



VRIJE UNIVERSITEIT BRUSSEL  
**Programming Technology Laboratory**  
Faculteit van de Wetenschappen

# **A Logic Meta-Programming Approach to Support Concern Interaction Detection**

**A Thesis submitted with a view to the obtainment of the degree of  
Master of Science in Computer Science**

**Andrea Veronica Vladu**

**Promoter: Prof. Dr. Theo D'Hondt  
Advisors: Kris Gybels and Johan Brichau**

**August 2004**

# Abstract

Being that software is still becoming more and more complex, the problems to be solved by software technology are becoming harder and the circumstances of software development call for a much more efficient process. This led focus of research in software engineering on finding methods that make software more easily maintainable. For software tools to become truly useful in aiding software developers, they need to be able to automate the everyday tasks of developers, they need to better automate the maintenance activities.

There exist techniques such as separation of concerns and incremental programming, which help software developers to build applications by allowing them to consider new features as independent increments of the application's basic service.

This thesis is about modeling concerns of software products and about the interactions that occur between these concerns. We will see approaches of concern modeling and will put a focus on concern interaction. To be able to maintain a piece of software it is necessary to find such interactions because concern interaction coincides with a better understanding of interrelationships of related items, that helps the programmer to find what else must be changed if one concern is changed.

In this dissertation, a description of concerns representing program structures linked to source code and organized in a concern tree view, based on Star Browser tool [RoWuytsSt] is used. Logic programming techniques allow identifying interactions between concerns and the links to the source code help the developer to perform software evolution tasks more systematically.

To validate the obtained results, we have used a software tool that describes concerns in terms of relations between program elements and defines the interactions between concerns, so that it is possible to automatically detect the source code related to the modifications needed in the task of the software evolution. We are able to detect a number of different types of concern interactions. Our approach is flexible enough to support user-defined concern interactions but is rigorous enough to allow automated interaction detection.

Using this tool, we have evaluated usefulness of concern interactions detection in a case study involving the evolution of the Soul tool. The results show that concern interactions detection is inexpensive and it can help developers to perform program evolution tasks more systematically.

# Acknowledgments

I owe a dept of gratitude to many people who have been important for my studies and the completion of this thesis.

In particular, I am deeply grateful to my two supervisors Kris Gybels and Johan Brichau for their continuous advice and support.

I am greatly indebted to Kris Gybels. He provided me valuable suggestions, ideas, comments, encouragement each time allowing me to improve my knowledge and to finalize my thesis.

I would like to thank to Theo D'Hondt, who gave me the opportunity to do the One year Master of Science in Computer Science that has been a wonderful experience.

I would like to express my sincere and special thanks to Roel Wuyts for spending his time with my questions, and for helping me every time I needed it.

I owe a lot of gratitude to all the people behind SOUL that was very useful in my work. My explicit thanks to Andy Kellens for developing the Relation Browser tool.

Thanks also to other people at the Programming Technology Lab, to the faculty, staff members and students, who have contributed to this thesis with their comments, help, and moral support.

Last, and certainly not least, I would like to thank my family, for giving me the opportunity to study, and for supporting me in various ways, too much to mention on one page.

# Table of Content

1. Introduction.....	6
1.1. Objectives .....	6
1.2. Motivation.....	6
1.3. Organization of the Dissertation .....	8
Chapter 2. Concern interaction .....	9
2.1. Definitions.....	9
2.1.1. Concerns .....	9
2.1.2. Feature.....	11
2.1.3. Aspect .....	12
2.2. Aspect-Oriented Programming (AOP) .....	13
2.3. Finding concerns .....	14
2.3.1. ASPECT BROWSER tool .....	14
2.4. Modeling concerns.....	16
2.4.1. Hyperspace approach to concern manipulation .....	17
2.2.5. Concern Space Modeling in COSMOS .....	21
2.5. Concern interaction.....	22
2.5.1. FEAT (Feature Exploration and Analysis Tool).....	22
2.3. Summary and Conclusions .....	26
Chapter 3. Logic Meta-Programming .....	28
3.1. Logic Meta-Programming.....	28
3.2. Logic Meta-Programming with SOUL .....	30
3.2.1. Syntax of SOUL.....	32
3.2.2. Smalltalk terms and Smalltalk clauses.....	32
3.2.3. LiCoR Library.....	34
3.3. SOUL predicates.....	36
3.4. Intentional source-code views.....	39
3.5. Conclusion .....	40
Chapter 4. The StarBrowser.....	42
4.1. Introduction.....	42
4.2. The Lightweight Classifications Model.....	42
4.3. The StarBrowser .....	45
4.4. Screenshots with StarBrowser .....	47
4.5. Conclusion .....	51
Chapter 5. Concern interaction using LMP .....	53
5.1. Introduction.....	54
5.2. Rules for concern interaction detection .....	56
5.3. Conclusion .....	61
Chapter 6. The Relation Browser: An experiment.....	63
6.1. The Relation Browser .....	63
6.2. An experiment.....	71
6.2.1. The problem.....	71
6.2.2. First phase .....	71

6.2.3. Second phase.....	74
6.3. Conclusion .....	76
Chapter 7. Conclusions .....	77
7.1. Summary .....	77
7.2. Conclusion .....	78
7.3. Future work.....	79
List of figures .....	81
List of tables.....	81
Bibliography .....	82

# 1. Introduction

## 1.1. Objectives

This dissertation is about Concern interaction detection by using Logic Meta-Programming in object-oriented programs, Smalltalk implementations in particular.

The goal of the dissertation is to use a classification of concerns - representing program structures linked to source code - that can help developers to perform software evolution tasks more systematically.

More specifically, we will provide a flexible framework that handles concern definitions and detects concern interactions. This framework will be implemented using the technique of logic meta-programming (LMP), based on StarBrowser and having Smalltalk as the target language to reason about.

For analyzing the concern interaction we will use the Relation Browser tool. In this way we expect to facilitate the software maintenance process.

## 1.2. Motivation

A major goal in software engineering is to reduce the cost of maintaining software systems. Finding methods that make software more easily maintainable has been one of the most prioritized challenges during the past decade. There exist techniques such as separation of concerns and incremental programming, which help software developers to build applications by allowing them to consider new features as independent increments of the application's basic service.

Software systems usually are large and complex. Because of their size and complexity, engineering such software systems is not an easy task. A large software system must achieve a lot of functionalities and on top of that it must do it fast, make efficient use of existing hardware etc. These requirements are also called the **concerns** of the software engineer. [KrisGybles01]

Additionally, during the development and maintenance of object oriented software, it is very likely that the code will experience modifications. When performing a

program evolution task, developers spend a significant amount of effort investigating source code in order to answer specific questions about the implementation of various mechanisms affecting the modification. In reality, in a large system, developers can examine only a small subset of the source code as part of a program evolution task. Determining which pieces of the code to examine is a complex problem that cannot be solved automatically with source code analysis techniques. [RobMur03]

Once the existing concerns are separated, the developer's task during the software evolution process is more reliable by using tools that helps the programmer to find the code fragment that must be changed if one concern is changed and that point to the source code related to the involved concern and the resulting software interaction.

Koen De Hondt, in his dissertation [KoDeHondt98] proposes software classification as an approach to architectural recovery in evolving object-oriented systems. The results of recovery are tangible entities in the software development environment.

Regarding software interactions, there exist several approaches about feature interactions. Straeten R. and Brichau J. [StraBric] have proposed the feature interaction detection in software engineering using logic.

In this dissertation, we will use the Star Browser tool for concern modeling. We will use logic meta-programming techniques to detect concern interaction, and the Relational Browser for experiments that validate this work.

Firstly, we will present existing approaches of concern modeling and interactions detecting and we will put a focus on Feat tool. [RobMur 03]

After having known these approaches for concern modeling, we extended the Star Browser tool for this purpose. This tool allows us to use a model that is flexible, precise, simple, robust, and tolerant to inconsistencies [RoWuytsSt]. Also, it can be easily extended to represent tree like view classifications in a similar way as Feat tool does.

For concern interaction detection we will use logic meta-programming techniques, that seem extremely well suited to query the object-oriented software and can automatically find interdependent software items. We are able to detect a number of different types of concern interactions in a flexible way that supports user-defined concerns. Our approach is rigorous enough to allow automated interaction detection. ,

and the Relational Browser for experiments that validate this approach for concern interaction detection

### **1.3. Organization of the Dissertation**

The rest of this thesis is organized as follows:

- Concern interaction;
- Logic Meta-Programming;
- StarBrowser tool;
- How to use LMP, StarBrowser and Relation Browser to do concern classification and to find interactions between them;
- Experiments;

Section 2 presents a concern interaction-survey. The technique of logic meta programming, in the first instance SOUL, in the form of an overview and context is presented in Section 3. An overview of the Star Browser tool is given in Section 4.

Section 5 details our approach in concern interaction detection, by using logic meta-programming, the StarBrowser.

Section 6 presents some experiments by using the Relation Browser tool, while Section 7 concludes this dissertation.

# Chapter 2. Concern interaction

This chapter gives an introduction to the subject of this dissertation. The first section consists in definitions about features, aspects, concerns and their interactions, considering existing terminology.

The second section is a survey about the existing tools for concern modeling.

Firstly, the Aspect Browser tool intended to find crosscutting concerns in software systems is presented. Aspect Browser is a graphical tool that helps developers to find concerns using lexical searches of the program text, using a map metaphor and an atlas metaphor to display and manage crosscutting concerns.

Secondly, some aspects of modeling concerns are tackled. The hyperspace approach, in which concerns are grouped into dimensions, giving hyperspaces, an explicitly multi-dimensional structure and Cosmos, a general purpose, multidimensional Concern-Space Modeling Schema are presented.

Finally, a technique that been proposed to help developers to identify concern interaction are given: Feat tool.

## 2.1. Definitions

When discussing about concern interaction it's important to consider existing terminology. This section overviews some definitions related to the presented subject. We distinguish the terms concern, feature and aspect according to our experiences as presented in the following paragraphs.

### 2.1.1. Concerns

A concern should be very generally defined as whatever is a matter of interest in the software, be it infrastructure, code, requirements, design artifacts, etc. [SuttRou]

In the literature there is not a well-established understanding of the notion of concern.

- In object-oriented methods the separated concerns are modeled as objects and classes, which are generally derived from the entities in the requirement specification and use cases;
- In structural methods, concerns are represented as procedures;
- In aspect-oriented programming, the term concern is extended with the so-called non-functional properties such as synchronization, memory management and persistency. In a sense one can consider this as a generalization of the notion of concern in the context of programming languages. [AskBed]

A concern is something we are concerned about (at the moment). Note that this is a temporal statement while a feature is permanent. A feature might become a concern if somebody is concerned about. On the other hand, a developer or stakeholder may be concerned about something which is not a feature. Therefore, not every concern results in a feature [Puver].

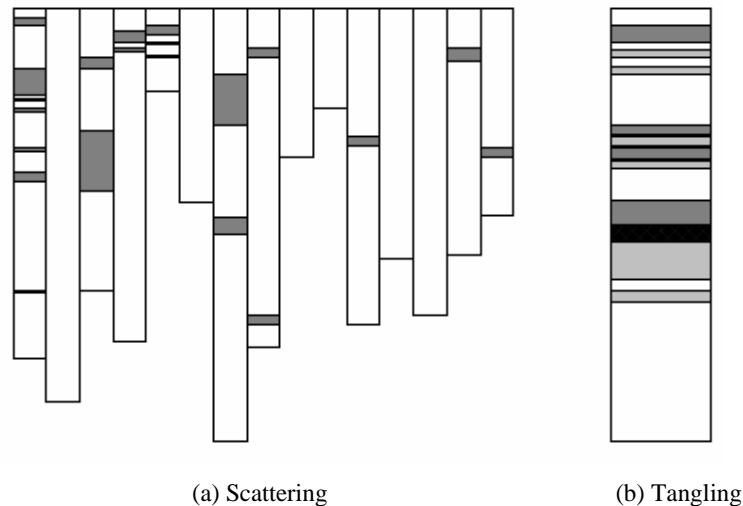
There is no global model of concerns, no systematic representation of concerns or a catalog of concerns. As a result, concerns in different work products or at different stages cannot be treated in the same way, we can't think about that in a uniform way, and concern-oriented tools and methods don't have a common foundation [SuttRou].

The separation of concerns (SOC) principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability and reusability.

Ideally, concerns, or features, in a software system are modularized making it easy for a software developer to identify and change a concern in a localized way, but it is impossible with existing approaches to modularize all concerns associated with a system.

As a consequence, in practice, the concerns a developer must consider during program evolution are not always well separated, and their implementation is often found to be scattered through different modules, and, at the same time, tangled within one module. Figure 1 illustrates schematically the scattering and tangling of concerns. The illustrations use the SeeSoft program view [StepEicSteff], where white rectangles represent the source code for a module. In the representation of concern scattering (*a*), gray rectangles indicate source code relevant to a concern. This source code is

scattered across multiple modules. In the representation of tangling (*b*), a single module is presented. The module comprises two concerns, shown by the boxes in the two different shades of gray. These two concerns also have overlapping code, represented by the area in black. In other words, tangling involves the presence of code implementing different concerns within a module.



**Figure 1** Concern scattering and tangling

Crosscutting concerns is very dependent on the context the kind of modeling being done, the formalism being used, the purpose of modeling, the approach to modeling and it may vary across life cycle stages or between projects.

### **2.1.2. Feature**

The concepts feature and feature interaction originated in the telephony domain. A feature in software engineering can be seen as a concern of the software application. The composition of features will always lead to interactions between features. Feature interference occurs when existing or new features interact such that a feature does not behave correctly [StraBric].

The term feature, in a broader sense is just as an extension to some basic functionality. It's an observable and relatively closed behavior or characteristic of a (software) part. A feature is something an application has to do, or any part or aspect of a specification, which the user perceives as having a self-contained functional role.

It may be composed from other features and a basic feature should be as small as possible (basic building blocks). A feature may be implemented as functionality or may have a non-functional nature. The term feature captures both.

A feature has the property that it is a service if it is localized in one component and if it refers to some functionality. However, a feature may be implemented in several components [Puver].

### **2.1.3. Aspect**

A concern that crosscuts a composition structure such as provided by a generalized procedural language (GP language) is an aspect. Aspects are explicit abstractions for representing crosscutting concerns.

In GP languages, functions, objects, modules etc. are seen as procedures because they are composed in a calls-upon or sends-message-to structure and each generalized procedure encapsulates a functional unit of the overall system. It is used the term module rather than the procedure to refer to the units of composition provided by a GP language, and the term modular composition to refer to its composition structure.

The intent of Aspect-Oriented Programming is to provide new abstraction and composition mechanism which do not follow the GP model so as to improve separation of aspects.

As such, a given design problem is decomposed into concerns that can be localized into separate modules and concerns that tend to crosscut over a set of modules. To specify the points that the aspect crosscuts a so-called pointcut specification is used. By providing explicit language constructs the notion of aspect becomes an explicit first-class abstraction. The crosscutting is actually localized in the pointcut specification.

Some examples of common aspects are: synchronization, logging, security, failure handling, debugging support etc.

## 2.2. Aspect-Oriented Programming (AOP)

AOP provides extensions to already existing programming languages, so that it becomes possible to modularize crosscutting concerns. AOP provides explicit abstractions for representing crosscutting concerns, called *aspects*.

Most of the existing AOP tools are based on the idea of having a language that specifies where the concern is spread across existing code and what should be done at these points. After the concerns have been specified a *weaver* is employed to weave the aspects into the existing source code, that is, the code without the tangled concerns.

An AOP language is not a stand-alone programming language. It merely complements an already existing language. Most of the time this underlying language is using the object-oriented paradigm, however this is not a requirement. In object-oriented programming the idea is to search for commonalities, and push them up in the hierarchy. With AOP the idea is to specify tangled concerns and represent them as first-class entities.

One important aspect of an AOP language is how it defines an aspect. To specify the points that the aspect crosscuts a so-called *pointcut specification* is used. A pointcut specification is in essence a predicate over the complete set of joinpoints that the aspect can crosscut. A pointcut specification can enumerate the joinpoints or provide a more abstract specification.

The crosscutting is actually localized in the pointcut specification. The pointcut specification indicates which points the aspect crosscuts but it does not specify what kind of behavior is needed. For this the concept of *advice* has been introduced. An advice is a behavior that can be attached before, around or after a joinpoint in the pointcut specification. Several aspect-oriented languages exist in the literature and they differ in the way of specifying aspects, pointcuts and advices.

A concept of AOP is that a programmer writes a crosscutting concern in an aspect, not many objects. A programmer writes aspects and links them to a target program using compiler or framework.

AOP has some disadvantages:

- While the understandability of the separate modules is improved, the understandability of the whole program becomes harder. This is because the weaver weaves code at some places where there is no explicit call to that code;
- Debugging of aspect oriented code is harder too, because debugging of the code is done over the weaved code and not in the separate modules.

## **2.3. Finding concerns**

Tool support can reduce the costs of software maintenance and evolution, but many tools are limited in their ability to manage information for large-scale software evolution. In this section we argue that the map metaphor can serve as an organizing principle for the design of effective tools for performing global software changes. We describe the design of Aspect Browser, developed around the map metaphor.

### **2.3.1. ASPECT BROWSER tool**

Concerns can be described in terms of subsets of the program text matching different queries.

The Aspect Browser is a tool that helps developers to find concerns using lexical searches of the program text. Concerns found in this fashion can be stored and viewed at different times to support program evolution tasks [GrisYu].

Aspect Browser is a graphical tool that uses the map metaphor to assist finding, exploring and manipulating crosscutting concerns. All the files in a program are displayed as a row of small windows in which each line of code in a file corresponds to a row of pixels in a window. Each occurrence of a crosscut is highlighted in a window with a specific color, like symbols on a map.

Aspect Browser's atlas metaphor feature can split the files of a large project into multiple views. It also enhances the scalability (AB does not have to load the entire source code into view) and the performance (commonly used views can be cached). The views can be used to organize code and to reduce the number of files the programmer has to focus on.

Aspect Browser has two parts:

*Aspect Emacs (AE)* is an Emacs-Lisp extension. It provides the core browsing capabilities of AB. An aspect in AE is defined as a pair of a regular expression and a color. When an aspect is activated, AE highlights the matching text in any displayed buffers with the aspect's corresponding color.

AE manages the aspects through a browser. The browser shows the pattern, color, match count, and annotation of each aspect. The user can perform a number of operations on aspects: create, delete, hide, show, show in a compressed view, change color, and edit annotation.

For finding the aspects, AE uses two simple external tools:

- The redundancy finder tool searches the entire project to find significant redundancies in the code, reporting any line that appears more than once. This approach is effective at identifying code written with copy-paste programming.
- The aspect finder tool extracts fragments of identifier names (tags) from source code according to a programmer specified naming convention. For example the name `delete_source_file` contains three tags, `delete`, `source` and `file`, separated by underscores. It is assumed that the programmer has used some conventions to “baptize” the variables.

*Nebulous* - is the visual part of the program. Each file is represented as a vertical strip, where the first row of pixels in the strip represents the first line of code, and so forth.

The aspects relieved in AE are displayed in their corresponding color. Because there are displayed a lot of files, the programmer perceives the real measure of scattering the aspect in the code. When more than one aspect is active, more colors are displayed. If more than one aspect resides in the same line, however, *Nebulous* shows the line in red to indicate that there is an “aspect collision”. This is necessitated by the low resolution of the view, but it also proves to be useful in software evolution tasks, as it provides a cue as to where aspects are likely interacting.

