

**An Integrated System to
Manage Crosscutting Concerns in
Source Code**



The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

An Integrated System to Manage Crosscutting Concerns in Source Code

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op vrijdag 25 januari 2008 om 10.00 uur
door

Marius Adrian MARIN

Diplomat Engineer in Civil Engineering – Buildings Services
Licentiate in Economics – Economic Cybernetics, Statistics and
Informatics
geboren te Boekarest, Roemenië

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. Arie van Deursen

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. A. van Deursen	Technische Universiteit Delft & Centrum voor Wiskunde en Informatica promotor
Dr. ing. L.M.F. Moonen	Technische Universiteit Delft
Prof. dr. P. Tonella	ICT-irst & Università degli Studi di Trento
Prof. dr. S. Demeyer	Universiteit van Antwerpen
Prof. dr. P. Klint	Centrum voor Wiskunde en Informatica & Universiteit van Amsterdam
Prof. dr. C. M. Jonker	Technische Universiteit Delft
Prof. dr. ir. H.J. Sips	Technische Universiteit Delft

Copyright © 2007 by A.M. Marin

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without the prior permission of the author.

ISBN 978-90-9022675-0

Author email: a.m.marin@tudelft.nl

To my parents,

Dan and Doina

Contents

Acknowledgments	xi
1 Introduction	1
1.1 Software Evolution in the Presence of Crosscutting Concerns	1
1.2 Problem Statement	4
1.2.1 Aspect Mining	5
1.2.2 Concern Modeling	5
1.2.3 Aspect-Oriented Programming and Refactoring Towards Aspects	5
1.2.4 Challenges and Problem Statement	6
1.3 Objectives	6
1.4 Research Method and Evaluation	7
1.5 Overview	8
1.5.1 A Study of Crosscutting Concerns	8
1.5.2 Crosscutting Concern Sorts	9
1.5.3 Crosscutting Concern Mining, Modeling and Refactoring using Sorts	10
1.6 Contributions	13
1.7 Road map	14
2 Identifying Crosscutting Concerns using Fan-in Analysis	17
2.1 Introduction	17
2.2 Aspect Mining: Background and Related Work	19
2.2.1 Terminology	19
2.2.2 Query-Based Approaches	22
2.2.3 Generative Approaches	23
2.2.4 Aspect Identification Case Studies	24
2.3 Aspect Mining Using Fan-in Analysis	25
2.3.1 A Fan-in Metric for Aspect Mining	25
2.3.2 Method Filtering	27

2.3.3	Seed Analysis	28
2.3.4	The Fan-in Tool FINT	29
2.4	The Case Studies	30
2.4.1	First Findings	31
2.4.2	Case Study Presentation	33
2.5	PETSTORE	34
2.6	JHOTDRAW	37
2.6.1	The Undo Concern	37
2.6.2	Persistence	39
2.6.3	Observers in JHOTDRAW	40
2.6.4	Other Concerns	41
2.7	TOMCAT	44
2.7.1	Lifecycle	45
2.7.2	Valves / Chain of Responsibility	46
2.7.3	Other Concerns	46
2.8	Discussion	48
2.9	Concluding Remarks	52
2.9.1	Contributions	52
2.9.2	Future Work	53
3	Applying and Combining Three Different Aspect Mining Techniques	55
3.1	Introduction	55
3.2	Background concepts	57
3.2.1	Fan-in	57
3.2.2	Concept Analysis	58
3.2.3	Terminology	60
3.3	The three aspect mining techniques	61
3.3.1	Fan-in Analysis	61
3.3.2	Identifier Analysis	62
3.3.3	Dynamic Analysis	63
3.4	Results of the Aspect Mining	64
3.4.1	The Fan-in Analysis Experiment	64
3.4.2	The Identifier Analysis Experiment	66
3.4.3	The Dynamic Analysis Experiment	68
3.5	Comparing the Results	69
3.5.1	Selected Concerns	69
3.5.2	Limitations	72
3.5.3	Complementarity	73
3.6	Toward Interesting Combinations	74
3.6.1	Motivation	74
3.6.2	Definition of the Combined Techniques	75
3.6.3	Analysis Indicators	75
3.6.4	Experimental Results	76

3.7	Summary and Future Work	78
4	Crosscutting Concern Sorts	81
4.1	Introduction	81
4.2	Crosscutting Concern Sorts	83
4.2.1	The Query Model	84
4.2.2	Description and Formalization of Sorts	84
4.3	Sort-Based Concern Modeling	89
4.3.1	SOQUET	91
4.3.2	Documentation of FigureChanged Observer	92
4.3.3	SOQUET Support for Software Evolution	93
4.4	Sorts in Practice	93
4.4.1	JHOTDRAW	94
4.4.2	Enterprise Applications	97
4.5	Sorts in Design Patterns	99
4.5.1	Interfacing Commands and <i>Adding variability</i> to Commands and Visitors	101
4.5.2	<i>Design enforcement</i> in Singleton and Prototype	102
4.5.3	Other Patterns	102
4.6	Discussion	104
4.6.1	Coverage of the Crosscutting Concerns by Sorts	104
4.6.2	Using Sorts in Aspect Mining and Refactoring	106
4.7	Related Work	107
4.8	Conclusions	108
5	A Framework for Evaluating and Combining Aspect Mining Techniques	109
5.1	Introduction	109
5.2	A Common Framework for Aspect Mining	111
5.2.1	Crosscutting Concern Sorts	113
5.2.2	Defining the Common Framework	113
5.3	Three Aspect Mining Techniques	115
5.3.1	Fan-in Analysis	115
5.3.2	Grouped calls Analysis	117
5.3.3	Redirections finder	118
5.4	Combining Techniques	119
5.4.1	Improving Precision	119
5.4.2	Improving Absolute Recall	120
5.4.3	Improving the Seed-Quality	120
5.5	Tool Support	123
5.6	Experiment	123
5.6.1	Applied Filters	124
5.6.2	Results	125
5.7	Retrofitting Existing Techniques	129

5.7.1	Role Superimposition	129
5.7.2	Consistent Behavior	131
5.7.3	Context Passing	132
5.7.4	Name-Based Approaches	133
5.8	Discussion	133
5.9	Related Work	136
5.10	Conclusions	136
6	An Integrated Strategy for Migrating Crosscutting Concerns	139
6.1	Introduction	139
6.2	Crosscutting Concern Sorts	140
6.3	An Integrated Migration Strategy	141
6.3.1	Aspect Mining	143
6.3.2	Concern Exploration	144
6.3.3	Concern Modeling and Documentation	144
6.4	Aspect Refactoring	145
6.5	Aspect Refactoring of JHOTDRAW	147
6.5.1	AJHOTDRAW	147
6.5.2	Consistent Behavior in Command	148
6.5.3	Undo Functionality	149
6.6	Discussion	154
6.6.1	Applicability in Practice	154
6.6.2	Benefits and Risks	155
6.6.3	Automation	156
6.6.4	Separation of Concerns	156
6.7	Related Work	157
6.8	Concluding Remarks	158
7	Conclusions	161
7.1	Summary of Contributions	161
7.2	Discussion and Evaluation	163
7.2.1	Revisiting Thesis Objectives	163
7.2.2	Independent and Integrated Migration Steps	163
7.2.3	Queries versus Aspects	164
7.3	Opportunities for Future Research	166
7.3.1	Aspect Mining	166
7.3.2	Crosscutting Concern Documentation and Modeling	167
7.3.3	Refactoring to Aspect-Oriented Programming	167
7.3.4	Integration of Migration Steps	169
7.4	Closing Remarks	169

A	FINT	181
A.1	Installation	181
A.2	User manual	181
A.2.1	Fan-in analysis	182
A.2.2	Grouped calls analysis	195
A.2.3	Redirections finder	195
A.2.4	Combination of techniques	203
A.2.5	Seeds management	203
B	SOrts QUeRY Tool (SOQUET)	207
B.1	Installation	207
B.2	User manual	207
B.2.1	Modeling and documenting concerns in SOQUET	208
B.2.2	Using SOQUET to aid program comprehension and software change tasks	223
	Samenvatting	235
	Curriculum Vitae	241

Acknowledgments

It does not happen often that you get the chance to acknowledge the people that make a difference for you, and, as I am getting to learn, people deserve to know that you appreciate them. I would start by saying that these 4 years I spent in Delft working on my PhD research have been particularly special for me, and not only because I got paid for doing what I like, but also because the people I met.

It was an honour and a privilege to meet and work with Arie (prof. van Deursen), and just as much of a pleasure. If it is true that meeting a great mentor is (mainly) a matter of chance, then I can surely call myself lucky.

During these years I collaborated, in various degrees, with a number of fellow researchers, and I had the opportunity to co-author papers with some of them: first of all Leon Moonen, then Paolo Tonella, Tom Tourwé, Kim Mens and Mariano Ceccato. I would like to thank them as well as the members of the committee for their valuable comments on this thesis: prof. dr. P. Tonella, prof. dr. S. Demeyer, prof. dr. P. Klint, prof. dr. C. M. Jonker, prof. dr. ir. H.J. Sips.

A number of people who experimented with the tools I developed, FINT and SO-QUET, kindly provided me with their feedback, which I very much appreciate.

I would also like to thank dr. Carlos Infante Ferreira, who supervised my Master's thesis in Delft, and who encouraged me to take on a PhD challenge. From the same group, I need to mention Dong-Seon Kim who was always available with a friendly advice.

Eamonn McDonagh willingly went over some of my first paper drafts, and it was always fun and good lessons to have his comments.

Coming back to SWERL, thanks to my "warriors", football and "borrel" teams (i.e., {{{Bas Cornelissen, Cathal Boogerd, Rui (...) Abreu}}, Bas Graaf, Marco Lormans (also a very enjoyable tennis partner)}, Ali Mesbah (yes, we still need to write "that" paper together), Leon Moonen (WCRE in Delft was the most pleasant conference to me!), Gerd Gross, Peter Zoetewij}}, all of which I so gladly joined. You guys keep up the good work!

Special thanks go to my room mates, Bas (alias sebas) and Andy Zaidman, who,

besides accepting the shades to be raised, helped me so kindly with the Dutch summary.

All the other colleagues in the group, particularly Frans Ververs, contributed to a special atmosphere.

I shall conclude this part with two friends from completely different parts of the world: Giorgio Alfarano and Ulysses Locadia. It is always great, and so comforting, spending time with you.

Last word is for my wife, Ioana: you know, all the good things are so because of you.

Delft
December, 2007

Marius Marin

Evolution of software systems accounts for the largest part of their lifecycle and costs. Software engineers therefore, more often than developing new systems, work on complex, existing ones that they have to understand in order to modify them. Understanding such systems requires insight into the various concerns the systems implement, many of which have to be inferred from source code. Particularly challenging for software comprehension, and consequently, software evolution, are those concerns said to be crosscutting: implementation of such concerns lacks modularity and results in scattered and tangled code.

The research presented in this thesis proposes an integrated approach to consistent comprehension, identification, documentation, and migration of crosscutting concerns in existing systems. This work is aimed at helping software engineers to more easily understand and manage such concerns in source code. As a final step of our approach, we also experiment with the refactoring of crosscutting concerns to aspect-oriented programming and reflect on the support provided by this new programming technique for improving modularization of concerns.

1.1 Software Evolution in the Presence of Crosscutting Concerns

Software engineers are often confronted with the daunting task of analyzing and understanding complex software systems into which they have little or no a priori insight. Many of these systems consists of millions of lines of code and interdependent projects developed by large teams. For example, the 2007 coordinated project release of the popular Eclipse¹ open development platform consists of 21 projects and over 17 million lines of code contributed by more than 310 developers. Compared to the previous year's release, these figures show that the code size has increased by around 100%.

¹<http://www.eclipse.org/>

Analyzing such systems is inherent in operation and maintenance of software, which is estimated to account for as much as 50 to 90% of the software's total costs [Sommerville, 2004; Erlikh, 2000; Pigoski, 1996].

In order to deal with this complexity and support the engineers in their comprehension tasks, techniques for modularization and separation of concerns have been proposed [Parnas, 1972; Dijkstra, 1997; Baldwin and Clark, 1999]. Nevertheless, complete separation of concerns is difficult or even impossible to achieve using modularization mechanisms available in today's most popular programming paradigms, such as object-oriented programming [Tarr et al., 1999]. In these paradigms, concerns like monitoring of objects' events or state, persistence, exception handling, security, auditing, and other various policies to be implemented consistently are typically non-modular, spanning multiple modules in a software system.

Unmodularized concerns are said to be *crosscutting* and exhibit symptoms like *scattering* – the implementation of a concern is spread over several program modules –, and *tangling* – a program module implements multiple concerns –. These symptoms are illustrated in Figure 1.1 for a crosscutting concern in JHOTDRAW, a framework for drawing applications which we shall analyze in detail in this thesis. The concern is part of an Observer pattern solution, which allows for automatic notification and update of a set of objects (i.e., the Observers) with the state changes of an object they depend upon (i.e., the Subject) [Gamma et al., 1994]. In our case, the notification is realized by invocations of a dedicated (changed) method by all the actions whose execution alters the state of the observed object. These invocations are shown as horizontal blue lines in Figure 1.1, and cut across multiple modules represented as rectangles for each class of a caller-method.

The scattering of the observers-notification concern is due to the multiple places where the invocation of the changed method needs to be inserted. Consequently, a modification in the requirements for the notification strategy implies changes to all the call sites of this method. Furthermore, a caller-method, like the one shown at the bottom of Figure 1.1, needs to address multiple, tangled concerns: besides its primary concern of modifying the *font* attribute of a text display, the method also notifies observers of this modification. Similarly, any new method added to the system that changes the state of a Subject object needs to be aware of and consistent with the notification concern, and implement the concern accordingly.

The challenges posed by crosscutting concerns are further apparent from the simplified implementation of the Observer design in JHOTDRAW, summarized in Figure 1.2: elements such as connections between figures or drawing views observe Figure objects for state changes. To comply with the design, a Figure not only needs to implement its core functionality, like drawing operations, but also a number of operations that allow observer-objects to be updated with any change in its state. These operations include the *willChange* and *changed* method to be invoked before and after a modification to a Figure respectively. In a real-life application, Figures might implement even more additional roles, like *persistence*, which requires that each Figure type defines operations to read and write itself from/to a storing device, or support for self-

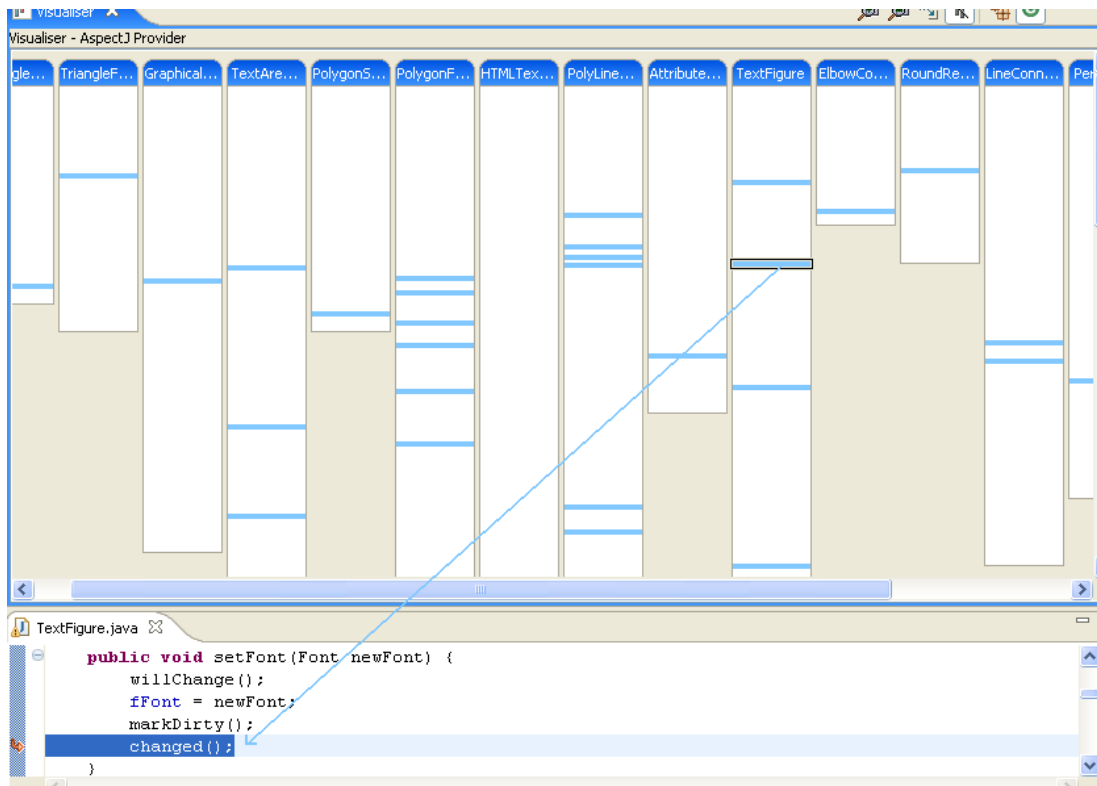
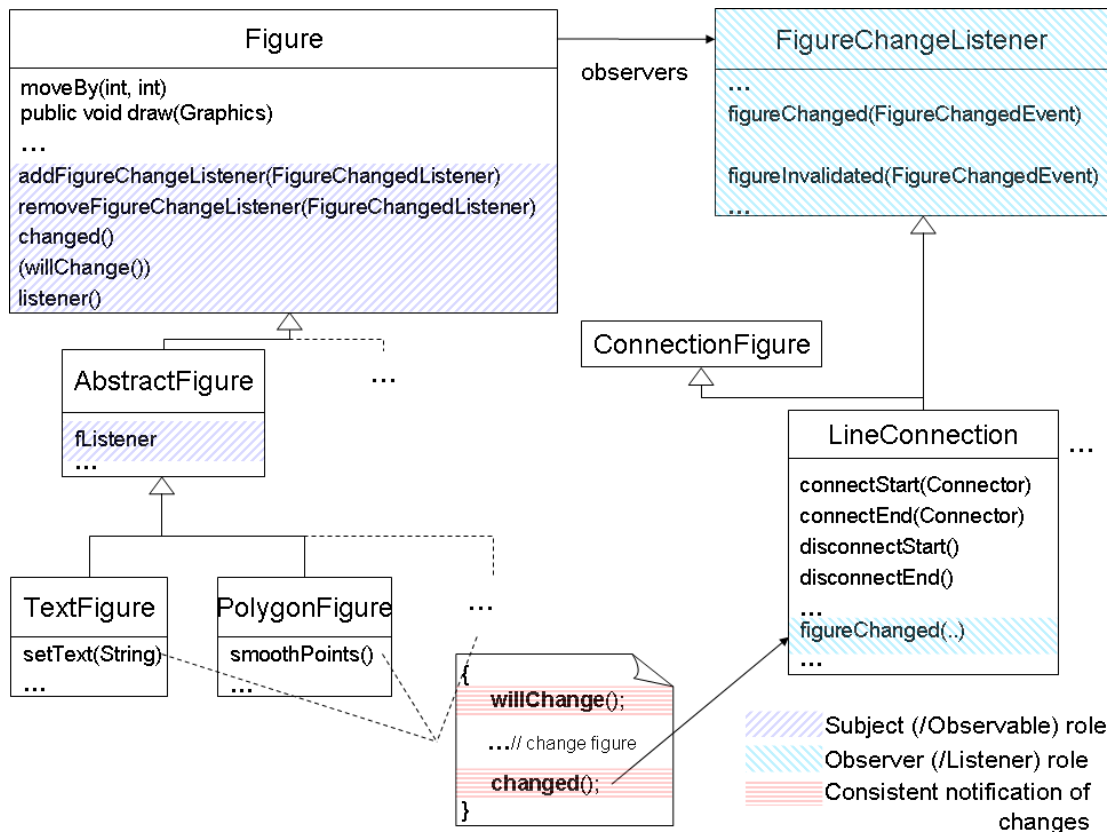


Figure 1.1: Scattering and tangling of the notification concern for figure changes in the JHOTDRAW drawing application.



cloning. Each of these different roles shows a distinct concern whose implementation is tangled with the other concerns in a sole module, namely the Figure type.

Lack of modularization of concerns hinders software comprehension: crosscutting, scattered concerns are difficult to recognize and reverse engineer from source code, and tangled code is hard to understand. Moreover, software evolution tasks might easily overlook crosscutting concerns as their underlying relations remain “hidden” in source code. This results in modifications or extensions to existing systems that are inconsistent with (crosscutting) policies and rules already present in those systems. Consequently, the new code breaks compliance with existing concerns, or duplicates their definition and implementation.

1.2 Problem Statement

The problem of crosscutting concerns has been investigated at various stages of the software lifecycle. Researchers have proposed solutions that include new programming techniques for software development, such as aspect-oriented programming (AOP) [Kiczales et al., 1997; Filman et al., 2005], software analysis techniques for

identification of concerns in source code (also known as *aspect mining*) [Marin et al., 2007a; Ceccato et al., 2006], or concern browsing and modeling approaches [Robillard and Murphy, 2002; Janzen and Volder, 2003; Harrison et al., 2004; Hajiyevev et al., 2006]. Below, we take a brief look at these approaches and then formulate our problem statement.

1.2.1 Aspect Mining

Aspect mining is a relatively recent research area aimed at developing (source code analysis) techniques and tool support for (semi-)automatic identification of crosscutting concerns in existing systems.

Identification of crosscutting implementation is a necessary first step to consider in order to ensure awareness of various concerns implemented by a system. As for the Observer example above, new elements added to a system need to know what functionality, other than their main concern, they have to implement in order to comply with existing design and requirements. Moreover, this step is important for understanding how crosscutting concerns occur in real life applications, how they are typically implemented, and what specific properties distinguish them from other concerns.

1.2.2 Concern Modeling

A next issue to consider is the representation of the identified crosscutting concerns in source code, to consistently describe, model and document them. A number of approaches to concern exploration, representation and source-code querying, like Concern Graphs [Robillard and Murphy, 2002] and the Concern Manipulation Environment [Harrison et al., 2004], have been proposed so far.

Concern modeling allows us to persistently document discovered concerns and emphasize those program elements that pertain to the implementation of these concerns. Moreover, such documentation can make explicit crosscutting relations between program elements, and hence help in conducting software comprehension and evolution tasks.

1.2.3 Aspect-Oriented Programming and Refactoring Towards Aspects

Aspect-oriented programming subsumes various programming techniques designed to support modularization of crosscutting concerns in source code by using new language constructs and composition mechanisms. The most popular of these approaches to date is AspectJ² [Kiczales et al., 1997], a Java language extension based on a *joinpoint* model. This model allows a programmer, for instance, to specify (in a declarative

²eclipse.org/aspectj/

way) sets of execution points in a program where a certain code, like the observers-notification invocation discussed earlier, to be executed.³

To improve modularity of concerns in existing systems by means of aspect-oriented techniques, we need to *migrate* these concerns by refactoring their implementation to aspect-oriented solutions.

