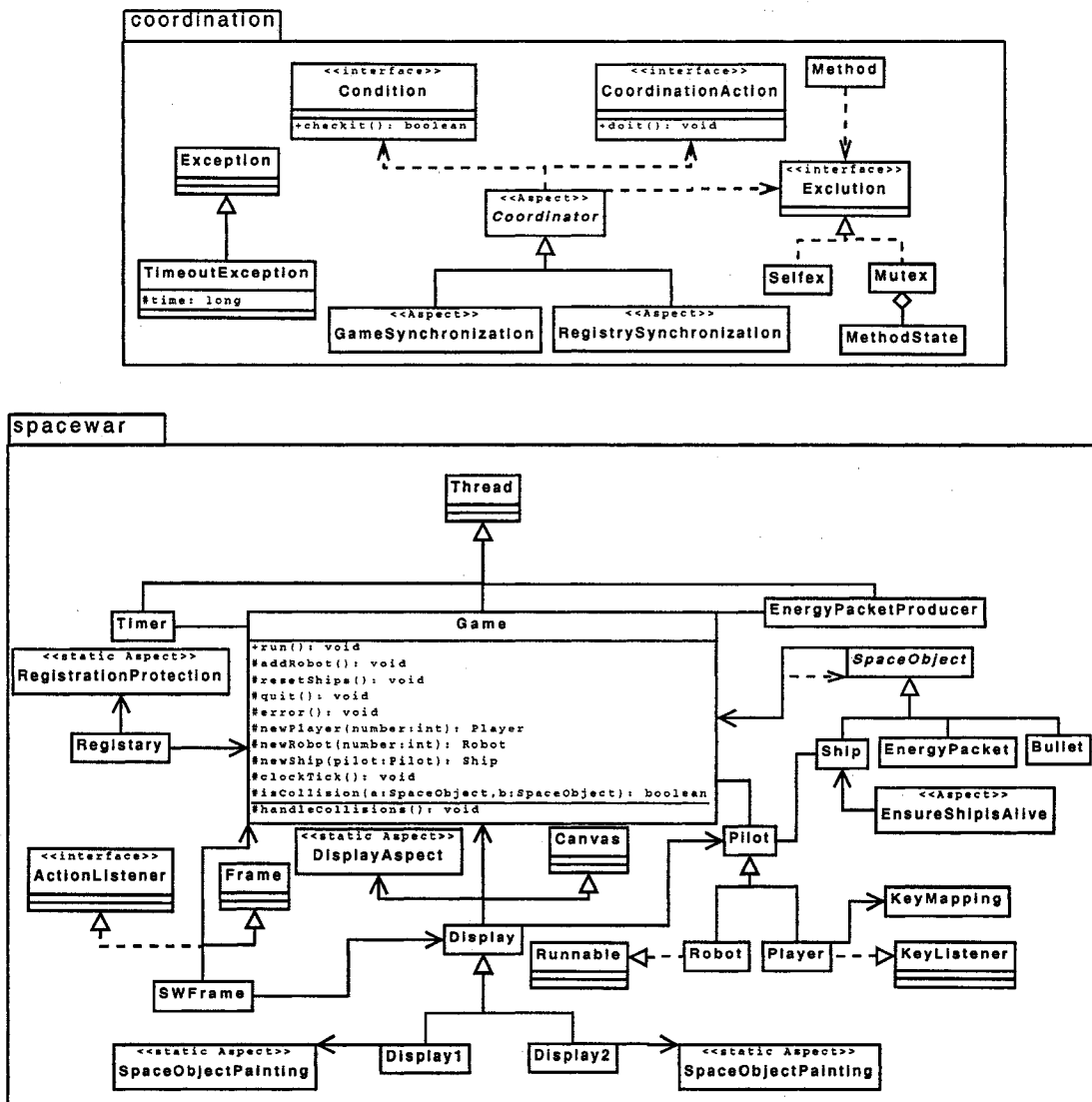


Figure 10.1: Partial class diagram of the Spacewar system.

```

1 package coordination;
2
3 public abstract aspect Coordinator {
4     private Hashtable methods = null;
5     private Vector exclusions = null;
6     abstract protected pointcut synchronizationPoint();
7     public Coordinator() {
8         methods = new Hashtable ();
9         exclusions = new Vector(5);
10    }
11
12    before (): synchronizationPoint() {
13        this.guardedEntry(thisJoinPointStaticPart
14                        .getSignature().getName());
15    }
16
17    after (): synchronizationPoint() {
18        this.guardedExit(thisJoinPointStaticPart
19                        .getSignature().getName());
20    }
21
22    public synchronized void addSelfex(String methName) {
23        Selfex sex = new Selfex (methName);
24        exclusions.addElement(sex);
25        Method aMeth = getOrSetMethod(methName);
26        aMeth.addExclusion(sex);
27    }
28
29    public synchronized void removeSelfex(String methName) {
30        for (int i = 0; i < exclusions.size(); i++) {
31            Exclusion sex = (Exclusion)exclusions.elementAt(i);
32            if ((sex instanceof Selfex) &&
33                (((Selfex)sex).methodName.equals(methName)))
34            {
35                exclusions.removeElementAt(i);
36                Method aMeth = getOrSetMethod(methName);
37                aMeth.removeExclusion(sex);
38            }
39        }
40    }
41    // some code
42 }

```

Table 10.1: Partial code of aspects 'Coordinator'.


```

[Type t, Method m] where
not (m hasModifier "private")
and
(
  t includes m
  or
  (exists[Type t2] where
    t2 isSupertypeOf+ t and
    t2 includes m)
  or
  exists[Aspect a] where
    declaresMethod(a, t, m)
)

```

(a) Considering all classes

```

[Method] from
[Type t, Method m] where
not (m hasModifier "private")
and
(
  t includes m
  or
  (exists[Type t2] where
    t2 isSupertypeOf+ t and
    t2 includes m)
  or
  exists[Aspect a] where
    declaresMethod(a, t, m)
)
where t hasName "Bullet"

```

(b) Considering only the class 'Bullet'

Table 10.2: Obtaining all messages to which an object of a given type can respond.

| Type | Method |