For each performed operation done by the programmer on an aspect, AE sends a message to *Nebulous* and this will change the display accordingly. In addition, a double-click on a strip in *Nebulous* window causes Emacs to open the file and display the portion of the file where the user has clicked. Zooming in this way leaves a red round cursor in the *Nebulous* view, helping the programmer stay oriented when returning to *Nebulous*.

However, the tool has some disadvantages:

- It supports the specification of concerns based on lexical matches to regular expressions and so, its expressive power is limited;
- It manages the aspects through a browser that shows the pattern, color, match count, and annotation of each aspect, so user encounters in case of a plurality of aspects. Programmers probably cannot view too many aspects at a time because of the overwhelming number of colors. Also, on a bigger project, the number of aspects will increase;
- It doesn't support the detection and reparation of inconsistencies between a concern and a base code; and the tools' ability is limited to capture relationships between scattered program elements

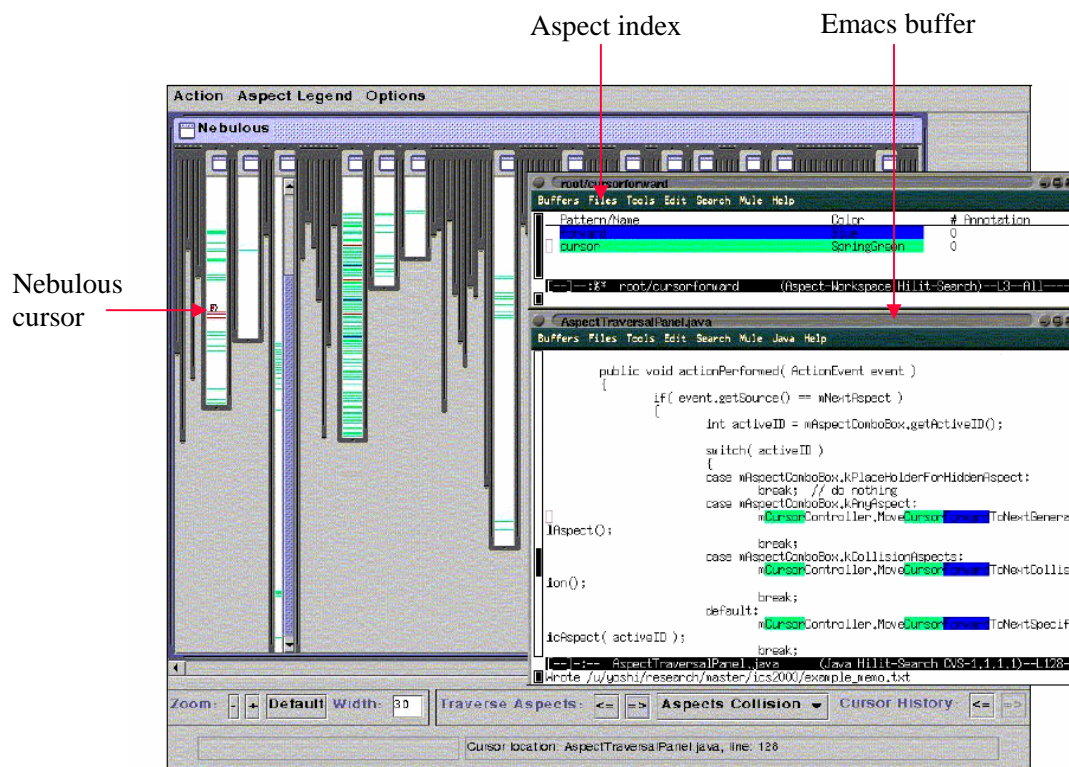


Figure 2 The AB windows

## 2.4. Modeling concerns

The job of a concern space is to organize the units in the body of software so as to separate all important concerns, to describe various kinds of interrelationships among concerns, and to indicate how software components and systems can be built to from the units that address these concerns.

To accomplish these goals, concerns have to be identified, which is the process of selecting concerns and populating them with the units that pertain to them.

The concerns must also be encapsulated such that they can be manipulated as first-class entities, that is, concerns must be modeled.

Once concerns have been encapsulated, it is possible to find interactions or integrate them to create software that addresses multiple concerns.

The following paragraphs present some approaches found in literature that uses concern modeling and attempt to find interrelationships between concerns.

### **2.4.1. Hyperspace approach to concern manipulation**

A hyperspace is a concern space specially structured to support multi-dimensional separation of concerns by organizing concerns in a multi-dimensional matrix. [OssTarr]

In hyperspace approach, concerns are grouped into dimensions (some kind of axis of reference), giving hyperspaces, an explicitly multi-dimensional structure. A dimension of concern is a set of concerns that are disjoint.

This implies that concerns in the same dimension cannot overlap, though concerns in different dimensions can and do. In [OssTarr] the authors use the term *multi-dimensional separation of concerns* to denote separation of concerns involving multiple, arbitrary kinds (dimensions) of concerns.

Separation according to these concerns *simultaneously*; i.e., a developer is not forced to choose a small number of "dominant" dimensions of concern according to which to decompose a system at the expense of others.

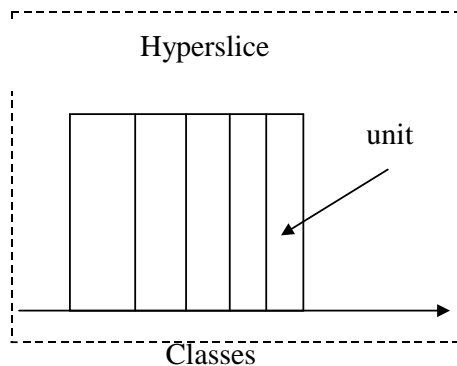
The authors define the following terms:

- A *unit* is a syntactic construct and might be: a declaration, statement, class, interface etc. There are: *primitive units*, which are treated as atomic (e.g. a method, instance variable, performance requirement) and *compound units*, which groups units together (e.g. a class, package, collaboration diagram).
- A *hyperspace* is a concern space whose structure supports multi-dimensional separation of concerns. A hyperspace also contains a set of *hypermodules*, which are modules based on concerns.

- Each hypermodule specifies a set of *hyperslices* - collections of units specified in terms of the concerns in the hyperspace-and a *composition rule* that specifies how the hyperslices are to be integrated. Hyperslices are generalizations of subjects from subject-oriented programming and hyperslice composition is derived from subject composition. Hypermodules are building blocks, and are not, in general, complete, executable programs. A *system* is a hypermodule that is complete, and can therefore run independently.
- A hyperspace can contain many hypermodules realizing different modularizations of the same units. Systems can be composed in many ways from these hypermodules.

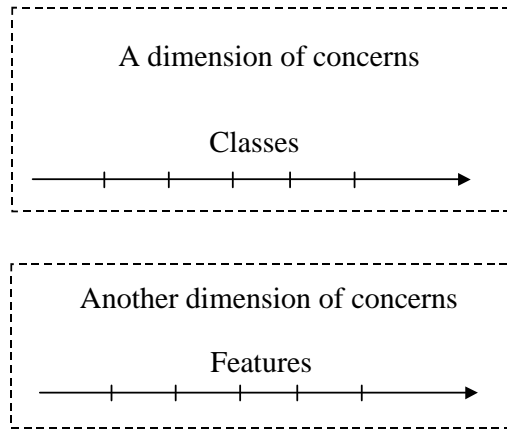
The main idea in this approach is that a mechanism of separations of concerns has to be followed by a mechanism of integration, which is the inverse operation, the fusion of concerns.

If it is desired a decomposition after the classes, it is defined a dimension, a kind of axis of reference, where all the classes desired are placed and the software is decomposed such as to make a correspondence between each class and unit. Because it is not sure that we refer all the software, to each of this axis it is added a concern called *none*, which contains the units form the software whom not apperatain to any of the desired concerns. This decomposition it is called hyperslice (Figure 3).



**Figure 3. Hyperslice decomposition after the classes**

For making the decomposition after the features, they have to be associated by the user with first-ordered classes. Likewise for the classes, the hyperslices are created (Figure 4)

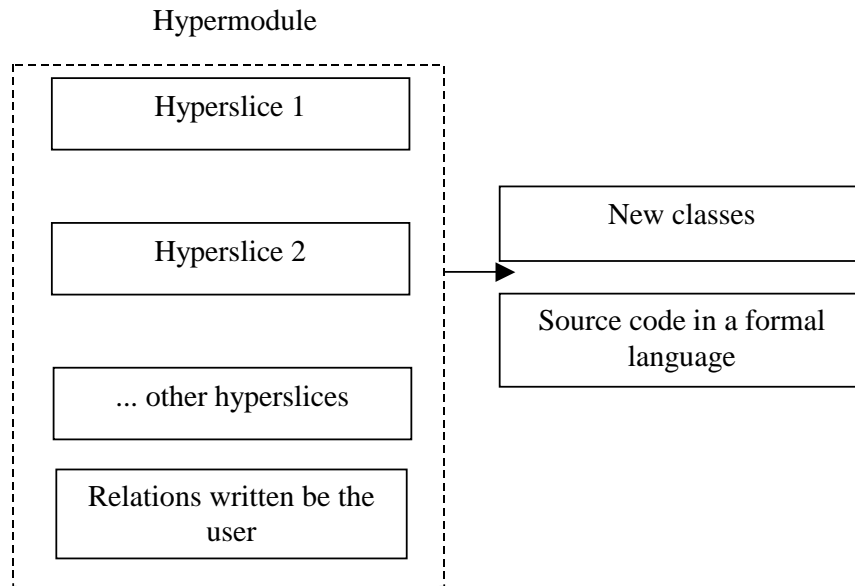


**Figure 4. Hyperslice decomposition after features**

The hyperslices can be created even if the code wasn't written, but the classes which are used are defined.

So, different decompositions of the same software after different criteria can be established, which are the hyperslices and which are memoryzed in a matrix of concern, where can be made some additions or changes.

The following step is the integration, where more hyperslices form a hypermodule. This is done by the user, by defining some relations in a Subject Oriented Programming language, through which is defined the way in which the new classes are formed in the case that are added new features or in which order the methods are called. As a result, new Java classes are created and also some kind of source code in a formal language (Figure 5). The obtained results are examined by the user and if they do not suit, other relations can be defined and the user can keep trying until the result is good or almost good. This can be used in "on demand re-modularization", namely creating different versions of the same program, or adding new features.



**Figure 5 A Hypermodule created in the Hyperspace approach**

Concern specifications in hyperspaces serve to identify the dimensions and their concerns, and to specify the coordinates of each unit within the matrix. Specifications can be extensional or intensional.

*Extensional specifications* explicitly enumerate the units in each concern.

*Intensional specifications* specify properties of concerns and units that can be used to determine whether a given unit pertains to a concern.

Concerns include classes and methods that are encapsulated such that they can be manipulated as first-class entities and organized in a multi-dimensional matrix.

The integration is an iterative process that makes use of a Subject Oriented Programming language, through which is defined the way in which the new classes are formed in the case that are added new features or in which order the methods are called.

We have studied the hyperspace approach, with the purpose to know how concerns are defined in other works in the literature and how the interrelationships between concerns are specified.

In hyperspace we have seen that concerns include classes and methods and no more refined granularities such as statements or variables are allowed. Also, concerns are organized in a complex multi-dimensional matrix and they make use of a Subject Oriented Programming language that specifies relationships between concerns.

Although this approach makes valuable contributions by satisfying some of the goals of multi-dimensional separation of concerns, like identification, encapsulation, and integration, is not clear what is the concerns structure and the nature of their interactions and the multi-dimensional matrix structure seems to be a bit complicate.

### 2.2.5. Concern Space Modeling in COSMOS

Cosmos is a general purpose, multidimensional Concern-Space Modeling Schema that allows concerns to be modeled independently of development formalism, tools and methods.

Cosmos includes three types of elements: concerns, relationships and predicates (Table 1).

**Table 1 Outline of the Cosmos concern-space modeling schema**

Core			Extensions
Concerns	Logical	Classifications, Classes, Instances, Properties, Topics	
	Physical	Collections, Instances, Attributes	
Relationships	Categorical	Classification, Generalization, Instantiation, Characterization, Topicality, Membership, Attribution	
	Interpretive	Significance	Admission, Contribution, Logical-Implementation, LogicalComposition, Motivation
	Mapping	Association	Affecting, Description, Modeling, Physical-Implementation, Representation
	Physical	PhysicalRelation	Connection, ConnectionTo, Physically-Affecting, PhysicalComposition
Predicates	<no subtypes defined>		

Concerns are divided into logical concerns and physical concerns.

Logical concerns represent the conceptual “matters of interest” in a software system. Examples include functionality, behavior, performance, robustness, state, coupling, configurability, usability, size, cost, and so on.

Physical concerns represent “real world” elements of a system, potentially including software, hardware, systems, and services. Physical concerns are represented in Cosmos for two main reasons. First, these are the things, such as software artifacts, to which our (logical) concerns apply and through which our logical concerns are realized. Second, these real-world things are of direct concern themselves, not just as derivative or supportive of logical concerns, but as work products, deliverables, and so on.

Relationships are divided into categorical relationships, interpretive relationships, physical relationships and mapping relationships.

Predicates represent integrity conditions over various relationships and can be classified accordingly. For example, categorical predicates apply to categorical relationships and interpretive predicates apply to interpretive relationships. One of the former is that no concern can be both a kind and an instance; one of the latter is that if one concern motivates another then the second concern should contribute to the first.

Cosmos provides a kind of semantic hyper-index for concerns in a software system and in its component artifacts. Concerns that will crosscut multiple objects should be noted in the design and then modeled separately.

## **2.5. Concern interaction**

Different techniques have been proposed to help developers identify concern interaction. This section presents some approaches for concern and features interaction detection.

### **2.5.1. FEAT (Feature Exploration and Analysis Tool)**

In Feat, the concerns are modeled as Concern Graph. [Feat] A Concern Graph (CG) is a collection of concerns relevant to a particular task that is displayed as a forest of trees. The root of each tree in the forest is a class that contributes to the implementation of the concern.

Displaying a CG as a collection of trees has two advantages:

- trees are easier to lay out than graphs;
- tree nodes can be collapsed to abstract information.

The main idea behind CGs is to use them to record your knowledge as you investigate various concerns of interest in a program. Once concerns are available in FEAT, they can be compared to see if they are inter-dependent.

FEAT has 3 main windows:

- Concern Graph – where it can see the graph;
- Participants - where all the program elements and their relations are displayed;
- Projections - where some details are displayed.

FEAT also allows the automatic display of the source code related to a concern (Figure 6).

A developer creates a Concern Graph (CG) by iteratively querying a model of the program, and by determining which elements and relationships returned as part of the queries contribute to the implementation of the concern.

In order to identify a concern, Feat allows some pattern search in the source code, so the user can inspect the appropriate code and he can build the concern, based on the obtained result.

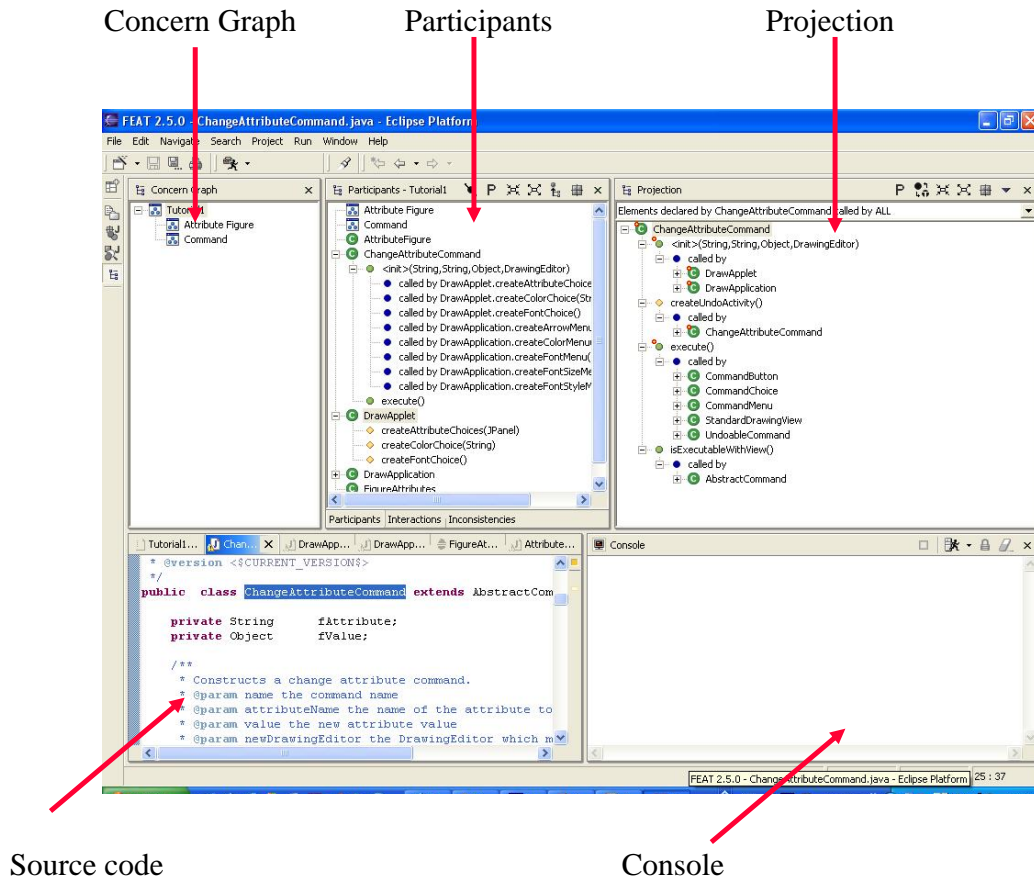
For example if the programmer knows that he has to modify something related to “Attribute”, the easiest way to specify this concern is to search in the source code after the “Attribute” string, and then he can build the concern, based on the obtained result. Looking only at these parts of the program, the user decides which classes and methods are added to the concern.

After that, the user has to see where the implicated classes are called, from who, where is defined a constructor etc. This is done in the Participants View where, with a right click, one can query the program. There are two kinds of queries:

- **Fan-out** - queries relations that describe how an element is referenced by other elements;
- **Fan-in** - queries relations that describe how an element references other elements.

Based on these queries, some relations in the program can be found, which are displayed in the Projection window.

The user looks in the appropriate source code and he can see that are other methods which are also used (these methods are called or they call the already found methods) and which has to be part of the concern.



**Figure 6 The FEAT windows**

The user is also allowed to see how concerns are interdependent. When a concern graph is subdivided into different sub-concerns, it is possible to analyze two concerns in the hierarchy to determine their interactions. To determine the interactions between two concerns, a developer selects the concerns in the Concern Graph View, right-clicks, and selects **Compare** from a pop-up menu. The results of the analysis appear in a view called the **Interactions View**, which overlaps with the **Participants View**. The **Interactions View** shows the participants of the two selected concerns side by side. Elements that are present in more than one concern are flagged with a red diamond. Elements that have relations to any element in other concern are flagged with a yellow diamond (Figure 7).

The advantage is that the program can find out automatically which elements are involved in the searched functionality, the appropriate source code is displayed and the user can do the changes easier. The relations can be either structural or behavioral (Table 2). Structural relations represent static, declarative relations between elements in a program. Behavioral relations represent code within a method.

The current Feat tool provides no support for the user to add new relations for investigating new kinds of feature interactions, the relations defined and the queries the user performs being fixed by the tool. Adding such support to adequately support the specific user-defined queries, by means of logic programming, is one of the goals of our work. This facility would increase the power of a tool that investigates the concern interactions.

