TAMPERE UNIVERSITY OF TECHNOLOGY

Institute of Software Systems

TOMMI REINIKAINEN

CONCERN MANIPULATION TOOLSET TO SUPPORT SOFTWARE COMPREHENSION

Master of Science Thesis

Subject approved by the department council on April 12, 2006

Examiners:	Prof. Tarja Systä	(TUT)
	Dr.Tech. Imed Hammouda	(TUT)

FOREWORD

I have written this thesis while working at the Software Systems Institute at Tampere University of Technology, between the months April and December during the year 2006. The thesis was written as a part of two research projects, INARI and SERIOUS. INARI is funded by a conglomerate of organisations, including Nokia, Plenware, John Deere and Tekes. SERIOUS is funded by Nokia Reasearch Centre and Tekes.

First and foremost I would like to thank Imed Hammouda, Professor Tarja Systä and Professor Kai Koskimies for the support, advice and motivation during my thesis work. Without them, this thesis surely would not have been completed. I would like to give special thanks to Imed, who, as the tutor of this master's thesis project, was a source of knowledge and inspiration at all times and made it possible for me to finish my work.

Special thanks are due to the people that worked beside me in INARI and SERIOUS, for their willingness to help and explain whenever I needed it the most.

I would like to thank all my friends and family for their support and interest in my work. Last but not least, I want to thank my girlfriend Johanna, who did more than she ever should have had to by understanding, supporting and motivating me all the way. This thesis is dedicated to her, for believing in me even when I did not believe in myself.

Tampere, February 16, 2007

Tommi Reinikainen
Sammonkatu 35 B 33
33540 Tampere
`<tommi.reinikainen@tut.fi>`

TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Information Technology
Institute of Software Systems

REINIKAINEN, TOMMI: Concern Manipulation Toolset to Support Software Comprehension

Master of Science Thesis, 44 pages, 1 Appendix page
March 2007
Major subject: Software engineering
Examiners: Prof. Tarja Systä, Imed Hammouda
Sponsors: Tekes, Nokia, Plenware, John Deere, TietoEnator

Keywords: Software comprehension, model-driven development, concerns, concern-based decomposition, software decomposition

Abstract

Software comprehension is an important part of modern software development. Due to the complex nature of software engineering, the limits of human capabilities in terms of software comprehension are quickly met without a remarkable investment in methods that further enhance software comprehensibility. This thesis examines a technique, concern decomposition, to support comprehension and analysis of a software system. The basis of this technique is to provide for the user a new abstraction level that allows for a non-invasive re-decomposition of an already decomposed software system. This technique is then further elaborated into a suite of tools that can be used to create and analyse a concern-based decomposition of an existing software system. In addition, the process of implementing this toolset in the context of UML is described extensively. Finally, the toolset is applied to solve a specific task regarding a software platform developed by mobile phone manufacturer Nokia and used in several of its mobile phone products.

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan osasto
Ohjelmistotekniikka

REINIKAINEN, TOMMI: Intressijaotellun Ohjelmiston Hallinointityökaluja Ohjelmistojen Ymmärtämisen Tukemiseksi

Diplomityö, 44 sivua, 1 liitesivu
Maaliskuu 2007
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Tarja Systä, Imed Hammouda
Rahoittajat: Tekes, Nokia, Plenware, John Deere, TietoEnator

Avainsanat: Ohjelmiston ymmärtäminen, intressiperustainen ohjelmiston jaottelu, ohjelmiston jakotekniikat, malliperustainen ohjelmistotuotanto

Tiivistelmä

Koodin ja koodista muodostuvan ohjelmiston ymmärtäminen on keskeisessä roolissa nykyaikaisessa ohjelmistotuotannossa. Johtuen ohjelmistotuotannon monimutkaisuudesta, ihmisen luontaisen käsityskyvyn rajat tulevat varsin varhaisessa vaiheessa vastaan. Tämän vuoksi on tärkeää panostaa uusien ymmärrystä helpottavien tekniikoiden kehitykseen. Tässä diplomityössä tutkitaan erästä ohjelmistojen ymmärrystä tukevaa tekniikkaa: intressiperustaista ohjelmiston jaottelua. Tämän tekniikan perustana on uuden abstraktiotason määrittely, jonka avulla tekniikan käyttäjä voi jaotella jo entuudestaan jaotellun ohjelmiston uudelleen koskematta alkuperäiseen jaotteluun. Näistä lähtökohdista johdetaan tässä diplomityössä joukko työkaluja, joiden avulla käyttäjä voi tehdä olemassa olevalle ohjelmistolle intressiperustaisen uudelleenjaottelun ja analysoida sitä. Lisäksi työssä käsitellään näiden työkalujen toteutusta käyttäen UML-mallinnuskieltä toteutuksen kohdeympäristönä. Lopuksi työkaluja sovelletaan ratkaisemaan ennalta määritelty ongelma liityen erääseen matkapuhelinvalmistaja Nokian tuottamaan ja useissa sen matkapuhelinmalleissa käytössä olevaan ohjelmistoalustaan.

CONTENTS

TERMS AND DEFINITIONS

CASE Computer-Aided Software Engineering, an umbrella term for software that assists in development and maintenance of software.

EBNF Extended Backus-Naur Form, a notation (aka. metasyntax) used to express context-free grammars.

HTML HyperText Markup Language, the predominant language used in creating web pages.

IDE Integrated Development Environment, a development environment usually consisting of a set of tools to aid in performing a specific task.

INARI Name of the project under which this thesis work was done. Also the name of the software that is the end product of that project.

JavaScript A scripting language commonly used to run client side functionality in a web browser.

Parser Generator A program that generates source code for a parser from a programming language description, such as an EBNF grammar.

Plug-In A computer program that serves as a part of a larger main application, usually extending its features.

SDK Software Development Kit, contains tools for creating software into a certain environment.

XML Extensible Markup Language, a general-purpose markup language used often to present information in a structured form.

XSL eXtensible Stylesheet Language, a family of transformation languages which allows one to describe how files encoded in XML standard are to be formatted or transformed.

1. INTRODUCTION

Complex software consists of simple parts and relations; the programmer's goal is human comprehension.

-Donald E. Knuth

1.1 Motivation

A modern software engineering effort is a massive undertaking. It is common for a single software project to last several years and employ hundreds, if not even thousands, of developers to produce hundreds of thousands of lines of code. Such an environment is quite far from the early days of software engineering, when a single programmer could be tasked with creating an entire program in a couple of months.

Such difficult circumstances require that special attention is taken in choosing what methods and conventions to use, to ensure that the effort pays off. One of the methods that must be carefully chosen is the strategy under which the software is decomposed. This thesis proposes that the software is to be decomposed according to the concerns of the various stakeholders. The decomposition is similar to that suggested by Tarr et al. in their research on concern-based decomposition. [TOHS99] [THO$^+$00] Though in some contexts the word "concern" brings with it a negative meaning, this thesis sees the existence and study of such concerns as purely beneficial.

When the software is decomposed according to concerns, the unit of this decomposition is also logically called a concern. Concern as a unit can be understood as a concrete representation of the concern of a particular stakeholder to the software system. For example, a stakeholder may be concerned about software security. This concern can be mapped to all the pieces of the software that in some way relate to the security of the software, from the viewpoint of this particular stakeholder. These pieces would thus form a single unit in this decomposition, a concern.

This thesis introduces a mechanism to concretise the abstract and immaterial stakeholder concerns by binding them to the concrete, physical elements that are part of a software. After these concrete representations of the concerns have been created, it is possible to further deepen one's understanding of the software by studying the relationships and dependencies between the various physical manifestations of these concerns. For this purpose, a set of tools and operations is needed, to compare and manipulate the many different concerns defined from the viewpoints of the many different stakeholders.

The concept of different viewpoints is what truly justifies proposing such a mechanism for decomposing a software system. To get a deep understanding of a software system, one must first be able to look at it from different perspectives. This is where traditional modularisation mechanisms come short. The decomposition of a software into traditional modules is always from a single dominant viewpoint, which often means that studying the software from another viewpoint is extremely cumbersome. This thesis aims at providing a set of tools that ultimately make it easier for an individual to come to an existing software project and gain understanding of the work already done, more effectively than it is currently possible. In short, this thesis aims to improve the comprehension of software.

1.2 Objectives

Most of the groundwork for this thesis has been laid down in the research of Imed Hammouda and Professor Kai Koskimies, from the Institute of Software Systems in Tampere University of Technology. The objective of this thesis is to further cultivate and refine this research work and to emphasise the programming comprehension and understanding point of view. Some parts of the thesis, like the mathematical set operations as concern operations (Section 2.4), have been derived entirely from the publications of the aforementioned scientists, especially from Hammouda's and Koskimies' publication "Concern Based Mining of Heterogeneous Software Repositories" [HK06]. Other parts, like tracking concern evolution (Section 2.5), were conceived while working on the thesis, but still owe much to the influence of the aforementioned Hammouda and Koskimies, as well as Professor Tarja Systä, members of the INARI and SERIOUS projects and other personnel at the Institute of Software Systems.

The aim of this thesis is to present a convenient set of tools for decomposing any given piece of software according to the concerns of the various stakeholders to that piece of software. To do so, it is important to build a strong foundation in the form of a unified theory for the concern manipulation concepts. Since simply defining a theory does little to establish the feasibility of such a conceptual toolset, however, the toolset was implemented and then placed into use in a real-life software development project. Such an environment was provided by Nokia, one of the industrial partners of the projects this thesis was developed for.

1.3 Thesis Structure

The thesis is structured in the following fashion:

In Chapter 2 a unified theoretical foundation is laid by introducing the core concepts for this thesis.

In Chapter 3 the concepts are brought towards a working implementation by defining what methods and structures can be used to implement them.

In Chapter 4 the actual process of implementing the conceptual model is explained in detail.

In Chapter 5 the implemented toolset is applied to a real life software system in an externally defined case study.

In Chapter 6 the advantages and limitations of the presented approach are discussed. Also, a summary of this thesis project is drawn.

2. CONCERNS AS UNITS OF SOFTWARE DECOMPOSITION

This chapter defines the theory around the thesis. This is done by defining the concept of a concern and comparing concern-based decomposition to traditional modularisation techniques. The thesis work is outlined by introducing a set of operations for concern manipulation and discussing effects of software evolution to existing concerns.

2.1 On software decomposition

Modern software engineering is a team effort. A prerequisite for a successful team effort is that all members of the team understand the effort as a whole and also their respective responsibilities, which are a part of that whole. One of the mechanisms that aids this understanding is an efficient *software decomposition strategy*. It allows the members of a software development team to comprehend and manage the developed software system with less effort. [TOHS99] It can even be argued, that without a decent decomposition strategy, comprehending the structure and architecture of even a medium-sized software system is close to impossible. [Mil56] Decomposing complex information into smaller, manageable bits is natural human behaviour and essential in the complex world of software development. That is not to say, however, that the only reason for why software decomposition is exercised is the increase in comprehensibility. After all, software is decomposable by its very nature.

Different decomposition strategies are often based on different programming paradigms. For example, during the golden age of procedural programming, the dominant decomposition strategy was *modular decomposition*, or *modularisation*. This decomposition strategy improved the flexibility of the code, in terms of re-use and distribution of work, as well as its comprehensibility. [Par72] Modularisation is based on simplifying a complex system by giving it a hierarchical structure and, simultaneously, limiting the amount of knowledge one has to have of the system to be a part of its development.

The advent of object-oriented programming languages brought changes to software decomposition strategies. [And91] In this object-oriented paradigm, software is decomposed into *classes*, which are usually real-world concepts relating to the problem field the software is addressing. *Object-oriented decomposition* can be seen as an extension of the classic modularisation, rather than a replacement. A good example of this is in the Java programming language, which is largely based on classes. Java programs are decomposed into both classes and modules, the latter being referred to as *packages*. An illustration of the relationships between classes and packages can be seen in Figure 2.1.