|--------|-----------------|
| Bullet | getsize |
| Bullet | handlecollision |
| Bullet | clocktick |

Figure 10.2: Messages to which an object of a given type can respond to in Spacewar example.

Declared method in inheritance hierarchy: In Table 10.3-(a), query

`findDeclaredMethod` identifies all methods defined by an aspect `AspectName` (through the inter type declaration) for the supertypes of a given type. By applying this query, one can find methods defined for supertype of class `Ship`.

The result of query `findDeclaredMethod TypeName, MethodName` in Figure 10.3-(b) shows that two aspects with the same name (`SpaceObjectPainting`) define two methods with the same name (`paint`) for class `SpaceObject` which is the supertype of class `Ship`.

Class `Display` has an inner aspect `SpaceObjectPainting`, which defines method `paint` for class `SpaceObject`.

Class `Display2` also has an inner aspect `SpaceObjectPainting` which defines method `paint` for class `SpaceObject`. By exploring the code we can see that the implementations of method `paint` in the two aspects are different.

```

[ Aspect a , Type t2 , Method m ]
where
  t2 isSuperType+ t and
  declaresMethod ( a , t2 , m ) and
  t hasName "ship"

```

(a) Textual query.

| AspectName | SuperTypeName | MethodName |
|-----------------|---------------|------------|
| spaceobjectp... | spaceobject | paint |
| spaceobjectp... | spaceobject | paint |

(b) Query result.

Table 10.3: Obtaining methods defined by an aspect for a supertype of a given type.

Chapter 11

Related work and evaluation

The current existing approaches for source code querying could be placed in the following three categories:

11.1 Logic-based query approaches

Logic-based query approaches such as SOUL [71], JQuery [69] [47], JTransformer [11], CodeQuest [42] use Prolog-based query engines such as TyRuBa [13] or Datalog [14] to perform query evaluation.

SOUL is a logic meta-language based on Prolog which is implemented in Visual Work Smalltalk [71]. It provides a declarative framework that allows reasoning about the structure of Smalltalk programs based on the parse tree representation.

In [69] the author implements a Java browser called JQuery as an Eclipse plug-in.