**Table 2 Concern Interaction in FEAT**

<b>Relation name</b>	<b>Relation type</b>	<b>Definition of the relation</b>
HasParameterType	Structural relation	<b>HasParameterType(x,y)</b> The function returns true if x is a method (abstract or concrete), y is a non-primitive type, and y is contained in the list of parameters of x
HasReturnType	Structural relation	<b>HasReturnType(x,y)</b> The function returns true if x is a non-constructor, noninitializer method(abstract or concrete), y is a non-primitive, non-void type, and y is the return type of x.
OfType	Structural relation	<b>OfType(x,y)</b> The function returns true if x is a field, y a non-primitive type (class or interface), and x is declared to be of type y.
ExtendsClass	Structural relation	<b>ExtendsClass(x,y)</b> The function returns true if x and y are both class types, and x is declared to extend y
ExtendsInterface	Structural relation	<b>ExtendsInterface(x,y)</b> The function returns true if x and y are both interface types, and x is declared extend y.
Implements	Structural relation	<b>Implements(x,y)</b> The function returns true if x is a class type, y an interface type, and x declares to implement y .
Overrides	Structural relation	<b>Overrides(x,y)</b> The function returns true if x is a concrete method, y is a method (concrete or abstract), and x overrides y.
Accesses	Behavioral relation	<b>Accesses(x,y)</b> The function returns true if x is a concrete method, y is a field, and x contains a field access expression referring to field y.

Calls	Behavioral relation	<b>Calls(x,y)</b> The function returns true if x is a concrete method, y a concrete or abstract method , and x contains a method invocation expression.
Creates	Behavioral relation	<b>Creates(x,y)</b> The function returns true if x is a concrete method, y is a class type, and x contains a class instance creation expression naming type y.

In order to automatically extract potential concern descriptions, the developers propose an algorithm that can generate concern descriptions based on a calculation of how related different elements were during a program investigation session.

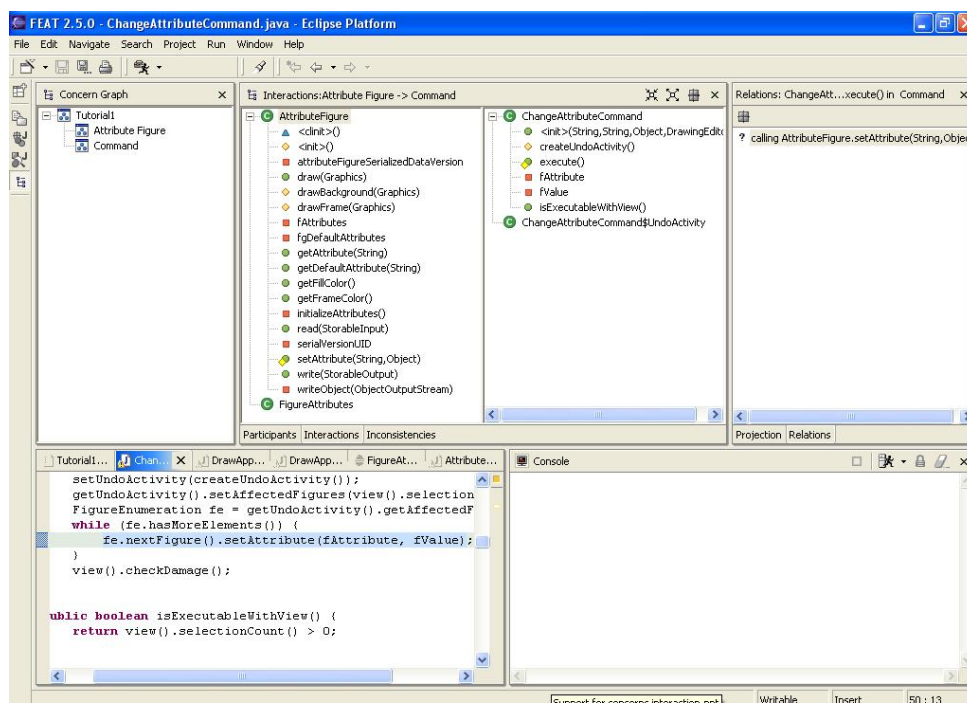


Figure 7 Interaction View in FEAT

## 2.3. Summary and Conclusions

This chapter presented first some definitions related to features, aspects and concerns. Being that the goal of this dissertation is to find concern interaction, it was important to consider existing terminology and definitions related to the subject.

Following this, an overview of the AOP and the existing tools intended to find concerns (Aspect Browser Tool), concern modeling (COSMOS and HyperJ) and concern interaction (Feat) was outlined.

The Aspect Browser tool demonstrates the applicability of the map metaphor to issues in software evolution. In the hyperspace approach, concerns are grouped into dimensions, giving hyperspaces, an explicitly multi-dimensional structure.

Cosmos is a general purpose, multidimensional Concern-Space Modeling Schema that allows concerns to be modeled independently of development formalism, tools and methods, but is a bit too abstract. We have studied Cosmos because it provides a kind of semantic hyper-index for concerns in a software system and in its component artifacts.

In order to accomplish the goal of this dissertation, we have to find interrelationships between concerns, so it is useful to document us about the studies and works in this domain. We found that the classification, the relationships and predicates outlined in Cosmos are very close with our purposes and we can use some of these semantic approaches in concern interaction detection that is presented in Chapter 5.

FEAT models concerns as Concern Graph that is displayed as a forest of trees. The concern graph model has many advantages, like: the concept of concern graphs is a model for describing concerns in source code based on relations between elements defined in a program; the general structure representing concerns is language independent; the concern graph model is precise, meaning that there exist a non-ambiguous mapping between any structure present in a concern representation and the corresponding source code.

Anyway, the Feat tool also has some drawbacks in detecting feature interactions:

- The relations defined and the queries the user performs, are fixed by the tool;
- The user cannot add new queries to investigate new kinds of feature interactions ;

The thesis purpose is to model concerns as Concern Graph in the Feat tool and to overcome the two major disadvantages of Feat mentioned above by using the logic meta-programming approach, particularly the SOUL language.

## Chapter 3. Logic Meta-Programming

Our approach uses logic meta-programming rules in order to detect concern interaction. We've chosen logic meta-programming because of its inherent declarative nature and because we need to extract information out of the source code.

The next sections define the concept of Logic Meta-Programming (LMP) as a variant of the logic paradigm that allows to query about code and express structural relations in various levels.

Also we explain some of the predicates provided by SOUL (which is an example of LMP system) that are further used for representing concerns, defining interactions, and for detecting advanced concern interactions, that are presented in Chapter 5.

### 3.1. Logic Meta-Programming

Logic programming is based on first-order predicate logic. Programs in a logic language are written by specifying as facts the base knowledge that is available about a problem and the relationships between this knowledge. The program will derive new information out of these facts by using rules, reasoning about knowledge, thanks to its capacity to support multi-way queries and the built-in techniques of unification and backtracking.

Logic programming languages are inherently high-level because they focus on the computation's logic and not on its mechanics. The result is that they are particularly suited for rapidly prototyping data structures and code to express complex ideas because the memory management, stack pointers, etc., are left to the computational engine. The engine incorporates logical inference, and is a powerful tool, which can be exploited in developing inference engines specific to a particular domain. [Flach]

Examples of logic programming languages are PROLOG and SOUL.

Meta-programming is the process of writing computer programs that can manipulate representations of other programs. A meta-program, regardless of the nature of the programming language, is a program whose data denotes another program.

Logic programming has been identified as very suited to meta-programming and language processing in general. The choice of logic programming as a basis for meta-programming offers several practical and theoretical advantages: among them, the possibility of tackling critical foundational problems of meta-programming within a strong theoretical framework, and the surprising ease of programming. Therefore it follows that logic programs can be used to specify base language programs indirectly.

In a declarative approach, it seems clear that design information is best expressed as logic facts and rules. Logic meta-programming has a declarative approach that focuses on *what* the base language does by means of present entities and relations, instead of *how* the computations are effectuated.

Logic meta-programming is an instance of declarative meta-programming, where a base program is represented indirectly by means of a set of logic propositions. These logic propositions are stored in a logic data repository and the link between the actually represented program and its logic representation is concretized under the form of a code generator. Basically the code generator performs queries to find out what it needs to know in order to construct the base program and outputs code in the base language.

The full power of the logic paradigm may thus be used to describe base-language programs indirectly. This can be achieved in various ways. In the case of Smalltalk, the reflection facilities can be used to implement a logic meta-layer on top of the Smalltalk meta-object protocol. [DeVolder99]

The logic meta-programming approach allows building software tools and environments, which belong to one of the following categories:

- verification of source code to some higher-level description, for example, conformance checking to coding conventions, to design models or to architectural descriptions;
- extraction of information from source code, for example, visualization, software understanding, browsing, generation of higher-level models or documentation, measurements, quality control;
- transformation of source code such as refactoring, translation, re-engineering, evolution, optimization;
- generation of source code.

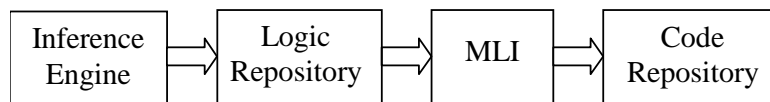
As already mentioned this dissertation uses a logic meta-programming approach for concern interaction detection. This approach fits with the broader research on logic meta-programming at the Programming Technology Lab (PROG) of the Vrije Universiteit Brussel.

### 3.2. Logic Meta-Programming with SOUL

SOUL is short for **S**malltalk **O**pen **U**nification **L**anguage. SOUL is an open, reflective logic programming language written in VisualWorks.

The SOUL system was developed by R.Wuyts at the Programming Technology Lab of the Vrije Universiteit Brussel [Rwuyts98].

SOUL consists of four major modules: the logic inference engine, the LiCoR library, the meta-level interface, and the base language code repository.



**Figure 8 SOUL modules**

The heart of the system is a logic Inference Engine. The meta-level interface (MLI) provides the representational mapping between the logic meta-language and the object-oriented base language. MLI is, in fact, a hierarchy of classes, sub-classing from an abstract MLI class, which declares the available meta-level interface.

By implementing a specific MLI subclass for every target language (or dialect), SOUL allows reasoning about multiple base-level languages (or dialects).

In the figure, the logic repository contains all logic predicates available to the Inference Engine. While a number of primitive logic predicates are distributed with SOUL, an additional library, called LiCoR consist in predicates intended for reasoning about object-oriented source code.

Figure 8 shows the different layers that are present in this logic meta-programming approach. When a query using a predicate of the Logic Repository to reason about code is performed in the inference engine, the predicate calls the MLI, which consults

the code repository to extract the actual structure from the code. The implementation of the predicates is relied on the MLI, which is language dependent.

SOUL allows reasoning about the structure of class-based object-oriented languages. The language that is reasoned about is represented using an internal parse tree representation, making SOUL a meta-language for Smalltalk.

The symbiosis between SOUL and Smalltalk was made possible by implementing SOUL entirely in Smalltalk and by using the reflective capabilities of the Smalltalk environment. It is thus possible to query Smalltalk code and derive relations between static elements. This allows us to express, extract and enforce the link between design and implementation.

Logic programs can be written in SOUL, that turns the Smalltalk code repository into a database that can be queried.

On the other hand, SOUL is based on TyRuBa, also a Prolog interpreter that has been extended by a quasi-quoting mechanism that enables the treatment of strings containing logic variables. By querying the interpreter, these logic variables get bound and the strings get glued together. This way, a query launched yields a result string representing anything the corresponding Prolog program describes.

Similarly with Prolog, SOUL language uses constants, variables, functors, lists, predicates, facts, rules and queries. SOUL allows also to write Prolog programs that:

- Query the underlying Smalltalk code base;
- Generate strings representing Smalltalk programs by using the quasi-quoting mechanism;
- By accessing the Smalltalk open programming environment, actually allow one to change the Smalltalk code base, and thus allow one to add generated code to the running Smalltalk environment.
- in a later version of SOUL, support for the Java programming language was added.

### 3.2.1. Syntax of SOUL

The syntax of the language is similar to that of the Prolog, but has an extension that allows logic clauses to manipulate elements from the Smalltalk language. It is even possible to execute blocks of Smalltalk code during the interpretation of queries.

The ways in which the syntax of SOUL differs from the syntax of Prolog are the following:

- The type of a name is not determined by whether that name starts with a lowercase or an uppercase letter. Whereas prolog recognizes variables by the first letter being an uppercase character, SOUL recognizes variables by a question mark preceding a name;
- The well-known :- sign has been replaced by the if keyword, so that rules are expressed by stating the head of the rule, followed by if, followed by the body of the rule. The body consists of predicates separated by commas, just like in Prolog; [DeMeuBrich03]
- Lists are delimited by less-than ("<") and greater-than (">") symbols, rather than square brackets as in Prolog.

To illustrate the difference of syntax between SOUL and Prolog we define the predicate sublist which tests whether a list is sub-list of another.

The Prolog implementation is the following one:

```
sublist ([ ,_ ) .
sublist ([ E | Rest ], List1 ) :-
member (E ,List1 ),
sublist ( Rest ,List1 ) .
```

While the SOUL implementation is given by:

```
sublist (<>,?).
sublist (<?e | ?rest > ,?list1 ) if
member (?e ,?list1 ),
sublist (? rest ,?list1 )
```

### 3.2.2. Smalltalk terms and Smalltalk clauses

Besides the small syntactic differences presented in the previous paragraph, SOUL has a number of distinguishing features not present in a typical Prolog implementation, most notably the Smalltalk terms and Smalltalk clauses.

This way, Prolog programs can alter Smalltalk code and continue their reasoning, based on these changes.

A terminology of ‘up’ and ‘down’ is used, based on the idea that the Prolog is reasoning about the ‘underlying’ Smalltalk. Hence, Prolog is the upper level (‘up’), while the Smalltalk is considered to be the lower level (‘down’). When the SOUL interpreter has to move a value from the up level to the down level, ‘the value is downed’. Conversely, when the SOUL interpreter moves a value from the Smalltalk level back to the Prolog level, the value is ‘upped’.

However, upping and downing is something that is done transparently for the programmer.

Instead, SOUL conceives two simple ways of connecting Smalltalk and Prolog: Smalltalk terms and Smalltalk clauses.

The use of Smalltalk terms enables the execution of blocks of Smalltalk code as part of logic clauses. To interpret these language primitives, the Smalltalk compiler is invoked to compute their value.

A Smalltalk term is a logic term containing any Smalltalk expression enclosed in square brackets [...] and the expression is evaluated each time the Prolog engine has to unify it with another term. This expression is allowed to contain logic variables that are downed to Smalltalk. Then the Smalltalk term in Smalltalk is evaluated. The result of the expression in the Smalltalk term is upped for the Prolog level and this result is used in the unification. This way it is possible to refer to real Smalltalk entities from within SOUL.

[MeBrMe03]

For example, in the following SOUL logic query:

```
if equals(?x, [ SortedCollection new ]).
```

expressions between brackets (‘[’ and ‘]’) are blocks of Smalltalk code, and the value of these expressions are the result of the evaluation of the block. In this case, [SortedCollection new] evaluates to an instance of the class SortedCollection, and this is the value bound to the variable ?x in the only solution for this example query:

```
1st solution: {?x ! aSortedCollection}
```

In addition to declarations of terms, blocks of Smalltalk code can be used also as clauses, with the same syntax. The only difference is that a Smalltalk clause is used in SOUL as a predication, and hence, it is required to always return either true or false.

For example, in the following rule, which verifies if an integer is odd:

```
odd(?anInteger) if  
[ ?anInteger odd ].
```

The clause [ ?anInteger odd ] calls the method odd of that object, which is supposed to be of type Integer. This method returns true or false, implying, respectively, the success or failure of the query.

The variable ?anInteger is replaced by the value to which it is bound at the time of clause evaluation. [DelValle03]

### 3.2.3. LiCoR Library

LiCoR (Library for Code Reasoning) is an additional library of over 500 logic rules, provided for reasoning about object-oriented source code.

LiCoR reifies object oriented code and provides basic predicates, typing predicates, code convention predicates, design pattern predicates, UML predicates. These predicates can be used for verifying entities or relations, or for extracting them, in order to reason statically about object-oriented source code.

LiCoR is largely language independent, relying on a language dependent meta-level interface that executes instructions for extracting the base language structure.

LiCoR has a layered structure[RoWuyts01]. The lower layers contain the more primitive predicates, and the higher layers use the lower ones to create more abstract reasoning predicates (Figure 9).

LiCoR reifies conceptual structural information about classes, methods, instance variables and inheritance.

The base layer contains predicates dedicated to querying the base level code by accessing the meta-level interface.

For example, the predicate:

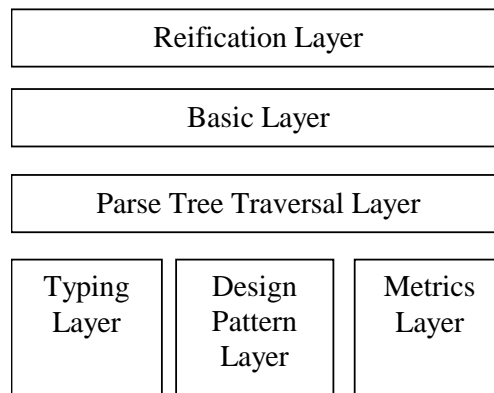
```
class(?c)
```

succeeds if the logic variable ?c is bound to a class that exists in the object-oriented source code.

If ?c is bound, it verifies whether the given value is indeed a class.

If ?c is unbound, class(?c) returns multiple bindings, using the generate predicate, which will successively bind ?c to an element of the collection of all classes returned by the meta-level interface:

```
class(?c) if atom(?c), [Soul.MLI current isClass: ?c].  
class(?c) if var(?c), generate(?c, [Soul.MLI current  
allClasses]).
```



**Figure 9 LiCoR structure**

LiCoR contains parse tree traversal predicates, which allow to recursively traverse the logical representations of the syntactic trees in a declaratory way.

Smalltalk is dynamically typed. The Typing Layer of LiCoR uses SOUL to reconstruct types of instance variables of classes in some possible ways:

- It finds all methods sent to the instance variable, and finds all classes that understand those methods;
- Uses the parse tree traversal predicates;
- Looks at assignments, factory methods etc.

The Design Pattern Layer contains logic facts and rules that express specific design pattern information, the classes involved, the restrictions and relations they follow, while the Metrics Layer consist in facts and rules intended to use different metrics related to the object oriented code.

### 3.3. SOUL predicates

In this thesis we focus on providing support for concern interaction detection, by means of finding the ways in which different software entities interact, in order to assist software evolution tasks.

Towards accomplishing this goal, we use the predicates provided by SOUL to represent code entities and for reasoning about classes, methods, variables and inheritance, to define and detect concern interactions.

In this section we explain some of the predicates provided by SOUL, that are further used for representing concerns, defining interactions, and for detecting advanced concern interactions, that are presented in Chapter 5.

**General logic predicates**, such as `findall`, `forall`, `not`, `or`, `and`, `append`, `list` etc. that are part of SOUL are used in a similar manner as in Prolog, with the mentioned syntax differences.

#### **Predicates reasoning about classes**

In this section, we give some examples of SOUL predicates reasoning about classes.

```
class(+?-?Class)
```

verifies whether `?Class` is a Smalltalk class or meta class. If `?Class` is variable it will be instantiated with a Smalltalk class or meta class. If `?Class` is bound it should be an existing class or meta class in the current Smalltalk image.