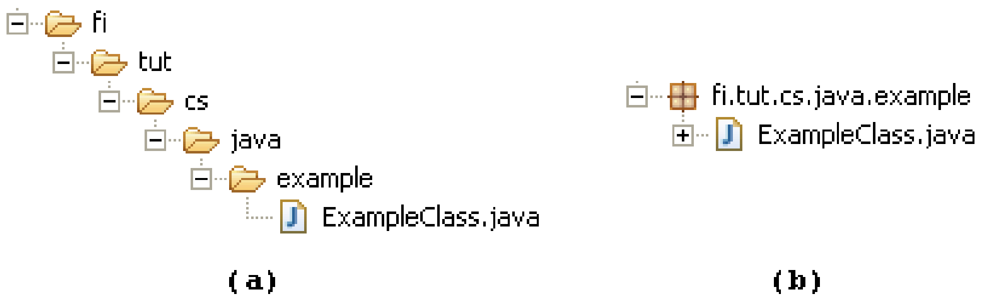


Figure 2.1: Packages (a) in the file system and (b) as they are seen in Java

Let us look at some of the drawbacks of this decomposition by using the Java decomposition system as an example:

- Java packages are meant to contain only source files (i.e. classes). While there is no explicit rule stopping the programmer from putting other kinds of files into packages [Sun99], there is an unwritten one that says that the packages should be kept clean of non-source code files, except for the occasional configuration file. In any case, scattering the documentation of a software system across different packages would be impractical.

- Each source file can only belong to a single package. A package commonly contains all the source files that are bound together by some logical rule (e.g. all the source files that contain code to create the user interface are placed inside the same package). Thus, if a source file contains code that makes it eligible to be placed into any of a number of packages, the file must still always be placed into a single, specific package. Doing so hides the fact that the source file actually could have been placed elsewhere as well. This restriction of modularising the software in only one way at a time is called the *tyranny of the dominant decomposition* [TOHS99] [BZL06].

Due to this tyranny of the dominant decomposition, Java formalisms allow us to organise a program only from a single viewpoint. This is a constraint on how a software written in Java can be organised. It also means that different stakeholders to the software always share the same viewpoint to the software, which is usually not desirable.

2.2 Concern-based decomposition

Before we venture any further, let us recapitulate a definition commonly used in software development: an *artefact*. An artefact is an organisable item in a software system. Examples of artefacts could be source files, documents, model drawings, resource files or even a user interface icon. To put it simply, everything that can be seen as a separate entity and that belongs to the software system in question is an artefact of that software system. This thesis treats all software as a collection of heterogeneous artefacts.

In this thesis work, the problems of modular decomposition that were presented in Section 2.1 are addressed by *concern-based decomposition*. Defining a *concern* is somewhat problematic. Here are some attempts at this:

- *[A] matter for consideration.* [Dic]
- *Any matter of interest in a software system.* [SR02]
- *A concern is an aspect of a problem that is critical or otherwise important to one or more stakeholders.* [Kan03]

The last of the three definitions is the most appropriate in the scope of this thesis. This thesis sees concerns as an alternative way to look at a software system: through the concerns of the different stakeholders to that system.

As concerns themselves are very abstract and thus separate from an actual software system, we need a mechanism to bind them to a system. This thesis calls this binding a *concern mapping*. When mapping the concerns, we identify which artefacts are addressed by which concern. Doing so, we bridge the gap between the abstract and immaterial concerns and the physical software system.

A concern mapping can be understood as a set of artefacts and their respective relationships, which can be seen as artefacts themselves. Any artefact may belong to any number of different concern mappings, which in turn allows the mappings to overlap (i.e. contain artefacts already found in other concern mappings). This is a fundamental difference between concern mappings and modules and is possible only because concerns are defined on a higher abstraction level than modules. Modules are the result of organisation of artefacts in a traditional, strictly hierarchical and non-relational file system, whereas concerns and concern mappings have no restrictions of that sort. The concern mapping concept is illustrated in Figure 2.2.

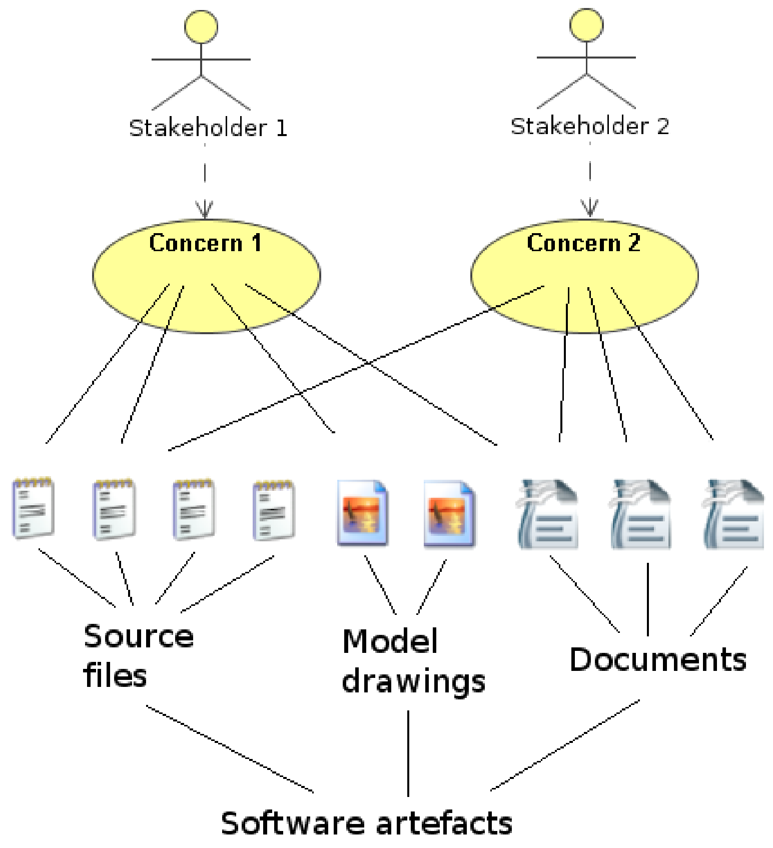


Figure 2.2: Relationships between concerns and software artefacts.

In this thesis, a concern is defined by a stakeholder to the system. This means that concerns are dualistic in nature: concerns define a certain feature of a software system, but also the viewpoint from which the feature is being studied. This allows for a single concern to be defined from the viewpoints of multiple stakeholders. Even though two stakeholders may be concerned about the same thing, the elements those concerns address (i.e. are mapped to) are probably different for each stakeholder, assuming they have different viewpoints to the system. As an example, let us assume an imaginary software project has two different stakeholders: a software engineer and a client. Both have a concern regarding the security of the system, but that does not mean the concerns equate on the artefact level. The software engineer may be more interested in the implementation of security in the system, whereas the client might be more interested in descriptive information regarding the planned, implemented or rejected security features, or possibly rationalisations behind decisions concerning security. This example is illustrated in Figure 2.3.

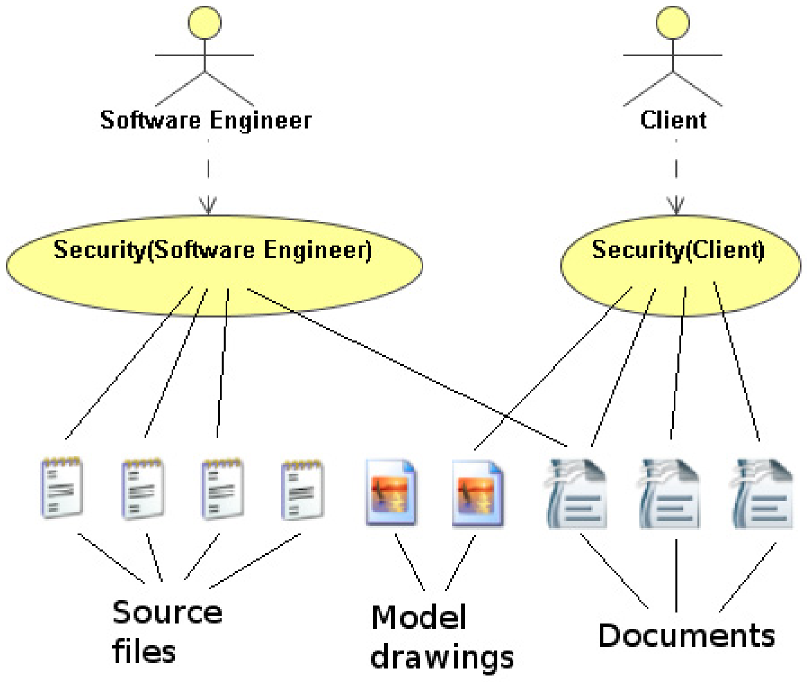


Figure 2.3: Same concern defined by different stakeholders.

The benefits of using concern-based decomposition are many. One of them is the possibility to decompose the system multi-dimensionally. [TOHS99] As an example of this, let us look at a typical system feature already used in the previous example, *security*. If there was such a thing as a security module in a modularly decomposed system, that module would of course contain source code that manages the security of the system. The system could also include other source code files that, though containing security-related code, are not included in the security module, because application security is not their primary emphasis. If a stakeholder of this system would like to view the system's security aspects, modular decomposition would offer him an incomplete set of source code files.

In the concern-based alternative, security is first identified as a concern after which a concern mapping for the security concern could be created. That mapping would bind all security-related artefacts. This would be beneficial for two reasons. Firstly, by mapping concrete elements to the concern, we can include all relevant artefacts regardless of whether security is their primary emphasis, as they can still be included in other concern mappings; inclusion does not exclude the artefacts from anywhere else. Secondly, restricting the included artefacts to source files would no longer be necessary. As the artefacts mapped by a concern mapping can be heterogeneous, the artefacts could also be documents, drawings or schematics that address security, without breaking any conventions or coding style rules that forbid placing artefacts other than source code into a module.

In the scope of this thesis, concern-based decomposition can be seen as an extension of modular decomposition. Concern mappings are not meant to replace modules. They are meant to fill the voids left by the modular decomposition, to increase the system's comprehensibility and to allow people outside the implementation team to have a clear vision of the system. Concerns can also be used to analyse and to keep track of dependencies between different parts of the program: if we know that two concerns overlap, i.e. that the concerns share an artefact, we also know that by modifying this artefact we are also modifying both concerns.

2.3 Cross-cutting concerns

The already mentioned overlapping of concerns is an important issue. It allows more flexibility in defining concerns, but more importantly, it allows us to identify and define concerns that could not otherwise have been defined. These special concerns are called *cross-cutting concerns*, because they cross-cut multiple parts of the system. This term originates from the research done by Gregor Kiczales and his research team at Xerox PARC. One of the outcomes of this research has been the concept of *aspect-oriented programming* [KLM+97], where the single unit of decomposition, an *aspect*, is very close to the concept of concerns studied in this thesis; hence the similarities in terminology. Another very similar concept is Hyper/J [OT00], a tool for multi-dimensional separation and integration of concerns in Java. HyperJ has been created by Peri Tarr, Harold Ossher and their team as part of the IBM alphaWorks [aWs] program.

The concept of cross-cutting concerns is easier to explain through an example. Let us loan one of the more popular examples of a cross-cutting concern (or an aspect) from the field of aspect-oriented programming: logging.

Assume that in our software system 30% of the classes do logging. If the system is decomposed solely by means of modular decomposition, the logging classes are scattered among the non-logging classes. There is no reasonable way to convey that a class does logging through modular decomposition conventions. One way to do this would be to add a character or a word into the name of the class or source file, which is not very rational concerning the limited importance of this information. Another alternative, which is even less feasible, is to create a module that contains all of the classes or source files that do logging. Since it is unlikely that logging is the main functionality of all these classes, this approach would also be impractical.

The scattered nature of cross-cutting concerns makes it difficult to properly highlight them with traditional decomposition mechanisms. The concern mechanism introduced in this thesis has no problems mapping cross-cutting concerns, as there are no limitations to whether or not a concern mapping may overlap another concern mapping.

The two important keywords concerning cross-cutting concerns are, depending on the point of view, *tangling* and *scattering*. On one hand artefacts of a cross-cutting concern mapping are scattered among the rest of the artefacts and among other concerns. On the other hand, when considering the issue from the point of view of a single artefact, we can say that this artefact is tangled into multiple concerns. Both of these relationship types are important and have served as inspiration for this thesis' concepts.

2.4 Concern Operations

After a software system has been mapped as concerns, we get a multi-dimensional and usually clearer view of the system. The organisation is no longer under the tyranny of the dominant decomposition and we may look at the system from many different viewpoints. We may also continue to look for new concerns by examining the relationships between the already existing mappings.