Most of the available refactoring solutions are examples-oriented [Laddad, 2003b]. Preliminary steps towards systematic, reusable solutions for refactoring to aspects have been taken by Hannemann et al. [2005], who proposed a role-based approach to refactoring design patterns, and Monteiro and Fernandes [2005], who initiated a catalog of fine-grained refactorings.

1.2.4 Challenges and Problem Statement

Despite this considerable research, a number of important challenges to enhancing the management of crosscutting concerns in source code remain open. The various solutions available to date are typically hard to integrate with each other and do not define uniform criteria for assessment. Even within the same approach, like for concern mining or refactoring, crosscutting concerns are addressed at different levels of granularity, which makes it difficult to compare and combine solutions. Furthermore, open tool support and detailed case-studies are rather scarce.

Similarly, the solutions to concern modeling do not distinguish specific characteristics of crosscutting concerns, and typically have a broader scope than these concerns, such as code browsing.

This thesis focuses on crosscutting concerns in existing systems and proposes to answer the research question of:

How can we consistently manage, i.e. identify, model, document and possibly migrate, crosscutting concerns in existing systems in order to better support program comprehension and effective software evolution?

1.3 Objectives

In answering our research question, we set the following objectives for our solution:

Objective 1 The solution should provide a coherent and consistent way to address crosscutting concerns in source code. Currently, the fairly comprehensive, yet vague, definitions of concerns in general, regarded as “any matter of interest in a software system” [Sutton and Rouvellou, 2005], or of crosscutting concerns in particular, (“properties” that “cannot be cleanly encapsulated in a generalized procedure” [Kiczales et al., 1997]) do not ensure such consistency. As a consequence, the aspect mining and

³We will give a more detailed introduction into AspectJ in Chapter 2, and report on our experience with applying it in the last chapter of this thesis.

refactoring approaches address concerns at various levels of granularity and complexity. The examples of concerns range from simple logging functions or authorization mechanisms, to complex designs, transactions management or business rules [Laddad, 2003b; Hannemann et al., 2005]. Such approaches are therefore difficult to integrate, to consistently assess, compare or combine.

Objective 2 Our solution should result in common benchmark(s) that allow others to experiment with new techniques for identification of concerns, and compare with our own results. Such benchmarks ask for detailed reports of the aspect mining results and a consistent system to present and document these results. Moreover, we aim at providing tool support to enable assessment of the proposed techniques on new benchmarks, as well as reproducible results.

Objective 3 At the time of writing, a multitude of aspect mining techniques exist. Unfortunately, their results are often hard to compare, and integrating multiple techniques into one tool has proved difficult, as argued before. Our solution should provide criteria and make it possible to integrate, compare, and evaluate different aspect mining techniques in a reproducible manner.

Objective 4 Managing crosscutting concerns consists of different steps, including identification, documentation and modeling, and refactoring to aspect-oriented programming. We aim at a well-integrated system allowing one, for example, to directly use aspect mining results in a concern modeling tool, which then can be used to come up with a suitable solution to refactor to aspects.

Objective 5 The concern documentation and refactoring solutions should ensure flexibility and re-usability so that they support (future) integration in development environments. This requires that the solutions aim at abstracting from particular concerns and are applicable to all concerns that share the same properties.

1.4 Research Method and Evaluation

The research methodology adopted in this thesis rests upon the following pillars:

- Use of descriptive case studies for obtaining a better understanding of the problem domain. This includes, for example, a detailed account of actual occurrences of crosscutting concerns in existing systems.
- Development of new theory, concepts, and techniques, such as novel aspect mining techniques, concern modeling approaches, or a new characterization of the notion of crosscutting concerns.

- Development of tools to permit application of the methods and techniques to existing software systems.
- Validation of the new methods and techniques through explorative case studies, in which the software tools developed are applied to a range of (open source, Java) systems.
- Analytical generalization of the case study results including a critical discussion of the case study findings. This evaluation is done per chapter, as well for the full thesis in the conclusions, based on the objectives proposed in the previous section.

Thus, in this thesis, tool development and case studies form an important part of the research methodology and evaluation approach, in line with observations from Kitchenham et al. [1995] and Yin [2003].

1.5 Overview

In order to address our research question and meet our objectives, we adopt the following approach:

1. We start by conducting a study of crosscutting concerns in actual systems. To support this study, we propose a new aspect mining technique.
2. Given our understanding of crosscutting concerns in actual systems, we propose a categorization of concerns in *sorts* by typical implementation idioms and specific relations.
3. Next, we use the crosscutting concern sorts to build an integrated system to manage crosscutting concerns in source code. The system consists of three main components, for aspect mining, for documentation and modeling of concerns, and for refactoring of concerns to aspect-oriented solutions, respectively.

Each of these steps will be discussed next.

1.5.1 A Study of Crosscutting Concerns

Our first step consists of acquiring a better understanding of what crosscutting concerns are, how they occur in practice, and how they are typically implemented in software systems. To this end, we conduct aspect mining tasks on a number of open source, object-oriented (Java) systems, from several application domains, that comprise over 500,000 non-comment lines of code. These systems include a framework for drawing applications (JHOTDRAW⁴), a J2EE enterprise application (Sun's Java PETSTORE

⁴<http://jhotdraw.org>

application⁵), and a servlet engine (TOMCAT⁶), as well as a J2EE-based application server (JBoss⁷), and the Java Development Tools component of the Eclipse integrated development environment (JDT plug-in⁸).

In a first experiment described in Chapter 2, we propose *fan-in analysis* as a general-purpose aspect mining technique. The technique searches for methods with large numbers of scattered callers, which are likely to implement concerns like logging, listeners updates, exception handling, etc. In our experiment, we apply fan-in analysis to three of the open-source systems just mentioned. The results of the experiments are covered in detail and show a significant variety of examples of crosscutting functionality, including concerns not previously discussed in literature.

We extend this experiment in Chapter 3 with a comparative study of fan-in analysis with two different aspect mining techniques developed by other research groups. This joint study uses JHOTDRAW as a common benchmark. This makes it possible to see what sort of crosscutting concerns are discovered by each technique, and whether these techniques yield overlapping results.

The joint study also revealed the inherent complexity of actually comparing aspect mining results. A major difficulty consists of the tedious effort of correlating mining results of different techniques due to the lack of a system to consistently describe these results and the identified crosscutting concerns. For instance, if we assume that one technique is able to identify the crosscutting roles in the Observer design, and another technique recognizes the crosscutting implementation of the notification mechanism, a question here is how to report these two valid results. The challenge lies in the fact that a common practice in aspect mining (and refactoring to aspects) is to report and describe results by referring to well-known examples from literature that discuss crosscuttingness. The Observer design is one such example [The AspectJ Team, 2003]. However, the two techniques in our case find distinct crosscuttingness in the design's implementation that can not be reported as the same result.

While Chapter 3 focuses on the various mining results obtained from the three techniques, Chapter 5 will address the comparison challenge, and propose a framework for consistently comparing and assessing the quality of aspect mining techniques.

1.5.2 Crosscutting Concern Sorts

The experience gained from our aspect mining experiments and case studies allows us to recognize and categorize *atomic* crosscutting concerns, i.e., concerns that cannot be decomposed into smaller, yet meaningful, concerns. We do so by distinguishing atomic concerns based on properties like their specific underlying relations and implementation idioms in object-oriented (Java) systems. For instance, concerns like logging, authorization and authentication checks, events notification, etc, follow a same idiom,

⁵<http://java.sun.com/blueprints/petstore>

⁶<http://tomcat.apache.org/>

⁷<http://www.jboss.org/products/jbossas>

⁸<http://www.eclipse.org/jdt/>

namely scattered invocations to the method implementing the crosscutting concern. These concerns can be grouped based on their shared idiom in a distinct category of concerns, which we can call *Consistent behavior*.

Similarly, the idiom to assign multiple roles to a class, like persistence or observability for changes, consists of *implement* relations for the members of each of the various roles. The concerns associated to these roles could be grouped together in a different category, such as *Role superimposition*.

The resulting categories are called concern *sorts*, which are discussed in Chapter 4. This chapter not only proposes sorts, but also presents a catalog of the most commonly encountered sorts.

We also observe that complex designs and mechanisms commonly acknowledged for their crosscutting properties can be described as compositions of the atomic concerns that we recognize. For example, the crosscuttingness in the Observer pattern discussed earlier consists of the composition of the two super-imposed roles, the Subject and the Observer roles to be implemented by observable and listener elements respectively, and the mechanism to consistently notify observers of changes in the subject's state. Each of these atomic concerns is an *instance* of one of the two different sorts introduced above.

1.5.3 Crosscutting Concern Mining, Modeling and Refactoring using Sorts

We use the classification of crosscutting concerns into sorts to address three important issues in managing crosscutting concerns in source code. First, we propose an evaluation framework for aspect mining. Second, we offer an innovative way of conducting concern modeling. Last but not least, we provide a systematic approach to refactoring object-oriented code towards aspect-oriented programming.

A common framework for aspect mining We use the classification of crosscutting concerns in sorts to define a common framework for consistent, idiom-driven aspect mining and assessment of mining techniques. The framework and its applications are described in Chapter 5.

The framework requires that a mining technique:

1. Defines its search-goal as instances of a specific sort. For example, Fan-in analysis aims at the *Consistent behavior* sort.
2. Describes the representation of its mining results. Fan-in analysis, for instance, reports results as method-call relations.
3. Defines a mapping between the representation of its results and the idiom typically used for the targeted sort. Fan-in analysis results, for example, can directly map its results onto the representation of the *Consistent behavior* sort, provided

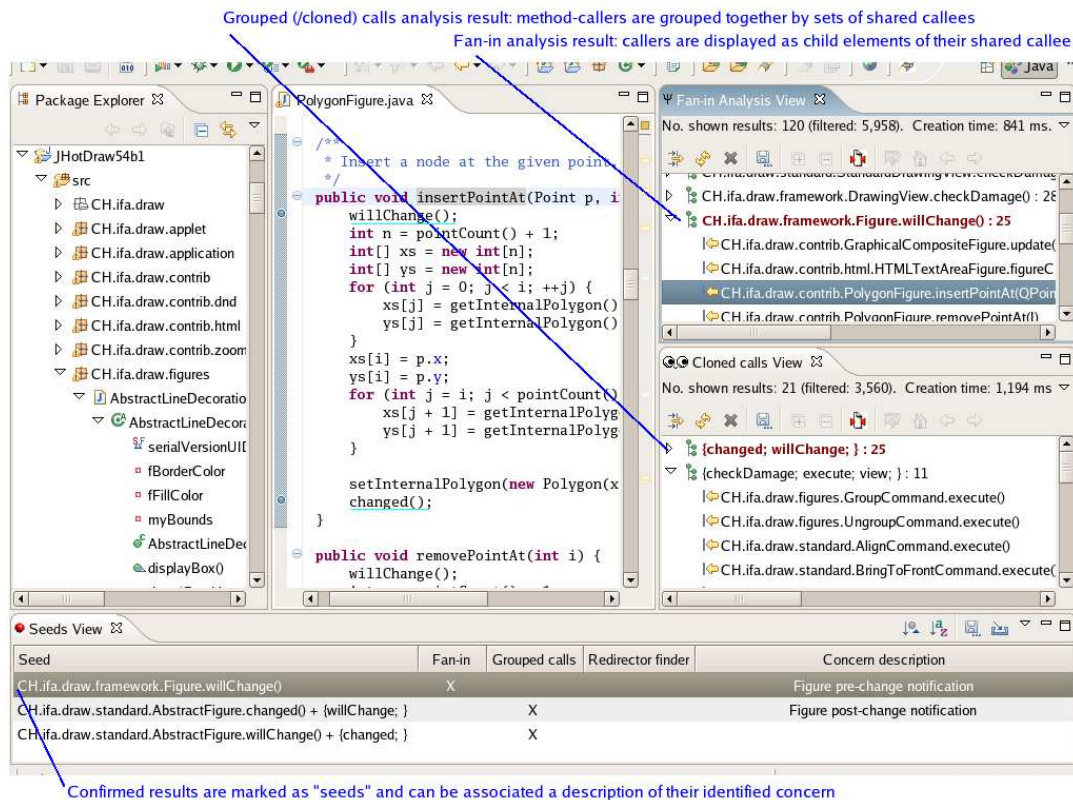


Figure 1.3: FINT views for source code analysis and management of the aspect mining results.

that they preserve the mapping of the endpoints of the relation: the crosscutting element is on the callee side, and the crosscut element is on the callers side. Mining results that do not map count as false positives.

4. Defines a set of metrics to assess its performance. For instance, a metric like *precision* can be used to indicate the percentage of valid results in the total set of reported results of a technique. A new metric that we propose is *seed quality*, which measures the mapping between a mining result and the crosscutting concern it identifies.

We use the framework to design two new aspect mining techniques that target different concern sorts. These, together with Fan-in analysis, are implemented in our aspect mining tool FINT, which is openly available as an Eclipse plug-in.⁹ Figure 1.3 shows results of two of the techniques in FINT, which are aimed at the *Consistent behavior* sort. The *Seeds* view, at the bottom of Figure 1.3, assists the user in managing the aspect mining results of the various techniques.

⁹<http://swerl.tudelft.nl/view/AMR/FINT>

Sort	Short description
<i>(Method) Consistent behavior</i>	A set of method-elements consistently invoke a specific action as a step in their execution.
<i>Redirection layer</i>	A type-element acts as a front-end interface having its methods responsible for receiving calls and redirecting them to dedicated methods of a specific reference, optionally executing additional functionality.
<i>Expose context (Context passing)</i>	Method-elements part of a call chain declare additional parameter(s) and pass it as argument to their callees for propagating context information along the chain.
<i>Role superimposition</i>	Type-elements extend their core functionality through the implementation of a secondary role.

Table 1.1: Sorts of crosscuttingness.

Moreover, in Chapter 5, we give an overview of the most important aspect mining techniques at the moment and position them into our framework.

Query-based documentation and modeling of concerns To document the identified concerns, we use sorts and formalize each sort by means of a query over a source code model, which we cover in detail in Chapter 4. The sort-query captures the sort's relation and describes its idiom.

As an example, the intent of the *Consistent behavior* sort is to extend the core concern of a set of methods by means of a systematic call to some specific functionality, such as notification of observers. Thus, the query for *Consistent behavior* reports all the call relations between two (user-)defined sets of program elements: one set consists of the crosscutting element, i.e., the callee, while the other set comprises the crosscut elements, i.e., the callers that are part of the concern of interest.

Similarly, the query for *Role superimposition* describes an *implement* relation between a set of program types, on the one side, and members that belong to a crosscutting role implemented by these types, on the other side. Other sorts, some of which are shown in Table 1.1, are formalized by similar, albeit sometimes more complex, queries, as we shall see in Chapter 4.

The queries form the basis for the *Sort Query Tool* (SOQUET¹⁰), our concern modeling and documentation tool, described in Chapter 4. Figure 1.4 shows how SOQUET can be used to document an instance of the *Consistent behavior* sort: the query receives a parameter to indicate the method whose calls are crosscutting, such as the notification method for Figure changes, and another parameter to define the collection of crosscut callers, which, in this case, includes only the set of those callers that are declared in the *Figure* type hierarchy.

The parameterized sort queries document concrete, atomic concerns in the code. These can be grouped together in composite, hierarchical concern models to further document complex features or designs, such as an Observer pattern. An example of concern model is shown in the same Figure 1.4.

¹⁰<http://swierl.tudelft.nl/view/AMR/SoQueT>

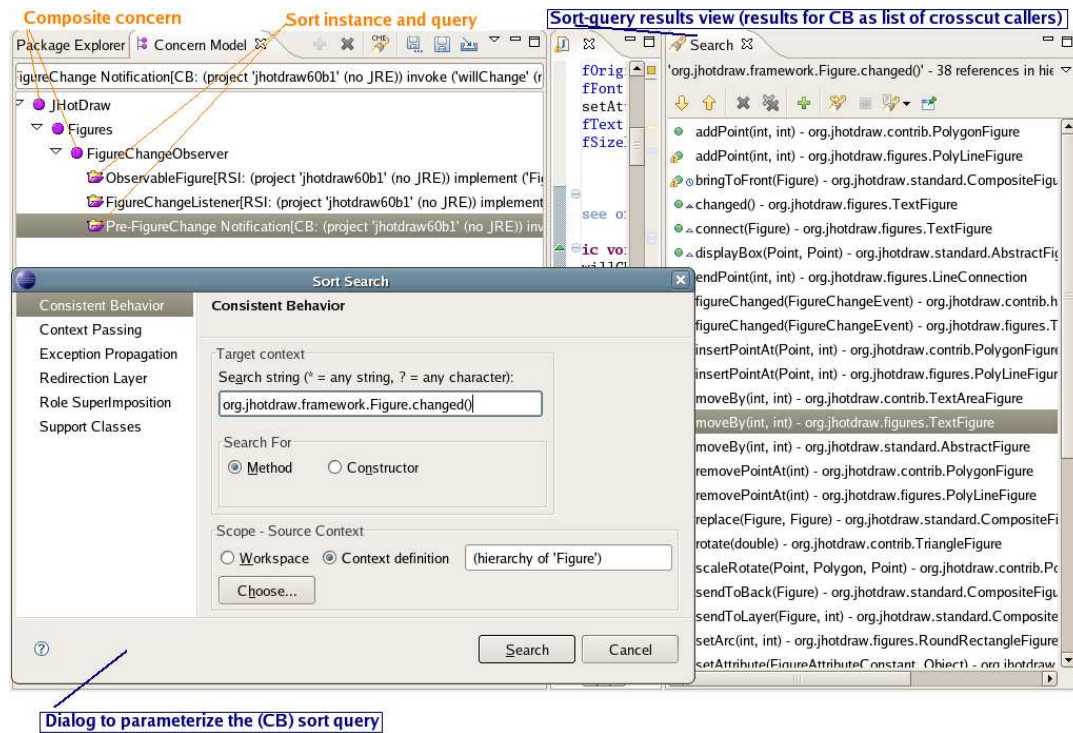


Figure 1.4: SOQUET views and dialogs.

Refactoring to aspect-oriented programming Last but not least, crosscutting concern sorts offer a way of conducting systematic refactoring of object-oriented systems towards aspects. For each sort, a specific refactoring can be defined. To actually refactor a sort instance, the corresponding sort query can be used as a starting point.

Sort-based refactoring ensures an optimal trade-off between the complexity of the refactoring and comprehensibility of the refactored concern: while addressing meaningful concerns, the refactoring (mainly) consists of one aspect language mechanism, which allows for a high degree of flexibility of the aspect solution for the various instances of a sort.

Furthermore, sorts form the glue for an integrated concern migration strategy, in which results from aspect mining can be directly used (via the corresponding sort-based documentation) as starting point for a subsequent refactoring. This integrated strategy is the topic of Chapter 6.

1.6 Contributions

The main contributions of the thesis can be summarized as follows:

- The most comprehensive report on aspect mining results and crosscutting concerns in source code available to date. We analyze and report in detail, in Chap-

ters 2, 3 and 5 on three relevant open-source systems.

- A set of three aspect mining techniques and tool support for these techniques and their combination, discussed in Chapters 2 and 5.
- A novel classification of crosscutting concerns on distinctive properties, and a tool-supported, query-based approach to documenting and modeling concerns, described in Chapter 4.
- A new approach to refactoring of concerns to aspect-oriented programming based on atomic crosscutting concerns, and a show-case for refactoring to aspects that is available as an open-source project, AJHOTDRAW. This is also the largest system publicly available to date that is the result of a refactoring towards aspects. The approach and its application are discussed in Chapter 6.
- An integrated migration strategy including steps for aspect mining, concern documentation and modeling, and aspect refactoring. This is presented in Chapter 6.

1.7 Road map

The chapters of this thesis cover three main research topics:

- The identification of crosscutting concerns in source code, also known as aspect mining, which is covered in Chapters 2, 3, 5;
- The systematic documentation and modeling of crosscutting concerns, which is described in Chapter 4;
- The refactoring of crosscutting concerns to aspect-oriented programming, which is covered in Chapter 6.

Each of the chapters in this thesis is directly based on at least one peer reviewed publication. While this results in some duplication, it also ensures that the various chapters can be read independently.

Most of the publications have been co-authored with Arie van Deursen and Leon Moonen; The publications of Chapter 3 have been co-authored with Mariano Ceccato, Kim Mens, Leon Moonen, Paolo Tonella, and Tom Tourwé. The following list gives an overview of these publications:

Chapter 2 This chapter has been accepted for publication in the Transactions on Software Engineering and Methodology (TOSEM) in January, 2007 [Marin et al., 2007a]. An earlier version of the chapter appeared in proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE), 2004 [Marin et al., 2004].

Chapter 3 This chapter is published in the Software Quality Journal (SQJ), 2006 [Ceccato et al., 2006]. An earlier version of the chapter appeared in proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC/ICPC), 2005 [Ceccato et al., 2005].

Chapter 4 This chapter integrates several publications from ACM Software Engineering Notes (proceedings of the International Workshop on the Modeling and Analysis of Concerns in Software), 2005 [Marin et al., 2005c], the proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM), 2005 [Marin et al., 2005a], the proceedings of the 14th IEEE Working Conference on Reverse Engineering (WCRE), 2007 [Marin et al., 2007b], and the proceedings of the 29th International Conference on Software Engineering (ICSE), 2007 [Marin et al., 2007d].

Chapter 5 The chapter is an extension and integration of two publications in the proceedings of the 13th IEEE Working Conference on Reverse Engineering (WCRE), 2006 [Marin et al., 2006a,b].

Chapter 6 This chapter will appear in the proceedings of the 7th IEEE International Conference on Source Code Analysis and Manipulation (SCAM), 2007 [Marin et al., 2007c].

Chapter 2

Identifying Crosscutting Concerns using Fan-in Analysis

Aspect mining is a reverse engineering process that aims at finding crosscutting concerns in existing systems. This chapter proposes an aspect mining approach based on determining methods that are called from many different places, and hence have a high fan-in, which can be seen as a symptom of crosscutting functionality. The approach is semi-automatic, and consists of three steps: metric calculation, method filtering, and call site analysis. Carrying out these steps is an interactive process supported by an Eclipse plug-in called FINT. Fan-in analysis has been applied to three open source Java systems, totaling around 200,000 lines of code. The most interesting concerns identified are discussed in detail, which includes several concerns not previously discussed in the literature on crosscutting concerns. The results show that a significant number of crosscutting concerns can be recognized using fan-in analysis, and each of the three steps can be supported by tools.

2.1 Introduction

Aspect-oriented software development (AOSD) is a programming paradigm that addresses *crosscutting concerns*: features of a software system that are hard to isolate, and whose implementation is spread across many different modules. Well-known examples include logging, persistence, and error handling. Aspect-oriented programming captures such crosscutting behavior in a new modularization unit, the *aspect*, and offers code generation facilities to *weave* aspect code into the rest of the system at the appropriate places.