The query language used for this tool is a logic programming language called TyRuBa based on Prolog. JTransformer [11] is a Prolog-based query and transformation engine for storing, analyzing and transforming fact-bases of Java source code.

JTransformer creates an AST representation of a Java project, including the complete AST of method bodies as a Prolog database.

CodeQuest [42] is a source code querying tool which uses *safe Datalog* as its query language, mapping Datalog queries to a relational database system.

In [33] the authors present a prototype tool for analysis and performance optimization of Java programs called DeepWeaver-1. This tool is an extension of the abc AspectJ compiler [16] which has a declarative style query language, (Prolog/Datalog-like query language) to analyze and transform code within and across methods.

The main problem of this group is the difficulty of encoding queries using the proposed language. Users would have to know Prolog should they want to have more complex queries, as their need might not already be implemented in the system. In contrast, our approach represents a way to compose queries with a small amount of effort from the maintainers. Our approach could completely erase the need for additional database expert in the browsing process. This is done through provisioning of visual support for each query operator available in the language. We also support storage and browsing of aspect-oriented programs as a super-set of object-oriented ones. Although some techniques have been adapted for querying AO programs, they are limited in terms of modeling the dynamicity of these systems. The dynamic features we addressed in our

model include context exposure and aspect instantiation control.

11.2 SQL-based approaches

SQL-based approaches (e.g. [60]) represent the program structure in the form of relational tables and use a relational database's native query engine to perform queries. Although relational storage mechanisms are highly investigated and established, and are very optimized for answering queries, they do not inherently provide a proper representation mechanism suitable for viewing program structures and, thus, creating a mismatch between complex program's structure and primary elements of the relational logical model, namely tables. This is partially due to the fact that hierarchical relationships such as modular containments between programming elements are modeled as flat relations. Another problem with this approach is the difficulty of encoding queries in logical level rather than conceptual ones. In this method, logical variables are substituted by column names, which demands users to memorize table structures. Moreover, since the relation columns does not support complex data types and meta-data is completely separated from the data itself, users have to frequently access the schema meta-data in an un-intuitive way.

11.3 OQL-based approaches

OQL-based approaches such as Semmler [68, 27] form the most recent wave of query tools published in recent years. They provide a intuitive interface through an abstract

object-oriented layer above the actual logic predicates.

Another problem with this approach is the difficulty of encoding queries in logical level rather than conceptual ones. In this method, logical variables are substituted by column names, which demands users to memorize table structures. Moreover, since the relation columns does not support complex data types and meta-data is completely separated from the data itself, users have to frequently access the schema meta-data in an unintuitive way.

11.4 Query-by-example approaches

Query-by-example approaches like JTL [23] aim at making queries simpler by making their syntax like the programming language itself.

11.5 Visual query approaches

Most conventional proposals discussed during 90s (e.g. proposals by Consens and Mendelzon [51, 25], GUPRO [30], MOOSE [29, 54], Rigi [67], CIA [20], the Software Bookshelf [35], SNiFF+ [19], and the work in [31]) have taken a pure database oriented approach and usually use a generic visual interface, neutral to software engineering domain (e.g. GraphLog [26], Hy+ [24], GraphQL [43], G-Log [55], and GREQL [48]), and thus, are confined to the capabilities of the visual interface itself. SWAGKit [12] with its SHriMP [66] views, GSEE [34], and the work adopted by [61] are exceptions which use their own unique visual query interfaces.

Following the shift of view during recent years toward specific purpose query engines, we focused on demands of software developers, rather than proving applicability of database theories on software domain.

To this end we could be viewed as an adaptation of conventional proposals in visual query domain (rather than their competitor) aiming at providing the same strength as current real world text based query tools by integration into IDEs (i.e. Eclipse), adaptation to new programming paradigms (i.e. OO/AO), and using more software-design-biased visual constructs.

Moreover, in our case, the evaluation against old tools in the same case study demands re-implementation of major parts of these tools (e.g. their fact extractor and probably the schema) for adapting them to Java/AspectJ.

11.6 Algebraic query approaches

[56, 57, 39] present and demonstrate the use of an algebraic source code query technique that blends expressive power with query compactness. For example, query framework of Source Code Algebra, or SCA, permits users to express complex source code queries and views as algebraic expressions.