It is reasonable to assume that some of the concern mappings overlap. By looking at two mappings and examining which parts of them overlap, and which do not, we are able to explicitly see the dependencies between two parts of the system. As was mentioned earlier, a concern mapping can be seen as sets of artefacts. As a logical continuation of this assumption, Hammouda and Koskimies suggest that mathematical set operations, loaned from the *naive set theory* [Hal60], be used to examine the relationships between concerns [HK06]. This thesis is based on the ideas presented in that publication.

The thesis focuses on a specific set of operations. Of the mathematical set operations, the ones implemented in the work leading to this thesis are *union*, *intersection* and *difference*. Their concern counterparts are defined in Table 2.1, with the corresponding set operation in parenthesis. A fourth operation, *nearest neighbourhood* (later just neighbourhood), was also implemented, although this is not originally a mathematical set operation. It is rather an operation that was proven very useful in the course of the work. A common feature for all of the operations is that they take existing concerns as their parameters and return the result as a new concern. The list of possible operations is in no way limited to the ones implemented here. For example, one of the unused mathematical set operations, the Cartesian product, is implementable in the concern realm. It was left unimplemented, however, as it did not seem be of much use in the scenarios handled in this thesis work.

Table 2.1: Concern operations

Operation name	**Symbol**	**Type**	**Description**	**Commutative**
Merge (union)	+	binary	Merging of concerns A and B results in a concern containing all the elements of A and B, excluding duplicates.	yes
Overlap (intersection)	&	binary	Overlapping A and B results in a concern containing all the elements that are common to both A and B.	yes
Slice (difference)	-	binary	The slicing of concern A with B results in a concern that contains the elements that belong to A but do not belong to B.	no
Nearest neighbourhood	|	unary	The nearest neighbourhood of concern A results in a concern that contains all elements that have a relationship with an element in A and that do not belong to A.	-

All the operations can be chained to form more complex expressions. The precedence between the binary operations is not defined (i.e. the precedence is from left to right), but the unary operation takes natural precedence before the binary ones. The order of execution can be altered by using parentheses. It is also important to note that the laws of distributivity and associativity for mathematical naive set operations [Hal60] are also valid for the concern operations.

As a simple demonstration of what can be done with the defined operations, let us assume an existing system contains the following concerns: *Security*, *Logging* and *Documents*. The system artefact repository contains source files and documents. "Security" contains all source files and documents that have to do with the system's security. "Logging" is mapped similarly to code files and documents that involve logging. "Documents" contains all the document type artefacts belonging to this system.

Some example questions that are answered using the above defined concern operations:

1. What artefacts of "Security" do logging?
    - *Security & Logging*
2. What are the documents that contain information about the system's security?
    - *Security & Documents*
3. What documents contain information of either logging or security (or both)?
    - *(Logging + Security) & Documents*
4. What non-document artefacts does "Logging" include?
    - *Logging - Documents*
5. Which artefacts have relationships with the artefacts in the concern "Security"?
    - $|$*Security*

2.5 Software Evolution -Driven Concern Evolution

As software systems evolve, so must the concerns. Since software evolution may entail anything from addition or omission of artefacts to subtle changes in relationships between artefacts, as well as merely changing a specific artefact, tracking the change in concerns is problematic at best and impossible at worst. Nevertheless, a tracking system is necessary. Without one, every new version would require a new mapping of concerns and artefacts.

If we assume that relationships between artefacts in a concern are artefacts themselves and that changing an artefact actually means replacing the old artefact with a new one, evolution of a software may cause any sequence of the following changes to a concern:

1. The concern may gain artefacts.
2. The concern may lose artefacts.
3. The concern may remain unchanged.

Note that the first two scenarios are not mutually exclusive but can both occur as the concern evolves. Also note that the third scenario is trivial. It does not require any actions, because in this scenario, the concern does not evolve.

If the concern loses artefacts, we can easily find the remaining artefacts of the original concern and thus the evolved concern. In some cases information on which artefacts were lost is just as important, if not more so, than what could be recovered from the old concern. If the concern gains artefacts, finding the evolved concern is not as straightforward. This can be illustrated more thoroughly by using an example:

Let us call the original version of the software (i.e. the set of artefacts that represents the entire software) Π. Furthermore, let us assume that there exists a concern λ that has been mapped to a subset of Π. Now, as Π evolves, we get a new set of artefacts that depicts the software after it has evolved. Let us call this set $\widehat{\Pi}$. Since the software has evolved, so must λ. The evolved version of this concern is $\widehat{\lambda}$. We may now define Equation 2.1.

$$\lambda \subseteq \Pi \ \wedge \ \widehat{\lambda} \subseteq \widehat{\Pi} \tag{2.1}$$

Now, if we look at a case where λ evolves by *losing* artefacts (denoted by the sub-index L in the variable name), we know that due to this the evolved concern and software system are actually subsets of the original concern and software, as shown in Equation 2.2. Furthermore, we can express both the evolved concern ($\widehat{\lambda}_L$, Equation 2.3) and the set of removed artefacts (aka. change, $\Delta\lambda_L$, Equation 2.4) by using the concern operations defined in Section 2.4. Also note that expressing this information only requires the original concern and the evolved software artefact set.

$$\widehat{\Pi}_L \subseteq \Pi \quad , \quad \widehat{\lambda}_L \subseteq \lambda \tag{2.2}$$

$$\widehat{\lambda}_L = \lambda \ \& \ \widehat{\Pi}_L \tag{2.3}$$

$$\Delta\lambda_L = \lambda \ - \ \widehat{\lambda}_L = \lambda \ - \ (\lambda \ \& \ \widehat{\Pi}_L) \tag{2.4}$$

If, on the other hand, the concern evolves by *gaining* artefacts (denoted by the sub-index G in the variable name), there is no way to determine the change in the evolved concern by only examining λ, Π and $\widehat{\Pi}_G$, which are the artefact sets we always have access to. In other words, $\widehat{\lambda}_G$ can not be expressed as a function of the other artefact sets, unlike in the previous scenario with the concern losing artefacts. Even though $\widehat{\Pi}_G$ has gained artefacts (and we know which these artefacts are), there is no way of telling whether or not a given gained artefact should be a part of $\widehat{\lambda}_G$, so a corresponding equation to Equation 2.3 does not hold in this scenario. From this information, we can derive Equation 2.5.

$$\widehat{\lambda}_G \neq \lambda \ \& \ \widehat{\Pi}_G \tag{2.5}$$

We may now conclude that if the concern evolves by gaining artefacts, we always need input from the stakeholder to define the evolved concern.

3. INFRASTRUCTURE FOR THE CONCERN MANIPULATION TOOLSET

This chapter presents a group of techniques that can be used when implementing the concepts presented in the previous chapter. It works as a stepping stone towards describing the implementation of the concern manipulation toolset. This chapter introduces *aspectual patterns*, which can be used as concern mappings in actual software development projects. Furthermore, the set of operations introduced in the previous chapter are now developed into an actual query language. Effects of concern evolution, when considered from the point of view of aspectual patterns, are also discussed.

3.1 Representing Concerns With Aspectual Patterns and Roles

As previously mentioned, concern mappings are used to concretise the concerns of the stakeholders. In this thesis work, *aspectual patterns* [HKK04] were used to represent concern mappings in the real-world software development process. One reason for this was that these patterns hold all the qualities that are required of a concern mapping; in brief, the capability to contain heterogeneous artefacts and to overlap. Another reason was that there was an existing implementation of aspectual patterns available at Tampere University of Technology as part of the INARI environment, which is more thoroughly introduced in Section 4.1.4.

Figure 3.1 shows a conceptual model of an aspectual pattern defined in UML [UML]. Patterns are essentially collections of roles, which in turn serve as attachment points to actual artefacts. In this thesis work, the pattern concept was used to represent the concerns and the concern elements (patterns and pattern roles respectively) as well as to provide a non-invasive way to gain access to the software repository elements. As may be concluded from the figure, patterns can be used for other purposes as well, such as for marking the implicit dependencies between roles that cannot be seen in a model diagram and for setting constraints for the different roles. However, these features are not used in the context of concern mapping.

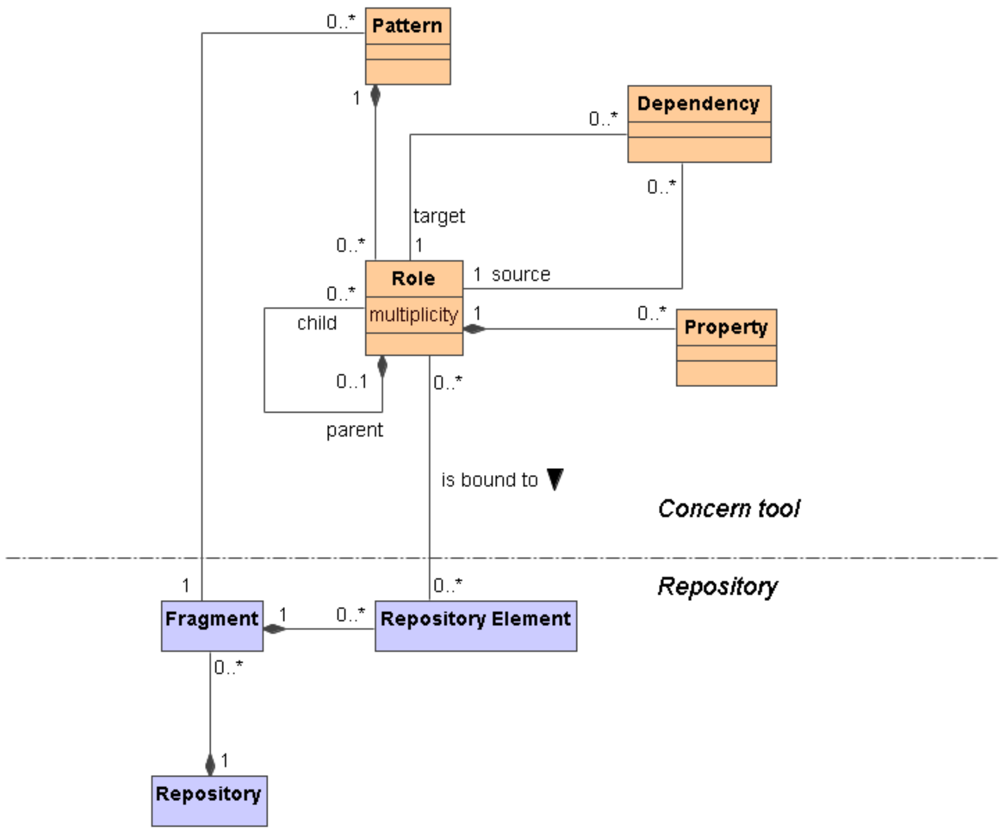


Figure 3.1: Conceptual model for aspectual patterns in UML [HK06]

Let us look at the relationship between a concern and an aspectual pattern. For each concern we wish to map, we create a new pattern. The pattern may or may not be named after the concern, although doing so is simpler and thus a good convention. After we have created our new and currently empty pattern, the next step is to populate it with the various artefacts that we have decided are connected to the concern the pattern is portraying. For each artefact to be bound, a new role must be created under the selected pattern. The role represents a single concern element (artefact) and also manages the connection, e.g. the binding to the actual artefact. Although any number of artefacts can be bound to a role, the concern manipulation toolset presented in this thesis operates under the assumption that there is only a single artefact bound to any given role.

The connection between a role and the artefact is "weak", meaning that a change in either may invalidate the connection. However, since the connection is meant to be navigated only from the role to the artefact, we only need to worry about situations where the artefact changes so that the connection is invalidated.

3.2 Concern Query Language

A set of operations to be used with mapped concerns was defined in Section 2.4. This operation set was by no means meant to be complete or even exhaustive, so a decision was made to create a simple but extensible query language that could be later supplemented with new operations, such as the Cartesian product mentioned in Section 2.4. The basic version of the language should have support for both unary and binary operations, chaining of operations and changing operation precedence with parentheses. From these requirements, a specification in Extended Backus-Naur Form (EBNF) was created, as illustrated in Table 3.1:

Table 3.1: EBNF of query language

<exp>	::=	<factor> { <binary_op> <factor> }
<factor>	::=	[<unary_op>] <primary>
<primary>	::=	<allowed_char> { <allowed_char> } | '(' <exp> ')'
<binary_op>	::=	'+' | '-' | '&'
<unary_op>	::=	'|'
<allowed_char>	::=	[a-zA-Z0-9]

The language does not define any explicit precedence rules between the different binary operations (the unary operation takes precedence over the binary operations by its nature), as it was not deemed necessary in the scope of this thesis. Introducing precedence rules into the language is relatively straightforward as well as expanding the language with new unary or binary operations or allowed characters.