Aspect mining is an upcoming research direction aimed at finding crosscutting concerns in existing, non-aspect-oriented code. Once these concerns have been identified, they can be used for program understanding or refactoring purposes, for example by integrating aspect mining techniques into the software development tool suite. In addition to that, aspect mining research increases our understanding of crosscutting

concerns: it forces us to think about under what circumstances a concern should be implemented as an aspect, it helps us find crosscutting concerns that are beyond the canonical ones such as logging and error handling, and it may lead to concerns that are crosscutting, yet not easily modularized with current aspect technology (such as, e.g., ASPECTJ).

In this chapter we propose *fan-in analysis*, an aspect mining approach that involves looking for methods that are called from many different call sites and whose functionality is needed across different methods, potentially spread over many classes and packages. Our approach aims at finding such methods by computing the fan-in metric for each method using the system's static call graph. It relies on the observation that scattered, crosscutting functionality is likely to generate high fan-in values for key methods implementing this functionality. Furthermore, it is consistent with the guidelines of applying aspect solutions when the same functionality is required in many places throughout the code [Colyer et al., 2005].

Fan-in analysis is a semi-automated process consisting of three steps. First, we identify the methods with the highest fan-in values. Second, we filter out methods that may have a high fan-in but for which it is unlikely that there is a systematic pattern in their usage that could be exploited in an aspect solution. Typical examples are getters and setters, as well as utility methods. Third, we inspect the call sites of the high fan-in methods, in order to determine if the method in question does indeed implement crosscutting functionality. This step is the most labor intensive, and it is based on an analysis of recurring patterns in, for example, the call sites of the high fan-in method. All steps are supported by an Eclipse¹ plug-in called FINT, which is also discussed in the chapter.

We discuss the application of fan-in analysis to three existing open source systems (the web shop PETSTORE, the drawing application JHOTDRAW, and the servlet container TOMCAT) implemented in Java. For all systems our approach found a number of interesting crosscutting concerns that could benefit from an aspect-oriented redesign.

When evaluating the quality of an aspect mining technique, two challenges have to be faced. The first is that a benchmark system must exist in which the crosscutting concerns are known already, for example because they have been identified by an expert. At the moment, such a benchmark does not exist. A growing number of aspect mining researchers, however, are using JHOTDRAW as their case study, which is thus evolving into such a benchmark system.

The second evaluation challenge is that the decision that a concern is crosscutting and amenable to an aspect-oriented implementation is a design choice, which is a trade-off between alternatives. Thus, there is not a yes/no answer to the question whether a concern identified is suitable for an aspect implementation. As a consequence, quantitative data on the number of false negatives (how many crosscutting concerns are missed) or false positives (how many of the concerns we identified are in fact not crosscutting) has a subjective element to it. This means that an evaluation of

¹ www.eclipse.org

an aspect mining technique just in terms of, for example percentages of false positives and negatives, or in terms of precision and recall, is an oversimplification.

To deal with these issues, we decided to discuss a substantial number of concerns found in considerable detail, explaining for what reasons they should be considered as crosscutting concerns. In order to encourage a debate on our results, we selected open source systems on purpose, allowing others to see all code details when desired.

As a result, the chapter can be read in two ways. First of all, it is the presentation of the fan-in aspect mining technique. Second, it is a discussion of those crosscutting concerns that were found in three open source systems by means of fan-in analysis – thus establishing a first step towards a common benchmark that can be used in further aspect mining research.

The scope of the present chapter is aspect mining itself. Using the aspect mining results, for example for refactoring to ASPECTJ, is a separate topic, for which we refer to, e.g., Binkley et al. [2005], as well as to our own work on reimplementing concerns discussed in this chapter, described in Chapter 6 of the thesis.

This chapter is organized as follows. We start out by surveying existing work in the area of aspect mining. Then, in Section 2.3, we present the fan-in metric, the analysis steps, as well as the Eclipse plug-in supporting fan-in analysis. In Section 2.4 we present an overview of the case studies. In Sections 2.5–2.7 we cover the results obtained from applying fan-in analysis to three open source case-studies presenting several of the concerns found in considerable detail. We reflect on these case studies, on the reasons for success, and on the limitations of our approach in Section 2.8. We conclude with a summary of the chapter’s key contributions and opportunities for future work.

We assume the reader has basic knowledge of aspect-oriented programming, and we refer to Gradecki and Lesiecki [2003], The AspectJ Team [2003], and Laddad [2003b] for more information.

2.2 Aspect Mining: Background and Related Work

Since aspect mining is a relatively recent research area, we start out by providing some uniform terminology. We then discuss the most important aspect mining approaches published to date.

2.2.1 Terminology

Sutton and Rouvellou [2005] provide a discussion on what constitutes a “concern”. Following them, we take concern generally to be “any matter of interest in a software system.” Concerns can live at any level, ranging from requirements, to use cases, to patterns and contracts. In this chapter we will focus on concerns that play a role at the source code level.

We distinguish between a concern’s *intent* and *extent*:

- A concern’s *intent* is defined as the objective of the concern. For example, the intent of a tracing concern is that all relevant input and output parameters of public methods are appropriately traced.
- A concern’s *extent* is the concrete representation of that concern in the system’s source code. For example, the extent of the tracing concern consists of the collection of all statements actually generating traces for a given method parameter.

In aspect mining, we search for source code elements that belong to the extent of concerns that *crosscut* the software system’s modularization structure. Such *crosscutting concerns* are not dedicated to a modularization unit like a single package, class hierarchy, class, method, but are *scattered* over all these units. As an example, the tracing concern will affect many different methods distributed over different packages or classes. A consequence of this scattering is *tangling*: modular units cannot deal exclusively with their core concern, but have to take into account the implementation of other concerns that crosscut their modularization as well.

Aspect-oriented software development aims at avoiding the maintenance problems caused by scattering and tangling by making use of the new aspect modularization construct. As a simple example, consider an implementation of the tracing concern in ASPECTJ², as shown in Figure 2.1. The *declare* statement at the top of the aspect body ensures that all classes contained in a particular package extend the *Traceable* interface, using a so-called inter-type declaration. The *Traceable* interface itself is provided in the subsequent lines, including a default implementation of the interface. In this way, the aspect extends multiple classes, thereby capturing the statically crosscutting nature of tracing. The remainder of the aspect captures the dynamic crosscutting, using a “pointcut” which intercepts all calls to public methods, and “around advice” that emits a string with the signature of the executing method just before and just after its execution. The aspect can be woven into the base code, keeping the latter *oblivious* to the tracing concern. This helps to reduce the tangling in the base code and provides a non-scattered implementation of the crosscutting concern. Furthermore, a (small) reduction in code size can be achieved if the crosscutting is sufficiently regular (as is the case with the tracing concern: the pointcut expression can quantify over all public methods).

Aspect mining aims at finding crosscutting concerns in existing, non-aspect-oriented code. Such concerns could possibly be improved by applying aspect-oriented solutions or can be documented for program comprehension purposes. The mining involves the search for source code elements belonging to the implementation of a crosscutting concern, i.e., which are part of the concern’s extent. We will refer to such code elements as *seeds*. Once we have found a single seed for a concern, we can try to expand the seed to the full extent of the concern, for example by following data or control flow dependencies.

²www.aspectj.org

```

package myaspects;
public aspect Tracing {

    declare parents: mypackage.* implements Traceable ;

    public interface Traceable {
        public void traceEntry(String methodSig);
        public void traceExit(String methodSig);
    }

    public void Traceable.traceEntry(String methodSig) {
        System.out.println("Entering_" + methodSig);
    }

    public void Traceable.traceExit(String methodSig) {
        System.out.println("Exiting_" + methodSig);
    }

    pointcut thePublicMethods(Traceable t) :
        target(t) &&
        execution(public * mypackage..*(..)) &&
        !within(Tracing );

    Object around(Traceable t): thePublicMethods(t) {
        t.traceEntry(thisJoinPoint.getSignature().toString());
        Object result = proceed(t);
        t.traceExit(thisJoinPoint.getSignature().toString());
        return result;
    }
}

```

Figure 2.1: ASPECTJ definition for the tracing concern

Aspect mining generally requires human involvement. Therefore, we will say that aspect mining tools yield *candidate seeds*, which can be turned into *confirmed seeds* (or simply “seeds”) if accepted by a human expert, or *non-seeds* if rejected. Sometimes a non-seed is also referred to as a *false positive* – a *false negative* then is a part of a known crosscutting concern, potentially detectable by the technique, but missed due to inherent limitations of the approach or due to the specific filters applied in it. The key aspect mining challenge is to keep the percentage of confirmed seeds in the total set of candidate seeds as high as possible, without increasing the number of false negatives too much. As we will see, with fan-in analysis this percentage is above 50%.

The origins of aspect mining can be traced back to the concept assignment problem, i.e., the problem of discovering domain concepts and assigning them to their realizations within a specific program [Biggerstaff et al., 1994]. Work on this problem has resulted in such research areas as feature location [Koschke and Quante, 2005;

Wilde and Scully, 1995; Xie et al., 2006], design pattern mining [Ferenc et al., 2005], and program plan recognition [Rich and Wills, 1990; Wills, 1990; van Deursen et al., 2000].

In aspect mining we specifically search for concerns (concepts) whose realization in a given program cuts across modular units. Several aspect mining approaches have been published, for which we propose a distinction between *query-based* and *generative* approaches. *Query-based* approaches start from manual input such as a textual pattern. *Generative* approaches, including fan-in analysis, aim at generating seeds automatically making use of, for example, structural information obtained from the source code. Below we discuss these two categories of aspect mining approaches. Moreover, we discuss techniques that are most closely related to our fan-in analysis.

2.2.2 Query-Based Approaches

Query-based, explorative techniques rely on search patterns provided by the user. Source code locations that match the pattern correspond to crosscutting concern seeds, which can subsequently be expanded to more complete concerns using a tool.

One of the first query-based tools, the Aspect Browser, uses lexical pattern matching for querying the code, and a map metaphor for visualizing the results [Griswold et al., 2001]. The Aspect Mining Tool AMT extends the lexical search from the Aspect Browser with structural search for usage of types within a given piece of code [Hannemann and Kiczales, 2001]. Both tools display the query results in a Seesoft-type view as highlighted strips in enclosed regions representing modules (e.g., compilation units) of the system [Eick et al., 1992].

AMTEX is an AMT extension that provides support for quantifying the characterization of particular aspects [Zhang and Jacobsen, 2003]. AMTEX, in turn, has evolved into PRISM, a tool supporting identification activities by means of lexical and type-based patterns called *fingerprints* [Zhang and Jacobsen, 2004]. A fingerprint can be defined, for example, as any method in a given class of which the name starts with a given word. A software engineer defining fingerprints is assisted by so-called *advisors*. PRISM currently provides a ranking advisor which reports the most frequently-used types across methods. This idea is akin to fan-in analysis, which reports the most frequently used methods across a system. There are, however, no reports about the successfulness of applying the approach implemented in PRISM to the identification of crosscutting concerns.

The Feature Exploration and Analysis Tool FEAT is an Eclipse plug-in aimed at locating, describing, and analyzing concerns in source code [Robillard and Murphy, 2007]. It is based on *concern graphs* which represent the elements of a concern and their relationships. A FEAT session starts with an element known to be a concern seed, and FEAT allows the user to query relations, such as direct call relations, between the seed and other elements in the program. The results of the query that are considered relevant by the user to the implementation of a (crosscutting) concern can be added to the graph-based representation of the concern.

The Concern Manipulation Environment CME aims at providing support across the whole lifecycle of an aspect-oriented development project [Harrison et al., 2004]. This support also includes aspect identification facilities through an integrated search component (Puma) that uses an extensible query language (Panther) [Tarr et al., 2004]. The Panther language includes the static part of the AspectJ pointcut language. CME also allows for concern management similar to FEAT. Most importantly, CME provides a possible infrastructure for the integration of different approaches to aspect mining, including seed identification and concern exploration and management.

Various query-based tools (the Aspect Browser, AMT, and FEAT) have been compared in a recent study [Murphy et al., 2005]. This study shows that the queries and patterns are mostly derived from application knowledge, code reading, words from task descriptions, or names of files. As the study shows, prior knowledge of the system or known starting points strongly affect the usefulness of the outcomes of the analysis.

2.2.3 Generative Approaches

The second group of aspect mining approaches aim at automatically generating cross-cutting concern seeds with a good quality: seeds that will reduce the effort of further understanding and exploring the concern. The approaches in this category can be described as *generative* techniques and will typically provide the input for the explorative approaches.

Many generative approaches use program analysis techniques to look for symptoms of code scattering and tangling and identify code elements exhibiting these symptoms that can act as candidate aspect seeds.

Shepherd et al. [2004] use clone detection based on program dependence graphs and the comparison of individual statement's abstract syntax trees for mining aspects in Java source code.

Three clone detection tools, implementing matching on tokens, abstract syntax trees, and on program dependence graphs, respectively, are evaluated by Bruntink et al. [2005] on an industrial C component. The starting point were four dedicated crosscutting concerns that were manually identified and annotated in the code beforehand. The evaluation assesses the suitability of clone detection for identifying these concerns automatically by measuring the coverage of the annotated concerns by detected clones.

Code clones in object-oriented systems would typically be refactored through method extraction [Fowler et al., 1999] which results in scattered calls to the extracted method [Laddad, 2003a]. Fan-in analysis looks for the concerns implemented by these scattered calls, which could be further refactored into aspect advice.

Dynamic analysis has been considered for aspect identification by examining execution traces for recurring execution patterns [Breu and Krinke, 2004] and by applying formal concept analysis to associate method executions to traces specific to documentation-derived use-case scenarios [Tonella and Ceccato, 2004a]. Particularly challenging for dynamic analysis techniques is to exercise all functionality in the system that could lead to aspect candidates. This implies that a preliminary activity is

needed in which use-case scenarios are defined for the system under investigation. Fan-in analysis does not require such a preliminary activity.

The first of the two dynamic techniques has been adapted recently to static analysis to search for recurring execution patterns in control flow graphs [Krinke, 2006]. The technique is similar in some respect to fan-in analysis, which searches for recurrent call relations. The experimental results of the technique are discussed by comparison with our own results reported for one of the analyzed systems, and show many common findings.

Formal concept analysis has also been applied in an identifier analysis that groups programming elements based on their names [Tourwé and Mens, 2004]. This analysis starts from the assumption that naming conventions can be used to relate the scattered elements of a concern. Although fan-in analysis could use naming conventions for the investigation of the automatically generated results, its primary functionality relies on structural relationships.

The suitability of refactoring certain interfaces implemented by a class has been investigated through a number of indicators like the naming pattern used by the interface definition, the coupling between the methods of the implementing class and the methods declared by the interface, or the package location of the interface and its implementing class [Tonella and Ceccato, 2004b]. By comparison with fan-in analysis which focuses on method seeds, this technique is directly targeting interface definitions for seed identification.

Besides our own experiments, assessments of fan-in analysis that we propose have been provided by Gybels and Kellens [2005] who used the metric as an approximate heuristic for measuring scattering. Another assessment of this analysis has been made available through the Timna framework [Shepherd et al., 2005a] which uses machine learning techniques to combine the results of several aspect mining techniques.

In their more recent work, Breu and Zimmermann [2006] search for concerns by analyzing the changes in the values of the fan-in metric between different versions of the system under investigation. The technique they propose examines the version history for insertions of method calls. Similar to fan-in analysis, a reported seed consists of a set of one or more methods with same call site locations. This technique could complement fan-in analysis by giving insight into the evolution of the metric's values in a system, and hence into the evolution of the concern of a method.

2.2.4 Aspect Identification Case Studies

The subject systems that we have analyzed in the previous [Marin et al., 2004] and present work have also been used by related research [Shepherd et al., 2005a, 2004; Janzen and Volder, 2003; Binkley et al., 2005] or in tool demonstrations (e.g., FEAT [Robillard and Murphy, 2007]). However, our work on fan-in analysis is the first attempt to establish a common benchmark for the development of aspect mining techniques, by explicitly reporting the results obtained for a number of case-studies and discussing them in significant detail. This work has been continued in a comparative

study [Ceccato et al., 2006] of the fan-in technique with the dynamic [Tonella and Ceccato, 2004a] and identifier analysis [Tourwé and Mens, 2004] approaches. The JHOTDRAW case-study targeted by the comparison experiment is intended to become the de-facto benchmark for aspect mining.

2.3 Aspect Mining Using Fan-in Analysis

Fan-in analysis fits in the category of generative aspect mining approaches. The main symptom of crosscuttingness it tries to capture is *scattering*: the code for one concern is spread across the system. If the scattered pieces of code have functionality in common, it is likely that this will have been factored out in helper methods. These methods are then called from many places, giving them a high fan-in value. In an aspect-oriented re-implementation of such concerns, the method would constitute (part of) the advice, and the call site would correspond to the context that needs to be captured using a pointcut.

Fan-in analysis consists of three steps:

1. Computation of the fan-in metric for all methods;
2. Filtering of the set of methods to obtain the methods that are most likely to implement crosscutting behavior;
3. Analysis of the remaining methods to determine which of them are part of the implementation of a crosscutting concern.

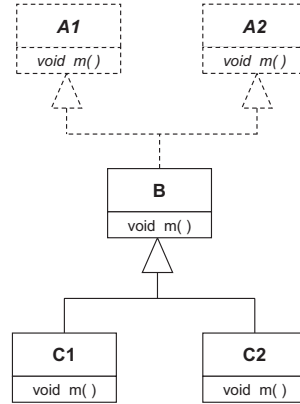
The next subsections describe each of these steps, as well as the tool FINT supporting these steps.

2.3.1 A Fan-in Metric for Aspect Mining

The metric we will use for aspect mining is based on method fan-in, which is a “measure of the number of methods that call some other method” [Sommerville, 2004]. Thus, we will collect the set of (potential) callers for each method — and the cardinality of this set gives the required fan-in value. The actual value, however, of method fan-in depends on the way we take polymorphic methods (callers as well as callees) into account.

Therefore, our first refinement is that we count the number of *different method bodies* that call some other method. Thus, if a single abstract method is implemented in two concrete subclasses, we treat these two implementations as separate callers.

Our second refinement deals with calls to polymorphic methods. Recall that we are interested in methods that are called from many different places, since these are potentially part of a crosscutting concern. If we find that a particular method *m* belongs to such a concern, it is very likely that superclass declarations or subclass overrides of



(a) Example Class Hierarchy

Call site	Fan-in contribution				
	A1.m	A2.m	B.m	C1.m	C2.m
f1(A1 a1) { a1.m(); }	1	0	1	1	1
f2(A2 a2) { a2.m(); }	0	1	1	1	1
f3(B b) { b.m(); }	1	1	1	1	1
f4(C1 c1) { c1.m(); }	1	1	1	1	0
f5(C2 c2) { c2.m(); }	1	1	1	0	1
Total fan-in	4	4	5	4	4

(b) Corresponding Fan-in Values

Figure 2.2: Example class hierarchy and corresponding fan-in values

m belong to that same concern. For that reason, if we see that method m' applies method m to an object of static type C , we add m' to the set of (potential) callers for each m declared in any sub- or superclass of C .

With this definition, (abstract) method declarations high in the inheritance hierarchy act as fan-in accumulators: whenever a specific subclass implementation is explicitly invoked, the fan-in of not only the specific but also of the abstract method is increased. In this way, if there are many calls to different specific implementations, we get a high fan-in value for the superclass method. An aspect-oriented reimplementa-tion would aim at capturing the many specific call sites into a pointcut, and invoke the abstract method in the advice, relying on polymorphism to dispatch to the proper specific implementation.

An example hierarchy is shown in Figure 2.2. The example illustrates the effects of various calls to a polymorphic method m in different positions in the class hierarchy. Note that, given our definition, the fan-in for method m in class $C1$ is not affected by calls to m defined in $C2$ and vice versa: the same holds for sibling classes $A1$ and $A2$.

Our last refinement is concerned with super calls. For super calls, we explicitly know which method is targeted, which therefore is the only method whose call set is extended.

Observe that there are multiple ways in which a fan-in metric can be defined. Historically, the notion of fan-in was introduced by Henry and Kafura [1981] as an indicator for coupling in procedural software. They include data access in fan-in as well, which we do not. An overview of coupling indicators for object-oriented systems is discussed by Briand et al. [1999]. In some cases these metrics are based on a derivative of the fan-in metric, which then often is taken at the class level (instead of the method fan-in we use) – see, e.g., Henderson-Sellers et al. [1996]. In other cases calls from private methods are excluded from the fan-in count.

2.3.2 Method Filtering

After having computed the fan-in values of all methods, we apply the following filters, in order to obtain a smaller set of methods with a higher chance of implementing crosscutting behavior.

First, we restrict the set of methods to those having a fan-in above a certain threshold. This can be an absolute fan-in value (say, 10) or a relative percentage (say, the top 5% of all methods ordered by their fan-in values). Note that an absolute value threshold not only acts as a filter, but also an indicator for the severity of the scattering.

In our case studies, we experimented with several values, and found 10 to be a useful trade-off between the number of concerns that one can find and the number of methods that need to be inspected.

Second, we filter getters and setters from the list of methods. This is either based on naming conventions (methods matching the “get*” or “set*” pattern) or on an analysis of the method’s implementation.

Last but not least, we filter utility methods, like `toString()`, classes such as *XMLDocumentUtils* containing “util” in their name, collection manipulation methods, and so on, from the remaining set. This is a manual step that may require some familiarity with the subject system. This familiarity can be improved after each iteration by looking at the results and analyzing apparent indicators like names or easily accessible documentation, such as descriptive comments in the code. The heuristics we used for identifying utility methods in our case studies are based on the following categories:

- Methods that belong to collection classes and/or packages. The JHOTDRAW case study, for example, comes with its own library for collection classes. We typically recognized these based on class or package names, such as *FigureEnumerator*, *HandleEnumerator*, *ListWrapper*, and so on.
- Documented utilities, based on naming and easily available documentation criteria. For example, for PETSTORE, the utility methods belong to two classes: *XMLDocumentUtils* and *PopulateUtils*, which creates and prints SQL statements

used to populate the sample database for the application. In TOMCAT, we marked classes from the *util.buf* package as utility, which deals with encoding and decoding buffers. We also marked the *util.digester.Digester* class as utility - the class is described as an XML parser in Tomcat's documentation.

2.3.3 Seed Analysis

Our final step is to conduct a manual analysis of the remaining set of methods. This analysis follows a number of guidelines, part of which benefit from automatic support. Reasoning about the reported candidates can take a top-down or bottom-up approach.

In the bottom-up approach we look for consistent invocations of the method with a high fan-in value from call sites that could be captured by a pointcut definition. Examples of such consistent invocations include:

- The calls always occur at the beginning or the end of a method;
- The calls occur in methods that are all refinements of a single abstract method, as, for instance, for contracts exercised across class hierarchies;
- The calls occur in methods with similar names, like handlers for mouse or key events;
- All calls occur in methods implementing a certain role, as, for example, listener-objects that register themselves as observers of a subject-object state.