11.7 Limitations

One of the limitations of our work is the possibility that there is a structural information not provided through the fact extractor. Since the fact extractor for AspectJ

language is an ongoing project, and thus, there are some information available in the source code which might not be captured by the fact extractor. This currently includes some constructs inside an expression and statements. We believe that there is a need to expand the scope of the fact extractor to include more detail information about the program.

Also since in current version of the tool we evaluate the queries by a user defined engine, there is not any optimization done to the queries. Thus the tool might not be as fast as similar database oriented approaches. In future we would like to substitute the query interpreter with a compiler to translate the query to an intermediate form and pass it for execution to an optimized query execution engine. This could also be accompanied with using existing DBMS systems with volatile storage.

In terms of representation mechanism we tried to adopt different views to help maintainers observing the query result from different point of view. However, our approach does not currently apply a specific strategy for visualization of large sets of query results (e.g. content summarization techniques adapted in SWAGKit [12]). This, currently leads to a bit of visual clutter in graph views. This clutter, however, is solvable using zooms available in the tool, but content summarizations would make this process easier for developers.

Chapter 12

Conclusion and recommendations

We have introduced visual+textual query composition as a generic technique for implementing highly customizable source code browsers [40]. We have demonstrated how visual query composition can be used in conjunction with existing software visualization techniques, and how the navigation model is capable of releasing the user from hand-coding the queries. We currently have automation as a proof of concept provided through a prototypical Eclipse plug-in tool supporting different programming paradigms such as procedural, object-oriented and aspect-oriented. Our work complements related work discussed in other proposals. However, we believe that it will help maintainers overcome the barriers of current tools for program comprehension and measurement.

A possible direction for future work could be providing support for non-volatile storage of the proposed codebase model. This could help the navigation and analysis of

legacy systems by providing efficient algorithms for managing large amount of non-volatile memory. An option could be using existing globally available database systems. The benefit of this approach is that the uniform query language (e.g. SQL) and its interpreters are publicly available. However, adaptation is needed to match the requirement of our model to these database systems, since the query composition method discussed here, requires more interactiveness than just 'query writing/execution/result representation' cycle. Moreover, this direction should focus on database support for complex data-types, meta-model access during the query composition, and ability to reuse queries and combine them either as higher-order functions or using set operations.

Another interesting subject for future work is to focus on the performance tradeoffs. That is to investigate the time and memory complexity of query systems discussed (e.g SQL, .QL, and datalog) and to find the proportionality between expressiveness and algorithmic complexity and efficiency. This could also be extended with a study on time complexity of source code to fact transformation.

Bibliography

[1] AJDT: AspectJ Development Tools - The Cross References view (XRef).

<http://www.eclipse.org/ajdt/xref/>

[2] AJDT: AspectJ Development Tools - The Visualiser.

<http://www.eclipse.org/ajdt/visualiser/>

[3] AspectC website.

<http://research.msrg.utoronto.ca/ACC>

[4] AspectJ website.

<http://www.eclipse.org/aspectj/>

[5] AspectR website.

<http://aspectr.sourceforge.net/>

[6] Chidamber and Kemerer Java Metrics.

<http://www.spinellis.gr/sw/ckjm/>

[7] Eclipse website.

<http://www.eclipse.org/>

[8] Graphical editing framework (GEF) website.

<http://www.eclipse.org/gef/>

[9] IMP website.

<http://www.eclipse.org/imp>

[10] Java Compiler Compiler [tm] (JavaCC [tm]) - the Java parser generator.

<https://javacc.dev.java.net/>

[11] JTransformer Framework website.

<http://roots.iai.uni-bonn.de/research/>.

[12] SWAG Kit website.

<http://www.swag.uwaterloo.ca/swagkit/>

[13] TyRuBa website.

<http://tyruba.sourceforge.net/>

[14] Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[15] Jonathan Aldrich. Open modules: Reconciling extensibility and information hiding. In *AOSD workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT)*, 2004.