3.3 Tracking Concern Evolution with Patterns

As it was shown in Section 2.5, tracking concern evolution is a complex matter. Using patterns does not change this, so the same limitations apply when studying the scenario where a concern evolves by gaining artefacts, namely that it cannot be done without stakeholder input. As previously mentioned, of the three possible evolution scenarios for a concern, the one where the concern does not change is trivial, and therefore will not be discussed here. The other two scenarios, however, are valid for further discussion.

3.3.1 Concern evolves by losing artefacts

In this scenario, the evolved concern is a subset of the original concern. When this is translated to patterns, we see that the pattern that depicts the evolved concern (later referred to as the evolved pattern) consists of roles and role bindings that can all be found from the pattern that depicts the original concern (later referred to as the original pattern). In fact, the evolved pattern actually consists of *the exact same set of roles as the original pattern did.* The actual difference is that some of the roles in the evolved pattern are not bound to any repository elements, simply because these elements do not exist in the evolved software artefact repository. This is illustrated in Figure 3.2.

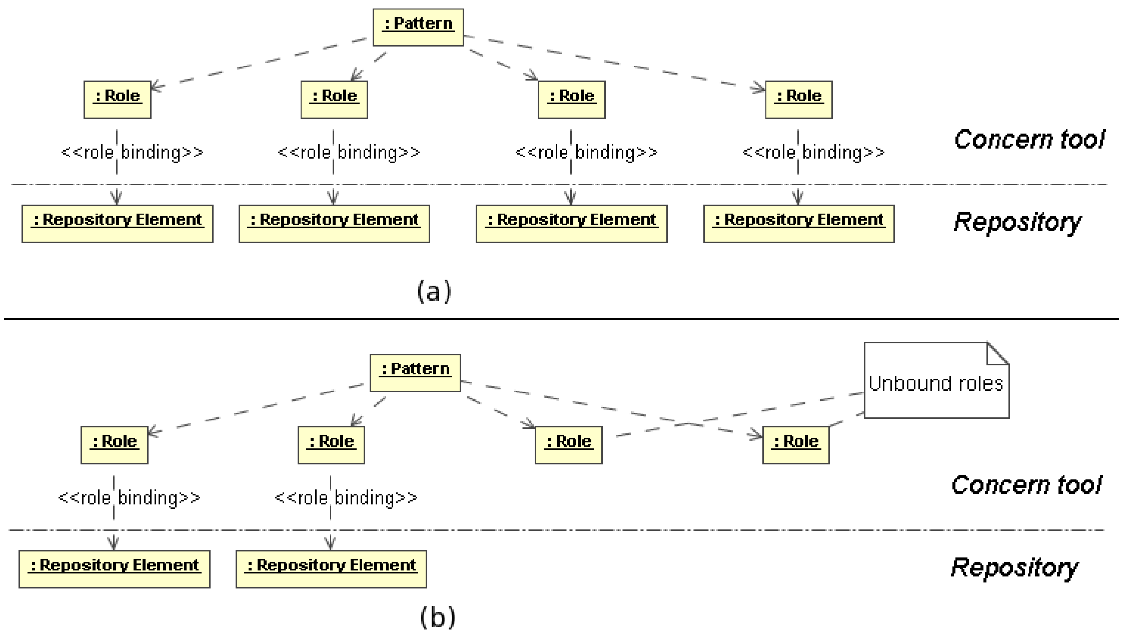


Figure 3.2: (a) The original pattern with all roles bound, (b) the evolved pattern with unbound roles

From the set of roles in the evolved pattern, we can separate those that are bound to a repository element and those that are not. The set of bound roles is actually the evolved concern, i.e. what is left of the original concern after the software has evolved. The set of unbound roles is the change in the concern. Both pieces of information can be useful, though usually in different situations.

3.3.2 Concern evolves by gaining artefacts

As was mentioned in Section 2.5, this scenario cannot be realised without stakeholder help. Unlike in the previous scenario, this time the evolved pattern has actually expanded its set of roles and role bindings. Finding these new roles and bindings is impossible independently. If the original pattern was formed by some kind of systematic rule (e.g. pattern consists of all repository elements whose name starts with the word "Abstract"), the evolved concern could perhaps be found by applying this rule to the evolved software. However, assuming this is what is wanted at all times will most likely eventually lead to undesirable results.

A relatively common scenario is the changing of an artefact that is bound to a pattern. In practice, this can be understood as the concern losing an artefact and gaining an artefact, as was mentioned in Section 2.5. On the pattern level, this situation is problematic only if the change that occurs in the artefact somehow changes the way the artefact is perceived from the outside. An example of this could be the changing of the artefact's name. However, as long as the artefact seems unmodified from the outside, any changes inside the artefact are allowed.

4. IMPLEMENTATION OF THE CONCERN MANIPULATION TOOLSET

This chapter discusses the implementation-specific details of the thesis work. The implementation environment is introduced along with some key software. The implementation of the different tools is discussed in depth and the implementation methods and philosophy are expressed.

4.1 Implementation environment

4.1.1 Eclipse

Eclipse [Ecl] is an open source, platform-independent integrated development environment (IDE). It has become one of the most (if not the most) popular tool for Java development, especially in the open source community. Eclipse is actually a collection of frameworks that fully support extending the IDE into whatever direction needed. This means that there are extensions to Eclipse for developing in languages other than Java, for developing to embedded devices, for controlling web servers and application servers from within Eclipse, etc. All the different added features (as well as the ones that come with Eclipse) are called Eclipse plug-ins.

4.1.2 Rational Rose

Rational Rose [Ros] is one of the older UML CASE-tools. Its history and development is entangled with the history of UML, as UML itself was created by three famous methodologists, James Rumbaugh, Grady Booch and Ivar Jacobson, who all worked at Rational at the time. At one point, Rose was seen as the de facto tool for UML modelling. It still continues to be in use in many of the world's largest software companies, though it is no longer in active development. It only supports UML 1.4, so it is in the process of phasing out as the industry moves more and more towards UML 2.0.

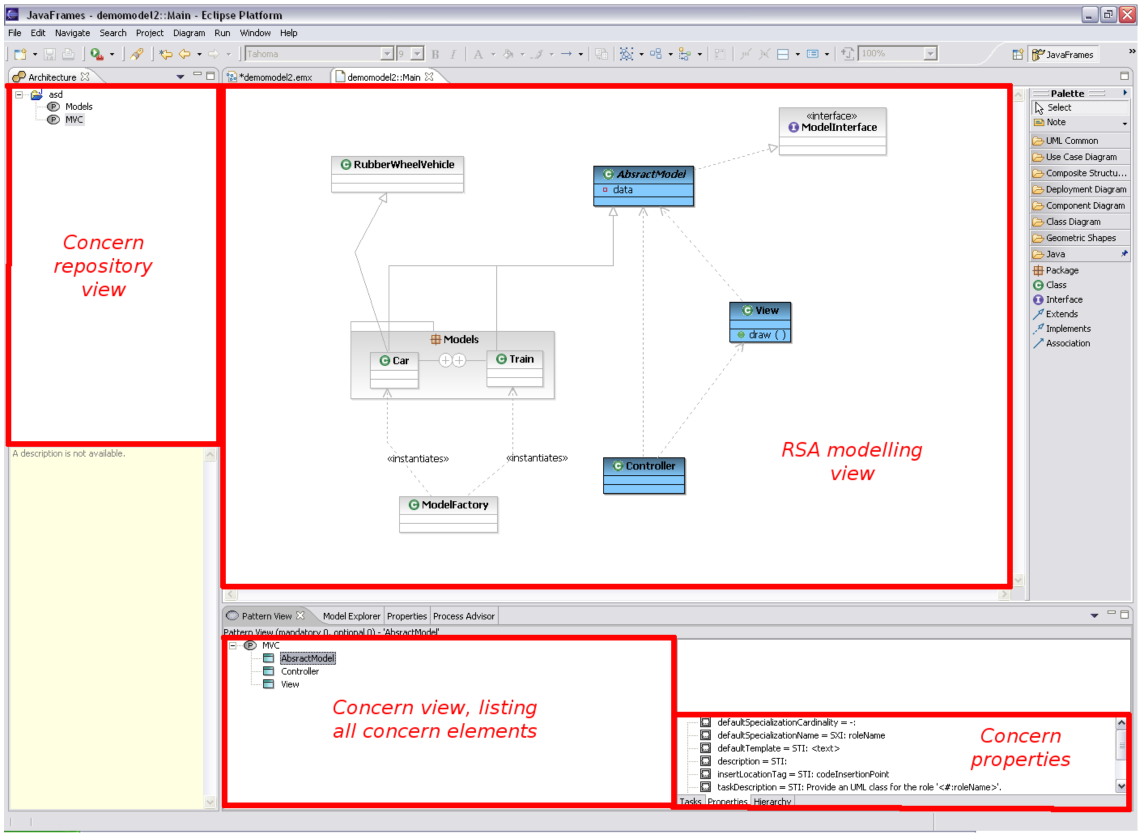
4.1.3 Rational Software Architect

Rational Software Architect [RSA] is an IDE developed by IBM and Rational (currently owned by IBM). It is built upon the technology used in Eclipse, but it extends Eclipse's feature set significantly. One of the central features in RSA are its UML modelling capabilities. It also enables software engineers to do model-driven development [OMG] due to its code generation capabilities and it maintaining a link between UML entities and program code. It is meant to replace Rational's previous modelling tool, Rational Rose, as the industry standard for UML modelling as it fully supports UML 2.0.

4.1.4 INARI

Integrated Archirecting Environment (INARI) [Ham05] is the programmatical bedrock for the concern toolset. INARI is a prototype toolset itself, intended to enable architects to do model-driven development. It is a long-running project that was first developed on the Eclipse platform and later ported onto RSA. INARI relies heavily on a UML modeller. During the Eclipse era this modeller was an external program, Rational Rose. The newer, RSA-based version of INARI uses RSA's built-in modeller.

The original implementation of the toolset was created as an Eclipse plug-in, on top of the Eclipse-based INARI. After most of the coding work had been finished, the toolset was ported onto INARI's RSA-version. The INARI UI can be seen in Figure 4.1.



```
Pattern View (mandatory 0, optional 0) - 'AbsractModel'
MVC
    AbsractModel
    Controller
    View
```

```
defaultSpecializationCardinality = -:
defaultSpecializationName = SXI: roleName
defaultTemplate = STI: <text>
description = STI:
insertLocationTag = STI: codeInsertionPoint
taskDescription = STI: Provide an UML class for the role '<#:roleName>'.
```

Figure 4.1: INARI environment UI

4.2 Tool Implementation

Early on in the thesis project, a decision was made to focus on enabling the concern operations to work on UML entities. While there is no reason why the operations themselves could not be used with code files, packages, documents or any other types of software artefacts, UML entities were seen as the best alternative when illustrating the uses of the concern operations.

Another early design decision was to create each tool in the toolset as a separate plugin, as shown in Figure 4.2. This decision was driven by several different factors. Firstly, it was intended from the beginning that the set of implemented tools would be later on supplemented with additional tools. By defining a common parent class for all the tools and separating them from each other by dividing them into different plugins, it also becomes easier to create new tools for the toolset. Secondly, the tools are relatively independent from one another. Separating them into different modules further underlines this fact and hopefully limits the amount of implicit dependencies between the different tools.