This predicate is implemented in four cases:

1. `class(+?Class)` checks whether `?Class` is an existing class

```
class(+?Class) if
```

```
    [ Soul.MLI current isClass: ?Class ]
```

2. `class(-?Class)` returns an existing class or metaclass

```
class(-?Class) if
```

```
    member(?Class, [ Soul.MLI current allClassEntities ])
```

3. `class(+?-?Class)` verifies whether `?Class` is a Smalltalk class or meta class. If `?Class` is variable it will be instantiated with a Smalltalk class.

```
class(?AClass) if
```

```
    atom(?AClass),
```

```
    [ Soul.ExplicitMLI current isClass: ?AClass ]
```

4. `class(-?AClass)` generates ordinary as well as metaclasses

```

class(?AClass) if
    var(?AClass),
    generate(?AClass, [ Soul.ExplicitMLI current allClasses ])

abstractClass(?class)

```

defines that a class is abstract if there is at least one abstract method the class can understand.

```

abstractClass(?class) if
    class(?class),
    one(abstractMethodInClass(?, ?class))

```

```

hierarchy(+?-?Ancestor, +- ?ADescendant)

```

states whether ?ADescendant belongs to the class hierarchy with root class ?Ancestor. A class is in the hierarchy of another class if it is a subclass (direct or indirect) of that class

This predicate actually belongs to the basic layer, because it can be written entirely in logic code by making use of the superclass/2 rule, as follows:

```

hierarchy(?Root, ?directSubclass) if
    subclass(?directSubclass, ?Root)

hierarchy(?Root, ?indirectSubclass) if
    subclass(?directSubclass, ?ASub)
    hierarchy(?directSubclass, ?indirectSubclass)

```

However, for performance reasons it is implemented it by calling Smalltalk directly.

### **Predicates reasoning about methods**

In this section, we give some examples of SOUL predicates reasoning about methods.

```

classImplementsMethodNamed(?class, ?selector, ?method)

```

states that ?class implements some method with name ?selector and parse-tree representation ?method

```

classImplementsMethodNamed(?class, ?selector, -?method) if
    methodWithNameInClass(?m, ?selector, ?class),
    parseTreeOfMethod(?method, ?m)

```

```
classImplementsMethodNamed(?class,?selector,+?method) if
    equals(?method,method(?class,?selector,?arguments,?tempor
aries,? statements))
```

### **Predicates reasoning about statements**

```
statement(?statement,?class,?method)
```

states that ?class implements some method with parse-tree representation ?method and statement ?statement.

```
statement(?statement,?class,?method) if
    classImplementsMethodNamed(?class,?method,?m),
    methodStatements(?m,?statements),
    member(?statement,?statements)
```

### **Predicates reasoning about variables**

```
classVar(+?-?Class,+?-?ClassVar)
```

defines the relationship between a class and its class variables. Both arguments may either be instantiated or not.

```
Case1: class variable is atom
classVar(?AClass,?Var) if
    atom(?Var),
    class(?AClass),
    [ Soul.ExplicitMLI current isClassVariable: ?Var inClass:
?AClass ]
Case2: class variable is variable
classVar(?AClass,?Var) if
    var(?Var),
    class(?AClass),
    generate(?Var,[ Soul.ExplicitMLI current
classVariablesInClass: ?AClass ])
```

```
instVar(+?-?InstVar)
```

checks whether ?InstVar is the name of an instance variable of some class, or returns the name of an existing instance variable when ?InstVar is unbound.

```
instVar(+?-?aClass,+?-?InstVar)
```

verifies the relationship between a class and its instance variable names. One or both arguments may be unbound in which case the appropriate values will be generated for them.

```
Case1: instVar(+?-?AClass,+?InstVar)
instVar(?AClass,?InstVar) if
    atom(?InstVar),
    class(?AClass),
    [ Soul.ExplicitMLI current isInstanceVariable: ?InstVar
inClass: ?AClass ]
```

```
Case 2: instVar(+?-?AClass,-?InstVar)
instVar(?AClass,?InstVar) if
    var(?InstVar),
    class(?AClass),
    generate(?InstVar, [ Soul.ExplicitMLI current
instanceVariablesInClass: ?AClass ])
```

Combining such predicates as presented bellow, one can write other predicates that can be used for concern representation and interaction detection, as presented in Chapter 4.

### **3.4 Intentional source-code views**

Intentional views enhance software understanding because they provide important knowledge about where and how certain concerns are implemented and how they relate with other concerns. Instead of describing different concerns in separate aspects that are weaved afterwards, the source code is modularized into a number of user-defined intentional views that may crosscut the actual implementation and that may be overlapping. Each intentional view corresponds to an important concern that maybe spread through the source code and groups a set of source-code entities that address this concern. [MensWer]

It becomes easier to manage the source code because important concerns have been made explicit in the intentional views, even if they are spread throughout the source code. When software evolves, it can be analyzed the constraints imposed by the intentional views and their relations to verify that no assumptions have been invalidate.

An intentional view makes use of L.M.P., particularly of SOUL and describes a set of source-code entities (classes, instances, variables, methods).

Using intentional descriptions has a series of advantages of over explicit enumerations of source-code entities:

- are often more concise;
- are more intuitive and intentional;
- are more robust towards changes;

The main disadvantages of this approach are:

- The efficiency of computation is low;
- The developer manually has to provide a description for a view, thus has to know about SOUL and about the structural information of elements in the view.

The main disadvantages of this approach are:

- The efficiency of computation is low;
- The developer manually has to provide a description for a view, thus has to know about SOUL and about the structural information of elements in the view.

The main disadvantage consists in the efficiency of computation and in fact that the developer manually has to provide a description for a view, thus has to know about SOUL and about the structural information of elements in the view.

### **3.5. Conclusion**

In this chapter we have presented the basic principles behind logic programming languages by means of the SOUL language. We discussed the notion of Logic Meta-Programming and outlined how we can write logic meta-programs with SOUL and the LiCoR library. In the context of this dissertation, we will use Logic Programming as the programming paradigm in which we implement the concern interaction detection tool.

The logic meta-programming approach presents a very natural way for reasoning about object oriented code and detecting concern interactions. This is achieved by using logic predicates to express and query the software structure.

Also, we've presented the Intentional source code-views, which make use of L.M.P and the source code is modularized into a number of user-defined intentional views.

In order to implement a concern interaction tool, we need a way for concern modeling. As presented in Chapter 2 there exist in literature some approaches to do this. We have chosen to represent concerns in a Feat like manner. In contrast with Feat tool, we use the Star Browser tool that provides an easy way to model and represent concerns. This tool is presented in the next chapter.

# Chapter 4. The StarBrowser

We have chosen to use the StarBrowser tool because it provides a tree like view model suited to concern modeling in a similar way as Feat tool does.

Also, it allows automatically code browsing and its user interface is very like Visual Works, the Smalltalk integrated development environment that is well known by the most part of programmers and so more easy to use.

The following paragraphs present the main features of the Star Browser tool, its advantages and the ways it can be extended.

## 4.1. Introduction

The *Star Browser* is a development browser that starts from the ideas of Koen D'Hondt, who had the goal to support the development process with classifications. Star Browser re-implements the classification model and the browser in a lightweight fashion, using meta-programming techniques and reflection.

By itself the StarBrowser provides only a toolbar, an interface to display classifications as a tree, a part where editors for these items can be shown, and a mechanism to allow a user to switch services using the ServicesConfiguration. It extends the classification model to support classes, methods, namespaces, packages, bundles, and parcels.

The *StarBrowser* consists of two parts: the *Lightweight Classification Model* and the *StarBrowser*, implemented in VisualWorks [Starindex]. The following paragraphs detail these parts.

## 4.2. The Lightweight Classifications Model

The Lightweight Classifications Model defines a software classification model as a meta-model that describes how software entities should be modeled to make them tangible in the software development environment.

The classification model allows to group all kinds of entities and to uniformly manipulate these entities and groups. Keeping all these entities together helps to reduce the complexity of the development process.

Particular about the model is that the grouping is independent of the manipulations, and that the model can be customized to either support particular groupings or to support particular manipulations or both. [Rwuyts03]

There are three main participants in the model: Item, Classification and ItemVisitor. A classification is a composite Item, and ItemVisitor is a visitor for items. There are no constraints on what can be classified, and the operations are separated from the items making this flexible as well.

A classification is a group into which items are classified, and basically is a container for items. Items do not have to be of the same type, and it provides behavior for enumerating and managing the items it contains. There are no restrictions on the number of items in a classification and an item may be classified into more than one classification. Since a classification is an item itself, it can be part of other classifications as well.

A service implements an action that can be performed on items. Depending on the kind of item, a service can perform a different action. But it should at least provide a default action that is able to process any kind of item that is passed. Examples of services can be a service that gets the children of an item, a label for an item, or a preferred editor.

The service configuration is a registry where services can register under a certain name to be retrieved by tools.

The basic model is implemented in Smalltalk. The implementation hierarchy has an abstract class AbstractItem, with a subclass for a wrapper that allows any object to be used as an item, and subclasses for the classifications.

Classifications can be extentional or intentional.

- An extentional classification just holds on to a collection of items that are just
- dragged from any kind of Smalltalk tool (stand-alone or embedded in the Star Browser) and dropped at their desired location. That way extentional classifications are used to group items of interest;

- An intentional classification uses a Smalltalk block to generate its contents whenever it is asked for.

The model can be extended in two ways:

- Operations on items are added as subclasses of `ItemVisitor`. A number of operations are provided, such as an *IconService* to get the icon for an item, a *LabelService* to get a short description, and more.
- Support for specific items is added by extending these items with an *itemActionFor:* method and extending the `ItemService` with a corresponding method.

The basic model also contains a *ServicesConfiguration*. This allows for dynamically extending the model with new kind of visitors and items. Basically, a `ServicesConfiguration` is a dynamic registration facility for visitors for registering their actions. The `ServicesConfiguration` is used by all classes that need to perform actions on items instead of referencing the visitors directly [Starindex].

The behavior of items is split in two: the behavior dealing with managing items (adding, removing, enumerating) is implemented on the items themselves while all other behavior regarding items (their icons, labels, editors) is implemented using the services, using the Visitor design pattern.

There motivations for choosing a visitor are:

- Keep the interface small. In the Smalltalk implementation, an `Item` can be any kind of Smalltalk object and only the method `acceptService:` has to be added to enable a `Visitor` to implement the services;
- The services can be changed at runtime, which is not possible if the behavior is directly implemented as methods on the item classes.

Objects are wrapped in a wrapper class `ObjectAsItemWrapper`, a subclass of class `Item` that implements the method `acceptService:` as follows:

```
ObjectAsItemWrapper_ acceptService: aService  
^self object acceptService: aService
```

Therefore any kind of object can be wrapped as an item provided it implements the method `acceptService:` where it determines what method needs to be called on the service to process that item.

### 4.3. The StarBrowser

On top of the Lightweight Classifications Model, a browser called StarBrowser is implemented. The goal of the StarBrowser is to allow the user to browse a Smalltalk environment and to classify anything.

Therefore it has full drag and drop support and allows the user to document his system in a flexible way. For example, while browsing a new piece of code the user can store some classes and methods that look interesting in an extentional classification, that can then later be used to remember these places.

Otherwise, the user can describe a classification in the Smalltalk block that defines an intentional classification, and the items of this classification will be calculated.

Besides a description with a Smalltalk block, the user can also add descriptions using Soul language.

An intentional classification can be recalculated, for example to bring it up to date with a changed implementation. No items can be added by drag and drop to intentional classifications, but an intentional classification can be converted into an extentional classification. This makes it easier to remove or add singular items to it.

Classifications support a full range of set operations (unions, subtractions and intersection) to make it easy to recompose their elements.

The StarBrowser has the following functionality:

- The classifications are on the left of the browser, in a Tree Widget. Classifications can contain anything;
- The children of an item are also defined by a service. By default only classifications have children (as defined by the ItemChildren service). When the ItemChildrenExtended service is used, classes get as children their protocols, that have as children their methods;
- Icons indicate the different kinds of items. This is defined by the current #icon service in the configuration menu;
- The text displayed for the item is defined by the #label service. A specific labeler can be easy added;
- By clicking on an item in the tree view, an editor for that item is displayed. For example, clicking an object opens an inspector on that object. Clicking on a class

or a method opens a Refactoring browser. Which editor is used for an item is defined by the #editor service in the configuration;

- The items in the Tree View have a context sensitive menu, as defined by the #menu service;
- The visitor used for an action can be chosen at any time in the Services menu, that lists all the registered services. Several #editor actions are registered. Which set is used can just be selected by clicking the appropriate visitor in the Configuration menu. The next item will be opened with the editor defined by the current edit service;
- Using the button with the light-blue background the Classifications view can quickly be removed, focusing only on the currently selected editor and clicking the button once more brings back the classifications view where it was;
- Classifications can be imported and exported in binary format so classifications can be transferred between images [Starindex].

These functionalities are implemented in a number of services:

- editor: The editor service is responsible for adding an editor on the currently selected item in the classifications list. This editor is embedded on the right of the classifications list. It integrates tools to edit all kinds of source code entities. The application returned by the editor service is integrated in the StarBrowser using VisualWorks' subcanvas technology. The toolbar of the editor (if there is one) is merged with the toolbar of the StarBrowser;
- icon: The icon service is responsible for showing the icon of an item in the classifications tree;
- label: The icon service is responsible for showing the label of an item in the classifications tree;
- menu: The menu service is responsible for returning the operate menu that users get when they right-click on an item in the classifications list .

Figure 10 shows the core concepts of the used model [Rwuyts03].

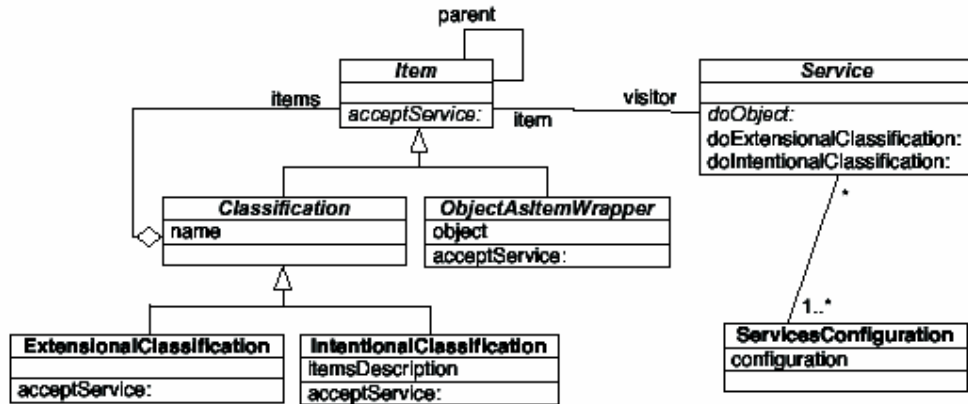


Figure 10 Diagram showing items, classifications, services and the service configuration

#### 4.4. Screenshots with StarBrowser

This paragraph presents some screenshots about how the StarBrowser works [Starscreen].

Figure 11 presents the classification tree: the Root Classification with an extentional classification (Favourites), and an intentional classification (Popular Classes).

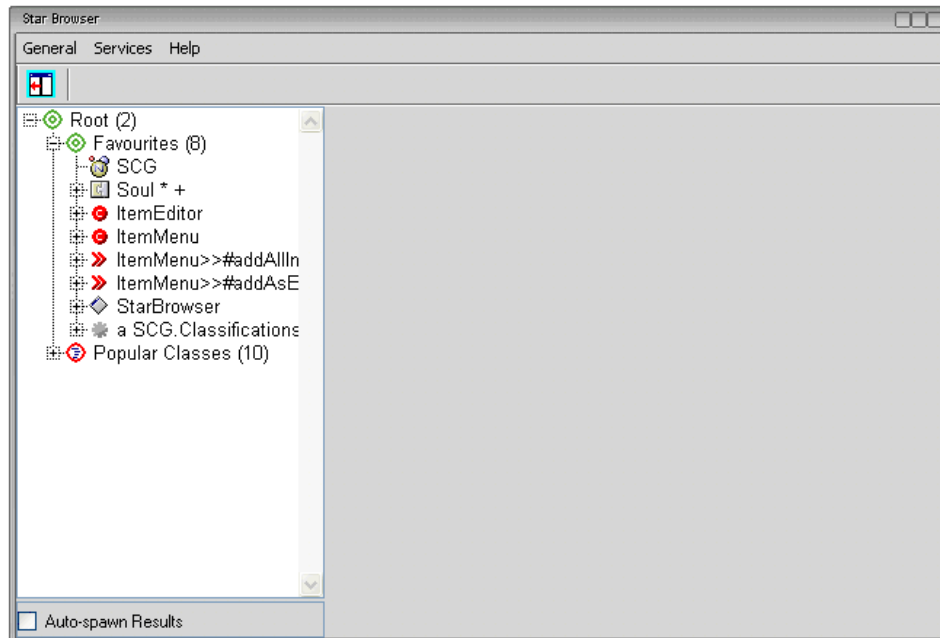


Figure 11 The classification tree

The extentional classification contains a namespace (SCG), a bundle (Soul), 3 classes (ItemEditor, ItemMenu and StarBrowser), two methods and an object (the singleton instance for the services configuration). These items were added by drag and drop.

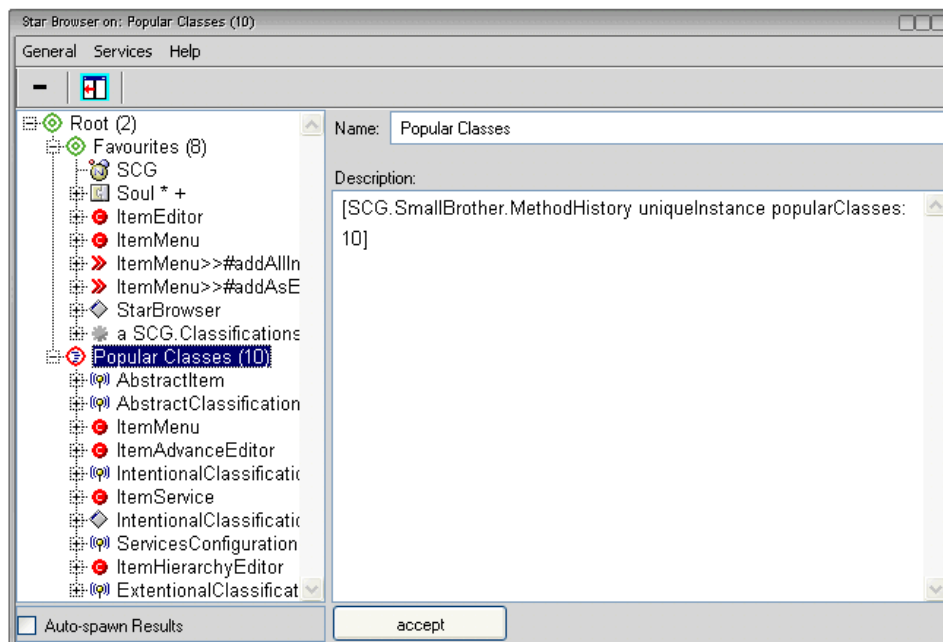
Since nothing is selected in the classification tree, the window in the right side is empty. However, selecting an item, results in showing in the right side the information about it. For example, selecting a class, a class browser is shown. The browser used is defined by the #editor service in the services menu.

Figure 12 shows an intentional classification selected. The right side window shows the definition of the Popular Classes classification.

When selecting an intentional classification, its implementation can be changed, using for example Soul.