The regularity of these call sites typically will make it possible to capture the calls in a pointcut mechanism, and the high fan-in method into advice. The main challenge of the bottom-up approach is to recognize these patterns leading to pointcuts. As we will see in the next section, it is possible to offer tool support here that helps the human engineer in conducting this analysis.

In the top-down approach, we take domain knowledge or knowledge of typical crosscutting concerns into account, as described by, e.g., Hannemann and Kiczales [2002] or Laddad [2003b]. For example, a number of design patterns define (crosscutting) roles and methods specific to these roles that can appear in the list of seeds. The human engineer can take advantage of such knowledge when analyzing the candidate seeds to recognize the pattern-specific roles. The Composite pattern, for example, defines roles and methods to allow parent-objects to refer and manipulate child-elements. Similarly, the methods in a decorator class are characterized by the consistent redirection functionality they implement.

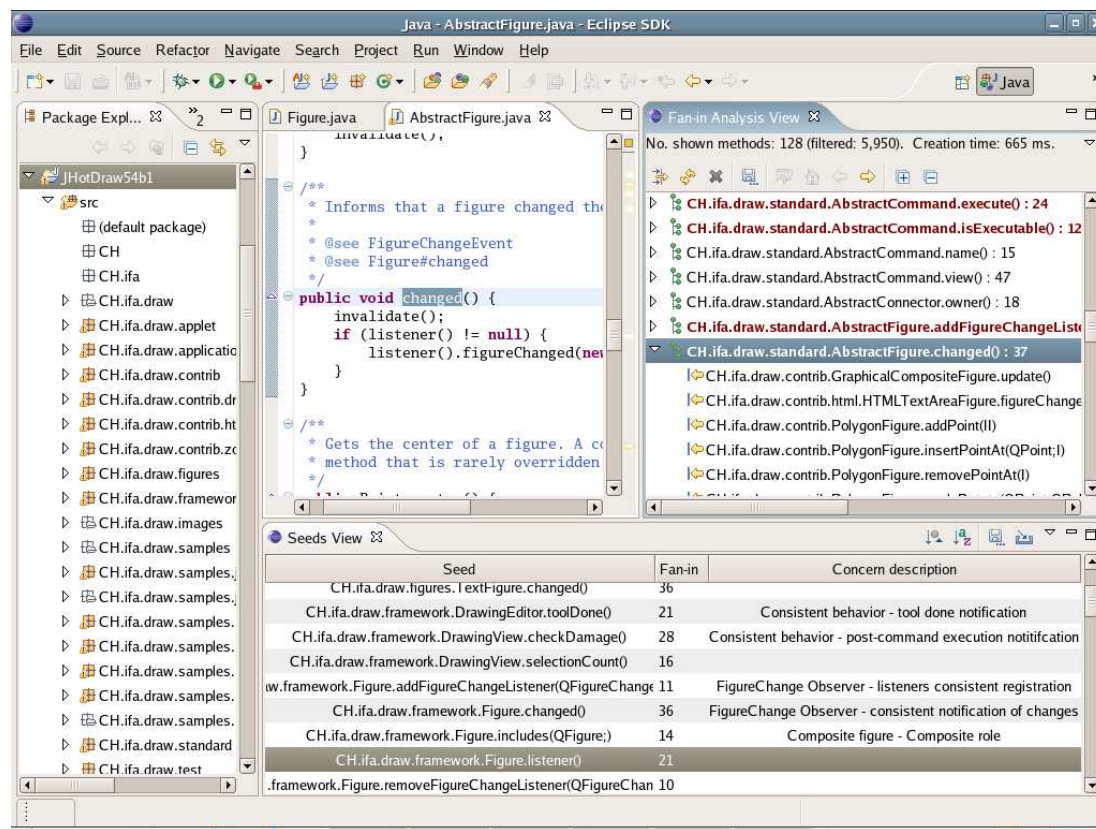


Figure 2.3: FINT in action, showing the *Fan-in Analysis View* (top right) and the *Seeds View* (bottom right).

2.3.4 The Fan-in Tool FINT

The Fan-in Tool FINT³ is an Eclipse plug-in that provides automatic support for the metric computation, method filtering, and candidate analysis steps of fan-in analysis.

To compute the fan-in metric, the tool first builds the abstract syntax tree for the user-selected sources, and then creates a call graph with the methods declared in the selected sources and their callees. The fan-in metric is derived from this graph, as described in Section 2.3.1. The results are displayed in the *Fan-in Analysis view*, shown in Figure 2.3, together with the list of callers for each method. The results can be ordered alphabetically or by their fan-in value. Optionally, the results can also be stored on file.

The same view is used for the filtering step of fan-in analysis. Thus, the user can indicate an absolute threshold for the fan-in value. Furthermore, the user can choose to filter out accessor methods by their signature based on the “get*” or “set*” naming convention, or based on their implementation.

³ <http://swierl.tudelft.nl/bin/view/AMR/FINT>. The features discussed in this chapter are part of FINT 0.6.

In addition to that, the user can indicate groups of elements whose methods are to be excluded from the callee or caller sets. Excluded callees are indicated as utility-methods and represent methods considered irrelevant for analysis. Similarly, the user-selected callers will not contribute to the fan-in metric of the analyzed methods. Both filters can be applied, for instance, to (JUnit) tests, which are neither relevant as candidate-seeds nor as callers. The user marks these elements in a browser window, which displays the Java elements in the hierarchy of the analyzed elements, similar to Eclipse's *Package Explorer* view. The user can select a check-box for the enclosing package, file, or declaring class of the method to be filtered.

Methods not declared in the analyzed sources, but called by analyzed methods are considered *libraries* and can optionally be included in the analysis. These methods cannot contribute to the fan-in metric of a method.

The *Fan-in Analysis View* is also the starting point for the last analysis step. From this view, the engineer can inspect the reported methods and their callers. Methods can be marked as seeds and added to the *Seeds View*, shown at the bottom of Figure 2.3. In this view, the seeds can be documented with a concern description, saved to a file or loaded from a previous analysis.

The analysis and seed views from FINT support the user in recognizing recurring patterns and similarities as discussed in the previous section, helping him or her in deciding whether one or more high fan-in methods belong to a crosscutting concern.

The various ways in which methods and call sites can be sorted and inspected in FINT help to discover such patterns. Furthermore, the tool provides automatic support for detecting some of the possible relations between the callers of an analyzed method, like grouping of the callers by common hierarchies or their declaring interfaces, by the position of the analyzed call, or by other callees shared by the callers.

As an example, Figure 2.4 shows the view for analyzing the callers of a method with a high fan-in value by investigating their declaring interfaces. The callers declared by the same interface are shown in a same, distinctive color. Such analysis is helpful, for example, in identification of (crosscutting) responsibilities that are to be fulfilled by a number of classes.

The same figure also shows a relational table for the callers of the method with the high fan-in value and the relative position of the call in the body of the caller. This analysis investigates whether the call occurs on the first, second, first before last, or last position. These positions would typically indicate a before or after advice as a natural aspect-refactoring solution for the candidate seed and its set of callers.

2.4 The Case Studies

We have applied fan-in analysis to several case studies, three of which we describe in detail in this chapter. All cases are open source systems, allowing validation of our results by others. The PETSTORE and JHOTDRAW systems are demonstration applications of J2EE technologies and design patterns, respectively. TOMCAT is the

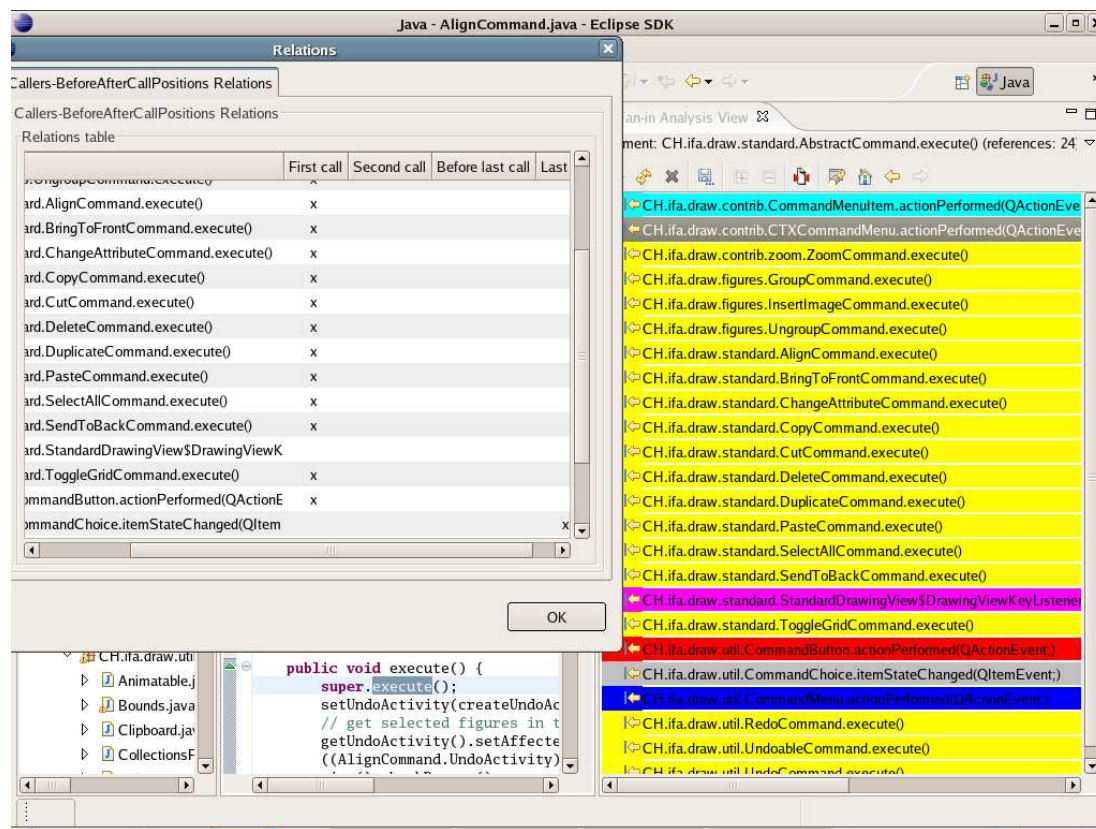


Figure 2.4: Seed inspection using FINT. The color codings in the right window indicate inheritance from common interfaces; the table at the left marks the positions of calls to a high fan-in method.

largest system, and one that is widely used in web servers all over the world.

Before going into detail in the case studies, we first discuss a number of general observations, and explain in what format we will present the three case studies.

2.4.1 First Findings

Key statistics for our case studies are provided in Table 2.1. A first observation that can be made from this table is that filtering methods above the threshold of 10 reduces the number of methods to be inspected to 1, 6, and 3 percent for PETSTORE, JHOTDRAW and TOMCAT, respectively. Figure 2.5 shows the fan-in distribution for the three case-studies. As can be seen, the vast majority of methods have a very low fan-in. The large percentage of methods with a fan-in value of 0 can be explained by the nature of the applications. PetStore, for instance, is a J2EE application and a number of calls are not explicit in the code but made by the container (EJB-specific methods).

A second observation that can be made from Table 2.1 is that the accessor and utility filters eliminate about half of the high fan-in methods. Note that the utility

	PETSTORE	JHOTDRAW	TOMCAT
size in non-comment lines of code	17,032	20,594	149,219
number of methods	1,917	3,230	13,489
methods with fan-in ≥ 10	16 (1%)	205 (6%)	424 (3%)
Statistics for methods with fan-in ≥ 10			
utility methods	3	20	16
accessors	5	71	181
confirmed seeds	7 (87%)	58 (51%)	164 (73%)
non-seeds	1 (13%)	56 (49%)	63 (27%)
concerns	5	10	10

Table 2.1: Key statistics of our case studies

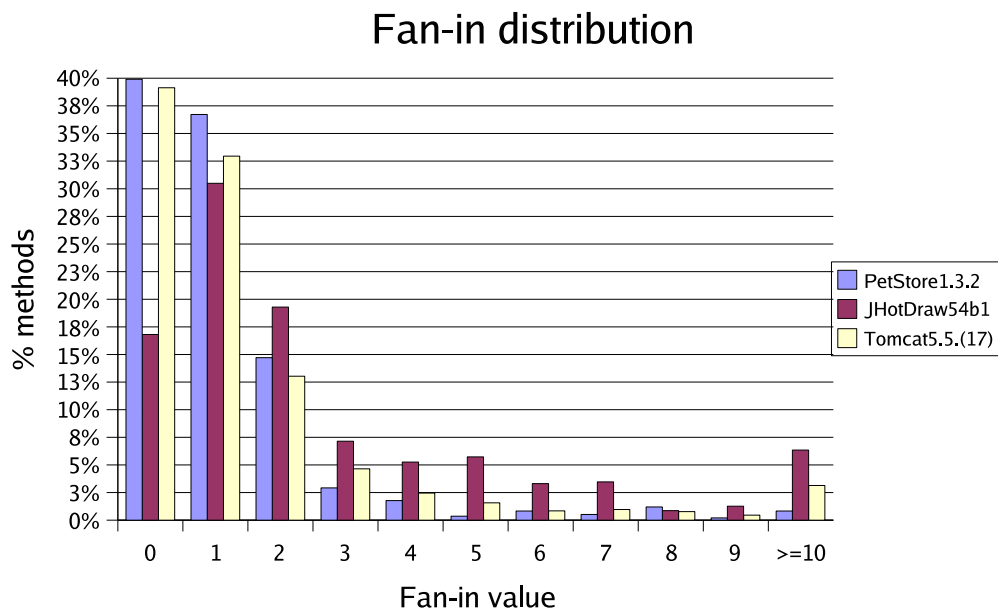


Figure 2.5: Fan-in distribution for the three case studies.

methods filtered out here are the ones that are part of the system under study. Utility methods in external libraries are not taken into account in the first place, and do not occur in the table. If necessary, the scope of the system under study can be extended to include certain libraries as well. This is a decision that requires a certain amount of domain knowledge, for example that a particular library is used for addressing a known crosscutting concern (we will encounter such a situation for the *logging* concern in the TOMCAT case study in Section 2.7).

The methods of the system under study that are not filtered out will give the set to be analyzed in a last, tool-assisted step. This should result in a classification as either a seed for a crosscutting concern, or as a non-seed. Our third observation from Table 2.1 is that for all cases, a significant percentage (87%, 51%, and 73% for the three cases) of the methods that need to be inspected manually turn out to be confirmed seeds. Thus, while this step may be more labor-intensive, it does give a good chance of finding crosscutting concern seeds.

A final observation is that there are many more seeds than concerns. This is due to two reasons. First, there may be multiple concern instances for one sort of concern. For example, JHOTDRAW makes use of more than one Observer. Second, a single concern is often identified through multiple seeds. For example, for the Observer design pattern, we may not only find a high fan-in for the notification method, but also for the methods for attaching different observers to a subject.

2.4.2 Case Study Presentation

In the next sections we discuss the PETSTORE, JHOTDRAW, and TOMCAT case studies. We particularly focus on the third step, in which seeds are either confirmed or rejected as belonging to a crosscutting concern, since this step implies various considerations, inherent in the mining process, about the classification of a candidate seed. For each case study, we discuss several of the concerns found in considerable detail, explaining why we think that they are crosscutting, and analyzing to what extent these concerns are amenable to an aspect-oriented re-implementation. A full list of all high fan-in methods and the concerns they belong to are publicly available on our web site⁴. The site furthermore describes which methods exactly were marked as utilities, thus making our experiments fully reproducible.

In order to give an impression of the limitations (and hence opportunities for improvement) of fan-in analysis, the next sections also discuss some of the false positives (rejected candidate seeds) and some of the concerns that are known from the literature or from related studies that our analysis missed (false negatives). Note that while we can compute the percentage of false positives (the number of non-seeds divided by the total number of seeds), we cannot determine the percentage of false negatives. This would require a common benchmark that documents all the crosscutting concerns exhibiting the symptoms (code scattering) targeted by fan-in analysis. At the time of

⁴ <http://swerl.tudelft.nl/bin/view/AMR/FanInAnalysisResults>

Method	Fan-in	Concern
XMLDocumentException(String)	27	Contract enforcement
ServiceLocatorException(Exception)	22	Exception wrapping
CatalogDAO SysException(String)	19	Exception wrapping
PopulateException(String, Exception)	11	Exception wrapping
TransitionException(Exception)	15	Exception wrapping
XMLDocumentException(Exception)	23	Exception wrapping and tracing for debugging
ejb.ServiceLocator()	30	Service locator
XMLDBHandler()	10	<i>False positive</i>

Table 2.2: PETSTORE high fan-in methods and concerns

writing, no such benchmark exists.

2.5 PETSTORE

The first case study we discuss is PETSTORE. This is a sample J2EE e-business application developed by SUN.⁵ It is a demonstration of a Web application allowing customers to purchase via a web browser. In addition, it includes modules to perform administration tasks like sales statistics, orders and shipping management, etc. PETSTORE is an application demonstrating the proper use of most of the J2EE concepts, and can be considered a well-designed system.

An overview of the methods with a fan-in of 10 and higher, their fan-in value, and the concerns they represent is given in Table 2.2. In this chapter we explain why these concerns are indeed crosscutting. Details on their refactoring towards ASPECTJ are presented by Mesbah and van Deursen [2005].

Service Locators The method with the highest fan-in value (30) belongs to the *ServiceLocator* class from the *ejb* package, which implements the J2EE pattern of the same name [Alur et al., 2003]. The intent of the pattern is to provide a single point of control to clients that need to locate and access a component or service in the business or integration tier. The common solution is to have a single instance of the service locator class for an application or, at least, for a tier and thus to have it implemented as a singleton. The advantages of this solution, however, are not always clear for the EJB-tier and thus the adopted solution can vary [Johnson, 2003].

PETSTORE contains two different service locators: the web-tier one is implemented as a singleton but the fan-in of the method returning the unique instance is only 7; the identified EJB-tier locator is not a singleton and the method reported is the constructor of the class.

⁵ <http://java.sun.com/blueprints/>, PETSTORE version 1.3.2.


```

public class InvoiceTD implements TransitionDelegate {

    /** sets up all the resources that will be needed to do
     *  a transition
     */
    public void setup() throws TransitionException {
        try {
            ServiceLocator sl = new ServiceLocator();
            qFactory = sl.getQueueConnectionFactory(JNDINames. ...);
            q = sl.getQueue(JNDINames. ...);
            queueHelper = new QueueHelper(qFactory, q);
        } catch (ServiceLocatorException se) {
            throw new TransitionException(se);
        }
    }

    /** Send an order approval to the OrderApproval Queue...
     */
    public void doTransition(TransitionInfo info)
        throws TransitionException {
        String xmlCompletedOrder = info.getXMLMessage();
        try {
            queueHelper.sendMessage(xmlCompletedOrder);
        } catch (JMSException je) {
            throw new TransitionException(je);
        }
    }
}

```

Figure 2.6: Error handling in PETSTORE

The service locator defines a consistent lookup mechanism for the dependencies of the various application components, which couples these components to the infrastructure framework and tangles them with the lookup logic.

A possible refactoring for the service locator is the *Dependency Injection* pattern (also called *Inversion of Control*) used in lightweight containers to avoid directly referencing a service locator [Fowler, 2004], a mechanism that resembles the aspect-oriented mechanisms for injection. For Singleton implementations, the aspect refactoring of the pattern [Murali et al., 2004] and the optional caching mechanism [Laddad, 2003b] are in place. The exception wrapping discussed next is also applicable to the Service Locator identified.

Exception Wrapping The majority of the seeds are constructors for PETSTORE exceptions. As an example, Figure 2.6 shows the *TransitionException* case, which is thrown from 15 catch blocks in different classes and packages.

As in the *InvoiceTD* class in the figure, most of the methods throwing the exception implement *doTransition(...)* and *setup()* declared by the *TransitionDelegate*

interface. All the transition delegates handle exceptions related to the particular functionality and re-throw *TransitionException*. This mechanism is common to many J2EE design patterns [Alur et al., 2003], such as *Business Delegate* discussed by Laddad [2003a]. The exception wrapping in *Business Delegate* aims at hiding the implementation details of a business service. The issue hidden in this case is the sort of exception that can be thrown by the actual implementation.

This consistent mechanism is spread over many places, and a refactoring solution is discussed by Laddad [2003a]. Aspects can be used to isolate the exception handling and to wrap the original exception thrown by the underlying implementation in the new exception. This will result in improvements in code size, localization and clarity. Studies of exception handling refactoring [Lippert and Lopes, 2000] show a reduction of catch statements when using AOP of up to 95%. For the case at hand, we found that the classes affected were reduced by 20% [Mesbah and van Deursen, 2005].

Contract Enforcement A method with a fan-in value of 27 is a constructor for the *XMLDocumentException* class. This exception is raised if the structure of the XML document does not comply with the expected structure. By examining the call sites, we were able to observe that 9 of them are from `DOM(Node)` methods, all throwing the exception if a certain compound condition fails. It turns out that all complex conditions share a common check, which can be easily factored out as an aspect by means of before advice – giving rise to the concerns similar to the pre- and post-condition (design by contract) examples discussed by The AspectJ Team [2003].

In this manner, the code will be better localized and new methods will be prevented from omitting the required checks.

Moreover, a set of another 14 call sites are methods of the same class that throw the reported exception if certain conditions do not hold. A sub-set of 11 methods from these callers check the same condition, namely the Boolean value of an input parameter.

Debug Information The *XMLDocumentException* class has a second constructor with a high fan-in. This constructor is (like for the business delegates) used as an exception wrapper. In addition to that, before being wrapped the exception at hand is written on the error output stream. This additional behavior (on top of the wrapping) can be added as another aspect, which indicates which exception should be printed before being wrapped. Turning printing debug information into an aspect helps to ensure a common debugging strategy, and to isolate the concern that is otherwise crosscutting.

False Positives The one case considered as non-aspect in the first set of candidates is an *XMLDBHandler* constructor with a fan-in value of 10. The callers are `setup(...)` methods in classes that populate the associated database tables with data from XML files. The `setup(...)` implementations are only slightly different: they return an instance of an anonymous inner class extending *XMLDBHandler* that is an XML filter. Because all the callers are well localized in a single package and there is only one `populate(...)` method that triggers the whole process at a client's request, we decided to label this candidate as non-crosscutting.

False Negatives As briefly mentioned at the beginning of this section, one of the missed concerns is the service locator in the web-tier, implemented as a singleton, but whose method for accessing the unique instance has a fan-in value of only 7.

A second concern potentially identifiable by fan-in analysis is transaction management. If J2EE's built-in transaction mechanism is used, the concern is well-isolated. PETSTORE, however, also includes explicitly encoded transaction management, which consists of calls to the Java Transaction API (JTA). In principle these can be detected by fan-in analysis, but since they belong to an external library, we normally would not include them in our analysis. Furthermore, the fan-in values for the two methods in the JTA API (the *javax.transaction.** package) used by PETSTORE code have a value (of just 2) well below our threshold.