During the implementation, the code had to be ported from Eclipse to RSA, as

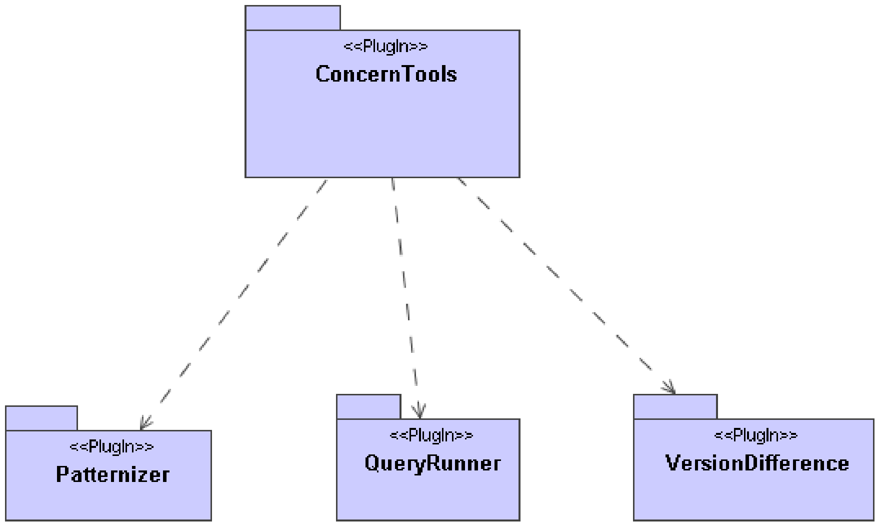


Figure 4.2: Structure of the different concern toolset plugins

the underlying platform, INARI, evolved. This was anticipated from the beginning of the project, but was not deemed to be problematic due to the fact that RSA is based on Eclipse. However, the fact that RSA is using an older version of Eclipse than what was used to implement the tools caused some problems. Namely, the old code had to be downgraded from Java 5.0 to Java 1.4, as RSA did not support some of the features found only in 5.0. This unfortunately caused some deterioration in the quality of the code, as many Java 5.0 -specific features had been used and were difficult to re-implement with 1.4. However, from the user point of view this operation caused hardly any changes to the toolset, disregarding some possible minor change in performance and user interface details.

The implementation work can be drawn roughly into five independent phases: creating user interfaces for all the different tools, building a parser for the query language, creating a tool for the creating of the initial concerns, creating a tool for running the queries and creating a tool for tracking concern evolution. These different phases are covered in the following subsections.

4.2.1 User interface

The creation of the user interfaces for the tools was greatly dictated (but also aided by) the fact that all of the tools were implemented as Eclipse plugins. Eclipse offers an internal workbench UI -plugin to aid plugin developers in creating user interfaces for their plugins. On top of the workbench UI, which defines attachment points to which the developer may attach her plugin on the user interface level, the Eclipse plugin developer SDK also offers the Standard Widget Toolkit (SWT) [SWT] and the JFace framework. These contain a considerable amount of readily usable tools, from low-level "widgets" such as buttons and text boxes (from SWT) to higher level constructs such as dialogs and wizards (from JFace), that free the plugin developer from having to re-invent the wheel in terms of user interface development.

The ideology behind the UI design was to keep everything as simple as possible, but at the same time offer the user a clear route to access all the different functionalities. It was also important that the UIs of the different tools were consistent in appearance.

The most challenging of the tools, in terms of UI implementation, was the patternizer tool (Figure 4.3). Not only is the UI set-up quite complicated with many different composite elements, but the tool UI also included several dialogs and wizard pages. To complicate matters even further, the UI was built so that the tool could later be extended with new element and constraint types, which are more thoroughly discussed in Section 4.2.3. All in all, implementing the UI made up a significant part of the entire implementation.

4.2.2 Query language

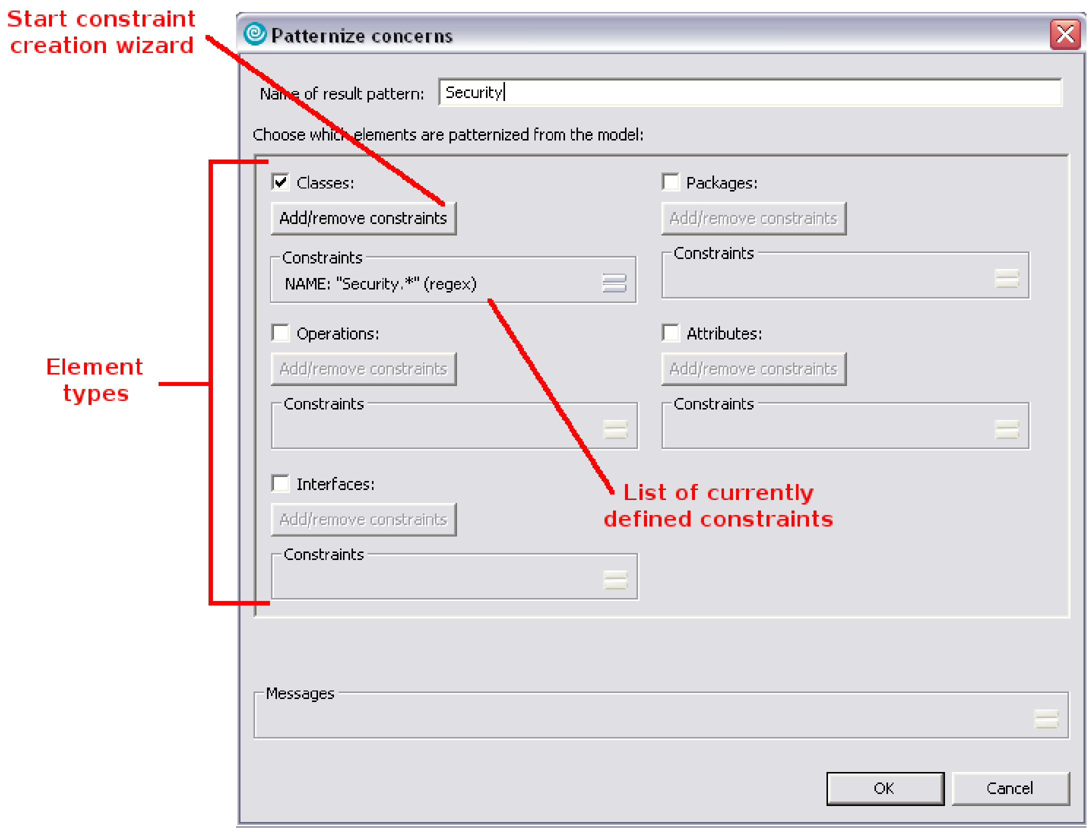
A parser for the query language, which was defined in Section 3.2, was implemented using ANTLR (ANother Tool for Language Recognition) [Par]. ANTLR is one of many parser generators available for Java. It was chosen due to its relative simplicity and its near-direct support for EBNF, in which the query language was originally described. ANTLR was used to generate a parser/lexer pair for the language using a EBNF-like language as input. The ANTLR input code can be found in Appendix 1. The generated parser is used as a validation and navigation tool when interpreting user-defined queries. It translates the query into an abstract syntax tree (AST), in which form the execution of a query is relatively simple, regardless of the complexity of the query.

After the AST has been formed, executing the query is a matter of traversing the tree from top to bottom (i.e. from the root node to the leaf nodes) while executing sub-parts of the query and passing the results of those as parameters to the next part. The order of execution for these sub-parts is defined by the EBNF grammar. The ANTLR-generated parser takes care that the correct order is represented in the AST.

Due to this EBNF-driven approach to the language design, changing or supplementing the grammar requires only little change to the actual code. All that is required is modifying the ANTLR description to be in accordance to the new grammar and regenerating the parser. Of course, modifying the grammar is only a part of the job: the functionality of the possible new operations must also be implemented.

4.2.3 Patternizer tool for creating concerns

At the beginning of this thesis project, it was decided that a tool was needed to streamline the creation of the initial concerns, on which the operations could be used. This tool, shown in Figure 4.3 came to be known as the patternizer (named after the use of patterns to represent concerns). The patternizing tool made it possible for the users to select which element types they wish to include into their concern and then to specify constraints that would further determine which elements are actually bound.


Start constraint creation wizard
Element types
List of currently defined constraints
Add/remove constraints
OK
Cancel

Patternize concerns

Name of result pattern: Security

Choose which elements are patternized from the model:

[✓] Classes:
Constraints
NAME: "Security.*" (regex)

[ ] Packages:
Constraints

[ ] Operations:
Constraints

[ ] Attributes:
Constraints

[ ] Interfaces:
Constraints

Messages

Figure 4.3: Screen shot of the patternizer tool user interface

The element types and their possible constraints were based on those of UML, as at this point it had already been decided that the implementation would be restricted to UML. As these UML-specific constraints are of little use outside UML, this design decision restricts the patternizer to creating concerns of only UML models. Future versions may include functionality to bind artefacts that do not belong to the UML environment.

The tool works by iterating through all the elements of the active UML model, checking whether or not the current element corresponds to the element type and constraints specified. When it finds a match, it adds the found element to the resulting pattern (or concern). The set of available constraints depends on the element type. For example, it is logical to include a constraint for a class element to examine if a class is a generalisation or a specialisation of another class. However, the same constraint does not work if patternizing packages. This means that the set of possible constraints is unique for each element type.

Many of the constraints use regular expressions. For example, for the name constraint (a constraint that filters unwanted elements based on their names and can be used with all element types) the user may define a regular expression for the name of the element. The element is included into the concern if and only if the name of the element matches the regular expression. The wizard for assigning constraints for a UML element is shown in Figure 4.4.

The patternizer was later extended to support adding elements manually to the active concern mapping straight from the model diagram. This functionality is illustrated in Figure 4.5.

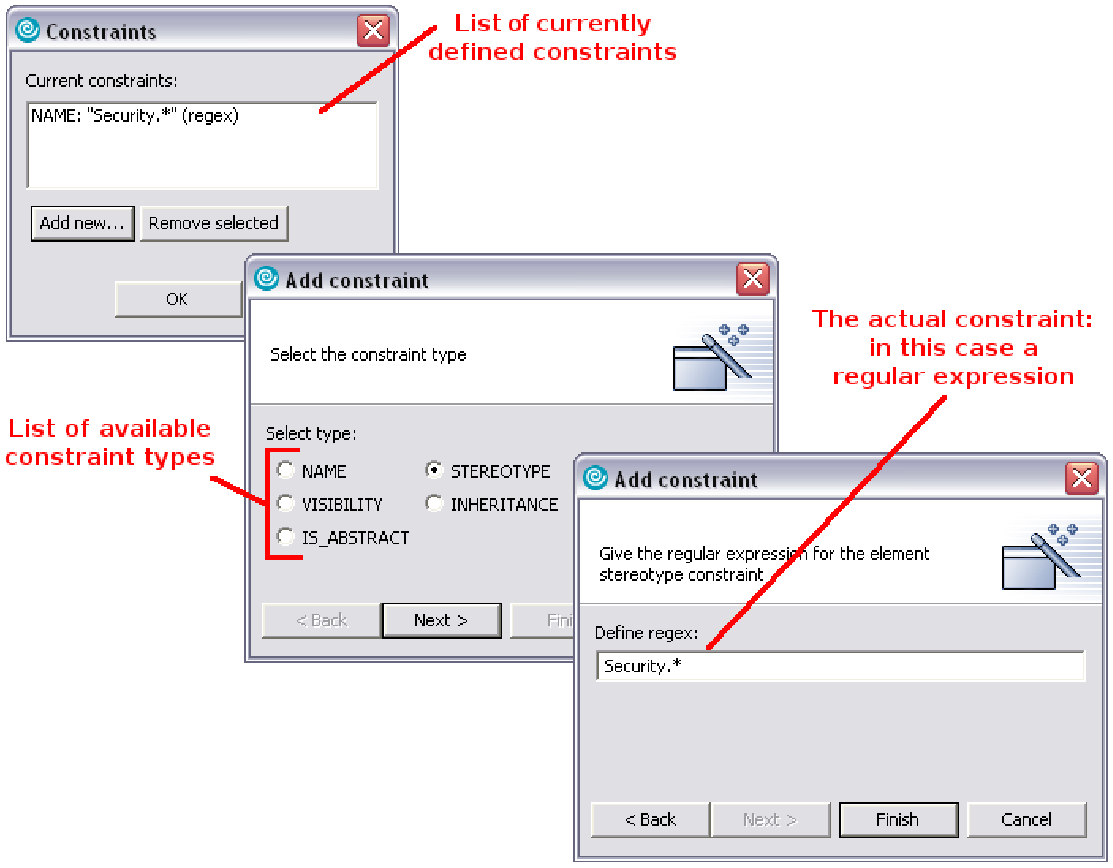

Constraints
Current constraints:
NAME: "Security.*" (regex)
Add new...
Remove selected
OK
List of currently defined constraints
Add constraint
Select the constraint type
Select type:
NAME
STEREOTYPE
VISIBILITY
INHERITANCE
IS_ABSTRACT
< Back
Next >
List of available constraint types
The actual constraint: in this case a regular expression
Add constraint
Give the regular expression for the element stereotype constraint
Define regex:
Security.*
< Back
Next >
Finish
Cancel