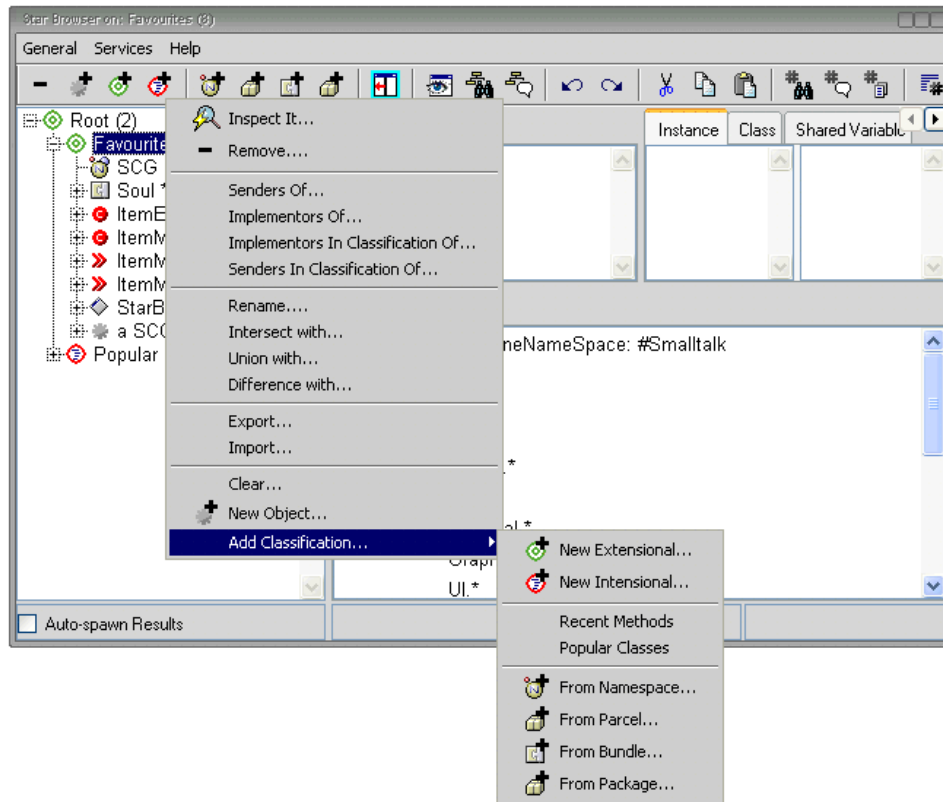
Right clicking on an element in the tree view brings up a context sensitive menu, with the contents defined by the *menu* service.

Figure 13 shows the menu for the extentional classification 'Favourites', that contains some common set-like operations, and operations to add items to the classification.



**Figure 12 Editing an intentional classification**

The services menu lists the different services that are registered. For example, for the #editor action three services are registered, and hence the user can choose between them. Figure 14 shows the ItemHierarchyEditor selected, while in Figure 15 ItemAdvanceEditor and then the Soul bundle is selected.



**Figure 13** The menu for extentional classification

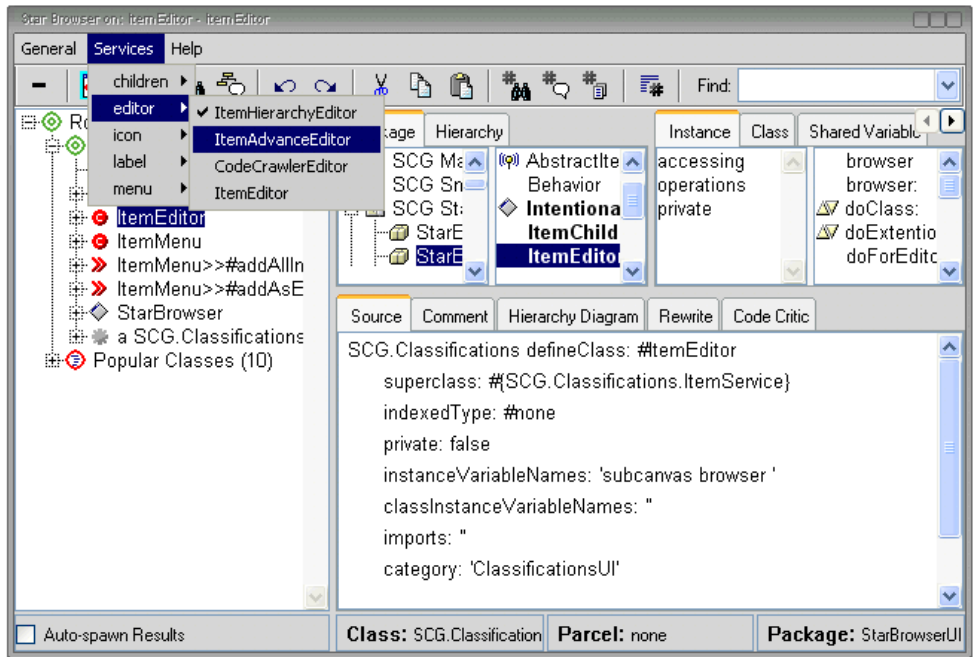


Figure 14 The services menu with the ItemHierarchyEditor selected

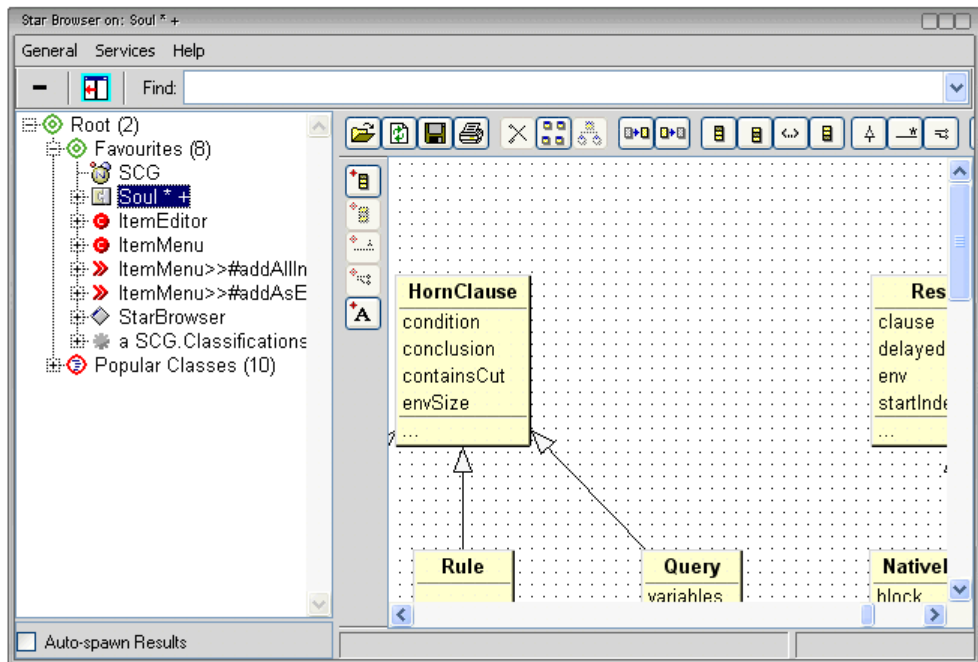
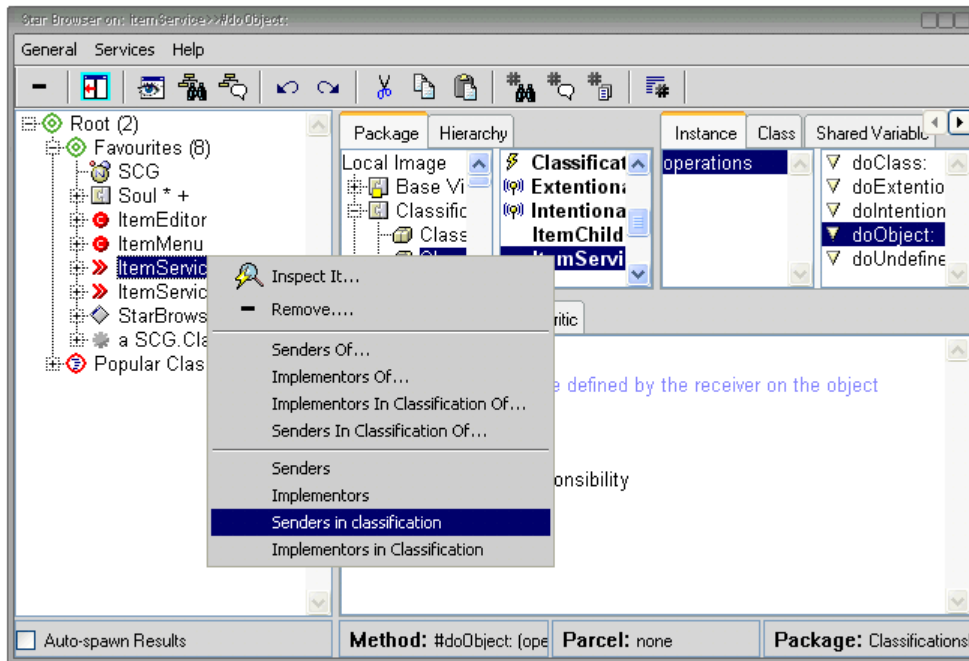


Figure 15 The services menu with the ItemAdvanceEditor selected

As result, the right pane shows an Advance UML Diagram for all the classes contained in Soul. Likewise, clicking on a classification will show the UML diagram for all the items in that classification.



**Figure 16 The Senders in classification for a method**

Using an UML tool in the classification model allows showing a class diagram for any kind of item that gets selected in the classifications tree. For example, selecting a namespace shows a class diagram for all the classes in this namespace, or showing a classification shows all the classes in that classification.

Besides being used for constructing working views on a system, classifications are also used as working contexts for some widely used commands. For example, the senders or implementors of a method can be looked for within the context of a classification. Figure 16 presents the Senders in classification for a method selected in the classification.

Selecting Senders Of... in the menu, the senders of the method are calculated in the context of the classification of that method, and then are added to the classification of the method, or shown in a new browser [Starscreen].

## 4.5. Conclusion

In this chapter we presented the StarBrowser development tool, with its two parts: the *Lightweight Classification Model* and the *StarBrowser*. We showed the Smalltalk

implementation of the classification model and the StarBrowser are an extensible browser that integrates different tools.

One of the advantages of the classification model consists in fact that it is lightweight, which make it easy to extend. This advantage is used to extend the StarBrowser with a Concern Interaction detection tool, which is presented in the following chapter.

## Chapter 5. Concern interaction using LMP

Detecting concern interaction in order to facilitate the software evolution task is becoming a common practice in diverse object-oriented software development environments, and there are several tools that perform this in an automated way: the best known are Aspect Browser tool and Feat tool. They all provide a number of facilities that help the software evolution task, detect concern interactions and allow performing automatic code inspection.

As presented in Chapter 2, the Aspect Browser is a tool that uses lexical searches of the program text, and a map metaphor to explore and manipulate crosscutting concerns, while FEAT models concerns as Concern Graph that is displayed as a forest of trees.

One of the major goal of this dissertation is to model concerns in a similar way as Feat tool does and to overcome the major disadvantage of this tool (the defined relations and the queries the user can perform are fixed by the tool so there is no possibility for the user to add new queries in order to investigate new kinds of feature interactions). We use the StarBrowser tool to model concerns as tree like view classifications and to allow an easy way for automatic code browsing.

This chapter first explains how logic meta-programming and the Star Browser tool are used for supporting the concern interactions detection process. We explain how concerns are modeled, how interactions are detected, and how related source code is browsed.

Another important point of this chapter is the definition of a number of predicates that determine the relations between the software entities, specially created in this work, and intended for concern interaction detection are explained.

## 5.1. Introduction

For defining and modeling concerns, we have extended the StarBrowser [Starindex]. As presented in Chapter 4, the StarBrowser allows easily the creation of classifications (by using a tree view and simply dumping the root of the classification in that view without checking that it is a graph or that it contains cycles) and implements the classification of software entities out of the Smalltalk image by means of drag & drop. The reason why we choose to extend the StarBrowser is that it is a complete framework for writing tools that use classifications.

We extended the StarBrowser with a new kind of classification: concern classification. We can create a new concern by adding a new Extensional classification to the Root of the graph defined in StarBrowser. Thereby a new empty concern is created with the name specified by the user such as “ConcernA”.

Then we can browse the source code and we can add elements to the “ConcernA” by means of drag & drop.

By extending the StarBrowser with concern classification, we are able to:

- create and delete concerns;
- classify concerns in classifications, moving and copying artifacts between concerns;
- remove artifacts from concerns;
- navigate through the concerns and through the source code due to the advanced browsing facilities of the tool;

While the StarBrowser allows to use almost all types of items that may occur in a Smalltalk Image, we use the following types of items we can put in a concern:

- Classes;
- Methods;
- Statements from the body of methods;
- Instance variables,

We also included the statements and instance variables because they might be useful when finding concern interactions, as presented in the following paragraph. Statements are necessary for us because sometimes concerns are not modularized using AOP (for example: update notification).

The StarBrowser already supplies the functionality for classifying classes and methods. Since the functionality for classifying statements is not implemented in the

Star Browser, we had to adapt the Learned Classifications toolbox and to also extend this one with the functionality for classifying instance variables.

Figure 17 shows a screenshot of the StarBrowser with some Concerns. On the left hand side of the screenshot it can be seen the concerns:

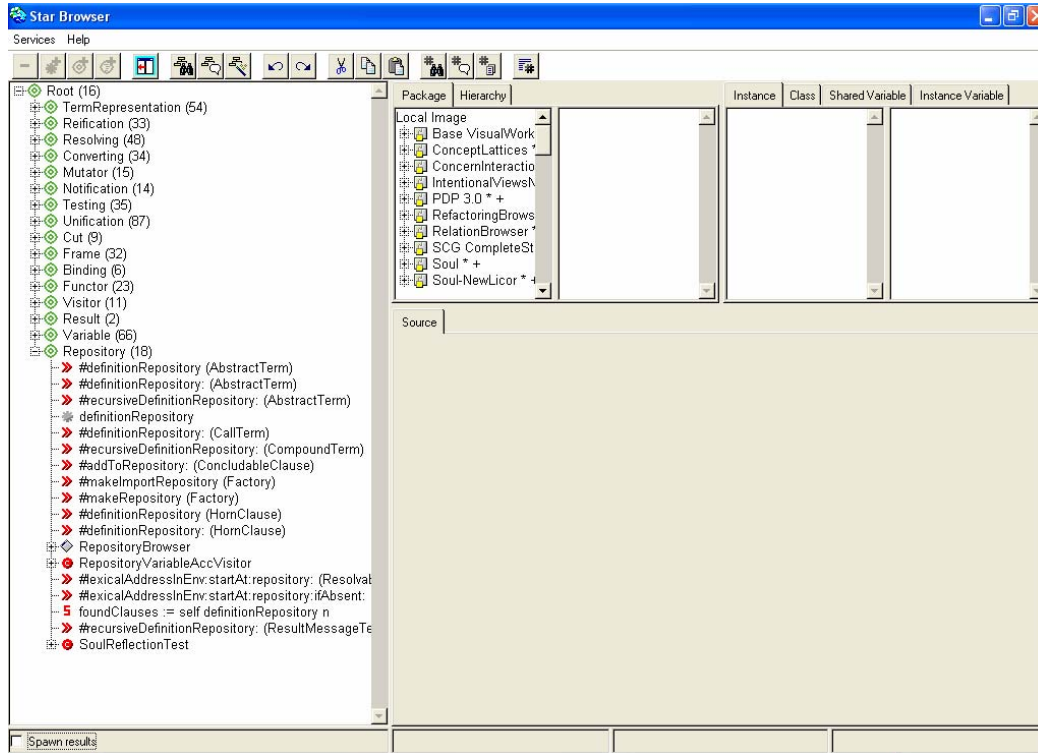


Figure 17: Concerns classification

Each of these concerns contains a certain number of software entities from the Smalltalk image. On the right hand side a class browser is opened on the currently selected item.

The concern interaction detection is implemented by making use of the facilities offered by SOUL and LiCoR.

## 5.2. Rules for concern interaction detection

Our tool uses a number of predicates (Table 3) that define relations between the software entities, specially created and intended for concern interaction detection.

Table 3 Summary of the newly defined relations

Relation name	Definition of the relation
<b>sameElementInteraction</b>	<b>sameElementInteraction(?c1,?c2,?interactionPoint)</b> The predicate checks if the two concerns (?c1,?c2) have a common element (?interactionPoint) which can be a class, a method, a statement or an instance variable.
<b>concernInteraction.⌋ ClassInNamespace</b>	<b>concernInteractionClassInNamespace(?c1,?c2,?class1,?class2,?namespace)</b> The predicate checks if the ?class1 from ?c1 and ?class2 from ?c2 are in the same ?namespace
<b>concernInteraction.⌋ CommonSelector</b>	<b>concernInteractionCommonSelector(?c1,?c2,?class1,?class2,?selector)</b> The predicate checks if the ?class1 from ?c1 and ?class2 from ?c2 have common ?selector
<b>concernInteraction.⌋ CommonStatements</b>	<b>concernInteractionCommonStatements(?c1,?c2,?class1,?class2,?selector,?stat1,?stat2)</b> The predicate returns the statements ?stat1 and ?stat2 of those methods in two given classes, ?class1 in ?c1 and ?class2 in ?c2, that correspond to the same method name ?selector
<b>concernInteraction.⌋ Override</b>	<b>concernInteractionOverride(?c1,?c2,?class1,?class2,?Method,?overridingMethod?Selector)</b> The predicate checks if the ?class1 from ?c1 and ?class2 from ?c2, if ?Method is implemented in ?class1, if ?overridingMethod is implemented in ?class2, if both methods have the same name ?Selector and if ?class1 is an ancestor class of the ?class2
<b>concernInteraction.⌋ Package</b>	<b>concernInteractionPackage(?c1,?c1,?class1,?class2,?Package)</b> The predicate checks if the ?class1 from ?c1 and ?class2 from ?c2 are in the same ?Package
<b>concernInteraction.⌋ UsesVariable</b>	<b>concernInteractionUsesVariable(?c1,?c2,?method1,?method2,?variables)</b> This rule verifies whether?c1 contains a ?method1 which uses the same ?variables with a ?method2 contained in the concern ?c2.
<b>concernInetraction.⌋ MethodUsesSameArg</b>	<b>concernInteractionMethodUsesSameArg(?c1,?c2,?class1,?class2,?selector1,?selector2,?commonArgs)</b> This rule verifies if ?c1 and ?c2 contains ?MethDef1, respective ?MethDef2,if these are implemented ?class1, respectively ?class2 and if they have like selector ?selector1, respectively ?selector2 and finally if the two selectors have some common arguments
<b>IndirectConcernInteraction.⌋ ClassCallsClass</b>	<b>IndirectConcernInteractionClassCallsClass(?c1,?c2,?class1,?class2,?calledClass)</b> This rule verifies if ?c1 contains a ?class1 and if ?c2 contains a ?class2 and if the ?class1 and

	?class2 calls the same ?calledClass which is neither in ?c1 nor in ?c2
<b>IndirectConcernInteractionClassCallsMethod</b>	<b>IndirectConcernInteractionClassCallsMethod(?c1,?c2,?class1,?class2,?selector,?method)</b> This rule verifies if ?c1 contains a ?class1 and if ?c2 contains a ?class2 and if the ?class1 and ?class2 calls the same ?method which is neither in ?c1 nor in ?c2

In order to adapt StarBrowser to our concern interactions detecting tool that uses a Soul like representation we had firstly defined a rule that associates a concern element to the StarBrowsers' internal representation:

```
➤ Rule      concernElement(?c,?soul) if
                member(?element,[?c unwrappedItems]),
                translatesTo(?element,?soul)
```

The concernElement relates two variables: the classification object and its Soul representation. The member predicate checks if the ?element is an element in the ?c concern. The ?c variable is a wrapped classification object, so it is necessary to unwrap it. The translatesTo predicate makes the translation from the Star Browser representation to the Soul representation or vice-versa if the ?element is a method or a statement otherwise the translation is not necessary.