2.6 JHOTDRAW

JHOTDRAW⁶ is an application framework for two-dimensional graphics. It is an exercise in developing software making use of design patterns [Gamma et al., 1994].

Our filters eliminated around half of the methods with top fan-in values. We were rather cautious not to eliminate too many methods. The only methods designated as “utility” are enumeration manipulators (e.g., `FigureEnumerator.hasNextFigure()/nextFigure()`).

An overview of the concerns found is given in Table 2.3. For each concern, it lists the number of different high fan-in methods that pointed to the concern, and the maximum fan-in value for this concern. In the next sections we discuss these concerns in more detail. Aspect solutions for some of these concerns are available through the open source AJHOTDRAW⁷ project, an ongoing activity to refactor JHOTDRAW to ASPECTJ starting from the results reported in the present chapter.

2.6.1 The Undo Concern

In the top of the list of methods sorted by fan-in, a number of methods point to the undo functionality, such as the `undo` method in *UndoableAdapter*. An undo in a graphical editor is clearly a concern that cuts across many features and activities, although textbooks on aspect-oriented programming, such as Gradecki and Lesiecki [2003], The AspectJ Team [2003], Laddad [2003b], do not discuss using aspects for undo functionality.

A (somewhat simplified) representation of the participating classes in the JHOTDRAW undo implementation is given in Figure 2.7. JHOTDRAW offers various sorts of *activities*, which are contained in a class hierarchy. Examples of concrete activities include handling font sizes, triangle rotation, or image rotation.

⁶ <http://jhotdraw.org/>, version 5.4b1

⁷ <http://ajhotdraw.sourceforge.net/>. AJHOTDRAW is described in more detail in Chapter 6 of this thesis.

Concern	No. of methods	Max fan-in
Adapter	1	37
Command	2	24
Composite	12	24
Consistent behavior	20	31
Contract enforcement	3	31
Decorator	6	57
Exception handling	1	11
Observer	10	37
Persistence	6	22
Undo	3	25

Table 2.3: Concerns found for JHOTDRAW, together with the number of high-fan in methods, and the highest fan-in among those methods.

The interface *Undoable* encapsulates the notion of undoing an action, for which it provides the undo method. Each class implementing a concrete activity that can be undone defines a static nested class conforming to this *Undoable* interface. The nested class knows how to undo the given activity, and has access to all the details of the activity that may be needed for this. Whenever the activity modifies its state, it also updates fields in its associated undo-activity needed to actually perform the undo. In addition to that, a list of *affected figures* is maintained, whose state must be adjusted if the activity is to be undone.

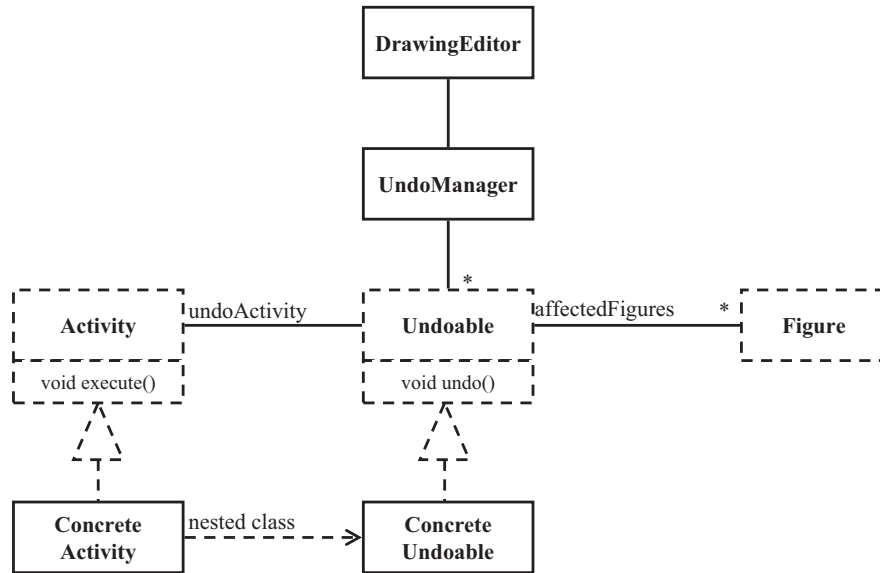
In JHOTDRAW, there are 22 activities that can be undone, causing the undo concern to be spread over these classes. This, in turn, leads to a high fan-in for the methods of, for example, *Undoable*, which helped us to identify this crosscutting concern.

An aspect-oriented solution for the undo concern is presented by Marin [2004]. It consists of a number of steps.

- First, the existing activities are extended with an association to their undoables by means of an inter-type declaration.
- Second, existing operations are extended with functionality to keep track of the old state so that the action can be undone. These existing operations can be captured using a pointcut, and then the updates can be contained in advice code.
- Last but not least, the various nested classes containing the undoable activities can be added by means of inter-type declarations.⁸

Thus, this refactoring captures the undo “protocol” in a pointcut and advice, ensuring that undo functionality is properly invoked whenever commands are executed. Fur-

⁸ The present version of ASPECTJ does not support introducing inner and static nested classes.

Figure 2.7: Participants for *undo* in JHOTDRAW.

thermore, the methods and (inner) classes devoted entirely to undo functionality are moved out of the command classes, and are modularized into an aspect.

2.6.2 Persistence

Another crosscutting concern that pops out clearly through a high fan-in is persistence. The concern was easily spotted, as there are six different methods involved, each having a name built from words like “read”, “write”, “storable”, “input”, and “output”. Storing and restoring figures is performed by methods inherited from the *Storable* interface. This interface offers methods to read one self from a *StorableInput* stream, or write one self to a *StorableOutput* stream.

The implementation of the persistence concern is spread over 36 classes. Figures implementing the *Storable* interface invoke several methods from the *StorableOutput* and *StorableInput* classes. The two classes are specialized in writing/reading various (primitive) types, (e.g., String, Color, int, etc.) to/from a storing device. This results in a high fan-in for their methods, which allowed us to detect the persistence concern using fan-in analysis.

The *Storable* interface can be considered a *secondary* interface, i.e., one that does not define the primary role of the implementing class but only adds supplementary functionality to it. An aspect-oriented implementation for this concern can super-

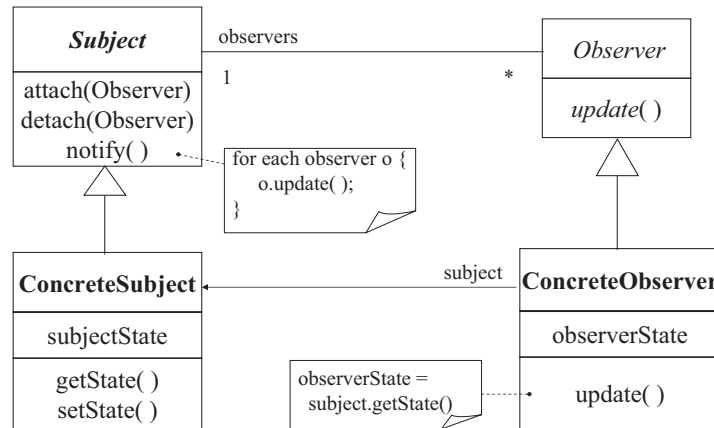


Figure 2.8: Class diagram illustrating the participants in the Observer design pattern.

impose such as secondary role onto relevant classes by means of inter-type declarations (as done in the AJHOTDRAW project). In this way, the persistence logic is isolated in the aspect, and figure classes need not contain any persistence-related code.

Observe that this refactoring merely moves methods from classes to aspects, and involves neither a pointcut nor advice. Thus, this refactoring does not have an effect on any *fan-in* value, and the methods from the *StorableOutput* and *StorableInput* classes will continue to have a high fan-in. In the original implementation, however, these calls came from many different classes or even different packages. In the aspect solution, all calls are from the persistence aspect. This suggests that it may be interesting to lift the call relation to the class, aspect, or package level, and count, for example, the number of other packages using a particular method. We have not yet explored this direction.

2.6.3 Observers in JHOTDRAW

Several methods with high fan-in point to instances of the *Observer* design pattern. Example methods include `Figure.addFigureChangeListener(...)` (fan-in 11) and `Figure.changed()` (fan-in 36).

The participants of the Observer design pattern are shown in Figure 2.8, taken from [Gamma et al., 1994]. One method that we expect to have a high fan-in is `notify`: this method is called for every different kind of change event the observer wants to hear about. Furthermore, we expect the fan-in for the `attach` and `detach` methods to be related to the number of observers involved. The `Observer.update()` method

```

public void execute() {
    // perform check whether view() isn't null.
    super.execute();

    // prepare for undo
    setUndoActivity(createUndoActivity());
    getUndoActivity().setAffectedFigures(view().selection());

    // key logic: cut == copy + delete.
    copyFigures(view().selection(), view().selectionCount());
    deleteFigures(view().selection());

    // refresh view if necessary.
    view().checkDamage();
}

```

Figure 2.9: (Simplified) execute method in JHOTDRAW exhibiting tangling.

is likely to have a low fan-in value, as it is only called from the `Subject.notify()` method.

These expectations are met in JHOTDRAW: The `Figure.changed()` method corresponds to the `Subject.notify()` and indeed has the highest fan-in, allowing us to discover this concern. Observers are called *Listeners* in JHOTDRAW, and the `addFigureChangeListener` corresponds to the `attach` method.

Matching on the naming conventions used in the first observer found led us to another instance of the pattern (with a somewhat lower fan-in). Thus, fan-in analysis provides initial seeds and application understanding, which then can be used by complementary techniques to identify further cross cutting concerns.

The Observer is a prototypical example of a design suitable for an aspect implementation: Inter-type declarations can be used to super-impose the *Observer* or *Subject* roles onto classes of interest, and pointcuts and advice can be used to weave in the appropriate calls to `notify()`.

The notification protocol used in JHOTDRAW is somewhat more complicated than a simple call to `changed()`. Before the change is being made, the affected figures should be invalidated, which should be done by means of a call to the method `willChange()` (fan-in value 25). Such a *policy enforcement* concern calls for an around advice, which helps to ensure that the protocol is properly implemented.

2.6.4 Other Concerns

Command and Related Concerns A method with high fan-in value (24) that is easy to connect to a design pattern is `AbstractCommand.execute()`. The crosscutting nature of the Command pattern is discussed by Hannemann and Kiczales [2002]. They propose a (fairly complex) aspect-oriented representation in which different roles (such

as the command *invoker* and *receiver*) are distinguished. The *protocol* between these is based on a pointcut capturing all places where invocations are required (for example when a GUI button is pressed). The advice then is to activate the receiver for the given invoker. This corresponds to calling the *execute* method, which in the aspect solution has a low fan-in, and in the non-aspect implementation a high one. The applicability of this solution to JHOTDRAW is not clear: isolating the Command concern in this way is complicated by the interaction with the *undo* and *redo* concerns.

The various implementations of the specific `execute()` commands exhibit two further concerns, as illustrated by the *CutCommand* example in Figure 2.9:

- Each `execute` implementation starts with a super call responsible for checking a common pre-condition, throwing an exception if it does not hold. This is a *Contract enforcement* concern as discussed for PETSTORE.
- Most `execute` implementations conclude with a check if the figure has been changed in order to trigger a refresh of the view if necessary. This is a *Providing consistent behavior* concern as discussed by The AspectJ Team [2003].

Factoring these (as well as the undo functionality) out of the code in Figure 2.9 would leave the `execute` method with just its core functionality, which is an implementation of the cut operation by means of a copy and delete operation.

Consistent Behavior The seeds reported by fan-in analysis cover 11 different instances of the “consistent behavior” concern. In other words, there are 11 different contexts into which a set of method-callers invoke a method with a high fan-in value as part of a consistent mechanism. Examples include the previously discussed notification to conclude the execution of commands, consistent (de-)activation of tools, initialization of tools, etc. Each of these 11 instances is a suitable candidate for replacement by an aspect solution by means of a pointcut and advice.

Composite High fan-in values are also obtained for the children manipulation methods from the Composite pattern (e.g., `add(Figure)`, fan-in value 13). The high fan-in in this case is largely due to the fact that these manipulation methods are widely used, but there was no systematic pattern in this usage. The high fan-in is not directly related to the crosscutting nature of the Composite pattern, and, consequently, not affected by a refactoring to the aspect-oriented Composite implementation suggested by Hannemann and Kiczales [2002] (which consists of one aspect containing inter-type declarations for the various composite participants).

Decorator, Adapter Several of the high fan-in methods are related to the Decorator or Adapter patterns. These patterns are different from, e.g., Command and Observer, which have characteristic methods likely to have a high fan-in (`execute` and `notify`, respectively). Instead, the Decorator and Adapter patterns make use of consistent forwarding, which allows us to recognize the relation with the pattern of the several methods with a high fan-in value reported for this concern (such as `DecoratorFigure.containsPoint`, fan-in value 15).

The aspect solution for these patterns as discussed by Hannemann and Kiczales [2002] is to drop the decorator and adapter classes altogether, directly weaving in the relevant decorations or adaptations in the appropriate classes. Whether this solution is applicable to JHOTDRAW is not clear, since JHOTDRAW relies on enabling or disabling decorations (which is less easy to do in the implicit aspect solution).

False Positives The group of false alarms for JHOTDRAW consists of 56 methods. More than half of these methods are implementations of two methods: `displayBox` and `containsPoint`. The first of the two returns the display box of a figure. The method has a high fan-in value because it supports many of the actions associated with a figure, like drawing or moving figures, etc. However, the callers could not be grouped by a clear relationship, and no clear call idiom could be observed when investigating the call sites.

Similar observations apply to the `containsPoint` method, which checks if a point is inside a figure. Except one implementation, which together with other reported methods in the *DecoratorFigure* class implement the consistent logic of redirecting incoming calls, *containsPoint* has been marked as a false positive.

Other false alarms include five `moveBy` methods from *Figure* classes, which implement actions for moving a figure, and a number of complex accessor methods that could not be filtered using the name or implementation criteria.

False Negatives As discussed for the identified Observer pattern instance, other instances of this pattern can be discovered starting from the fan-in seeds. The *Drawing* classes, for example, are part of a different Observer implementation and define role-specific methods with names that are similar to those in the *Figure* classes: `add/removeDrawingChangeListener(...)`. These role methods have lower fan-in values because the *Drawing* Observer implementation has a smaller extent, with fewer classes that register as *Drawing* observers.

The comparison experiment using JHOTDRAW as common benchmark revealed a few concerns missed by fan-in analysis [Ceccato et al., 2006]. One of these concerns is a *Visitor* pattern instance. The pattern defines specific roles and methods, such as the visit operations for the *Visitor* role, and the `accept` method implemented by the *Visitable* elements. The `visit` method in the *Visitor* role would collect calls from all the *Visitable* classes that pass self-objects as arguments to this method for being visited. A large number of *Visitable* elements would therefore increase the fan-in value of the visitor method. However, in JHOTDRAW only two *Figure* classes implement the methods to accept visitors. The large majority of figures do not override the default implementation for this task, which also implements the tree traversal for composite elements.

We have found implementation of the Visitor pattern through the role-specific methods by applying FINT to its own source code, as well as in TOMCAT, as we shall see in the next section.

Concern	No. of methods	Max fan-in
Chain of responsibility (pipeline)	24	18
Command	2	16
Composite	9	37
Consistent behavior	34	90
Contract enforcement	9	46
Lifecycle	73	34
Logging	1	10
Observer	6	56
Redirector	4	25
Visitor	1	28

Table 2.4: Concerns found for TOMCAT, together with the number of high-fan in methods, and the highest fan-in among those methods.

2.7 TOMCAT

Apache TOMCAT is the servlet container that is used in the reference implementation for Sun’s Java Servlet and JavaServer Pages technologies. TOMCAT is developed within the open-source Jakarta project at the Apache Software Foundation.⁹ The main elements of TOMCAT are the servlet container called Catalina, the JSP engine called Jasper, and the TOMCAT connectors. We analyze and discuss the results for version 5.5(.17) of TOMCAT¹⁰.

The main architectural components of TOMCAT are shown in Figure 2.10 [Moodie, 2005]. The outer *Server* component offers a number of *Services* through various *Connectors*. The default connector implements HTTP. The *Engine*, *Host* and *Context* components are all *container components*, representing the top-level container, the virtual host, and the actual web application, respectively. Inside containers there can be *nested components* which can provide various administrative services. Some components can be contained more than once and are marked with a star in the figure. Particularly relevant for our discussion are the nested components called *Valves*: these can intercept a request and process it before it reaches its destination.

The crosscutting concerns found for TOMCAT are summarized in Table 2.4. Again, some of the concerns are related to crosscutting behavior as encountered in design patterns, but there are also some concerns not previously described. Below we elaborate some of the concerns in more detail.

⁹ <http://jakarta.apache.org/tomcat/>

¹⁰ <http://tomcat.apache.org/tomcat-5.5-doc>

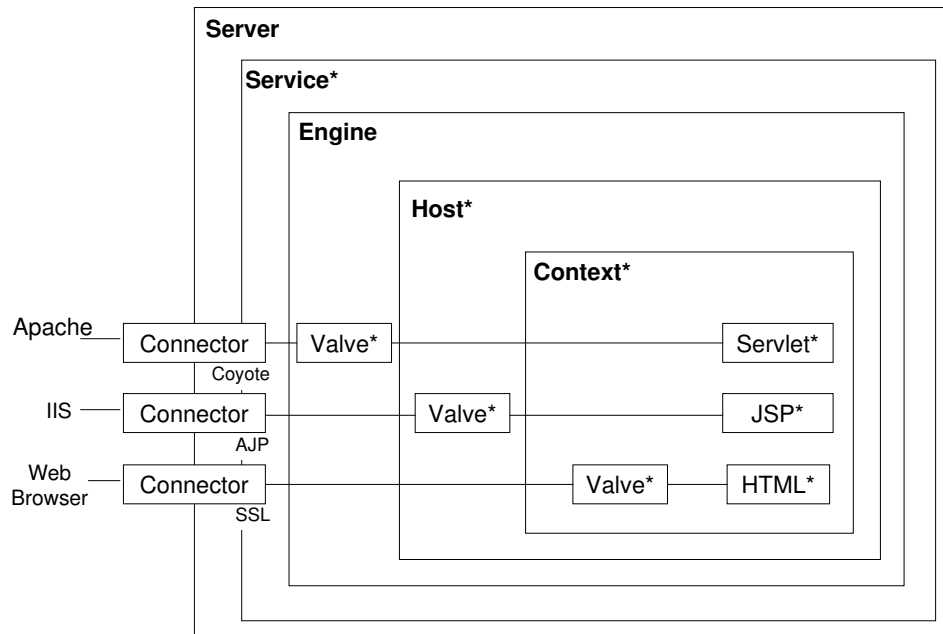


Figure 2.10: Example TOMCAT configuration

2.7.1 Lifecycle

Lifecycle is a common interface for several Catalina components, providing a consistent mechanism to start and stop the component. It is a secondary interface, adding new, supplementary capabilities to the core logic of the implementing classes. *Lifecycle* is implemented by more than 40 classes. The start and stop methods for these classes have fan-in values varying between 25 and 34. The set of results of fan-in analysis comprises 73 implementations of these two Lifecycle methods.

The start and stop methods are part of a particular type of *consistent behavior* scheme: The start operation has to be called before any public method of the component, while stop terminates the object's use and should be the last call for a component's instance. Furthermore, implementors of the *Lifecycle* interface have to adopt the *Subject* role from the *Observer* pattern: listeners can be added which must be notified of start or stop events. The key methods to support these operations have fan-in values as high as 56.

The Lifecycle concern can be seen as a generalization of the use of `stop()` methods to remedy Java's expensive finalization mechanism [Vickers, 2002; Goetz, 2004]. Those methods take care of cleaning up the object's resources inside the program code to avoid the overhead of having finalizers but will result in crosscutting for the object's clients.

The Lifecycle concern is complex, comprising several crosscutting concerns. Al-

though aspect-oriented solutions have been presented for some parts of it, a complete refactoring solution remains an open issue. One of the problems is that the type of consistent behavior needed by the concern cannot be expressed in a pointcut-based aspect language like ASPECTJ (because it requires specifying “before accessing any public methods of class” and “after last use of class”).

2.7.2 Valves / Chain of Responsibility

A method occurring around 20 times in the seed list is the `invoke(...)` method in the *Valve* hierarchy. Valves are nested components that implement a pluggable request-processing operation for an associated container. Valves are connected through a pipeline structure, in which each valve passes the request to the `invoke` method of the next valve in the pipeline. Examples of valve classes include *AccessLog Valve* to create standard web servers log files, *RemoteAddress Valve* to filter the requests by the IP address of the client that submitted them, or *SingleSignOn Valve* to grant user access to the web applications associated with a virtual host.

The pipeline organization of the valves is implemented using the *Chain of responsibility* pattern [Gamma et al., 1994]. This implies that a valve’s core logic is crosscut by the functionality of retaining the reference to the next valve in the pipeline and consistently passing the invocation to it. Furthermore, the various implementations of the `invoke` method are tangled with other concerns. The *AuthenticatorBase* abstract class, for instance, implements the basic functionality of the request authentication valve. However, its `invoke` method also performs logging operations for debugging activities. Similarly, the previously mentioned *AccessLog Valve* implements a *timing* operation for the request/response operation it has to log. An aspect-oriented solution for the *Chain of responsibility* pattern is provided by Hannemann and Kiczales [2002].

2.7.3 Other Concerns

A number of architectural components of TOMCAT and Catalina are Container elements. The *Container* interface defines these elements as *Composite* structures. Standard implementations of the interface are abstractions of the TOMCAT container components, like *StandardEngine* or *StandardContext*. Fan-in analysis identifies the children manipulation methods specific to the *Composite* structure of these components and reports them as concern seeds (fan-in values of up to 37).

In the same category of design patterns, a number of seeds correspond to the Observer, like the notifier for Container events (`ContainerBase.fireContainerEvent(...)`) (fan-in value 55) and the `execute` method of the *Command* pattern implementation (fan-in value 16). Similar to the cases discussed for JHOTDRAW, the Command seed methods reported for TOMCAT are also part of a *contract enforcement* that consist of a pre-execution attribute validation. The contract is implemented as a call to the method in the super class. Other seed results include methods that participate in the implementation of consistent *redirection* functionality (*Wrappers*); the

methods implement non-trivial accessors that are invoked by a large number of methods that simply redirect their callers to dedicated methods of the reference returned by the reported seed. The fan-in values for these seeds are up to 25.

Different *pre-condition check enforcements* are also part of the various implementations for the *Lifecycle* `start` and `stop` methods. The reported seed method in this case is the constructor of the exception thrown if the pre-condition does not hold (fan-in value 32).