Figure 4.4: Screen shot of the constraint creation process

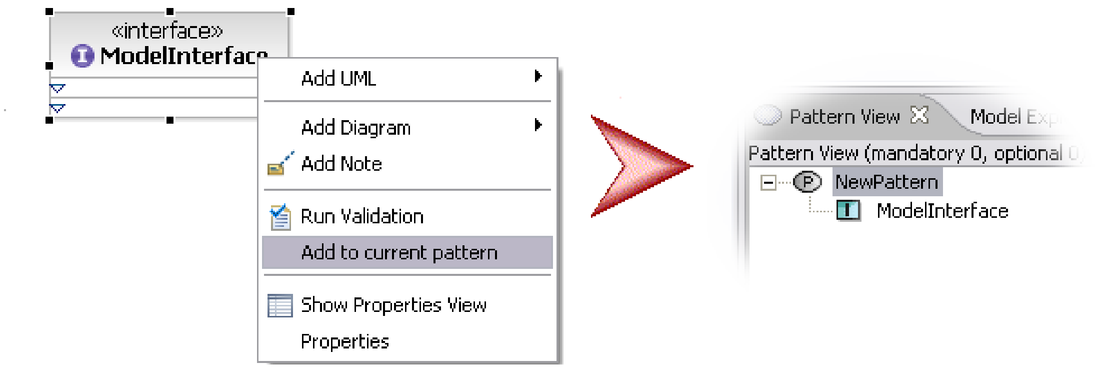
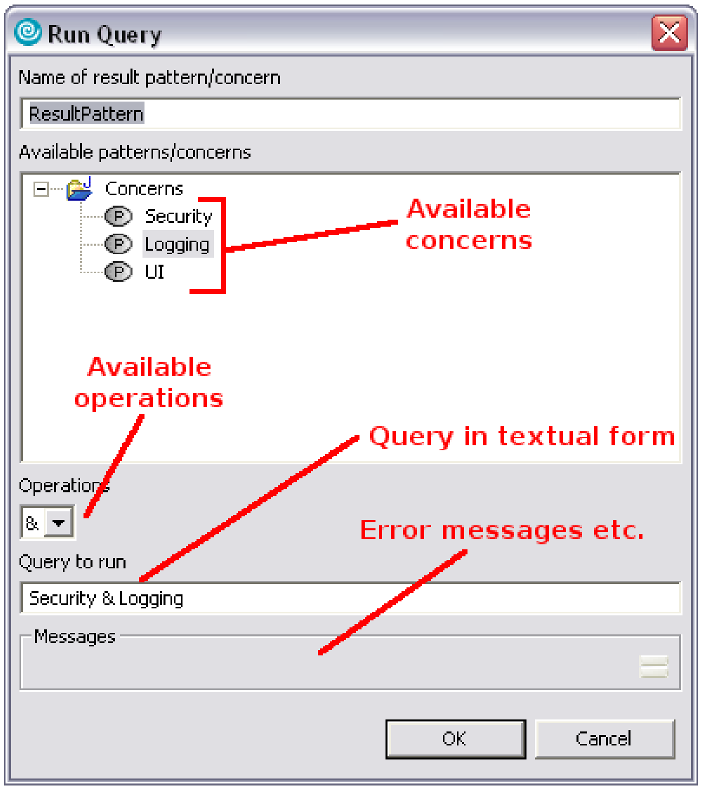


Figure 4.5: Adding a UML interface element to the active concern mapping by hand.

4.2.4 Concern operations

When designing the implementation for the query tool and for the various concern operations (screen shot of the query tool shown in Figure 4.6), some details had to be carefully addressed beforehand. It was clear from the beginning that the set of operations should in no way be limited to the ones we had decided to implement. As the grammar was designed to be extendible in the future, the same philosophy should be employed with the implementation of the operations. This thinking resulted in writing the code so that adding new operations would be relatively simple and, more importantly, would not affect the execution of existing operations.



Name of result pattern/concern: ResultPattern

Available patterns/concerns:
- Concerns
  - Security
  - Logging
  - UI

Query to run: Security & Logging

Messages

Figure 4.6: Screen shot of the query running tool user interface

Another thing that merited extra consideration was the chaining of several operations to form complex queries. This issue was simplified by the fact that the separately generated parser would create a directed tree of the different operations, which was easy to follow from top to bottom while executing operations one by one. The only problem left to be resolved was how to combine the middle products into the final result. All middle products of a complex query are concerns themselves, but from the point of view of the user they are not very interesting. A mechanism had to be crafted to enable the use of concerns for middle products without filling the concern repository with unwanted concerns.

The implementation of the operations became more complex because of the fact that, especially for operations derived from the mathematical set operations, a heavy use of recursion was required. There was also a number of situations where there was no clear way to implement a certain operation for certain parameter concerns. As an example from UML, imagine a situation where we slice a concern with another concern, the latter consisting only of a single, empty UML package. This could be interpreted in a number of ways. For example, the package found in the slicing concern (e.g. the concern that *slices* the other concern, see Section 2.4 for a more in-depth explanation) may be understood to mean the package and also every other element inside that package (analogous to for example removing a directory in a file system). On the other hand, it can also be interpreted so that the slicing should have any effect to the target concern only if an exactly similar UML element structure is found. In this case, exactly similar would mean an empty package.

These questions are left open by the theoretical definition of the operations, simply because they are dependent on the chosen implementation. The theory cannot dictate implementation-level details and the end result is always formed from the subjective decisions of the implementer. For the scenario described above, it was decided that this particular implementation would treat UML packages in the same manner as directories in a traditional file system. That means that if the slicing concern consists of packages, the packages found are removed with whatever content they may have. However, if the slicing concern consists of packages and, for example, UML classes inside those packages, the elements targeted to be sliced are only those classes that can be found from those packages.

4.2.5 Version tracking tool

The implementation of the version tracking tool, shown in Figure 4.7, was quite straightforward. The functionality of the tool is quite simple: it goes through the original concern and tries to find elements from the model that are similarly named.

At one point during the development process, the version tool was planned to be more complex. It was envisioned that the tool would include support for analysing the structure of the model to find structural similarities between the old and new versions of the model. The tool would also be able to successfully guess which new elements would be included into the evolved concern as the underlying model evolved. While determined in Section 2.5 that there is no absolute way to say which

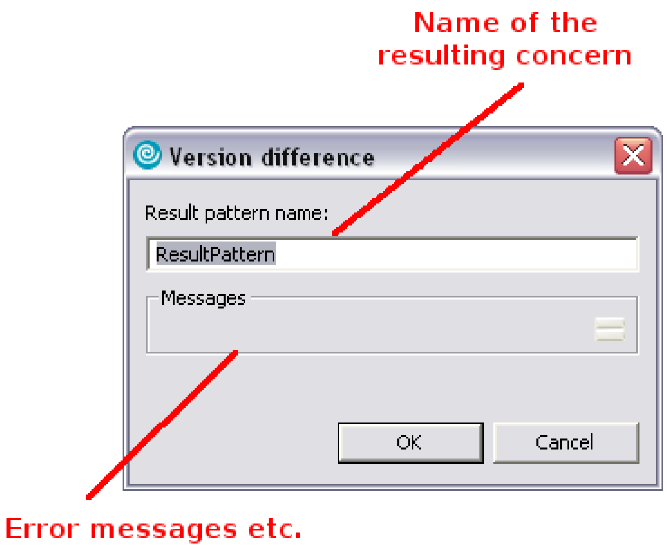


Figure 4.7: Screen shot of the version difference user interface

elements are to be added to the evolved concern as the model evolves, one can always make educated guesses. Unfortunately, for the system to work the "guessing algorithm" must be very sophisticated to avoid getting constant false positives (e.g. elements included into the evolved concern that do not actually belong there). Implementing such an algorithm was seen as beyond the scope of this thesis, but could be implemented as a future improvement of the toolset.

As it stands, the version tracking tool works more as a conveniency tool to streamline the transition of concerns from an old model to a newer one, rather than a full-fledged concern evolution analysis tool. However, justification for its existence is irrefutable. It fills an important void in the toolset, enabling the toolset to be used in a constantly evolving environment.

5. CASE STUDY

A case study was undertaken to test the applicability of the concern manipulation toolset in a practical environment. The toolset was to be tested on a reverse engineered model of Nokia's [Nok] ISA software platform.

5.1 Background

ISA is a proprietary software platform created and maintained by Nokia. It includes the operating system used in all of Nokia's Series 40 mobile phones. It is a closed system that is controlled by Nokia alone and does not support native third-party applications (Mobile Java applications provide an exception to this rule, though their access to the OS core functionality is very limited).

ISA contains a wide array of functionalities, ranging from user interface driven applications, such as the phone book or the calendar, to complex, hardware-oriented functionalities such as telephony or camera operation. All this dictates that, on the source code level, ISA is a fairly complex system and dispersed in nature. This is not aided by the fact that ISA is in a continuous state of evolution to include more features and functionalities. Also, as is quite commonly known, software evolution tends to further increase complexity and weaken the understandability and maintainability of a software system, regardless of the quality of the implementers and the maintainers. [Har03]

Another feature underlining the dispersed nature of ISA is that for each product (e.g. mobile phone) that is running ISA software, a configuration specific for that product exists, defining which of the features of ISA are to be included in that product. No one mobile phone contains all the possible features that ISA contains, so what is actually installed on a product is always a subset of the entire ISA feature set. These characteristics of ISA provide a fine test environment for the concepts and tools presented in this thesis.

5.2 Goals

ISA can be understood as a collection of compilation units containing modules, each of which is dependent on a number of header files. At some point during the development, a design decision was made to place the different header files in a single, global directory to streamline the compilation. However, as the platform evolved and grew, the number of header files placed in the global directory multiplied and reached a size that started to cause more problems than it solved.

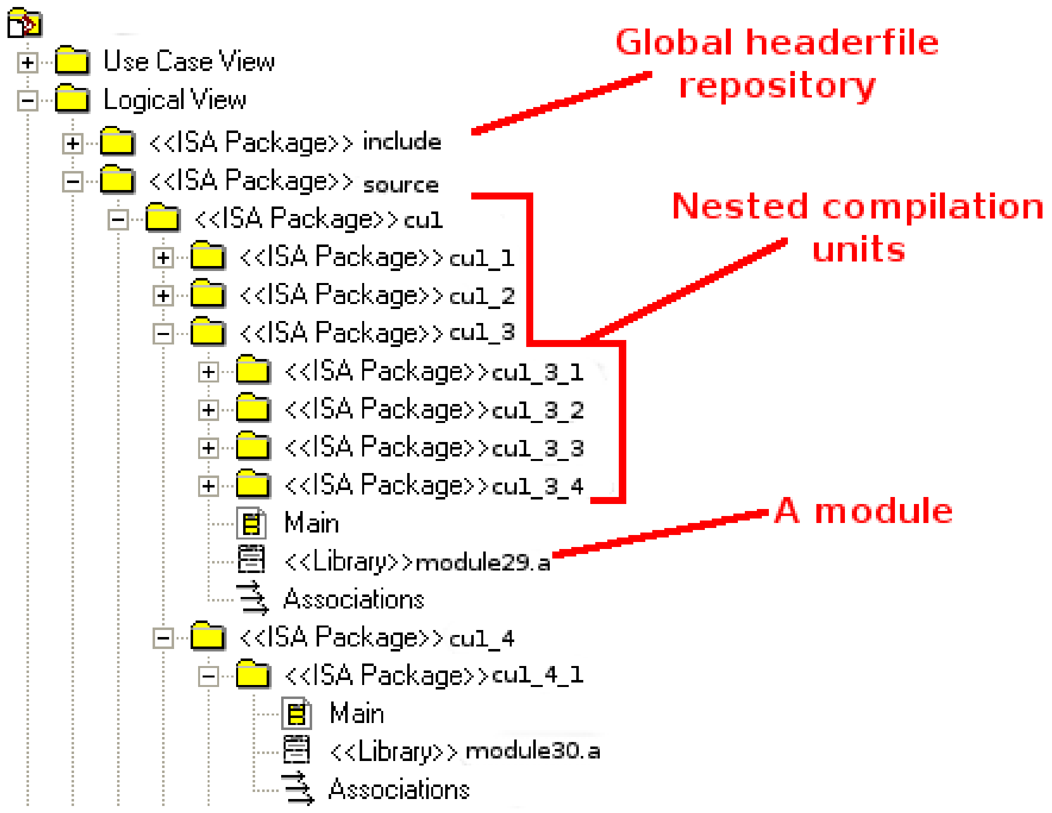


Figure 5.1: Structure of the ISA dependency model (element names changed for confidentiality reasons)

The primary objective of the case study was to help solve the aforementioned problem: to reduce the amount of dependencies between ISA's global header file repository and the different modules. To achieve this, Nokia Research Center (NRC) created a model that focused on presenting the hierarchy between different compilation units and modules and the dependencies each of those modules have with the header files. One of the goals of this case study was to identify header files that were endpoints for only a small amount of modules (0-3 dependencies). Another goal was to find the specific compilation unit, under which all modules dependant on a specific header file resided. The results could then be used to find an optimal location for each header file in the hierarchy of compilation units.