While we make use of the StarBrowsers's Extentional classification menu to concern classifications, we had to define the following rule:

```
➤ Rule      extentionalClassification(?c)if ?c
                isKindOf:[SCG.Classifications.ExtentionalClassification]
```

This predicate checks if the concern ?c is an instance of the ExtentionalClassification class. This is necessary since our classification made in Star Browser is an extentional one. In some cases where were many elements to add to a concern, we've first made a soul intentional classification (which permits to add soul code in Star Browser) and then we transformed it into an extentional one. A concrete example is for the Reification concern. First we create the soul intentional classification and then we execute the rule that we wrote:

```
methodInProtocolHierarchy([Soul.TermRepresentation],[#reification],
                          ?Method)
```

To explain this rule, we will present its implementation:

➤ **Rule**     `methodInProtocolHierarchy(?SuperClass,?Protocol,?Method)`  
                   `if`  
                   `methodInProtocol(?Class,?Protocol,?Method),`  
                   `hierarchy(?SuperClass,?Class)`

The above predicate states that a ?Method of class ?SuperClass is in a protocol ?Protocol if the same ?Method of class ?Class is also in the same ?Protocol and if the variable ?SuperClass is a super class of the variable ?Class.

So in our example after we accept the `methodInProtocolHierarchy` predicate, in the Reification concern, all the methods ?Method of the class `TermRepresentation` or of an other descendent of him which is in the protocol `[#reification]` will be added. After that, only by right clicking on the Reification concern and choosing from the pop-up menu the `AsExtentional` option will have another Reification concern which is the wanted extentional concern.

➤ **Rule**     `sameElementInteraction(?c1,?c2,?interactionPoint) if`  
                   `concernElement(?c1,?interactionPoint),`  
                   `concernElement(?c2,?e),`  
                   `equals(?e,?interactionPoint)`

The `sameElementInteraction` relates three variables: the two concerns (?c1 and ?c2) and the interaction point between them which can be a class, a method, a statement or an instance variable. The first `concernElement` predicate verifies if ?interactionPoint is an element in the ?c1 concern and the second one verifies if ?e is an element in the concern ?c2. Finally, the `equal` predicate checks if the two elements (?e and ?interactionPoint) are the same.

➤ **Rule**     `concernInteractionClassInNamespace(?c1,?c2,?class1,`  
                   `?class2,?namespace) if`  
                   `concernElement(?c1,?class1),`  
                   `concernElement(?c2,?class2),`  
                   `class(?class1),`  
                   `class(?class2),`  
                   `classInNamespace(?class1,?namespace),`  
                   `classInNamespace(?class2,?namespace)`

The `concernInteractionClassInNamespace` verifies if the concern ?c1 contains a class ?class1 which is in the same namespace like ?class2 contained in the concern ?c2.

➤ **Rule**     `concernInteractionCommonSelector(?c1,?c2,?class1,?class2,`

```

?selector) if
concernElement(?c1,?class1),
concernElement(?c2,?class2),
class(?class1),
class(?class2),
commonSelector(?class1,?class2,?selector)

```

The concernInteractionCommonSelector verifies if the concern ?c1 contains a class ?class1 and if this class has a common selector with ?class2 contained in the concern ?c2.

```

> Rule    concernInteractionCommonStatements(?c1,?c2,?class1,
        ?class2,?selector,?stat1,?stat2) if
concernElement(?c1,? class1),
concernElement(?c2,? class 2),
class(?class1),
class(?class2),
commonStatements(?class1,?class2,?selector,?stat1,?stat2)

```

The concernInteractionCommonStatements verifies if the concern ?c1 contains a class ?class1 and if it has common statements with ?class2 contained in the concern ?c2.

```

> Rule    concernInteractionOverride(?c1,?c2,?class1,?class2,
        ?Selector) if
[1]    concernElement(?c1,?class1),
[2]    class(?class1),
[3]    concernElement(?c2,?class2),
[4]    class(?class2),
[5]    or(and(classImplementsMethodNamed(?class1,?Selector,
[6]                                                ?Method),
[7]    classImplementsMethodNamed(?class2,?Selector,
[8]                                                ?overridenMethod),
[9]    overridingSelector(?class2,?Selector,?overridenMethod,
[10]                       ?class1)),
[11]    and(classImplementsMethodNamed(?class1,?Selector,
[12]                                                ?overridenMethod),
[13]    classImplementsMethodNamed(?class2,?Selector,?Method),
[14]    overridingSelector(?class1,?Selector,?overridenMethod,
[15]                       ?class2)))

```

In the concernInteractionOverride the 1-4 lines verify if the concern ?c1 contains a class ?class1 and if the concern ?c2 contains a class ?class2. Then in the lines 5-8 we verify with the classImplementsMethodNamed if the method ?Method is implemented in the class ?class1 and that it has like selector ?Selector, and also if the method ?overridenMethod is implemented in the class ?class2 and that it has the same selector ?Selector. In the 9-10 lines with the overridingSelector predicate we verify if the ?Selector is an overriding selector and if ?class1 is an ancestor class of the class

?class2. In the next lines (11-15) the vice-versa is also taking into consideration, meaning that this time the ?class2 is the ancestor class of >class1.

```
➤ Rule      concernInteractionPackage(?c1,?c1,?class1,?class2,
      ?package) if
      concernElement(?c1,?class1),
      concernElement(?c2,?class2),
      class(?class1),
      class(?class2),
      classInPackage(?class1,?package),
      classInPackage(?class2,?package)
```

The concernInteractionPackage verifies if the concern ?c1 contains a class ?class1 and if the concern ?c2 contains a class ?class2 which are in the same package ?Package.

```
➤ Rule      concernInteractionUsesVariable(?c1,?c2,?method1,?method2,
      ?variables) if
      concernElement(?c1,?method1),
      concernElement(?c2,?method2),
      method(?method1),
      method(?method2),
      usesVariables(?method1,?variables),
      usesVariables(?method2,?variables)
```

This rule verifies whether the concern ?c1 contains a method ?method1 which uses the same ?variables with a method ?method2 contained in the concern ?c2.

```
➤ Rule      concernInteractionMethodUsesSameArg(?c1,?c2,?class1,
[1] ?class2,?selector1,?selector2,?commonArgs) if
[2] concernElement(?c1,?MethDef1),
[3] concernElement(?c2,?MethDef2),
[4] classImplementsMethodNamed(?class1,?selector1,?MethDef1),
[5] classImplementsMethodNamed(?class2,?selector2,?MethDef2),
[6] commonArguments(?class1,?class2,?selector1,?selector2,
[7] ?commonArgs)
```

The concernInteractionMethodUsesSameArg verifies if the concern ?c1 contains a method definition ?MethDef1, if the concern ?c2 contains a method definition ?MethDef1 (lines 1 and 2), In the next two lines, with the predicate classImplementsMethodNamed, we verify if this methods are implemented in ?class1, respectively ?class2 and if they have like selector ?selector1, respectively

?selector2. In the 5 - 7 lines we check if the two selectors have some common arguments.

```
➤ Rule      IndirectConcernInteractionClassCallsClass(?c1,?c2,
               ?class1,?class2,?calledClass) if
[1]   concernElement(?c1,?class1),
[2]   concernElement(?c2,?class2),
[3]   class(?class1),
[4]   class(?class2),
[5]   class(?calledClass),
[6]   classCallsClass(?c1,?calledClass),
[7]   classCallsClass(?c2,?calledClass),
[8]   not(concernElement(?c1,?calledClass)),
[9]   not(concernElement(?c2,?calledClass))
```

The `IndirectConcernInteractionClassCallsClass` verifies if the concern `?c1` contains a class `?class1` and if the concern `?c2` contains a class `?class2` (1-4 lines). The predicate `classCallsClass` on lines 5 and 6 verifies if the `?class1` and `?class2` calls the same `?calledClass` which should neither be in the concern `?c1` nor in the concern `?c2` (8-9 lines).

```
➤ Rule      IndirectConcernInteractionClassCallsMethod(?c1,?c2,
               ?class1,?class2,?selector,?method) if
[1]   concernElement(?c1,?class1),
[2]   concernElement(?c2,?class2),
[3]   class(?class1),
[4]   class(?class2),
[5]   classCallsMethod(?class1,?method),
[6]   classCallsMethod(?class2,?method),
[7]   not(concernElement(?c1,?method)),
[8]   not(concernElement(?c2,?method))
```

The `IndirectConcernInteractionClassCallsMethod` verifies if the concern `?c1` contains a class `?class1` and if the concern `?c2` contains a class `?class2` (1-4 lines). The predicate `classCallsMethod` on lines 5 and 6 verifies if the `?class1` and `?class2` calls the same `?method` which should neither be in the concern `?c1` nor in the concern `?c2` (8-9 lines).

### 5.3. Conclusion

In this chapter, we have explained how to support different steps of the concern interaction detection process using logic meta-programming. Thanks to powerful features as backtracking and unification, we can write complicated predicates related

to the structure of the code in a straightforward, understandable and concise way. This includes the detection of concern interactions. A number of predicates, specially created in this dissertation, and intended for concern interaction detection are explained.

The strength of our approach is the flexibility it offers in two respects: the kinds of interactions that can be detected, and the user-defined queries that can be answered. First, our approach does not force the user to know about Soul and automatically detects a number of interactions that also can be easily extended.

Second, the tool is not restricted to generating a fixed set of interactions, but allows the user to define interactions of interest by declaratively defining the relations to be found and the logic predicates to be applied. This makes it possible to create views of varying granularity and so to ask questions.

# Chapter 6. The Relation Browser: An experiment

In this chapter we will first present the Relation Browser tool and give an overview of its functionalities and how to use them.

In the second part of the chapter we will use this tool to perform some experiments involving Soul and finally we will present the obtained results.

## 6.1. The Relation Browser

As seen before, the StarBrowser tool can be used to model concerns as a tree classification in order to facilitate code browsing. The Relation Browser tool is another tool implemented in Smalltalk which allows detection of possible concern interaction and the result is showed with more detail.

It has two main functionalities:

- Checking whether a particular quantified relation between two concerns holds.
- Deducing the relations that hold

These functionalities will be presented in more detail later in this section based on a concrete example. In this example we want to check if one or more classes from TermRepresentation concern and one or more classes from Cut concern are in the same package. In order to check this interaction we have to write the following predicate:

```
➤ Rule      classInSamePackage(?class1,?class2) if
                class(?class1),
                class(?class2),
                classInPackage(?class1,?package),
                classInPackage(?class2,?package)
```

To add the `classInSamePackage` predicate to the Relation Browser we have to assert this rule in the `RelationBrowser.PredefinedClassificationRelation` layer:

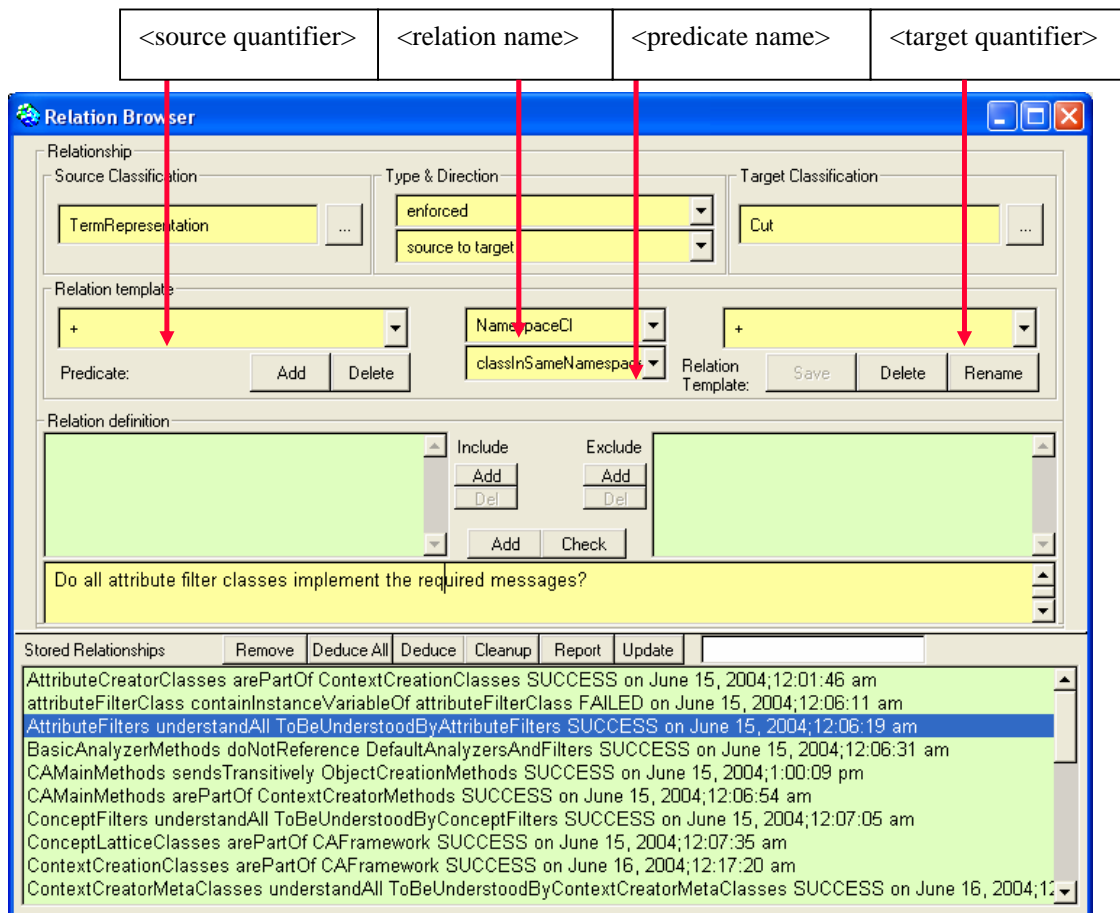
```
elementRelation(classInSamePackage,2,classInSamePackage)
```

Then we can select it in the ‘<predicate name>’ drop-down menu and then we can save these relations rank by rank in a new template via the “Save” button. As result, a rule will be created automatically:

```
classificationRelation(PackageCI,exists,exists,  
                        classInSamePackage,sourceToTarget)
```

and the saved relation template will appear in the ‘<relation name>’ drop-down menu.

The main window (Figure 18) allows the user to interact easily with the tool:



**Figure 18 A user case of the Relation Browser window**

The ‘Relationship’ set of frames guides the user to select the type of relation he wants to verify, the direction of the relation (meaning that the rule can be tested from the

‘source to target’ or from the ‘target to source’), including the considered concerns, the predicate and – what is specific to this tool – the type of quantification of the relation i.e. not only it checks if a relation exists between concerns but it also determines how strong is this relation, how many elements of each concern participate to the relation (Table 4).

**Table 4 Symbols**

Symbol	Meaning
+	at least one
!	exactly one
A	All
~	Zero
?	at most one
*	Zero or more

So we will select “+” on the left and “+” on the right (Figure 18) because we look for at least one concern interaction between two concerns.

While in the StarBrowser tool we had to perform the Soul query by ourselves, in the Relation Browser tool will check the relation for every tuple in the cross-product of the Source and Target set by launching automatically a query in Soul. For our example the query is the following:

```
➤ Query concernNamed(?c1,['TermRepresentation']),
      concernNamed(?c2,['Cut']),
      classInSamePackage (?class1,?class2)
```

So by using the Relation Browser the predicates are simpler because they are applied to the search space already reduced to the selected concerns (via SourceClassification and TargetClassification frames and their drop-down menu containing all the classifications made in StarBrowser) while in the StarBrowser tool we narrow the search space in the predicate itself (via the concernElement predicate as seen in the previous chapter).

### ***Deducing the relations that hold:***

The 'Deduce all' button allows finding automatically all the verified relations among all the possible combinations of a set of several concerns and predicates.

This opens a new window called 'Deduce'. In this window we can select one or more concerns in the source and in the target column and one or more predicates in the third column (Figure 19) and press the 'Deduce' button.

After a while, the tool will return the list of valid relations among all the possible combinations of selected concerns and predicates already saved in their respective relation template. In our case, as we can see in the figure, there are five valid relations detected by the tool.

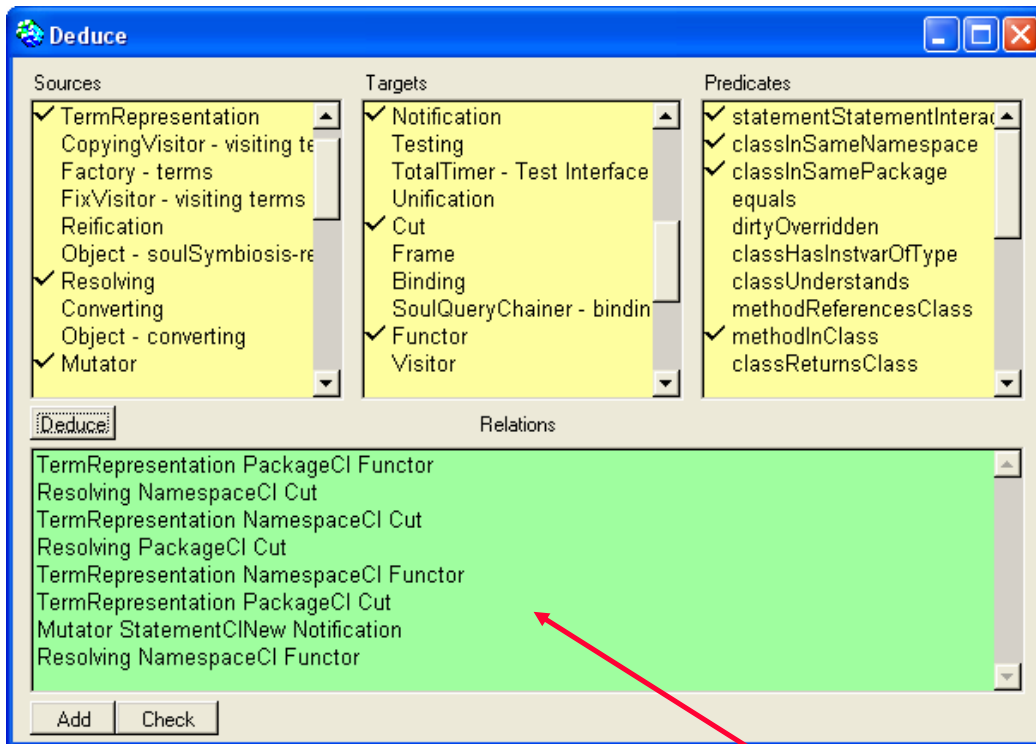


Figure 19 The Deduce window

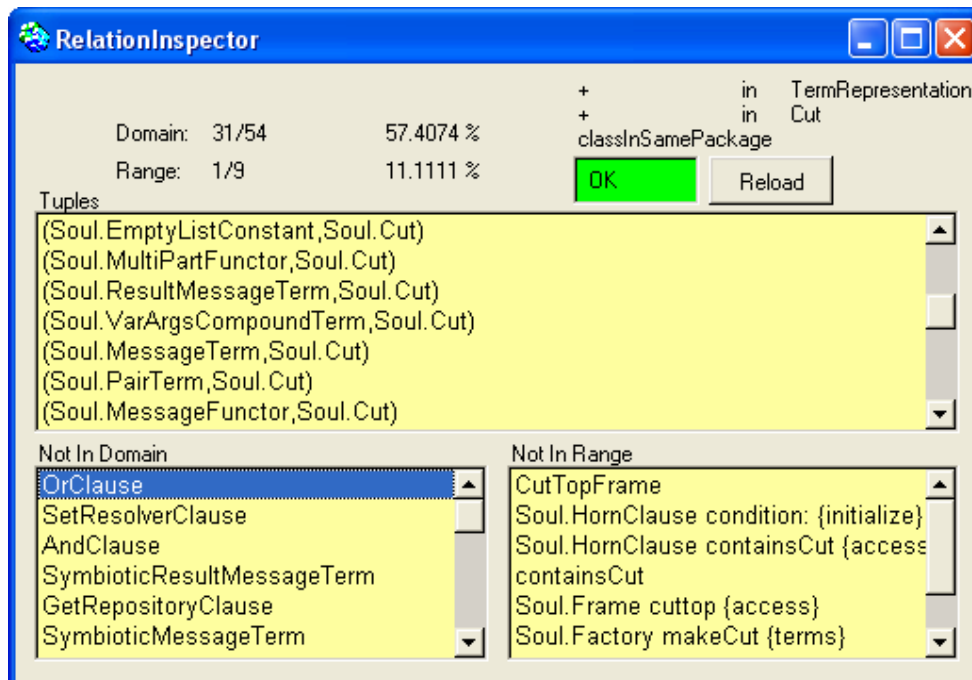
<the valid relations>

### ***Checking whether a particular quantified relation between two concerns holds:***

To see details about a relation, we can press the 'Check' button in the Relation Browser main window or in the Deduce window (after the selection of the desired relation) and the Relation Browser tool will check the relation for every tuple in the

cross-product of the Source and Target set by launching automatically a query in Soul. Then we can store it by pressing the ‘Add’ button.