The *logging* concern is particularly interesting because of the new implementation strategy in version 5.x of TOMCAT. This concern used to be implemented in the previous versions using *Logger* classes that were part of the Catalina API. However, the current implementation uses logging functionality available through specialized, external libraries. Although we have been able to directly identify *logging* methods in the analyzed code (e.g., *ModuleClassLoader*), as well as logging functionality tangled with the implementation of other seed methods, a number of direct *logging* seeds are missed. This is due to our choice not to include library components in the analysis, as discussed in Section 2.4.

The remaining seeds include, besides other instances of the concerns already discussed, a large number (up to 25) of different instances of the *consistent behavior* concern, as well as seeds for the super-imposed role in the *Visitor* pattern.

False Positives A group of 13 false alarms consists of methods in the *JspReader* and *ServletWriter* classes. The first class is an input buffer for the JSP parser, and the reported methods are utilities for parsing JSP files, like methods to match an input String in a file or to skip space-characters. The callers are methods in the *JSP Parser* class.

The methods reported for *ServletWriter* print String elements in various formats to an output stream. The callers of these methods belong to the *Generator* class, which outputs Java code from an internal, tree-based (XML) representation of JSP documents.

These classes could have been considered as *utility*, if we would have had more detailed knowledge about the system prior to analysis.

Among the other false alarms there are 12 implementations of the `store` method in the *StoreFactoryBase* hierarchy. The classes in this hierarchy are specialized in storing configuration elements, such as *Server*, *Service*, *Engine*, or *Context* to a XML configuration file (`server.xml`). The callers of the reported methods are declared in classes in the same hierarchy or are overloaded implementations of the `store` method in the class *StoreConfig*. This class is part of the same concern as the reported methods and so no crosscutting element could be identified.

False Negatives The literature on TOMCAT discusses hardly any crosscutting concerns, making it difficult for us to assess whether there are any interesting false negatives we missed. The crosscutting concern that is discussed widely for TOMCAT is logging, and often it is mentioned as an example of poor modularization. As already discussed, fan-in analysis helps us to identify several seeds for the logging concern.

However, the analyzed version of TOMCAT is extensively using logging methods declared by external libraries (the *org.apache.commons.logging.** package). By canceling the filter for library methods in FINT and looking for calls to externally declared methods, we noticed that there are 19 methods from the logging package that are referred from the analyzed TOMCAT sources. From these ones, 13 methods belong to the *Log* class and show a fan-in value higher than the considered threshold of 10. The fan-in value for the logging method for debugging (*Log.debug*), for example, is as high as 465.

2.8 Discussion

High Fan-in as Indicator As we have seen in the previous case studies, fan-in analysis identifies high fan-in methods, applies a series of filters to these methods, after which more than half of the remaining methods turn out to be related to a crosscutting concern.

We can distinguish three main situations in which a high fan-in value indicates the presence of crosscutting concerns:

- The method has a high fan-in because it is part of a *dynamic* crosscutting mechanism. The typical refactoring will be to capture the call sites through a pointcut, and to move the method call to advice. Examples that we encountered include exception wrapping, contract enforcement, observer notification, and life cycle.
- The method has a high fan-in because it is used by a *static* crosscutting mechanism. A typical example is a secondary interface that must be implemented by a series of classes. The various implementations are likely to make use of the same helper methods, giving these a high fan-in. The refactoring is to collect all these interface implementations into one or more inter-type declarations. This we encountered for the persistence concern.
- The method has a high fan-in because it is part of a concern that plays a key role in the design. The method happens to be part of a crosscutting concern, which will benefit from an aspect-oriented refactoring. The refactoring, however, will not affect any of the call sites of the high fan-in method. This we encountered for the composite concern.

These situations are not mutually exclusive. In many cases, a concern involves static as well as dynamic crosscutting, as we have seen for the undo concern. We then are likely to see multiple seeds, which may either point us to the static or to the dynamic crosscutting behavior.

The Type of Concerns Identified The fact that we were able to find similar aspects in various case studies suggests that their identification is not accidental. We identified various crosscutting concerns that are discussed in the literature, including those that

stood at the origins of aspect-oriented programming. In addition, we have identified a number of new aspects, such as *Undo* and *Lifecycle*. Given the different nature of the three case studies, we feel that these results can also be achieved for other cases.

A notable source of crosscutting behavior is formed by various design patterns: for both JHOTDRAW as well as TOMCAT they account for approximately half of the concerns identified. This suggests that it may be worthwhile to investigate the use of design pattern mining techniques (see, e.g., Ferenc et al. [2005]) for aspect mining purposes.

Reasoning about Seeds and Non-seeds One of the subjective elements of our aspect mining approach is the third step in which the human engineer has to distinguish seeds from non-seeds. We adopted the following reasons for classifying a high fan-in method as a seed:

- We were able to link the method to a concern that is known to be crosscutting.
- We considered the method's concern to be conceptually separate from the key functionality of the calling classes. Thus, it would be meaningful to make the base implementation oblivious of method's concern.
- We could discover an idiom, recurring patterns, or other similarities in for example the call sites found, suggesting an implicit relationship between these call sites that could be made explicit through a pointcut with advice.
- We were able to identify a refactoring to ASPECTJ that may be beneficial in terms of modularization, flexibility, or evolution. Usually, these refactorings were composed from basic refactorings as included in the catalogs provided by Laddad [2003a] and Monteiro [2004].

When rejecting a high fan-in method as a seed, we were not able to achieve any of the above.

Utility Filtering A step requiring some manual effort is the filtering of utility methods. The intent of this is to remove groups of methods for which it is a priori obvious that they do not belong to crosscutting concerns. It is not necessary to capture all utility methods. Therefore, the amount of effort involved in this step is very limited: if it is not immediately clear if something is a utility, it is simply safe not to filter the method, and analyze it in detail if it turns out to have a high fan-in.

Percentage of False Positives The percentage of false positives in the three case studies is 13%, 49% and 27% for PETSTORE, JHOTDRAW, and TOMCAT, respectively (see Table 2.1). Based on these figures, and based on experiments we conducted with other systems, we conjecture that 50-75% of the candidate seeds that we identify automatically can be confirmed as belonging to a crosscutting concern.

Note that this percentage is conservative in two ways: First, we only discarded classes or methods as utilities when this was immediately obvious. Second, we only

confirmed seeds when we clearly could see the crosscutting nature of the underlying concern. In other words, it is possible that with a more involved analysis of some of the *non seeds* from Table 2.1 these could turn out to be crosscutting concerns as well. For this reason, it is reasonable to expect that other systems will exhibit a similar (or perhaps higher) success rate.

False Negatives While working on our case studies, analyzing their design and implementation in considerable depth, we did encounter several crosscutting concerns not found through fan-in analysis, some of which were discussed in the previous sections. As an example, for PETSTORE we found transaction management, scattered implementations of the Serializable interface, and opportunities for making use of ASPECTJ's approach to imitating multiple inheritance [Mesbah and van Deursen, 2005]. As for the logging example discussed for TOMCAT, key methods implementing transaction management are likely to be missed as well, because they are part of imported libraries that we do not include in our analysis. Crosscutting concerns found in JHOTDRAW through other aspect mining approaches are discussed and compared by Cécato et al. [2006]. An example concern fan-in analysis did not find is bringing a figure to the front or sending it to the back, simply because the methods involved were not called sufficiently often.

Based on these observations, we can make the following more general claims about the sort of crosscutting concerns that will not be found through fan-in analysis. First, the “footprint” of the concern should be above the threshold. Thus, if the concern involves dynamic crosscutting, the number of scattered calls should be higher than the threshold. Furthermore, if the crosscutting is purely static, the concern will usually not be found, unless the scattered implementation relies on shared functionality, and the number of call sites is higher than the threshold.

Note that the effect of the threshold is twofold. First of all, it helps us reduce the number of methods to be inspected. In addition to that, it allows us to find those aspects that are likely to significantly influence the modularity of the source code. Thus, while we certainly miss some crosscutting concerns, we are likely to find the ones that are most scattered, and hence good candidates for refactoring.

Percentage of False Negatives How to arrive at a percentage for false negatives is less clear. This would require a report of all the crosscutting concerns that could be found in the case studies considered. Such reports have not been available prior to our experiments. Furthermore, such a report would be affected by the difficulty of deciding objectively what is and what is not a crosscutting concern.

The way to achieve progress in this direction is by establishing a common benchmark of known crosscutting concerns in existing systems. Such a benchmark would not only be a simple list, but also a summary of the reasons why certain concerns are deemed crosscutting. Our coverage of the concerns we found through fan-in analysis is aimed at establishing and promoting such a benchmark.

Seed Inspection Effort How much effort is involved in inspecting seeds by hand? An important observation to make is that in many cases it is possible to decide for a group

of methods together whether they constitute a seed. One reason for this is our treatment of polymorphism. In our definition of the fan-in metric one call could increase the metric value for several methods in the hierarchy of the invoked callee. Therefore, method implementations in the same hierarchy, which most commonly implement the same concern, also share many of their callers.

Such situations are very common in the cases analyzed. In TOMCAT, for instance, the over 200 seed and non-seed methods are implementations or declarations of only less than 100 distinct methods. As another example, the set of candidates for JHOTDRAW includes more than 20 implementations of the `displayBox` method, which we marked as non-seed. Grouping methods by their declarations as supported by FINT considerably reduces the investigation effort required for each method.

FINT offers further ways to reduce the manual effort involved in seed inspection. This includes various analyses to detect relations between the callers of a reported method with a high fan-in value, as discussed in Section 2.3.4. For example, by examining the callers (of any of the around 20 reported implementations) of the `invoke` method in TOMCAT's *Valves pipeline* concern, FINT shows that more than 80% of these callers are also `invoke` methods in *Valve* classes. The tool groups these callers as shown in Figure 2.4. Such relations are present for a significant number of discovered seeds, including crosscutting elements discussed for JHOTDRAW's *Undo* concern and the concerns in the *Command* hierarchy, as well as *Exception wrapping* concerns in PETSTORE.

Required Expertise Level How much domain knowledge or expertise is required for conducting fan-in analysis? For the bottom-up approach, when we look for consistent invocations of the method with a high fan-in value from call sites that could be captured by a pointcut definition, little specific knowledge is needed. For the top-down approach more a priori knowledge is required. The top-down approach relies on easily observable relations between tool-reported candidates and known examples of crosscutting functionality; design patterns are the most common in our cases. The rules we employed for associating patterns to candidates are simple: the methods are part of the roles defining the design patterns and/or they execute actions specific to responsibilities of participants in the pattern implementation (e.g., delegations of actions).

Note, however, that many crosscutting concerns described in the context of design pattern implementations will typically be found by means of the bottom-up approach as well. For instance, calls to notification methods in implementations of the Observer pattern, or invocations to the action of the next element in a pipeline (chain of responsibility) are typical examples of crosscutting concerns targeted by fan-in analysis. In this case, the discussion of the patterns serves to describe the larger context into which the crosscutting concern occurs.

AspectJ Fan-in analysis is an aspect mining approach that is entirely independent of ASPECTJ or any other aspect-oriented language. Fan-in analysis is a technique for understanding a system's modularization, helping developers to find crosscutting concerns. Some of these can be candidates for a refactoring towards ASPECTJ (as

discussed for PETSTORE and JHOTDRAW by Mesbah and van Deursen [2005] and Marin et al. [2005b]). For other concerns, alternative aspect-oriented solutions, such as composition filters [Bergmans and Aksit, 2001] or the inversion of control pattern [Fowler, 2004], while for still other concerns present aspect-oriented languages do not offer a suitable modularization mechanism yet.

Thus, fan-in analysis is not only a possible first step in refactoring to aspects. It also is a program comprehension technique that can help to understand crosscutting concerns in existing applications.

The Fan-In Metric The variant of the fan-in metric we have used, has been optimized for aspect mining purposes, and, as shown in this chapter, has brought us good results. An open question is whether this metric can be further improved. One possible route would be to lift the fan-in metric to the class, inheritance hierarchy, or package level, as we briefly discussed for the persistence concern of JHOTDRAW in Section 2.6.2. Fine tuning the metric such that it reflects, e.g., call site locations instead of the mere number of methods containing call sites is an issue for further research.

2.9 Concluding Remarks

2.9.1 Contributions

We consider the following as our three key contributions.

First of all, we propose a new, metrics-based, aspect mining approach. The approach aims at capturing crosscutting concerns by focusing on methods that are called from many places, and hence have a high fan-in. Our case studies show that after appropriate filtering more than 50% of these methods turn out to belong to a crosscutting concern.

Our second contribution is FINT, a tool that is freely downloadable that supports fan-in analysis. FINT not only shows how the fan-in metric and the filters can be implemented, but also offers support for the final manual step consisting of exploring the high fan-in methods and their call sites, and managing the seed-methods.

The third contribution consists of the extensive case studies we conducted. We argue in detail why we think that certain concerns are crosscutting in three existing open source Java systems. Some of these concerns were not previously described in the literature as crosscutting (such as undo or lifecycle). Moreover, in most cases we discuss alternative aspect-oriented implementations of these concerns. The resulting list of concerns and their manifestation in the three systems is relevant not only for fan-in analysis: it is of value for the validation of any aspect mining approach.

In addition to that, we offer an explanation of our results by identifying the factors contributing to the success of fan-in analysis as an aspect mining approach, as well as the limitations of the approach.

2.9.2 Future Work

We are presently in the process of extending our results along the following lines.

First, we are considering various extensions to FINT. One route is to integrate FINT with other concern elaboration tools, such as FEAT [Robillard and Murphy, 2007] or the Concern Manipulation Environment CME [Harrison et al., 2004]. We could use such tools to explore and describe a concern or feature to its full extent, starting from the (partial) set of elements and relations identified by FINT as part of the crosscutting concern implementation.

Another option is to combine FINT with other automated aspect identification techniques, such as, for example, techniques based on formal concept analysis, identifier analysis, or clone detection. A prerequisite for combination is to be able to assess and compare aspect mining techniques and their results.

In addition to that, we continue to elaborate our case studies. This will provide further data on optimal threshold values, typical number of concerns that can be found in existing applications, and figures for the percentages of false positives and false negatives.

The results presented in this chapter show that the recognized crosscutting concerns follow various implementation idioms. Fan-in analysis is particularly suited for identifying method invocations that cut across a set of other methods. However, concerns like those encountered in the Decorator pattern are typically less likely to occur among the results of this technique. In Chapter 5 of this thesis, we shall take a number of steps towards design of mining techniques that target specific implementation idioms, and their implementation in FINT.

One of our activities directly related to one of the case studies presented in this chapter is AJHOTDRAW, a sourceforge project in which we offer an aspect-oriented re-implementation of JHOTDRAW, based on the concerns found in the present chapter.¹¹ In this way, the case studies presented here form the starting point for a benchmark for comparing aspect mining and refactoring approaches.

¹¹<http://sourceforge.net/projects/ajhotdraw/>. Also discussed in Chapter 6.

Chapter 3

Applying and Combining Three Different Aspect Mining Techniques

Understanding a software system at source-code level requires understanding the different concerns that it addresses, which in turn requires a way to identify these concerns in the source code. Whereas some concerns are explicitly represented by program entities (like classes, methods and variables) and thus are easy to identify, crosscutting concerns are not captured by a single program entity but are scattered over many program entities and are tangled with the other concerns. Because of their crosscutting nature, such crosscutting concerns are difficult to identify, and reduce the understandability of the system as a whole.

In this chapter, we report on a combined experiment in which we try to identify crosscutting concerns in the JHotDraw framework automatically. We first apply three independently developed aspect mining techniques to JHotDraw and evaluate and compare their results. Based on this analysis, we present three interesting combinations of these three techniques, and show how these combinations provide a more complete coverage of the detected concerns as compared to the original techniques individually. Our results are a first step towards improving the understandability of a system that contains crosscutting concerns, and can be used as a basis for refactoring the identified crosscutting concerns into aspects.

3.1 Introduction

The increasing popularity of aspect-oriented software development (AOSD) is largely due to the fact that it recognises that some concerns cannot be captured adequately using the abstraction mechanisms provided by traditional programming languages. Several examples of such *crosscutting* concerns have been identified, ranging from simple ones such as logging, to more complex ones such as transaction management [Fabry, 2005] and exception handling [Lippert and Lopes, 2000].

An important problem with such crosscutting concerns is that they affect the un-

derstandability of the software system, and as a result reduce its evolvability and maintainability. First of all, crosscutting concerns are difficult to understand, because their implementation can be scattered over many different packages, classes and methods. Second, in the presence of crosscutting concerns, ordinary concerns become harder to understand as well, because they get tangled with the crosscutting ones: particular classes and methods do not only deal with the primary concern they address, but also may need to take into account some secondary, crosscutting concerns.

Several authors have presented automated code mining techniques, generally referred to as *aspect mining* techniques, that are able to identify crosscutting concerns in the source code. The goal of these techniques is to provide an overview of the source-code entities that play a role in a particular crosscutting concern. This not only improves the understandability of the concern in particular and of the software in general, but also provides a first step in the migration towards applying aspect-oriented software development techniques. However, since the research field is still in its infancy, very few experiments have been conducted on real-world case studies, comparisons of different techniques are lacking, and no agreed-upon benchmark is available that allows to evaluate the existing techniques.

This chapter reports on an experiment involving three independently developed aspect mining techniques: fan-in analysis [Marin et al., 2007a], identifier analysis [Mens and Tourwé, 2005; Tourwé and Mens, 2004] and dynamic analysis [Tonella and Cécato, 2004a]. In the experiment, each of these techniques is applied to the same case study: the JHotDraw graphical editor framework. The goal of the experiment is not to identify the “best” aspect mining technique, but rather to mutually compare the individual techniques and assess their major strengths and weaknesses. Additionally, by identifying where the techniques overlap and where they are complementary, the experiment allows us to propose interesting combinations and to apply these combinations on the same benchmark to verify whether they actually perform better.

The JHotDraw framework which we selected as benchmark case was originally developed to illustrate good use of object-oriented design patterns [Gamma et al., 1994] in Java programs. This implies that the case study has been well-designed and that care has been taken to cleanly separate concerns and make it as understandable as possible. Nevertheless, JHotDraw exposes some of the modularisation limitations present even in well-designed systems, and contains some quite interesting crosscutting concerns.

The contributions of this chapter can be summarised as follows:

- We provide an overview of the major strengths and weaknesses of three aspect mining techniques. This information is valuable for developers using these techniques, as it can help them choosing a technique that suits their needs. Other aspect mining researchers can take this information into account to compare their techniques to ours, or to fine-tune our techniques;
- We discuss how the individual techniques can be combined in order to perform better, and validate whether this is indeed the case by applying the combined techniques on the same benchmark application and comparing the results;

```
interface A {  
    public void m();  
}  
class B implements A {  
    public void m() {};  
}  
class C1 extends B {  
    public void m() {};  
}  
class C2 extends B {  
    public void m() { super.m(); };  
}  
class D {  
    void f1(A a) { a.m(); }  
    void f2(B b) { b.m(); }  
    void f3(C1 c) { c.m(); }  
}
```

Figure 3.1: Various (polymorphic) method calls.

- We present a list of all crosscutting concerns that the three techniques identified in the JHotDraw framework. Such information is valuable for other aspect mining researchers who want to validate their techniques, and might lead to JHotDraw becoming a de-facto benchmark for aspect mining techniques;

The chapter is structured as follows. Section 3.2 introduces the necessary background concepts required to understand the three aspect mining techniques explained in Section 3.3. Section 3.4 presents the results of applying each technique on the common benchmark, while Section 3.5 uses these results for discussing the benefits and drawbacks of each technique with respect to the others. Based on this discussion, Section 3.6 presents useful combinations of the techniques, and reports on the experience of applying such combinations on the benchmark application. Section 3.7 presents our conclusions. For an overview of related work concerning aspect mining, we refer to the previous chapter and the papers discussing the individual techniques [Marin et al., 2007a; Mens and Tourwé, 2005; Tourwé and Mens, 2004; Tonella and Ceccato, 2004a].

3.2 Background concepts

3.2.1 Fan-in

The *fan-in* metric, as defined by Henderson-Sellers [1996], counts the number of locations from which control is passed into a module. In the context of object-orientation, the module-type to which this metric is applied is the method. We define the *fan-in* of

a method M as the number of distinct method bodies that can invoke M . Because of polymorphism, one call site can affect the fan-in of several methods: a call to method M contributes to the fan-in of M , *but also* to all methods refined by M , *as well as* to all methods that are refining M (see the previous chapter).

Method	Potential callers	Fan-in
A.m	D.f1, D.f2, D.f3	3
B.m	D.f1, D.f2, D.f3, C2.m	4
C1.m	D.f1, D.f2, D.f3	3
C2.m	D.f1, D.f2	2

Figure 3.2: Fan-in values for program in Figure 3.1.

As an example, Figure 3.2 shows the calculated fan-in for the methods named m in the program of Figure 3.1. Note that $D.f3$ is reported among the potential callers of $B.m$, even though this situation cannot actually occur at run-time. However, the resulting effect of having higher fan-in values reported for methods in super-classes is arguably positive for the purpose of the present analysis, as it emphasizes the concern implemented by the super-class method, which generally is addressed by its overriding methods as well.

3.2.2 Concept Analysis

Formal concept analysis (FCA) is a branch of lattice theory that can be used to identify meaningful groupings of *elements* that have common *properties* [Ganter and Wille, 1997].¹

Programming language	object-oriented	functional	logic	static typing	dynamic typing
Java	✓	-	-	✓	-
Smalltalk	✓	-	-	-	✓
C++	✓	-	-	✓	-
Scheme	-	✓	-	-	✓
Prolog	-	-	✓	-	✓

Table 3.1: Programming languages and their supported programming paradigms.

FCA takes as input a so-called *context*, which consists of a (potentially large, but finite) set of *elements* E , a set of *properties* P on those elements, and a Boolean *incidence relation* T between E and P . An example of such a context is given in Table 3.1,

¹We use the terms *element* and *property* instead of *object* and *attribute* used in traditional FCA literature, because these latter terms have a very specific meaning in object-oriented software development.

which relates different programming languages and properties. A mark \checkmark in a table cell means that the element (programming language) in the corresponding row has the property of the corresponding column.

Starting from such a context, FCA determines *maximal* groups of elements and properties, called *concepts*, such that each element of the group shares the properties, every property of the group holds for all of its elements, no other element outside the group has those same properties, nor does any property outside the group hold for all elements in the group. Intuitively, a *concept* corresponds to a maximal ‘rectangle’ containing only \checkmark marks in the table, modulo any permutation of the table’s rows and columns.

Formally, the starting context is a triple (E, P, T) , where $T \subseteq E \times P$ is a binary relation between the set of all elements E and the set of all considered element properties P . A *concept* c is defined as a pair of sets (X, Y) such that:

$$X = \{e \in E \mid \forall p \in Y : (e, p) \in T\} \quad (3.1)$$

$$Y = \{p \in P \mid \forall e \in X : (e, p) \in T\} \quad (3.2)$$

where X is said to be the *extent* of the concept ($Ext[c]$) and Y is said to be its *intent* ($Int[c]$). It should be noticed that the definition above is not “constructive”, being mutually recursive between X and Y . However, given a pair (X, Y) , it allows deciding whether it is a concept or not. FCA algorithms provide constructive methods to determine all pairs (X, Y) satisfying the constraints (1) and (2).