- [16] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, 2005.
- [17] Benjamin B. Bederson, Ben Shneiderman, and Martin Wattenberg. Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies. *ACM Transactions on Graphics*, 21(4):833–854, 2002.
- [18] Jens Marco Bendisposto. A framework for semantic-aware editors in eclipse. Master’s thesis, Heinrich-Heine-Universitt, INSTITUT FR INFORMATIK, 2007.
- [19] Walter R. Bischofberger. Sniff: A pragmatic approach to a c++ programming environment. In *Proceedings of the C++ Conference*, pages 67–82, 1992.
- [20] Y.-F. Chen, M.Y. Nishimoto, and C.V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, 1990.
- [21] E. F. Codd. Relational completeness of data base sublanguages. In R. Rustin (ed.): *Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, pages 65–98, 1972.
- [22] Wesley Coelho and Gail C. Murphy. ActiveAspect: presenting crosscutting structure. In *Proceedings of the ICSE Workshop on Modeling and Analysis of Concerns in Software*. ACM Press New York, NY, USA, 2005.

- [23] Tal Cohen, Joseph Gil, and Itay Maman. JTL: the Java Tools Language. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '06)*. ACM Press New York, NY, USA, 2006.
- [24] Mariano Consens and Alberto Mendelzon. Hy+: a hygraph-based query and visualization system. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, pages 511–516, New York, NY, USA, 1993. ACM.
- [25] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering (ICSE '92)*, pages 138–156. ACM, 1992.
- [26] Mariano P. Consens and Alberto O. Mendelzon. The g+/graphlog visual query system. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data (SIGMOD '90)*, page 388. ACM, 1990.
- [27] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .QL for source code analysis. In *Proceedings of the 7th Working Conference on Source Code Analysis and Manipulation (SCAM '07)*, pages 3–16. IEEE Computer Society, 2007.

- [28] Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge, and Bruce W. Ballard. LaSSIE - a knowledge-based software information system. In *Proceedings of the 12th International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 1991.
- [29] Stéphane Ducasse, Michele Lanza, and Sander 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)*, 2000.
- [30] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO-Generic Understanding of Programs An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2):47–56, 2002.
- [31] Juergen Ebert, Bernt Kullbach, and Andreas Winter. Querying as an enabling technology in software reengineering. *Proceedings of the Third European Conference on Software Maintenance and Reengineering (CSMR '99)*, 00:42, 1999.
- [32] Laleh Mousavi Eshkevari, Venera Arnaoudova, and Constantinos Constantinides. Comprehension and dependency analysis of aspect-oriented programs through declarative reasoning. In *Proceedings of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL '08)*, pages 35–52. Springer, 2008.

- [33] Henry Falconer, Paul H. J. Kelly, David M. Ingram, Michael R. Mellor, Tony Field, and Olav Beckmann. A declarative framework for analysis and optimization. In *Proceedings of the 16th International Conference on Compiler Construction (ETAPS-CC '07)*, 2007.
- [34] Jean-Marie Favre. GSEE: A generic software exploration environment. *Proceedings of the 9th International Workshop on Program Comprehension (IWPC '01)*, 00:0233, 2001.
- [35] P.J. Finnigan, R.C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, and H. Muller. The Software Bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.
- [36] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, 1991.
- [37] Herve Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16(2):153–185, 1984.
- [38] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
- [39] Hamoun Ghanbari and Constantinos Constantinides. An algebraic query method for static program analysis and measurement. In *Proceedings of the 17th International Conference on Software Engineering and Data Engineering (SEDE '08)*,

2008.

- [40] Hamoun Ghanbari, Constantinos Constantinides, and Venera Arnaoudova. A hybrid query engine for the structural analysis of java and aspectj programs. In *Proceedings of the 15th International Working Conference on Reverse Engineering (WCRE '08)*, 2008.
- [41] Martin Graham, Jessie B. Kennedy, and Chris Hand. A comparison of set-based and graph-based visualisations of overlapping classification hierarchies. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '00)*, pages 41–50. ACM Press New York, NY, USA, 2000.
- [42] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with Datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP '06)*, pages 2–27. Springer, 2006.
- [43] H. He and A.K. Singh. GraphQL: Query Language and Access Methods for Graph Databases. Technical report, Technical report, Department of Computer Science at University of California, Santa Barbara, 2007.
- [44] Brian Henderson-Sellers. *Object-oriented metrics: Measures of complexity*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1995.
- [45] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.