The result will be shown in the Relation Inspector window as for example in Figure 20. In this figure we can see that in the TermRepresentation concern 31 of the 54 elements are part of the relation ‘Tuples’ with 1 of the 9 elements of the Cut concern. The  $31 \times 2 = 62$  couples of elements satisfying the relation are listed in the Tuples frame, the  $54 - 31 = 23$  elements from the TermRepresentation concern which are not part of the relation are listed in the ‘Not In Domain’ frame and the  $9 - 1 = 8$  elements of the ‘Cut’ concern which are not part of the relation are listed in the ‘Not In Range’ frame.



**Figure 20 The Relation Inspector window for the classInSamePackage predicate**

When maintaining the code, the developer can discover mistakes thanks to the Relation Browser tool. Then, before updating the code and/or the classifications, the developer can use the include/exclude facility of the tool, as explained below, to see exactly how to fix the relations.

As already mentioned, a relation is composed of ‘tuples’ of elements of each concern and the ‘Relation definition’ section (Figure 18) allows modifying these tuples:

- With the ‘Include’ frame, the user can create tuples which are not in the relation.

In our example (Figure 20) we can see that the “OrClause” element of TermRepresentation and the “CutTopFrame” element are not parts of any tuple of the relation. We can add a ‘fake’ tuple between TermRepresentation and Cut concerns (like in Figure 21) implying this element via the Include frame.

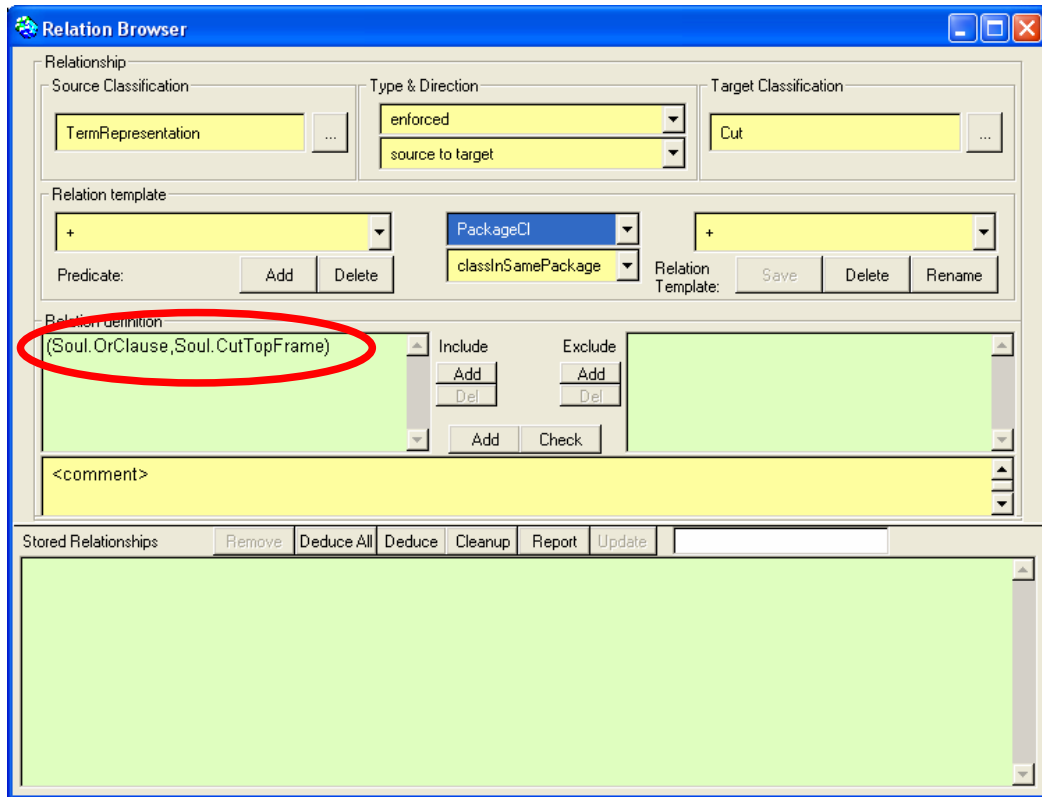
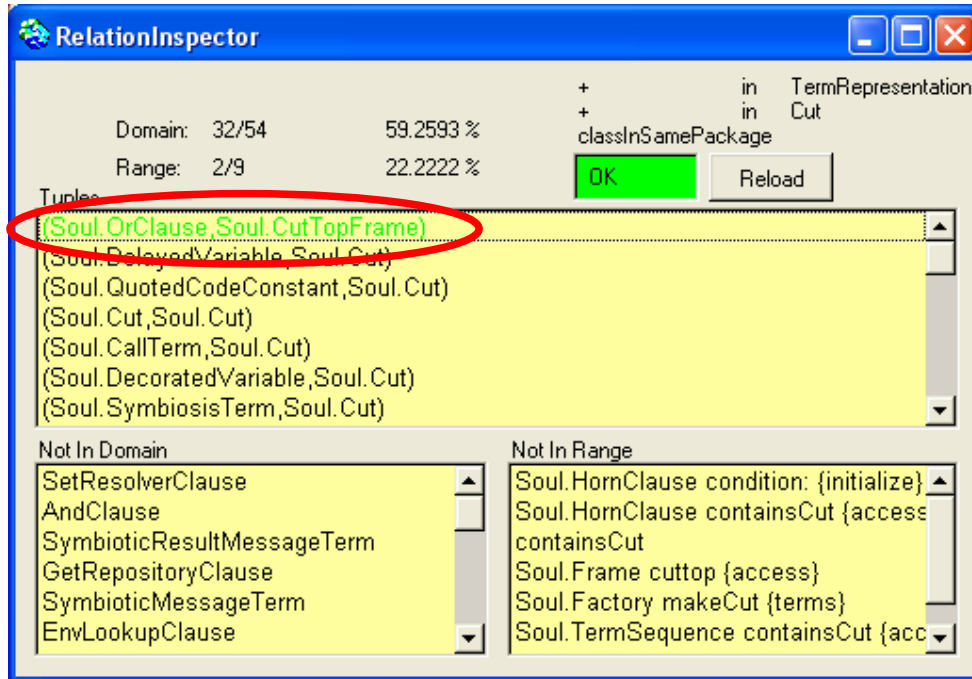


Figure 21 A Relation Browser window containing a ‘fake’ tuple

So when we check it, we can see in the RelationInspector window (Figure 22) that now in the TermRepresentation concern 32 of the 54 elements are part of the relation ‘Tuples’ with 2 of the 9 elements of the Cut concern. The introduced tuple is shown in green.



**Figure 22** A RelationInspector window containing a ‘fake’ tuple

- With the ‘Exclude’ frame, the user can as well exclude tuples which are part of the relation. As an example we excluded the tuple “Soul.PosVariable, Soul.PosCut” (Figure 23Figure 25). After we check this relation we can see in the RelationInspector window (Figure 24) that this time the “PosVariable” element of TermRepresentation concern is in the ‘Not In Domain’ frame and only concern 30 of the 54 elements are part of the relation. The excluded tuple is shown in red.

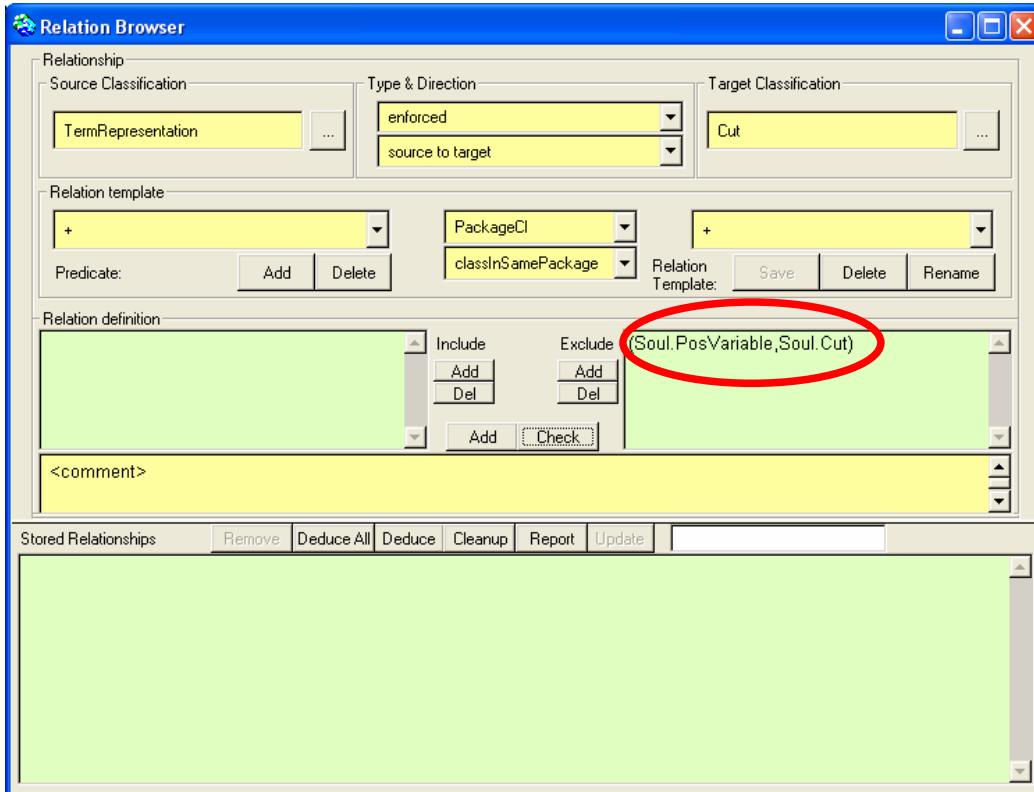


Figure 23 A Relation Browser window with an excluded tuple

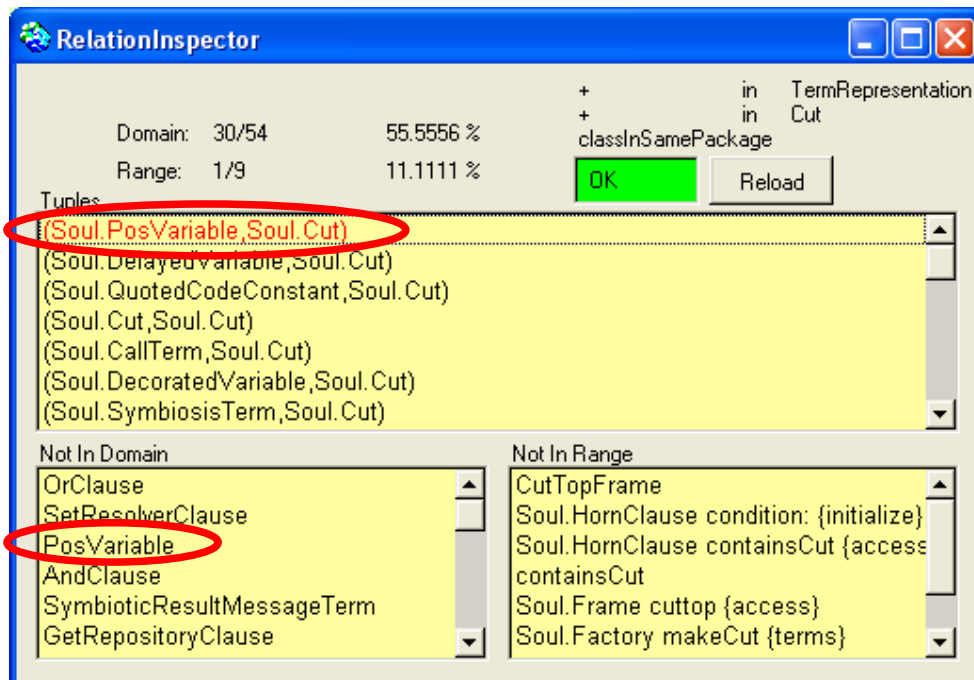


Figure 24 A Relation Inspector window with an excluded tuple

As a result the number of elements participating to the relation can vary and alter the quantization of the relation so the user can see the impact of modifications of the relation.

## **6.2. An experiment**

In this section we will see a concrete usage of the Relation Browser tool to observe concerns during software evolution and we will apply this experiment to the Soul tool.

### **6.2.1. The problem**

Our example of software evolution will be decomposed in two phases:

In a first phase a first developer writes some code, classifies it into concerns in StarBrowser and writes some rule to verify that concern interaction are correctly implemented using the Relation Browser tool and, in a second phase, a second developer adds some code, classifies it into the existing concerns and inadvertently breaks the previous rule because he is not aware of the Observer Pattern [DesignPat] details used by the first developer.

### **6.2.2. First phase**

As we know the Observer Pattern describes a collaboration involving two roles: Subject (or Observable) and Observer. An implementation of the Observer Pattern in Smalltalk requires that Observer is responsible to understand the message 'update:' while Subject should propagate change events via 'changed:' and manage its dependents (observers) using 'addDependent:' and 'removeDependent:'.

For this example, the concern interaction that our first developer wants to guarantee is one specific interaction that should be present in any Observer Pattern: for each statement changing the value of a variable, which we will put in the Mutator concern, there should be a corresponding statement propagating this information to its dependents, which we will put in the Notification concern.

In our example we are interested only in the messages of the Subject containing the 'changed:' message, for example 'functor:' of the CompoundTerm class where we added on purpose the notification:

```

functor: newFunctor
    functor := newFunctor.
    self changed: #functor

```

It is here composed of:

- a Mutator statement to modify the object: `functor := newFunctor.`
- a Notification statement to send a message to every dependant: `self changed: #functor.`

So the interaction between Mutator elements and Notification elements can be expressed as: *“for every element of Mutator concern it exists at least one element of Notification of which the changed object is the affected variable of the Mutator”*.

Therefore to check this interaction the first developer writes the following predicate:

➤ **Rule**

```

statementStatementInteraction(?statementSB1,?statementSB2)
    if
        translatesTo(?statementSB1,?statement1),
        translatesTo(?statementSB2,?statement2),
        SBstatement(?statementSB1,?class,?selector),
        SBstatement(?statementSB2,?class,?selector),
        or(equals(?statement1,assign(variable(?1),?val)),
        equals(?statement1,send(variable(?var1),?1,
        <variable(?var2)>))),
        equals(?statement2,send(variable([#self],[#changed:],
        <literal(?1)>)))

```

This rule verifies if the concerns ?c1 and ?c2 contain a statement ?statement1, respective ?statement2 , if this statements are in the same selector ?selector from class ?class. Then we check if ?statement1 is an assignment or a send message and if ?statement2 is a ‘self changed:’ statement.

Then, as seen in the previous section, he adds this predicate to the RelationBrowser.PredefinedClassificationRelation layer.

After that he uses the Relation Browser tool to deduce the Mutator – Notification interaction, documents it and saves it (Figure 25) to be able to see later when the code has evolved if the deduced relation still holds.