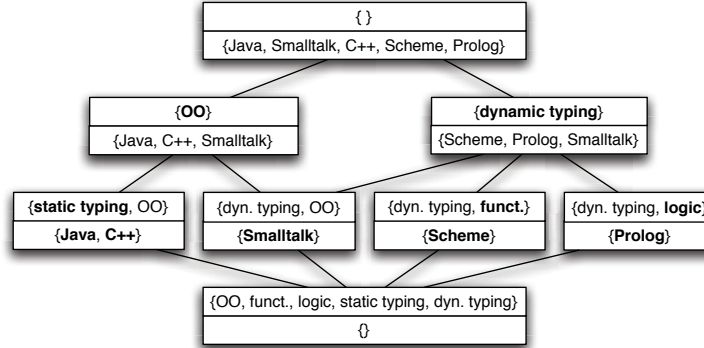


Figure 3.3: The concept lattice for Table 3.1.

The containment relationship between concept extents (or, equivalently, intents) defines a partial order over the set of all concepts, which can be shown to be a lattice [Ganter and Wille, 1997]. Figure 3.3 shows the concept lattice corresponding to Table 3.1. The lattice’s bottom concept contains those elements that have all properties. Since there is no such programming language in our example, that concept contains no elements (its extent is empty). Similarly, the top concept contains those properties that hold for all elements. Again, there is no such property (the concept’s intent is empty). Other concepts represent related groups of programming languages,

such as the concept $(\{Java, C++\}, \{static\ typing, OO\})$, which groups all statically-typed object-oriented languages, a sub-concept of all OO languages. Intuitively, the sub-concept relationship can thus be interpreted as a specialization of more general notions. Elements (resp. properties) in boldface are those that are most concept-specific, being attached to the largest lower bound (resp. least upper bound) concept. When using the so-called *sparse labeling* of the concept lattice, only these boldface labels are retained, without loss of information.

More precisely, when using *sparse labeling*, a node c is marked with an element $e \in Ext[c]$ only if it is associated with the most specific (i.e., lowest) concept c having e in the extent; a node c is marked with a property $p \in Int[c]$ only if it is associated with the most general (i.e., highest) concept c having p in its intent. The (unique) node of a lattice L marked with a given element e is thus:

$$\gamma(e) = \inf\{c \in L \mid e \in Ext[c]\} \quad (3.3)$$

where *inf* gives the infimum (largest lower bound) of a set of concepts. Similarly, the unique lattice node marked with a given property p is:

$$\mu(p) = \sup\{c \in L \mid p \in Int[c]\} \quad (3.4)$$

where *sup* gives the supremum (least upper bound) of a set of concepts. The set of elements in the extent of a lattice node c can then be computed as the set of all elements at or below c , while the set of properties in its intent are those marking c or any node above c .

The labeling introduced by the functions μ and γ give the most specific concept for a given element (resp. property). Thus, with *sparse labeling*, the elements and properties that label a given concept are those that characterize it most specifically. Sometimes it is convenient to get the labels of a given concept through the following functions:

$$\alpha(c) = \{p \in P \mid \mu(p) = c\} \quad (3.5)$$

$$\beta(c) = \{e \in E \mid \gamma(e) = c\} \quad (3.6)$$

$\alpha(c)$ gives the set of properties labeling a concept c , while $\beta(c)$ gives the concept's elements, according to the *sparse labeling*.

3.2.3 Terminology

We conclude this background section by introducing some terminology that will be used throughout the remainder of this chapter.

A concern is a collection of related source-code entities, such as classes, methods, statements or expressions, that implement a particular functionality or feature of the application. A *crosscutting* concern is a concern whose entities are not captured into a single localised abstraction, but are scattered over many different locations and tangled with other concerns.

A (concern) seed is a single source-code entity, such as a method, or a collection of such entities, that strongly connotes a crosscutting concern. It offers a starting point for further exploration and understanding the whole extent of that concern's implementation.

A candidate seed is identified by an automated aspect mining technique as a potential concern seed but is not yet confirmed to be an actual concern seed or rather a false positive.

Seed expansion is the manual or automated process of completing the set of source-code entities constituting a seed into the entire set of source-code entities of which the crosscutting concern corresponding to that seed consists.

3.3 The three aspect mining techniques

In this section, we give a brief overview of three techniques, developed independently by different research groups, that support the automated discovery of crosscutting concerns in the source code of a software system that is written in a non aspect-oriented way.

3.3.1 Fan-in Analysis

Crosscutting functionality can occur at different levels of modularity. Classes, for instance, can assimilate new concerns by implementing multiple interfaces or by implementing new methods specific to super-imposed roles. At the method level, crosscutting in many cases resides in calls to methods that address a different concern than the core logic of the caller. Typical examples include logging, tracing, pre- and post-condition checks, and exception handling. It is exactly this type of crosscutting that fan-in analysis tries to capture.

When we study the mechanics of AOSD, we see that it employs the so-called *advice* construct to eliminate crosscutting at method level. This construct is used to acquire control of program execution and to add crosscutting functionality to methods without an explicit invocation from those methods. Rather, the crosscutting functionality is isolated in a separate module, called aspect, and woven with the method implicitly based on the advice specification.

Fan-in analysis reverses this line of reasoning and looks for crosscutting functionality that is explicitly invoked from many different methods scattered throughout the code. The hypothesis is that the *number of* calls to a method implementing this crosscutting functionality (fan-in) is a good measure for the importance and scattering of the discovered concern.

To perform the fan-in analysis, a fan-in metric was implemented as a plug-in for

the Eclipse platform², and integrated it into an iterative process that consists of three steps:

1. Automatic computation of the fan-in metric for all methods in the investigated system.
2. Filtering of the results from the previous step by
 - eliminating all methods with fan-in values below a chosen threshold (in the experiment, a threshold of 10 was used);
 - eliminating the accessor methods (methods whose signature matches a *get*/set** pattern and whose implementation only returns or sets a reference);
 - eliminating utility methods, like `toString()` and collection manipulation methods, from the remaining subset.
3. (Partially automated) analysis of the methods in the resulting, filtered set by exploring the callers, call sites, naming convention used, the implementation and the comments in the source code.

Besides code exploration, the tool supports automatic recognition of a number of relations between the callers of a method, such as common roles, consistent call positions, etc.

The result of the fan-in analysis is a set of candidate seeds, represented as methods with high fan-in.

3.3.2 Identifier Analysis

In the absence of designated language constructs for aspects, naming conventions are the primary means for programmers to associate related but distant program entities. This is especially the case for object-oriented programming, where polymorphism allows methods belonging to different classes to have the same signature, where it is good practice to use intention-revealing names [Beck, 1997], and where design and other programming patterns provide a common vocabulary known by many programmers.

Identifier analysis relies on this assumption and identifies candidate seeds by grouping program entities with similar names. More specifically, it applies FCA with as elements all classes and methods in the analyzed program (except those that generate too much noise in the results, like test classes and accessor methods), and as properties the identifiers associated with those classes and methods.

The identifiers associated with a method or class are computed by splitting up its name based on where capitals appear in it. For example, a method named

² <http://swert.tudelft.nl/view/AMR/FINT>

`createUndoActivity` yields three identifiers `create`, `undo` and `activity`. In addition, we apply the Porter stemming algorithm [Porter, 1980] to make sure that identifiers with the same root form (like `undo` and `undoable`) are mapped to one single representative identifier or ‘stem’. It is these stems that are used as properties for the concept analysis.

The FCA algorithm then groups entities with the same identifiers. When such a group contains a certain minimum number of elements (in the experiment, a threshold of 4 was used) and the entities contained in it cut across multiple class hierarchies, the group is considered a candidate seed. The only remaining but most difficult task is that of deciding manually whether a candidate seed is a real seed or a false positive. To help the developer in this last task, the *DelfSTof* source-code mining tool presents the concepts in such a way that they can be browsed easily by a software engineer and so that he or she can readily access the code of the classes and methods belonging to a discovered seed.

3.3.3 Dynamic Analysis

Formal concept analysis has been used to locate ‘features’ in procedural programs [Eisenbarth et al., 2003]. In that work, the goal was to identify the computational units (procedures) that specifically implement a feature (i.e., requirement) of interest. Execution traces obtained by running the program under given scenarios provided the input data (dynamic analysis).

In a similar way, dynamic analysis can be used to locate aspects in program code [Tonella and Ceccato, 2004a] according to the following procedure. Execution traces are obtained by running an instrumented version of the program under analysis, for a set of scenarios (use-cases). The relationship between execution traces and executed computational units (methods) is subjected to concept analysis. The execution traces associated with the use-cases are the elements of the concept analysis context, while the executed methods are the properties. In the resulting concept lattice (with sparse labeling), the *use-case specific* concepts are those labeled by at least one trace for some use-case (i.e. α contains at least one element), while the concepts with zero or more properties as labels (those with an empty α) are regarded as *generic* concepts. Thus, use-case specific concepts are a subset of the generic ones.

Both use-case specific concepts and generic concepts carry information potentially useful for aspect mining, since they group specific methods that are always executed under the same scenarios. When the methods that label one such concept (using the sparse labeling) crosscut the principal decomposition, a candidate aspect is determined.

Formally, let C be the set of all the concepts and let C_s be the set of use-case specific concepts ($|\alpha(c)| > 0$). A concept c is considered a candidate seed *iff*:

Scattering: $\exists p, p' \in \beta(c) \mid \text{pref}(p) \neq \text{pref}(p')$

Tangling: $\exists p \in \beta(c), \exists c' \in \Omega, \exists p' \in \beta(c') \mid c \neq c' \wedge \text{pref}(p) = \text{pref}(p')$

where $\Omega = C_s$ for the *use-case specific* seeds, while $\Omega = C$ for the *generic* seeds. The first condition (*scattering*) requires that more than one class contributes to the functionality associated with the given concept ($\text{pref}(p)$ is the fully scoped name of the class containing the method p). The second condition (*tangling*) requires that the same class addresses more than one concern.

In summary, a concept is a candidate seed if: (1) *scattering*: more than one class contributes to the functionality associated with the given concept; (2) *tangling*: the class itself addresses more than one concern.

The first condition alone is typically not sufficient to identify crosscutting concerns, since it is possible that a given functionality is allocated to several modularized units without being tangled with other functionalities. In fact, it might be decomposed into sub-functionalities, each assigned to a distinct module. It is only when the modules specifically involved in a functionality contribute to other functionalities as well (i.e. the second condition) that crosscutting is detected, hinting for a candidate seed.

3.4 Results of the Aspect Mining

In this section, we present the results of applying each technique to version 5.4b1 of JHotDraw, a Java program with approximately 18,000 non-commented lines of code and around 2800 methods. We mutually compare the results of the techniques, and discuss the limitations of each technique as well as their complementarity.

3.4.1 The Fan-in Analysis Experiment

As described in Subsection 3.3.1, fan-in analysis first performs a number of successive steps to filter the methods in the analyzed system. The threshold-based filtering, which selects methods with high fan-in values, kept around 7% of the total number of methods. The filters for accessors and utility methods eliminated around half of the remaining methods. In the remaining subset, more than half of the methods (52%) were categorized as seeds, based on manual analysis.

Table 3.2 gives an overview of the types of crosscutting concerns that were identified and the seeds that led to their identification. Several of these concern types, such as *consistent behavior* or *contract enforcement* [The AspectJ Team, 2003], have more than one instance in JHotDraw; that is, multiple unrelated (crosscutting) concerns exist that conform to the same general description. For example, one instance of *contract enforcement* checks a priori conditions to a command's execution, while another instance verifies common requirements for activating drawing tools. The number of different instances that were detected is indicated in the # column.

We distinguish three different ways in which the fan-in metric can be associated with the crosscutting structure of a concern implementation (also indicated in Table 3.2):

Concern type	#	Seed's description
Consistent behavior	4	Methods implementing the consistent behavior shared by different callers, such as checking and refreshing figures/views that have been affected by the execution of a command.
Contract enforcement	4	Method implementing a contract that needs to be enforced, such as checking the reference to the editor's active view before executing a command.
Undo	1	Methods checking whether a command is undoable/redoeable and the <i>undo</i> method in the super-class, which is invoked from the overriding methods in subclasses.
Persistence and resurrection	1	Methods implementing functionality common to persistent elements, such as read/write operations for primitive types wrappers (e.g., Double, Integer, etc.) which are referenced by the scattered implementations of persistence/resurrection.
Command design pattern	1	The <i>execute</i> method in the command classes and command constructors.
Observer design pattern	1	The observers' manipulation methods and <i>notify</i> methods in classes acting as subject.
Composite design pattern	2	The composite's methods for manipulating child components, such as adding a new child.
Decorator design pattern	1	Methods in the decorator that pass the calls on to the decorated components.
Adapter design pattern	1	Methods that manipulate the reference from the adapter (<i>Handle</i>) to the adaptee (<i>Figure</i>).

Table 3.2: Summary of the results of the fan-in analysis experiment.

1. The crosscutting functionality is implemented through a method and the cross-cutting behavior resides in the explicit calls to this method. Examples in this category include *consistent behavior* and *contract enforcement*.
2. The implementation of the crosscutting concern is scattered throughout the system, but makes use of a common functionality. The crosscutting resides in the call sites, and can be detected by looking at the similarities between the calling contexts and/or the callers. Examples of concerns in this category are *persistence* and *undo* (see Chapter 2).
3. The methods reported by the fan-in analysis are part of the roles superimposed to classes that participate in the implementation of a design pattern. Many of these roles have specific methods associated to them: the *subject* role in an Observer design pattern is responsible to notify and manage the observer objects, while the *composite* role defines specific methods for manipulating child components. In general, establishing a relation between these seed-methods and the complete concern to which they appertain might require a better familiarity of the human analyzer with the code being explored, than for the previous two categories. However, many of these patterns are well-known and have a clear defined structure, which eases their recognition [Hannemann and Kiczales, 2002].

For more details regarding fan-in analysis and a complete discussion of the JHotDraw results, we refer to Chapter 2.

3.4.2 The Identifier Analysis Experiment

Applying the identifier analysis technique of Subsection 3.3.2 on JHotDraw yielded 230 concepts and took about 31 seconds when using a threshold of 4 for the minimum number of elements in a concept. With a threshold of 10, the number of concepts produced was significantly less: only 100 concepts remained after filtering, for a similar execution time.³ In both cases, 2193 elements and 507 properties were considered. It is a good sign that the number of properties is significantly smaller than the total number of elements considered, as it implies that there is quite some overlap in the identifiers of the different source-code entities, which was one of the premisses of the identifier analysis technique.

The manual part of the experiment, i.e. deciding which concepts were real seeds, was much more time-consuming. Overall, this took about three days for the experiment with threshold 4, where 230 seed candidates needed to be investigated. For each of the discovered concepts, the code of the entities in its extent had to be inspected to decide whether (most of) these entities addressed a similar concern. Other than allowing to

³Whereas the threshold of 4 was chosen arbitrarily, the threshold of 10 was determined experimentally: below that threshold the amount of concepts that were regarded as noise was significantly higher than above the threshold.

Crosscutting concern	Concept(s)	#elements	Some elements
Observer	change(d) check listener release	67 14 65 12	figureChanged(e) checkDamage() createDesktopListener() ...
Command execution	command executed execut(able)	4 51	commandExecuted(...) commandExecutable(...)
Undo	undo(able) redo(able)	53 14	createUndoActivity() redo()
Visitor	visit	12	visit(FigureVisitor)
Persistence	file storable load register	15 5 8 7	registerFileFilters(c) readStorable() loadRegisteredImages loadRegisteredImages
Drawing figures	draw	112	draw(g)
Moving figures	move	36	moveBy(x,y) moveSelection(dx,dy)
Iterating over collections	iterator	5	iterator(), listIterator(), ...

Table 3.3: Selection of results of the identifier analysis experiment.

browse the source code of the elements in the extent of a concept, the DelfSTof code mining tool provided no direct support for this.

Table 3.3 presents some of the seeds discovered by manually analyzing the classes and methods belonging to the extent of the concepts produced by the FCA algorithm. The first column names the concern, the second column shows the identifiers shared by the elements belonging to the concept(s) corresponding to that concern. The third column shows the size of the extent for each concept. Finally, for illustration purposes, the fourth column shows some program entities appearing in the extent of the discovered concepts.

Out of 230 candidate seeds, 41 seeds were retained, when using a threshold of 4 for the minimum number of elements in a concept. These discovered concerns were classified in three different categories:

1. Some of these concerns looked like aspects in the more traditional sense (e.g., *observer*, *undo* and *persistence*).
2. Many other concerns seemed to represent a crosscutting functionality that was part of the business logic (e.g., *drawing figures*, *moving figures*). The distinction between these two first categories was somewhat subjective, however.

Crosscutting concern	Concepts	Methods
Undo	2	36
Bring to front	1	3
Send to back	1	3
Connect text	1	18
Persistence	1	30
Manage handles	4	60
Manage figure change event	3	8
Move figure	1	7
Command executability	1	25
Connect figures	1	55
Figure observer	4	11
Add text	1	26
Add URL to figure	1	10
Manage figures outside drawing	1	2
Get attribute	1	2
Set attribute	1	2
Manage view rectangle	1	2
Visitor	1	6

Table 3.4: Summary of the results of the dynamic analysis experiment.

- Three Java-specific concerns were discovered (e.g., *iterating over collections*) that are difficult to factor out into an aspect because they rely on or extend specific Java code libraries.

3.4.3 The Dynamic Analysis Experiment

The dynamic analysis technique of Subsection 3.3.3 is supported by the *Dynamo* aspect mining tool⁴. The first step required by *Dynamo* is the definition of a set of use-cases. To accomplish this task, the documentation associated with the main functionalities of JHotDraw was used to define a use-case for each functionality described in the documentation. Amongst others, a use-case was created to draw a rectangle, one to draw a line using the scribble tool, one to create a connector between two existing figures, one to attach a URL to a graphical element, and so on. In total, 27 use-cases were obtained. When executed they exercised 1262 methods belonging to JHotDraw classes, so that the initial context for the concept analysis algorithm contained 27 elements and 1262 properties. The resulting concept lattice contained 1514 nodes.

Among the concepts in the lattice, 11 satisfied the crosscutting conditions (scatter-

⁴Available from <http://star.itc.it/dynamo/> under GNU General Public License (GPL).

ing and tangling), described in Section 3.3, for the use-case specific concepts, while 56 (including the 11 above) satisfied the conditions for the generic concepts. Next, both the use-case specific and generic concepts were revisited manually, to determine which ones could be regarded as plausible seeds and which ones should be considered false positives. The criterion followed in this assessment was the following: a concept satisfying the crosscutting conditions is considered a seed if

- it can be associated to a single, well-identified functionality (this usually accounts for the possibility to give it a short description that labels it), and
- some of the classes involved in such a functionality have a different primary responsibility (indicating crosscutting with respect to the principal decomposition).

Of course, due to the nature of crosscutting concerns and the related design decisions, some level of subjectivity still remains (as is the case for the other techniques).

In the end, the list of candidate seeds shown in Table 3.4 was obtained. The four topmost concerns are use-case specific. As apparent from the second column of the table, and as was the case for the identifier analysis experiment, some crosscutting concerns were detected by multiple concepts. In total, among the 56 generic concepts satisfying the crosscutting conditions, 24 concepts were judged to be associated with 18 crosscutting concerns.

The methods associated with each candidate seed (counted in the last column of Table 3.4) are indicative of the “aspectizable” functionality. Although they may be not the complete list (dynamic analysis is partial) and may contain false positives, they represent a good starting point for a refactoring intervention aimed at migrating the application to AOSD.

3.5 Comparing the Results

In this section we discuss some selected concerns that were identified by the different techniques. We selected some concerns that were detected by all three techniques, as well as a representative set of concerns that were detected by some techniques but not by others. This allows us to clearly pinpoint the strengths and weaknesses of each individual technique.

3.5.1 Selected Concerns

Table 3.5 summarises the concerns we selected. The first column names the concern. The other columns show by what technique(s) the concern was discovered: if a technique discovered the concern, we put a + sign in the corresponding column, otherwise a - sign is in the table.

Concern	Fan-In Analysis	Identifier Analysis	Dynamic Analysis
Observer	+	+	+
Undo	+	+	+
Persistence	+	+	+
Consistent behavior / Contract enforcement	+	-	-
Command execution	+	+	+
Bring to front / Send to back	-	-	+
Manage handles	-	+	+
Move Figures	+ (discarded)	+	+

Table 3.5: A selection of detected concerns in JHotDraw.

Observer

The Observer design pattern is an example of a concern reported by all techniques. Other examples include *Command execution*, *Undo* functionality and *Persistence*, whose implementation in JHotDraw is described in detail in Chapter 6. Their identification should come as no surprise, because they correspond to well-known aspects, frequently mentioned in AOSD literature, or to functionalities for which an AOSD implementation looks quite natural.

Concerns identified by all three techniques are probably the best starting point for migrating a given application to AOSD, because developers can be quite confident that the concern is very likely to be an aspect. However, the fact that only four of such concerns were discovered, stresses the need for an approach that combines the strengths of different techniques.

Contract enforcement / Consistent behavior

The *contract enforcement* and *consistent behavior* concerns [The AspectJ Team, 2003] generally describe common functionality required from, or imposed on, the participants in a given context, such as a specific pre-condition check on certain methods in a class hierarchy. An example from the JHotDraw case is the *Command* hierarchy for which the *execute* methods contain code to ensure the pre-condition that an ‘active view’ reference exists (is not null).

We classify these concerns as a combination of contract enforcement and consistent behavior since these types often have very similar implementations, and choosing a particular type depends mainly on the context and on (personal) interpretation.

Fan-in analysis is particularly suited to address this kind of scattered, crosscutting functionalities, which involve a large number of calls to the same method, while the other two techniques potentially miss it. In fact, contract enforcement and consistent

behavior are usually associated with method calls that occur in *every* execution scenario, so that they cannot be discriminated by any specific use-case. On the other hand, identifier analysis will miss those cases where the methods that enforce a given contract or ensure consistent behavior do not share a common naming scheme.

Command execution

This concern deals with the executability and the actual execution of objects whose class belongs to the *Command* hierarchy. Identifier analysis identified a concept which contains exactly the *execute* methods in the *Command* hierarchy. Dynamic analysis identified the classes containing *isExecutable* methods. Indeed, the *execute* methods all have the same name and manual inspection showed they exhibit similar behavior: they nearly all make a super call to an *execute* method, invoke a *checkDamage* method and (though not always) invoke a *setUndoActivity* and *getUndoActivity* method. A similar argument can be made for *isExecutable*.

Hence, whereas identifier and dynamic analysis may not detect the more generic Contract enforcement / Consistent behavior aspect directly, they can identify some locations (pointcuts) where potentially such an aspect could be introduced.