The data on the modules and their dependencies was presented as a Rational Rose UML 1.X model. An example screen capture of this model can be seen in Figure 5.1. In the model, the modules that are dependent on the header files are presented as UML classes (tagged with the stereotype «Library»), while the header files are depicted as UML interfaces. Compilation units are depicted as UML packages (with stereotype «ISA Package») and the dependencies as UML dependencies. The model consists of the (global) repository of header files and a hierarchy of compilation units, each containing one or more modules. To show the dependencies for a single module more clearly, the model also contains a view for every compilation unit showing all the modules and the header files those modules are dependent on. An example of such a view can be seen in Figure 5.2. Though the view is associated to a specific compilation unit, that does not mean that the header files presented in that view actually reside in that compilation unit in the hierarchy, nor that they should reside there. The view can be understood as a cross-cutting view of the system from the perspective of that compilation unit.

5.3 Main Results

The key in solving the problem described in the previous section is the neighbourhood operation, defined in Section 2.4. Running the neighbourhood operation on a concern that consists of a header file returns a concern that lists every module that has a dependency relationship with that header file. The result concern also contains the structural location of the found modules, in a similar manner, as can be seen later in Figure 5.3. After getting this result concern, it is straightforward to deduce the number of dependencies for that header file, as well as for the topmost compilation unit in the hierarchy.

Some modifications to the original concern query tool were made to facilitate running the operation for all of the header files as a batch run. It was decided that due to the thousands of resulting concerns, the results of the batch run would be presented in a more convenient format instead of the result patterns in the INARI tool. It was argued that browsing the results in INARI would be too slow in terms of usability and performance.

Instead, an XML-based [XML] result format was developed, allowing only the essential parts of the resulting data to be presented. For each header file, we would only present information on the name of the file, the amount of modules dependent on it, the topmost compilation unit containing those models, and the location of

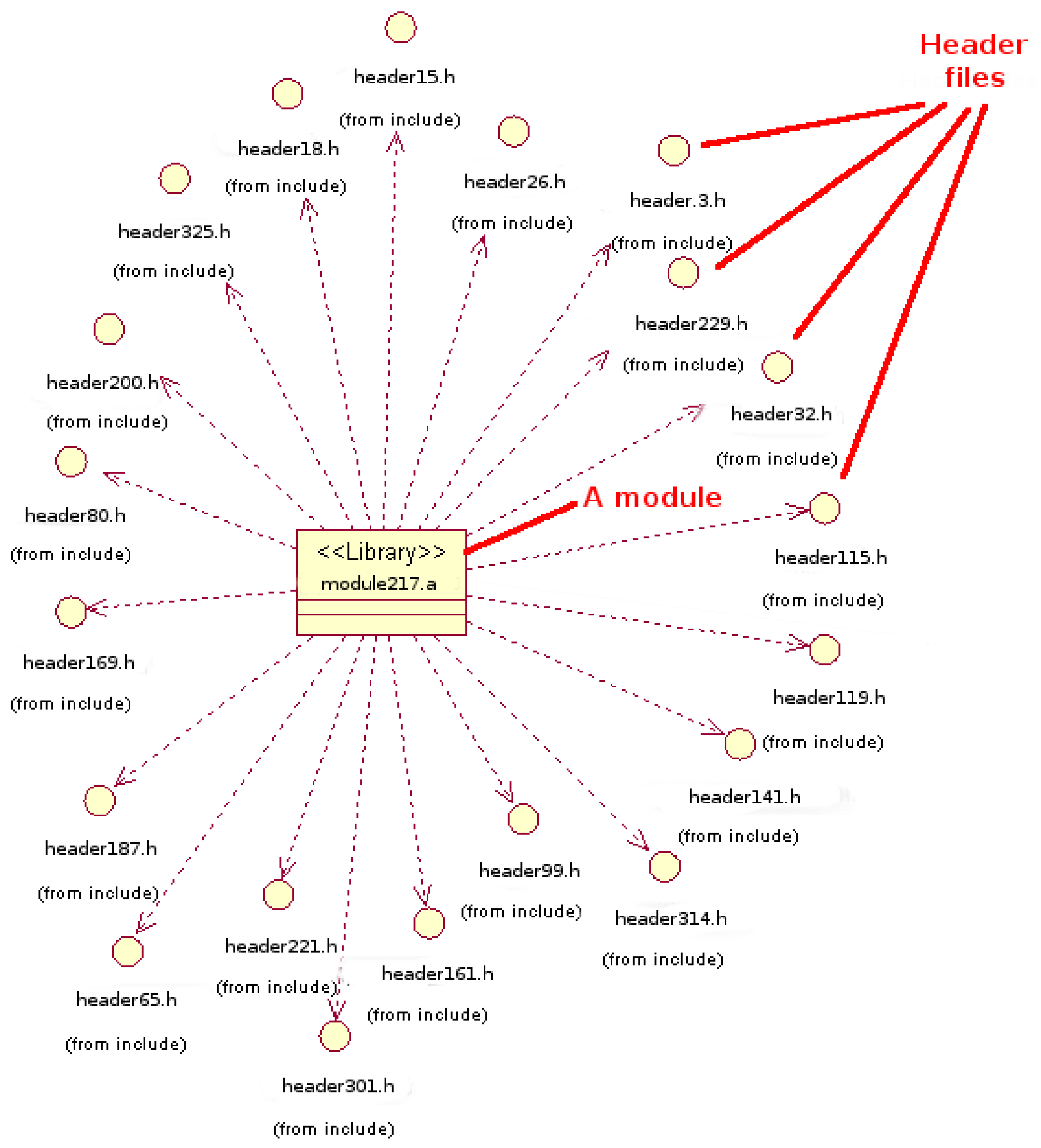
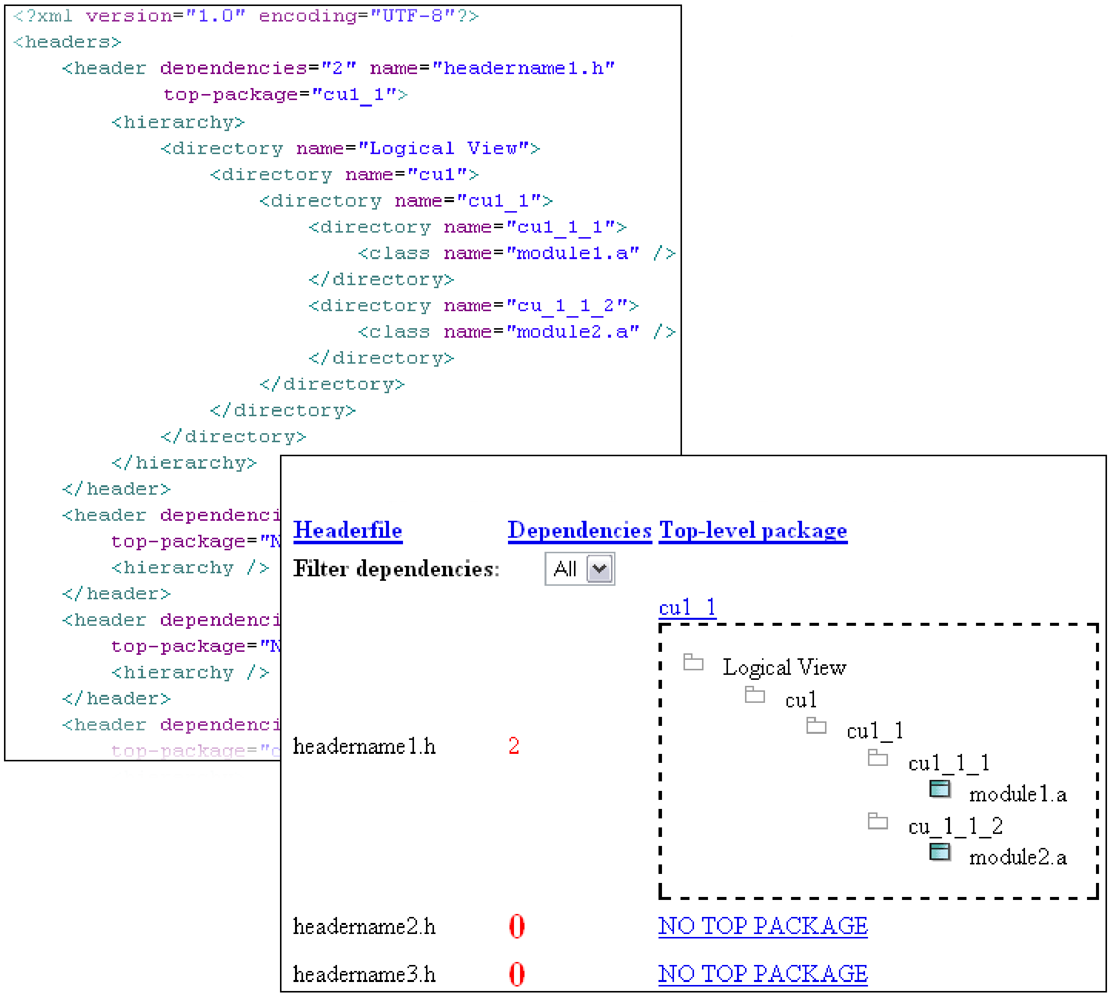


Figure 5.2: A screen capture of the ISA dependency model (element names changed for confidentiality reasons)



```
<?xml version="1.0" encoding="UTF-8"?>
<headers>
    <header dependencies="2" name="headername1.h"
            top-package="cu1_1">
        <hierarchy>
            <directory name="Logical View">
                <directory name="cu1">
                    <directory name="cu1_1">
                        <directory name="cu1_1_1">
                            <class name="module1.a" />
                        </directory>
                        <directory name="cu_1_1_2">
                            <class name="module2.a" />
                        </directory>
                    </directory>
                </directory>
            </directory>
        </hierarchy>
    </header>
    <header dependenci
        top-package="N
        <hierarchy />
    </header>
    <header dependenci
        top-package="N
        <hierarchy />
    </header>
    <header dependenci
        top-package="c
```

Headerfile **Dependencies** **Top-level package**

Filter dependencies: All

headername1.h	2	cu1_1 Logical View cu1 cu1_1 cu1_1_1 module1.a cu_1_1_2 module2.a
headername2.h	0	NO TOP PACKAGE
headername3.h	0	NO TOP PACKAGE

Figure 5.3: Converting the XML result into HTML (element names changed for confidentiality reasons)

the dependent modules in the compile unit hierarchy. This XML-data was then transformed into HTML [HTM] using XSL [XSL] transformation. In HTML-form, and with the help of some JavaScript, the result data could be ordered by the different features or filtered according to the amount of dependencies. A mock-up example of the resulting XML- and HTML-formatted data is shown in Figure 5.3.

5.4 Other uses for the concern manipulation toolset

The case study that was requested by NRC, although clearly a valid proving ground for the concern manipulation toolset, was somewhat limited in scope to what could be done with the toolset. For that reason it is necessary to study other potential scenarios where the toolset can be used for benefit. These scenarios include tracking the evolution of the concerns in the ISA model, as well as studying how different product configurations relate to one another.