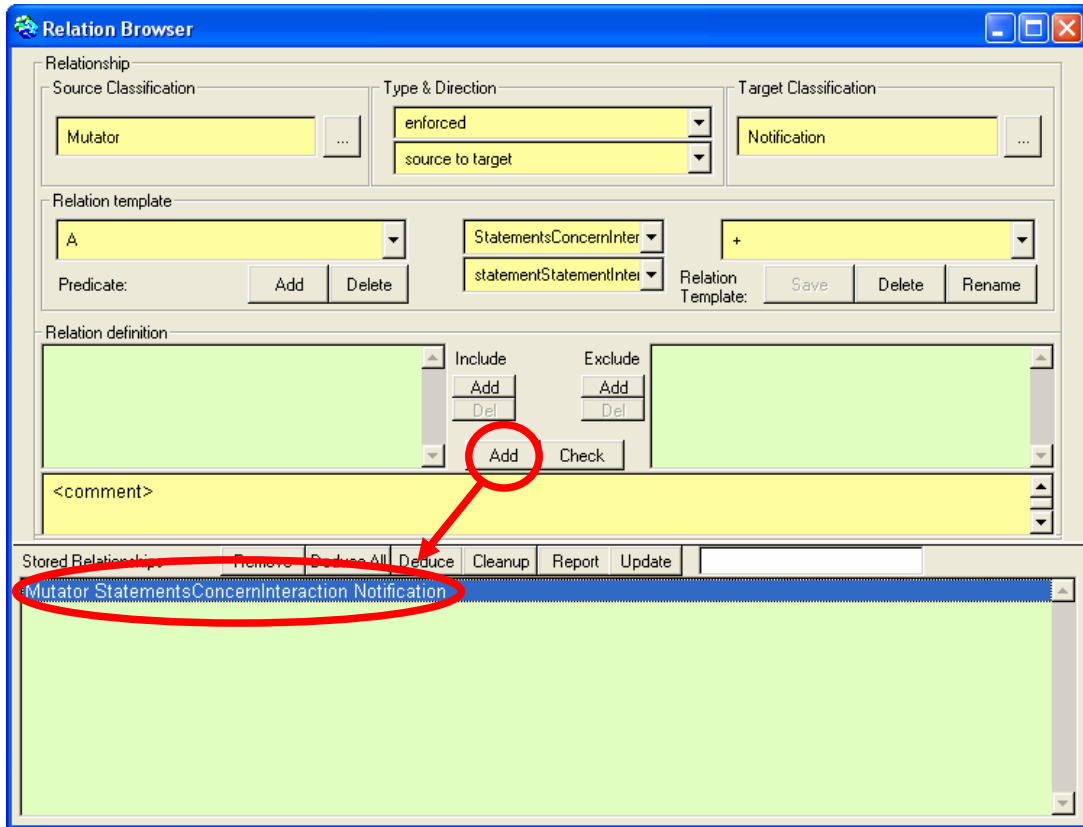


Figure 25 Saving the StatementsConcernInetraction relation

At this point, if the developer inspects the stored relation (by right-clicking on it) he sees that the relation is satisfied (Figure 26):

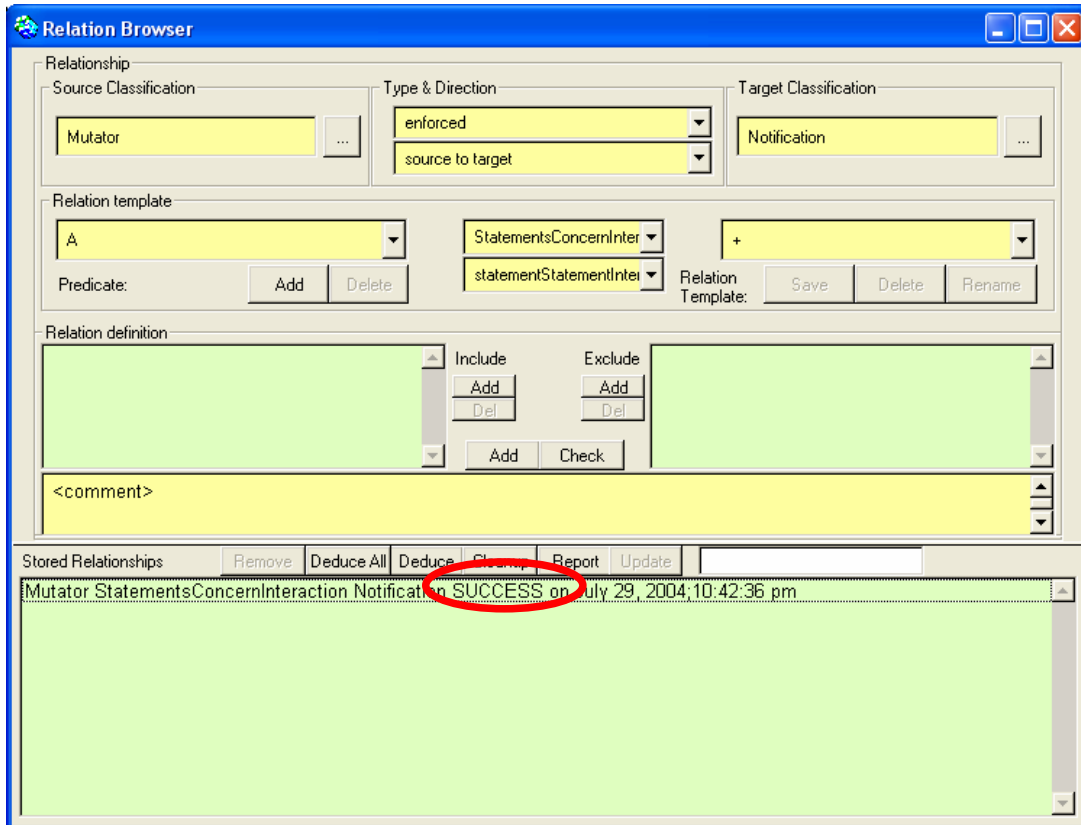


Figure 26

### 6.2.3. Second phase

In this phase, after a while, a second developer modifies the code without knowing the necessity to respect the relation between Mutator and Notification elements therefore he adds a few Mutator elements without adding the corresponding Notification elements, as we can see in the following example:

```

decorators: aCollection
    decorators := aCollection
    
```

But when he verifies his code with the RelationBrowser tool to check his modifications, he sees immediately that he broke the previous relation (Figure 27) and he can inspect it to see what the faulty elements are (Figure 28).

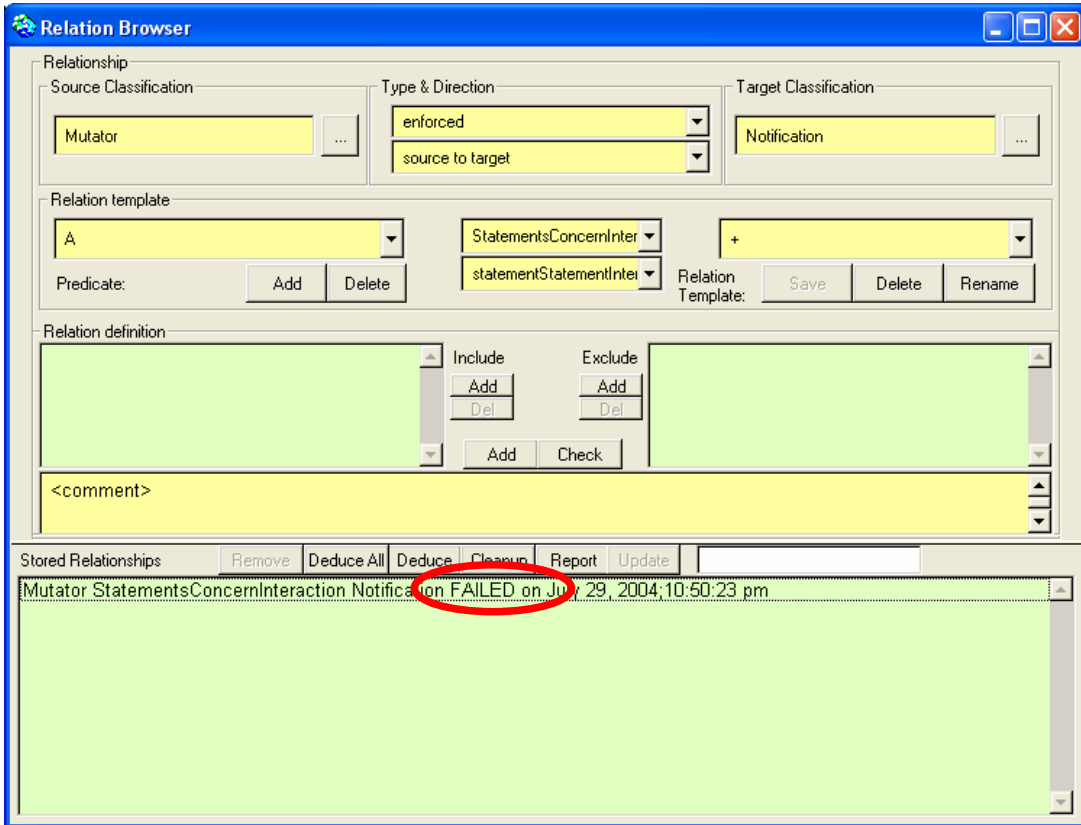


Figure 27

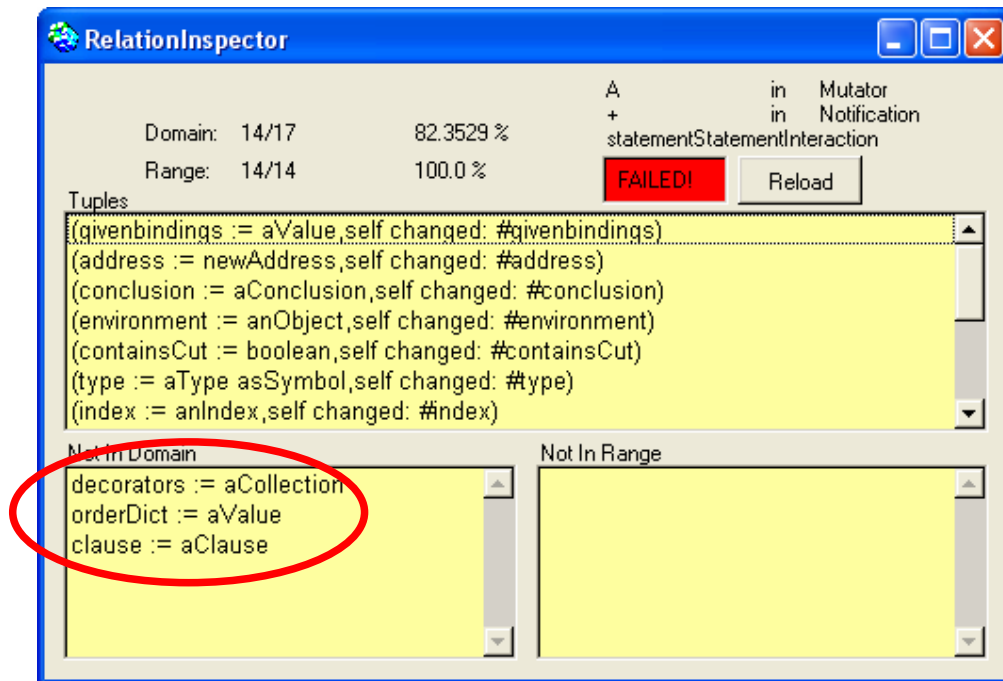


Figure 28

### **6.3. Conclusion**

In this chapter we presented an overview of the Relation Browser tool, which is able to find the relations existing between concerns. We also gave a concrete example of usage of this tool.

Then we applied concern interaction to software evolution by using the Relation Browser tool. As we have seen this tool is a very efficient one, since the user is able to detect quickly if a relation succeeds or fails (by simply inspecting it) after the software has evolved.

# Chapter 7. Conclusions

## 7.1. Summary

Concern interaction detection is the task of concern modeling and finding concern interactions, for a better understanding of interrelationships of related items and to help developers to find what part of the software must be changed if one concern is changed.

Chapter two squared up to definitions about features, aspects, concerns and their interactions, considering existing terminology. We started our discussion with an overview of the existing tools for finding concerns, modeling concerns and identifying the interactions between them. For us, the most interesting tool was the Feat tool since here concerns are modeled as a concern graph and supports concern interaction detection.

Chapter three gave a general introduction to logic meta-programming languages and explains some of the predicates provided by SOUL and by the LiCoR library.

Chapter 4 presented the Star Browser tool that is used in our approach to model and represent concerns. We have chosen to use the StarBrowser tool because it can be easily extended to represent tree like view classifications in a similar way as Feat tool does, it allows automatically code browsing and its user interface is very like Visual Works, the Smalltalk integrated development environment. The sections of this chapter present the main features of the Star Browser tool, with its two parts: the *Lightweight Classification Model* and the *StarBrowser*, relieving its advantages and the ways it can be extended.

Chapter 5 was about the goal of this dissertation: detecting concern interaction in order to facilitate the software evolution task. This chapter explained how logic meta-programming, the Star Browser tool are used for supporting some automated steps of the concern interaction detection process, focusing on concern modeling, interaction detection and related source code browsing. A number of predicates that define relations between the software entities and intended for concern interaction detection was given.

Chapter six contains an overview of the Relation Browser tool, which is able to find the relations existing between concerns. A concrete example of usage of this tool is presented and then, by making use of this tool, some experiments involving Soul are performed, including the obtained results. As seen from the case study presented our approach is rigorous and flexible.

## **7.2. Conclusion**

As mentioned, a number of tools provide support for concerns interaction detection. The most known ones are Feat tool and Aspect Browser tool. They all provide a number of facilities that ease the software evolution task detect concerns interaction, allows performing automatic code inspection. While several tools are implemented, there is no single tool that supports all necessary steps of the concern modeling, interaction detection process, allow specific user defined queries, automatic code browsing.

Our approach situates in the framework of finding more efficient ways for concern interactions detection. The goal of this dissertation is to increase the power of concern interactions detection tools, by adding support to adequately enable the specific user defined queries, by means of logic programming. This facility would overcome the current Feat tool drawback, which consist in fact that it provides no support for the user to add new queries in order to investigate new kinds of feature interaction, the relations defined and the queries that the user performs being fixed by the tool.

By means of logic meta-programming we defined appropriate predicates intended to perform concern interaction detection, we have been able to define complete and partial interactions, most of them highly time-consuming to be detected manually and we applied our approach in software evolution tasks.

The contributions of this dissertation are still considered:

- Being that the goal of this dissertation is to find concern interaction, it was important to consider existing terminology and definitions related to the subject.
- The existing tools intended to concern interaction are explored and some approaches of modeling concerns are described.

- The Star Browser tool was extended to allow concern modeling and instance variables handling.
- Our approach allows user-defined relations and queries in order to investigate new kinds of concern interaction, overcoming by this point of view the Feat tool.
- A logic meta-programming approach is used as a mechanism for detecting concern interaction. The ability to query the structure from a declarative point of view, gives a more flexible and rigorous way to concern interaction detection.
- Predicates that define relations between the software entities and intended for concern interaction detection are presented.
- Experiments using Relation Browser intended to detect concern interaction in software evolution tasks involving Soul validate the obtained results, showing that it is possible to automatically detect the source code related to the modifications needed in the software evolution.
- Likewise, using and extending of already existing software such as StarBrowser tool meets the goal of software reuse and makes it possible to grow the productivity of the software activities.

### **7.3. Future work**

We conclude this dissertation by presenting some future research topics.

- We are continuing this work in applying our approach to several Smalltalk applications. This will allow us to refine our analysis rules and to discover which kinds of interactions detection could be most useful in software evolution tasks.
- We intend to explore if such a tool makes it possible to modularize aspects of a system's implementation that previously could not be modularized. We are thinking to automatically detect different kinds of concerns such as physical concerns, logical concerns, refining them to functionality, behavior, performance, robustness, state, coupling, usability, size and so on.
- Based on some case studies, the defined predicates can be classified in light, heavy and so on concern interactions detection capabilities in order to allow the developer to gradually perform the software evolution task.
- The Aspect Mining Tool (AMT) provides an open multi-modal analysis framework for concern identification and system understanding, which uses succinct lexical descriptions and type patterns. We intend to explore the

applicability of our tool in the aspect mining process, by describing the crosscutting structure of an aspect by means of LMP.

## List of figures

Figure 1 Concern scattering and tangling .....	11
Figure 2 The AB windows.....	16
Figure 3. Hyperslice decomposition after the classes .....	18
Figure 4. Hyperslice decomposition after features .....	19
Figure 5 A Hypermodule created in the Hyperspace approach .....	20
Figure 6 The FEAT windows .....	24
Figure 7 Interaction View in FEAT .....	26
Figure 8 SOUL modules .....	30
Figure 9 LiCoR structure .....	35
Figure 10 Diagram showing items, classifications, services and the service configuration .....	47
Figure 11 The classification tree.....	47
Figure 12 Editing an intentional classification .....	48
Figure 13 The menu for extentional classification .....	49
Figure 14 The services menu with the ItemHierarchyEditor selected .....	50
Figure 15 The services menu with the ItemAdvanceEditor selected.....	50
Figure 16 The Senders in classification for a method.....	51
Figure 17: Concerns classification.....	55
Figure 18 A user case of the Relation Browser window .....	64
Figure 19 The Deduce window.....	66
Figure 20 The Relation Inspector window for the classInSamePackage predicate .....	67
Figure 21 A Relation Browser window containing a ‘fake’ tuple .....	68
Figure 22 A RelationInspector window containing a ‘fake’ tuple.....	69
Figure 23 A Relation Browser window with an excluded tuple.....	70
Figure 24 A Relation Inspector window with an excluded tuple.....	70
Figure 25 Saving the StatementsConcernInetraction relation .....	73
Figure 26 .....	74
Figure 27 .....	75
Figure 28 .....	75

## List of tables

Table 1 Outline of the Cosmos concern-space modeling schema .....	21
Table 2 Concern Interaction in FEAT .....	25
Table 3 Summary of the newly defined relations .....	56
Table 4 Symbols .....	65

# Bibliography

- [AskBed] Aksit M, Bedir Tekinerdogan B. , Bergmans L “The Six Concerns for Separation of Concerns” Software Engineering, Dept. of Computer Science, University of Twente.
- [CamGrif92] E. Cameron, N. Griffeth, Y. Lin, and H. Velthuijsen. “Definitions of Services, Features, and Feature Interactions”, December 1992. Bellcore Memorandum for Discussion, presented at the International Workshop on Feature Interactions in Telecommunications Software Systems.
- [DeVolder99] De Volder, K. “Aspect-oriented logic meta-programming”. In Proceedings of International Reection 1999 Conference, volume 1616 of Lecture Notes in Computer Science, pages 250 -272. Springer-Verlag, 1999.
- [DeMeuBrich03] De Meuter W, Brichau J, Mens K “SOUL Manual” 2003.
- [DelValle03] Del Valle J.G. “Towards round-trip engineering using logic meta-programming “Master thesis EMOOSE 2003.
- [Feat] “Feat User Manual” <http://www.cs.ubc.ca/~mrobilla/feat/manual-1.9-1.html>.
- [DesignPat] Gamma E., Helm R., Johnson R., Vlissides J., “Design Patterns: elements of reusable object-oriented software”, Addison-Wesley,1995.
- [GriYuaKat] Griswold W., Yuan J., Katoy Y. “Exploiting the Map Metaphor in a Tool for Software Evolution” (Aspect Browser).
- [Kellens03] Kellens, A. “Using Inductive Logic Programming to Derive Software Views” Master dissertation Vrije Universiteit Brussel, 2003.
- [KoDeHondt98] Koen De Hondt “A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems” dissertation 1998.
- [KrisGybels 03] Kris Gybels “Aspect-oriented programming using a logic meta-programming languageto express corss-cutting through dynamic joinpoint structure”, 2001.
- [MensWer] Mens K., Mens T., Wermelinger M. “Supporting software evolution with Intentional Software Views”.
- [Mens1] Mens K “Automating Architectural Conformance Checking

- by means of Logic Meta-Programming” PHD dissertation, Vrije Universiteit Brussel, 2000.
- [MeBrMe03] De Meuter W., Brichau J., Mens K”SOUL Manual (draft)” Vrije Univ. Brusselles, 2003.
- [OssTarr] Ossher H., Tarr P. “Multi-Dimensional Separation of Concerns and The Hyperspace Approach“, IBM T.J. Watson Research Center, P.O. Box 704 Yorktown Heights, NY 10598.
- [Puver] Pulvermuller, E. Speck, A. et all “Feature Interaction in Composed Systems” Position Paper. Institute for Program Structures and Data Organization, Universitat Karlsruhe, Germany.
- [RobMur 03] Robillard M. P., Murphy G. C. “Automatically Inferring Concern Code from Program Investigation Activities”, IEEE Computer Society Press, October 2003. IEEE, 2003.
- [RoWuytsSt] <http://homepages.ulb.ac.be/~rowuyts/StarBrowser/index.html>.
- [RoWuyts01] “A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation” dissertation 2001.
- [Rwuyts98] R. Wuyts. “Declarative reasoning about the structure of object-oriented systems” In Proceedings of TOOLS USA 1998, pages 112{124. IEEE Computer Society Press, 1998.
- [Rwuyts03] R. Wuyts, S. Ducasse “Unanticipated Integration of Development Tools using the Classification Model” Preprint submitted to Elsevier Science, 2003.
- [StepEicSteff] [Stephen G. Eick, Joseph L. Steffen, and Eric E. Summer, Jr. Seesoft—A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11), 1992.
- [StraBric] Straeten R., Brichau\_ J. “Features and Feature Interactions in Software Engineering using Logic” System and Software Engineering Lab Programming Technology Lab, Vrije Universiteit Brussel, Belgium Vrije Universiteit Brussel, Belgium.
- [SuttRou] Sutton S.M.Jr., RouvellouI. “Concern Space Modeling in Cosmos” IBM T. J. Watson Research Center Hawthorne, NY 10532.
- [Starscreen] <http://homepages.ulb.ac.be/~rowuyts/StarBrowser/screenshots.html>.
- [Starindex] <http://homepages.ulb.ac.be/~rowuyts/StarBrowser/index.html>.