- [46] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.
- [47] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 178–187. ACM Press, 2003.
- [48] M. Kamp. GReQL. *Eine Anfragesprache für das GUPRO-Repository, Sprachbeschreibung (Version 1.2)*. in [8], pages 173–202, 1998.
- [49] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, 1997.
- [50] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of the 3rd AOSD Workshop on Linking Aspect Technology and Evolution (LATE '07)*, New York, NY, USA, 2007. ACM.
- [51] Alberto O. Mendelzon and Johannes Sametinger. Reverse engineering by visualizing and querying. *Software - Concepts and Tools*, 16(4):170–182, 1995.
- [52] Russell Miles. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

- [53] Michael L. Nelson. A survey of reverse engineering and program comprehension. In *In ODU CS 551 Software Engineering Survey*, page 2, 1996.
- [54] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of moose: an agile reengineering environment. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE '05)*, pages 1–10. ACM, 2005.
- [55] Jan Paredaens, Peter Peelman, and Letizia Tanca. G-log: A graph-based query language. *IEEE Transactions on Knowledge and Data Engineering*, 07(3):436–453, 1995.
- [56] Santanu Paul and Ataul Prakash. A query algebra for program databases. *IEEE Transactions on Software Engineering*, 22(3):202–217, 1996.
- [57] Santanu Paul and Atul Prakash. Querying source code using an algebraic query language. In *Proceedings of the International Conference on Software Maintenance (ICSM '94)*, pages 127–136. IEEE Computer Society, 1994.
- [58] Knut H. Pedersen and Constantinos Constantinides. AspectAda: aspect oriented programming for ada95. In *Proceedings of the 2005 Annual ACM SIGAda International Conference on Ada*, pages 79–92. ACM, 2005.
- [59] J.-Hendrik Pfeiffer and John R. Gurd. Visualisation-based tool support for the development of aspect-oriented programs. In *Proceedings of the 5th International*

- Conference on Aspect-Oriented Software Development (AOSD)*, pages 146–157. ACM Press New York, NY, USA, 2006.
- [60] Awais Rashid and Neil Loughran. Relational database support for aspect-oriented programming. In *Revised Papers from the International Conference NetObject-Days on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 233–247. Springer-Verlag, 2003.
- [61] Steven P. Reiss. A visual query language for software visualization. *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, 00:80, 2002.
- [62] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering (FSE '04)*, pages 147–158. ACM Press, 2004.
- [63] Spencer Rugaber. Program comprehension for reverse engineering. In *Proceedings of the AAAI Workshop on AI and Automated Program Understanding*, 1992.
- [64] Robert W. Sebesta. *Concepts of programming languages (7th Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2005.
- [65] Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Addison-Wesley, 2006.

- [66] Margaret-Anne Storey, Casey Best, and Jeff Michaud. SHriMP views: an interactive environment for exploring Java programs. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC '01)*, page 111. IEEE Computer Society, 2001.
- [67] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. Rigi: a visualization environment for reverse engineering. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 606–607, New York, NY, USA, 1997. ACM.
- [68] Mathieu Verbaere, Elnar Hajiyev, and Oege De Moor. Improve software quality with SemmleCode: an eclipse plugin for semantic code search. In *Companion to the 22nd ACM SIGPLAN Conference on Object Oriented Programming, Systems Languages and Applications (OOPSLA'07)*, pages 880–881, New York, NY, USA, 2007. ACM.
- [69] Kris De Volder. JQuery: A generic code browser with a declarative configuration language. In *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL '06)*, 2006.
- [70] Anneliese von Mayrhauser and A. Marie Vans. Program understanding – a survey, 1994.

- [71] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of the 26th International Conference on Technologies of Object-Oriented Languages and Systems (TOOLS-USA '98)*, pages 88–102. IEEE Computer Society, 1998.
- [72] Sai Zhang and Jianjun Zhao. Change impact analysis for aspect-oriented programs. Technical report, Center for Software Engineering, Shanghai Jiao Tong University, 2007.