5.4.1 Tracking evolution of ISA models

The dependency model received from Nokia was based on a specific weekly build of ISA. To elaborate the case study, we were able to get access to another model based on newer build of ISA. While it is outside the scope of the original boundaries of the case study, this offered an opportunity to test the version tracking aspect of the toolset. Unfortunately, due to the fact that the complete result data of the original case study was presented only in XML/HTML-format, the version tracking could only be done to single header files at a time instead of the entire studied data. A similar, customised batch operation could have been implemented for studying the evolution of the different dependencies, as was done with the original case study. However, it was decided that this did not fit into the scope of the thesis and thus the idea was discarded.

What was learned was that, indeed, transition from an earlier version of the model to a newer one was easier by using the version tracking tool. After the individual concerns, bound earlier to the old version of the model, were remapped to the new version, the user only needed to go through the concerns one by one, delete the elements that had not been bound and add those thought to belong to the concern in question in the new version of the model. Of course, concerns that had been created as a result of queries can usually not be remapped so easily, because the original query result was for the older version of the model. In such a case, to get a valid concern one must first remap the original, fundamental concerns that had been used as parameters for the query and only then re-execute the query. This is the only way to be absolutely certain that the contents are in accordance with what was expected.

One specific evolutionary feature that was was the change in the global header file repository. Since the dependency analysis was done for both of the versions of the dependency model, we already had a concern mapping containing all the header file elements from both of the models. To examine the evolutionary change in the model, the version tracking operation was run to the header concern in the later version in regard to the older version's model (i.e. the header elements of the later version were remapped to the older version). From the result it was possible to determine what header files had been added to the model between the two versions, as those elements understandably could not be bound to the elements found in the older version. The operation could have also been executed in the opposite direction, in which case we could have learned which header files had been dropped from the system between the two versions.

5.4.2 Mapping and studying product configurations

Another place where concerns can be used is in the mapping of the different product configurations in the model. As was previously mentioned, no single product contains the entire set of modules ISA has to offer. Defining each configuration as its own concern would allow us to study how the different configurations overlap and relate to one another. Also, by using the version tracking tool, we could very easily see if there is a problem with one of the product configurations due to the evolution of the system: elements of a certain product configuration concern could not be entirely mapped to the new version if some of the elements were missing.

As a practical example, let us assume there exists a concern mapping that maps all the required modules to build a certain product, hereafter denoted as product A. After the underlying platform evolves, we run the version tracking tool for product A in respect to the new version of the model. The resulting concern mapping shows us whether or not the artefacts that were earlier needed to build a working product A are still present in the model. Missing artefacts may mean that a mistake was made, however, they may also simply mean that some artefacts have been combined, but the contents still exist. Nevertheless, the original product configuration is clearly no longer valid.

5.4.3 Mapping product features as concerns

The following scenario somewhat overlaps with the one presented in the previous section. The premise is that, instead of products, we identify functional features (e.g. the calendar) and create concern mappings for those. After we have done this, we can create any combination of features to form a new product configuration. By using the union operation we simply combine all the different feature concerns into a composition concern that is bound to all the necessary artefacts for that configuration.

The usefulness of this approach is limited by the fact that most of the features may already be split into their separate modules and are therefore already mapped. Whether or not this is the case is unclear from the models that we worked on. The models are clearly heavily modularised, but whether a single module equates to a single feature or service, we do not know. If it were commonplace that the features would cross-cut different modules, this approach could be seen useful, as it would define clear boundaries for a specific feature.

5.5 Tool evaluation

The case study gave an interesting perspective to the theory and implementation of the concepts presented in this diploma thesis. Some problems materialised upon applying the tools to the case study. One of these was the performance of the tools when handling large base material. Manipulating and generating results from the base models required processing time of several minutes up to an hour, depending on the complexity of the action. While it is good to remember that the analysed model was quite substantial in size, there is no denying that the tool implementation could have been more efficient performance-wise. As was mentioned in Section 4.2, one of the implementation philosophies was the possibility to extend the toolset at a later time. As it is well known that extensibility often implies a decrease in performance (and, equally importantly, vice versa), the performance problems did not come as a surprise. To further explain the limitations in performance, it should also be noted that the toolset was built upon an extensible architechting environment (INARI), which in turn was also implemented on top of an extensible platform (Eclipse).

Another, less predicted deficiency were the constrictions set by the user interface of the tools, especially when analysing large models. It was clear from very early on when implementing the case study that the user interface could not meet the requirements of the case study and thus had to be bypassed. Improving the user interface to scale for larger tasks is a candidate for future development.

On the positive side, the case study proved that there indeed is use for tools such as those implemented in this project. Analysing ISA by hand would have required considerably more time than it did when using the toolset to do the mechanical work. It was also fairly easy to customise the toolset to fit the specific needs of this case study (i.e. to create a single new action that composed the existing functionality in a way that was needed by the case study scenario). This is a positive consequence of the decision to emphasise extensibility during implementation. It can also be argued that time saved with a better performing tool would have been lost when customising the tool for this specific need.

Also, even such a restricted case study proved that there is much promise in the tool proposed. It is relatively easy to envision a number of similar analysis- and enhancement -driven scenarios, where the toolset would come in handy. In such situations it would be good to have a customisable multi-purpose set of tools that can be modified with little effort to fill the respective needs. That said, it is important to note that the toolset implemented is only a prototype and as such requires further development and enhancement before such roles can be realistically envisioned.

All in all, the case study can easily be called a success. Not only were the requirements of the client met, but the client was also pleased with the results and the way they were portrayed. It is still unclear whether the toolset will be taken into active use by the client. However, the fact that an additional analysis was requested for a newer version of the model, does seem promising.

6. CONCLUSIONS

Concern-based decomposition of software is a powerful tool in managing the ever increasing complexity of software systems. This thesis presents a set of mechanisms that can be used to take advantage of a software system organised into concerns. It also argues that the correct use of these mechanisms will lead to a better understanding of complex software systems.

Although the implementation of the concepts presented in this thesis is partly constrained to UML, the concepts and ideas scale beyond UML to practically any environment or platform. In fact, there is no reason why a similar implementation of these concepts could not be made for environments outside even the scope of software engineering itself. Controlling complexity and enhancing comprehensibility is certainly important in fields other than just software engineering.

The primary goals of this thesis were to serve as an example implementation of such a mechanism and as a proof of concept. The implemented toolset was very successfully used in a non-theoretical environment, which meant that the original expectations were surpassed: the implementation went beyond being a mere proof of concept to being a tool justified to exist in its own right. Still, much work is left to be done and, even in the scope of this thesis, more could have been done.

The main positives of this diploma thesis are that the originally quite humble goals of creating a toolset for creating and querying concern mappings were reached and, on most fronts, surpassed. Many new ideas were conceived and included as part of this thesis work. The final suite, while far from perfect, is a fully functional and feature-rich tool that gives an array of opportunities and possibilities to those that appreciate the concepts behind it. The case study was the final proof that uses and interest for such a tool indeed exists, and that the implementation itself was mature enough to cope with the challenges placed upon it in the case study.

Some aspects of the thesis project still left room for improvement. For example, now that the benefit of hindsight is available, setting the final boundaries for the thesis was done too late in the project. Though it might have limited what could have been accomplished in this thesis, setting clearer boundaries earlier into the project would have made a difference in the writing process and would have also made it easier to schedule the entire project. This is especially important when bearing in mind the inexperience of the writer. Likewise, work on the case study had begun too late. Since the underlying theme of this thesis is largely dictated by the case study, the fact that it was completed so late in the project caused some complications in this thesis' writing process. These were clearly the most significant deficiencies faced when writing this thesis.

Implementing the concern manipulation toolset and writing this thesis was an effort that took close to nine months of full-time, active development. When examined from that perspective, it is truly a pleasure to say that the project was successful and filled all the expectations that were put upon it. From an extremely subjective viewpoint, there seems to be no reason why the concept of concern-based decomposition could not be applied successfully in many future situations. What remains to be seen is whether it is perceived useful enough to become commonly applied in software development.

BIBLIOGRAPHY

[And91] John A. Anderson. Manageable object-oriented development: Abstraction, decomposition, and modeling. *Proceedings of the conference on TRI-Ada '91: today's accomplishments; tomorrow's expectations*, pages 199–212, December 1991.

[aWs] IBM. alphaWorks research program. http://www.alphaworks.ibm.com/, January 2007.

[BZL06] Silvia Breu, Thomas Zimmermann, and Christian Lindig. Mining eclipse for cross-cutting concerns. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 94–97, New York, NY, USA, 2006. ACM Press.

[Dic] Merriam-Webster Online Dictionary. Concern. http://www.m-w.com, October 2006.

[Ecl] Eclipse Foundation. Eclipse. http://www.eclipse.org, January 2007.

[Hal60] P.R. Halmos. *Naive Set Theory.* Litton Ed. Publ. Inc., 1960.

[Ham05] Imed Hammouda. *Multi-Dimensional Structuring of Software Systems: Tools and Applications.* PhD thesis, Tampere University of Technology, 2005.

[Har03] Maarit Harsu. *Ohjelmion ylläpito ja uudistaminen.* Talentum, first edition, 2003.

[HK06] Imed Hammouda and Kai Koskimies. Concern based mining of heterogeneous software repositories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 80–86, New York, NY, USA, 2006. ACM Press.

[HKK04] Imed Hammoud, Mika Katara, and Kai Koskimies. A tool environment for aspectual patterns in UML. In *Proceedings of WoDiSEE*, pages 58–65. IEE, May 2004.

[HTM] World Wide Web Consortium. HyperText Markup Language. http://www.w3.org/MarkUp/, January 2007.

[Kan03] Mohammed M. Kandé. *A Concern-Oriented Approach to Software Architecture.* PhD thesis, École Polytechnique Fédérale de Lausanne, 2003.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[Mil56] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.

[Nok] Nokia Incorporated. http://www.nokia.com/, January 2007.

[OMG] Object Management Group. http://www.omg.org/, January 2007.

[OT00] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[Par] Terence Parr. Another tool for language recognition (antlr). http://www.antlr.org, December 2006.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Ros] Rational Software. Rational Rose. http://www.rational.com/, January 2007.

[RSA] IBM Rational. Rational Software Architext. http://www-306.ibm.com/software/awdtools/architect/swarchitect/, January 2007.

[SR02] Stanley M. Sutton, Jr. and Isabelle Rouvellou. Modeling of software concerns in cosmos. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 127–133, New York, NY, USA, 2002. ACM Press.

[Sun99] Sun Microsystems. *Code Conventions for the Java Programming Language*, 1999. http://java.sun.com/docs/codeconv/, October 2006.

[SWT] Eclipse Foundation. Standard Widget Toolkit. http://www.eclipse.org/swt/, January 2007.

[THO$^+$00] Peri Tarr, William Harrison, Harold Ossher, Anthony Finkelstein, Bashar Nuseibeh, and Dewayne Perry. Workshop on multi-dimensional separation of concerns in software engineering (workshop session). In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 809–810, New York, NY, USA, 2000. ACM Press.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[UML] Object Management Group. Unified Modelling Language. http://www.uml.org/, January 2007.

[XML] World Wide Web Consortium. Extensible Markup Language. http://www.w3.org/XML/, January 2007.

[XSL] World Wide Web Consortium. The Extensible Style Sheet Language Family. http://www.w3.org/Style/XSL/, January 2007.

APPENDIX 1: ANTLR CODE

```
header {
  package fi.tut.cs.practise.inari.queryrunner.parser;
}

class QueryParser extends Parser;
options {
  buildAST=true;
  defaultErrorHandler=false;
}

query:      factor ((MERGE^|OVERLAP^|SLICE^) factor)* ;
factor:     (EXCLUSION^|BOUNDARY^)? primary ;
primary:    ALLOWED_CHAR | LPAREN! query RPAREN! ;

class QueryLexer extends Lexer;
options {
    k=2;
    charVocabulary='\u0000'..'\u007F'; // allow ascii
}

LPAREN       : '(' ;
RPAREN       : ')' ;
QUOTE  : '"' ;
MERGE        : '+' ;
OVERLAP      : '&' ;
SLICE        : '-' ;
EXCLUSION    : '#' ;
BOUNDARY    : '|' ;
ALLOWED_CHAR : (('a'..'z')|('A'..'Z')|('0'..'9'))+ ;
WS           : ( ' '
               | '\r' '\n'
               | '\n'
               | '\t'
               )
               { $setType(Token.SKIP); };
```