Bring to front / Send to back

The functionality associated with this concern consists of the possibility to bring figures to the front or send them to the back of an image. When exercised, it executes specific methods that have a low fan-in, hence they were not detected by fan-in analysis. Identifier analysis also missed them, because there were not enough methods with a sufficiently similar name to surpass the threshold. Hence, dynamic analysis is the only technique that identified this concern. This example is a good representative of crosscutting concerns that are reported only by dynamic analysis: whenever the methods involved in a functionality are not characterized by a unifying naming scheme (or there are not enough of them), neither do they have high fan-in, the other two techniques are likely to fail.

Manage Handles

A crosscutting functionality is responsible for managing the handles associated with the graphical elements. Such handles support interactive operations, such as resizing of an element, conducted by clicking on the handle and dragging the mouse. This seed is interesting because it is detected by dynamic analysis and by identifier analysis, but in different ways. Identifier analysis detects this concern based on the presence of the word 'handle' in identifiers. Consequently, it misses methods such as *north()*, *south()*, *east()*, *west()*, which are clearly related to this concern, but do not share the lexicon with the others. On the other hand, dynamic analysis reports both the latter methods and (some of) those containing the word 'handle'. However,

since not all possible handle interactions have been exercised, the output of dynamic analysis is partial and does not include all the methods reported by identifier analysis.

The *manage handles* concern was missed by the fan-in analysis because the calls are too specific: they are similar but different calls instead of one single called method with a high fan-in.

Moving figures

The three techniques discard concerns on different bases: some of the concerns are filtered automatically while others are excluded manually. The *move figures* concern, seeded by the *moveBy* method in the *Figure* classes, is one example where different, subjective decisions can be made depending on whether the concept is classified either as a candidate aspect or as part of the principal decomposition. The *moveBy* methods allow to move a figure with a given offset. The team which used fan-in analysis argued that the original design seems to consider this functionality as part of a *Figure*'s core logic. The other two teams considered it as part of a crosscutting functionality and included it in the list of reported seeds.

This example highlights the difficulty of deciding objectively on what is and what is not an aspect and corroborates our choice to conduct a qualitative, instead of a quantitative, comparison.

3.5.2 Limitations

As a consequence of applying each technique to the same case, some of the limitations of the respective techniques have become obvious. For example, we obtained a better idea of potential 'false negatives', i.e. concerns that were not identified by a particular technique but that were identified by another. Below, we summarise some of the discovered limitations. In the next section we then describe how to partly overcome these limitations by combining different techniques.

Fan-in analysis mainly addresses crosscutting concerns that are largely scattered and that have a significant impact on the modularity of the system. The downside of this characteristic is that concerns with a small code footprint and thus with low fan-in values associated, will be missed. For example, the identification of *Observer* design pattern instances is dependent on the number of classes implementing the observer role. These classes contain calls to specific methods in the *subject* class for registering as listeners to the subject's changes. The number of observer classes will determine to a large extent the number of calls to the registration method in the subject role. A collateral effect is the anticipated unsuitability of the technique for analysing small case studies.

Identifier analysis tends to produce a lot of detailed results. However, these results typically contain too much noise (false positives), so a more effective filtering of the discovered concepts, as well as of the elements inside those concepts, is needed. In

Technique	Concerns
Dynamic analysis	18
Fan-in analysis	16
Dynamic analysis \cup Fan-in analysis	30
Dynamic analysis \cap Fan-in analysis	4

Table 3.6: Concerns identified by either dynamic or fan-in analysis.

addition, the discovered concepts are often incomplete, in the sense that they do not completely “cover” an aspect or crosscutting concern. Often, more than one concept is needed to describe a single concern, as was the case for the *Observer* aspect. The individual concepts themselves may also need to be completed with additional elements that are not contained in those concepts. This was the case for the *Undo* aspect: in addition to the methods with ‘undo’ or ‘undoable’ in their name, some of the methods calling these undo methods need to be considered as part of the core *aspect* as well.

Dynamic analysis is partial (i.e., not all methods involved in an aspect are retrieved), being based on specific executions, and it can determine only aspects that can be discriminated by different execution scenarios (e.g., aspects that are exercised in every program execution cannot be detected). Additionally, it does not deal with code that cannot be executed (e.g., code that is part of a larger framework, but that is not used in a specific application).

3.5.3 Complementarity

The three proposed techniques address symptoms of crosscutting functionality, such as scattering and tangling, in quite different ways. As shown in Table 3.6, fan-in analysis and dynamic analysis show largely complementary result sets: among the 30 concerns identified by either dynamic or fan-in analysis, only 4 are identified by both techniques. This is an expected result. Fan-in analysis focuses on identifying those methods that are called at multiple places. However, when a method is called many times, it is likely to occur in most (if not all) execution traces. Hence, no specific use-case can be defined to isolate the associated functionality, and dynamic analysis will fail to identify it as a seed.

Identifier analysis is the least discriminating of the three techniques and has a large overlap with the other two techniques. When a concern can be identified through fan-in analysis and/or dynamic analysis, identifier analysis can often isolate it too, since a common lexicon is often used in the names of the involved methods.

In the next section, we will use these observations to propose a new aspect mining technique that is a clever combination of the three individual techniques.

3.6 Toward Interesting Combinations

Based on the discussion in the previous section, this section presents three combined aspect mining techniques and reports on the results of applying these combined techniques on the JHotDraw application. Based on the analysis indicators of *recalled methods* and *seed quality* we compare whether these combined techniques provide a more complete coverage of the detected concerns than each of the original techniques individually.

3.6.1 Motivation

As has been explained in the previous sections, the fan-in analysis and dynamic analysis techniques are largely complementary, and address different symptoms of cross-cutting. An obvious and interesting combination of these techniques thus consists of simply applying each technique individually and taking the union of the results. Additionally, the seeds in the intersection of the results (if any) are likely to represent the best aspect candidates, because both techniques identify them. This was illustrated in our experiment, in which both techniques identified the *Observer*, *Undo*, *Persistence* and *Command execution* candidates.

As for other combinations of the techniques, two interesting observations were considered. First, the manual intervention required by identifier analysis is very time-consuming and is not justified by the fact that it produces more interesting results. This makes the technique less suited than the others for large(r) cases. Second, both fan-in analysis and dynamic analysis identify only candidate seeds that serve as a starting point for seed expansion. Dynamic analysis in particular suffers from this problem as it is based on a (necessarily partial) list of execution scenarios. Similarly, fan-in analysis is only focused on invocations of high fan-in methods, which represent just a portion of the whole concern. Interestingly, while performing fan-in analysis and dynamic analysis, we observed that the classes and methods in the seed expansion often exhibited similar identifiers.

Consequently, we believe better results can be obtained if we use identifier analysis as a seed expansion technique for the seeds identified by either fan-in analysis or dynamic analysis, or by the seeds identified by both these techniques. In this way, the search space for identifier analysis is reduced significantly, and more automation is provided for the manual seed expansion needed by both fan-in analysis and dynamic analysis. A final manual refinement step is anyway necessary, since the expanded seeds may contain false positives and negatives.

In the remainder of this section, we will present three different techniques: a combination of fan-in analysis with identifier analysis, of dynamic analysis with identifier analysis, and of the union of fan-in analysis and dynamic analysis with identifier analysis.

3.6.2 Definition of the Combined Techniques

The combined techniques work as follows:

1. Identify interesting candidate seeds by applying fan-in analysis, dynamic analysis or both to the application;
 - For candidate seeds identified by dynamic analysis, (manually) filter out those methods that do not pertain to the concern;
2. For each method in the candidate seed, find its enclosing class, and compute the identifiers occurring in the method and the class name, according to the algorithm used by identifier analysis;
3. Apply identifier analysis to the application, and search for a concept, among the concepts it reports, that is “nearest”. The nearest concept is the concept that contains most of the identifiers generated in the previous step. If more than one nearest concept exists, take the union of all their elements.
4. Add the methods contained in the nearest concept(s) to the candidate seed.
5. Revise the expanded list of candidate seeds manually to remove false positives and add missing seeds (false negatives).

In what follows, we experimentally validate these techniques on the JHotDraw case.

3.6.3 Analysis Indicators

Before applying the combined techniques, we define two measures to validate the results. A common way to measure classification techniques is to use *precision* and *recall* as performance indicators. Unfortunately, this requires information about all crosscutting concerns present in the application, and this is not available. Therefore, we have chosen alternative metrics, which we use for measuring the quality of the individual *seeds* obtained using the various techniques. We call these metrics *recalled methods* and *seed quality*.

Recalled methods is the number of methods reported in a seed that actually belong to the crosscutting concern.

Seed quality is the percentage of a seed’s recalled methods with respect to the total number of methods in the seed. This indicator estimates how difficult it is to spot a concern in the methods provided by the seed.

With respect to the definitions above, it is important to remark that for fan-in, two interpretations of seeds are possible: the first takes only the callees with high fan-in into account; the second interpretation includes, besides the callees with high fan-in, also all callers to these methods. These differences stem from the fact that the fan-in technique is actually based on the *call-relation* and the interpretations use either one or both sides of the relation in seed representations. During exploration these differences are not that important because we can easily navigate from caller to callee and vice versa. However, when we start assessments based on counting elements, these interpretations do have considerable impact.

In the first case, the number of recalled methods will be low (since call-sites are not considered in the seeds), and the seed quality will always be 100% since the high fan-in callees belong to the concern by definition. The second interpretation will result in higher values of recall and yields a more complete picture of the concern. However, lower values for seed quality are possible since not all calls may be caused by a crosscutting concern.

The next section describes the results of applying combined techniques on the JHot-Draw application, and evaluates the above indicators before and after the experiment. We include results for both interpretations of fan-in seeds discussed above.

3.6.4 Experimental Results

Table 3.7 shows the values of the indicators before and after the completion experiment (based on the first interpretation of seeds for fan-in). Although the completion technique can be applied to all concerns identified by either fan-in analysis or dynamic analysis, we performed the experiment only on the concerns identified by all three techniques. The sole reason is that we need to assess how the completion technique influences the recalled methods and seed quality indicators as compared to their initial values, which can only be done for the *Undo*, *Command execution*, *Persistence* and *Observer* concerns.

When looking at the common results, it is important to note that fan-in seeds point to distinct crosscutting concerns *sorts* that can occur as parts of more complex structures like implementations of the *Observer* pattern [Marin et al., 2005c,a], a topic we will explore in depth in the next chapter. In the experiments, these are grouped to obtain the same level of granularity obtained by the other techniques.

A deeper look into the results of the completion with identifier analysis reveals interesting information: For the *Undo* concern, the results of both fan-in analysis and dynamic analysis improve a lot in terms of recalled methods (from 23 and 3 up to 183 and 94). There is a negative impact on the seed quality for (completed) dynamic analysis (from 64% down to 55%), but the seed quality for fan-in plus identifier analysis remains at 100%. For the *Command execution* and *Persistence* concerns, the number of recalled methods increases significantly for the completion technique (from 20 and 3 up to 132 and from 29 and 6 up to 104), while the seed quality remains at the same level.

Concerns	Undo		Command execution	
Technique	Recalled Methods*	Seed Quality*	Recalled Methods*	Seed Quality*
Dynamic analysis	23	64%	20	80%
Fan-in analysis	3	100%	3	100%
Dyn \cup Fan-in	24	63%	22	81%
Dyn + Identifier	183	55%	132	80%
Fan-in + Identifier	94	100%	132	80%
(Dyn \cup Fan-in) + Identifier	183	55%	132	80%

Concerns	Persistence		Observer	
Technique	Recalled Methods*	Seed Quality*	Recalled Methods*	Seed Quality*
Dynamic analysis	29	97%	3	100%
Fan-in analysis	6	100%	10	100%
Dyn \cup Fan-in	32	97%	13	100%
Dyn + Identifier	104	100%	121	14%
Fan-in + Identifier	104	100%	146	15%
(Dyn \cup Fan-in) + Identifier	104	100%	146	15%

Table 3.7: Recalled methods and seed quality before and after completion (*based on the first interpretation of seeds for fan-in)

For the *Observer* concern, the results are less encouraging than for the other concerns. Even though the number of recalled methods increases for the completion technique, the quality of the seeds drops to an unacceptable level (from 100% down to 14% and 15%). Clearly, the completion does not provide a good expansion of the original seeds. Closer inspection reveals that no clearly distinctive naming convention has been used to implement the *Observer* concern. The *Undo*, *Command execution* and *Persistence* concerns employ distinctive identifiers such as *undo/undoable*, *execute/command* and *store/storable*, which are used extensively only within the concern implementation. Consequently, the completion provided by identifier analysis gives good seed expansions. However, the identifiers used for the *Observer* concern are the more general *figure/update/...* that are used extensively throughout the application, and not only in the concern implementation. Therefore, identifier analysis is not able to provide a good expansion for the seeds found by the other techniques.

An overview of results based on the second interpretation of seeds for fan-in, i.e. taking also the call-sites into account, is shown in Table 3.8. For the *Undo* concern, we show both the individual values for each of the three high fan-in callees reported as seeds earlier and the recall and seed quality of the combination of these three. The seed quality is lower than 100% in these cases since some of the calls found were not considered to be part of the actual crosscutting concern. For the *Observer* concern we

Seed	Recalled Methods	Seed Quality
Undo (callee #1)	24	92%
Undo (callee #2)	25	88%
Undo (callee #3)	24	83%
Undo (combined)	73	88%
Observer (combined)	83	100%

Table 3.8: Recalled methods and seed quality for fan-in analysis based on the second interpretation of seeds for fan-in

only show the value for the combined high fan-in callees since it would go too far to go over all individual values here. The seed quality is 100% in these cases since there are no calls from outside this concern to the reported callees.

The seeds identified by fan-in analysis and their quality measures, for the aforementioned as well as for the other concerns, are available online⁵.

3.7 Summary and Future Work

The purpose of the chapter was to compare three different aspect mining techniques, discuss their respective strengths and weaknesses by applying them to a common benchmark application, and develop combined techniques based on this discussion.

We observed that all three techniques were able to identify seeds for well-known crosscutting concerns, but that interesting differences arose for other concerns. These differences are largely due to the different ways in which the techniques work. Fan-in analysis is good at identifying seeds that are largely scattered throughout the system and that involve a lot of invocations of the same method, but it cannot be used to analyse smaller applications. Identifier analysis is able to identify seeds when the associated methods have low fan-in, but only if these methods share a common lexicon. The main drawback of this technique is the large number of reported seeds that had to be inspected manually. Finally, dynamic analysis is able to find seeds in the absence of high fan-in values and common identifiers, but the technique is only partial because it relies on execution traces.

We also observed that the three techniques are quite complementary: fan-in analysis and dynamic analysis require a manual effort to expand the seeds into full concerns, whereas identifier analysis covers a large part of a concern, but requires extensive filtering of the reported seeds. Hence, to improve automation of both fan-in analysis and dynamic analysis, and to reduce the search space for identifier analysis, we proposed a combined technique in which seeds from either fan-in analysis or dynamic analysis

⁵<http://swerl.tudelft.nl/view/AMR/CombinationResults>

are expanded automatically by applying identifier analysis. To verify the performance of this combined technique, we applied it to JHotDraw and interpreted the results in terms of two indicators: *recalled methods* and *seed quality*. The measures show that for three out of the four concerns we considered, the combined technique outperforms the individual techniques. In only one case, the combined technique performed worse.

Future work mainly consists of extending our comparison with other aspect mining techniques, and potentially proposing new interesting combinations with such techniques. This will not only allow us to come up with better (combined) aspect mining techniques, but will also allow us to evaluate the three considered techniques even better, as new concerns will be identified that we were not aware of. Additionally, we could come up with extra quality indicators that complement the *recalled methods* and *seed quality* indicators, and empirically establish their validity by considering other benchmark applications as well.

Our analysis of crosscutting concerns in real-life software systems (totaling over 500,000 lines of code), and in reports from literature, shows that many of these concerns are compositions of primitive building blocks, which are atomic crosscutting concerns. Moreover, our study indicated a number of properties that allow for the categorization of these blocks into crosscutting concern sorts. We use the concern sorts to describe the crosscutting structure of many (well-known) designs and common mechanisms in software systems.

In this chapter, we formalize the notion of crosscutting concern sorts by means of relational queries over (object-oriented) source models and describe a number of commonly encountered sorts. Based on these queries, we present a concern management tool called SOQUET, which can be used to document the occurrences of crosscutting concerns in object-oriented systems. We assess the sorts-based approach by using the tool to cover various crosscutting concerns in two open-source systems: JHOTDRAW and Java PETSTORE.

4.1 Introduction

The typically ill-modularized, scattered and tangled implementation of crosscutting concerns in existing software systems is known to be a challenge to understanding, and hence to the maintenance and evolution of these systems. Despite significant research efforts on the design and development of aspect-oriented languages, as well as on concern identification techniques (i.e., aspect mining), there is still little consensus on what exactly constitutes a crosscutting concern, and how such concerns can be recognized, understood, and clearly documented in source code.

The need for a coherent system to address and represent crosscutting concerns occurs for many, different steps towards better management of concerns in source code. For example, we need to be able to consistently document aspect mining results in

order to ensure common benchmarks for comparison and combination of mining techniques. Similarly, a consistent approach to modeling and documentation of concerns helps in exploring existing systems and in becoming aware of the crosscutting concerns that they implement. We believe that (enabling) consistent understanding and documentation of crosscutting concerns in existing code is the key to making such systems easier to comprehend and maintain.

Over the last three years, we have analyzed crosscutting concerns in a range of Java systems, including JBoss, TOMCAT, JHOTDRAW, and the J2EE PETSTORE, totaling over 500,000 lines of code. A detailed description of the crosscutting concerns in the latter three of these systems is provided in Chapters 2 and 3 of this thesis.

In our study, we found several “building blocks” for crosscutting concerns. These show typical, idiomatic crosscutting implementations. An example is the superimposition of a new role on an existing class. An instance of such a role superimposition can be found in the drawing application JHOTDRAW, in which all classes representing figure types that are to be stored on file should implement the “Storable” interface. In AspectJ, such a crosscutting concern would typically be implemented through an “introduction” mechanism.

Another building block we noted involves “consistent behavior”. As an example, again from JHOTDRAW, the `execute` methods as occurring in the *Command* hierarchy consistently invoke their super method in order to check certain preconditions. Another consistent behavior occurs at the end of these `execute` methods, which have to refresh the drawing view upon completion. Here the AspectJ equivalent is a pointcut and advice.

The building blocks we observed are *atomic* concerns, i.e., concerns that cannot be naturally decomposed into smaller, yet meaningful concerns. These atomic concerns can be categorized by distinctive properties, such as their specific underlying relations and implementation idioms in source code. For instance, typical implementations of tracing [The AspectJ Team, 2003], authorization checks [Laddad, 2003b], or notification of listener-objects as part of the *Observer* design solution [Kiczales and Mezini, 2005a,b] follow the same idiom that consists of method invocations. We distinguish each category of concerns that share their implementation idiom as a concern *sort*. A number of concern sorts, which we shall discuss in detail in this chapter, are briefly described in Table 4.1.

The concern sorts can be used on their own, but can also be composed to construct more complex designs or features. For example, the Observer pattern, often used as an example of a design whose implementation is crosscutting, can be seen as consisting of two role superimpositions (one for the Subject and one for the Observer role), and two consistent behaviors (one for the notification and another for the observer registration). Likewise, we have seen several well known complex crosscutting concerns that can be composed from our sorts, like transaction management and undo support.

In this chapter, we introduce the notion of concern sorts and discuss in detail a number of these sorts. Particularly, we set out to provide answers to a range of open questions: What exactly is a “sort”? Is there a way to formalize this notion? Based on

such a formalization, would there be a way to unambiguously identify the occurrence of a sort in source code? Can we offer tool support for documenting crosscutting concerns based on sorts? And, last but not least, how “typical” are the sorts we have proposed? Do these sorts indeed occur in practice? How often? How can certain well known crosscutting concerns as occurring in existing systems be captured using sorts?

Subsequent chapters will focus on specific applications of concern sorts, particularly on consistent aspect mining and refactoring: we shall look at how the sorts’ distinctive properties can be used for identification of sort instances in source code, and for design of re-usable solutions for migration to modular, aspect-based implementations of concerns.

The main contributions of this chapter are:

- We recognize and describe a number of commonly encountered sorts of crosscutting concerns, showing their specific implementation idioms and a significant number of examples of the sorts instances.
- We formalize the notion of crosscutting concern sort by characterizing each sort by a specific query over a model of the source code, and present these queries.
- We present SOQUET, an Eclipse plug-in that implements the queries for each sort, which can be used to document crosscutting concerns in Java applications.
- We provide an in depth study of sorts occurring in existing Java systems. In particular, we use SOQUET to document a variety of crosscutting concerns as presently implemented in the drawing application JHOTDRAW, and the web application PETSTORE. We also show how the sorts can describe the crosscutting concerns in the design patterns solutions discussed by Gamma et al. [1994].

The next sections each cover one of these contributions, after which we conclude with a discussion on these results, a survey of related work, and an outlook towards future work.

4.2 Crosscutting Concern Sorts

Crosscutting concern sorts are generic descriptions of atomic crosscutting concerns that share a specific relation and implementation idiom. Atomic concerns are therefore *instances* of a particular sort, which can serve as building blocks of more complex concerns or designs.

In our analysis of crosscutting concerns in source code, we recognized several sorts of crosscuttingness, which recur in many well known and lesser known crosscutting concerns. In this chapter, we cover in detail six most commonly encountered sorts, discussing their general intent and specific implementation idioms. For each concern sort, we also show various instances, i.e., crosscutting concerns encountered in practice or presented in literature. Moreover, we look at how we can formalize the notion of

Sort	Short description
<i>(Method) Consistent Behavior</i>	A set of method-elements consistently invoke a specific action as a step in their execution.
<i>Redirection Layer</i>	A type-element acts as a front-end interface having its methods responsible for receiving calls and redirecting them to dedicated methods of a specific reference, optionally executing additional functionality.
<i>Expose Context (Pass Context)</i>	Methods in a call chain consistently use parameter(s) to propagate context information along the chain.
<i>Role Superimposition</i>	Type-elements extend their core functionality through the implementation of a secondary role.
<i>Support Classes for Role Superimposition</i>	Type-elements implement secondary roles by enclosing support classes. The class nesting mechanism enforces (and defines) the relation between the role of the enclosing class and that of the support class.
<i>Exception Propagation (Declare throws Clause)</i>	Method-elements in a call chain consistently (re-)throw exceptions from their callees in the absence of an appropriate answer.

Table 4.1: Sorts of crosscuttingness.

concern sorts. To that end, we express the six sorts, shown in Table 4.1, as queries over a meta-model describing object-oriented source code.

4.2.1 The Query Model

Our query model is aimed at providing a standard, formalized description of the relations underlying each of the crosscutting concern sorts. The model consists of a generic query definition and a set of query templates (*sort queries*) that capture the relations specific to each of the sorts.

A sort query is a binary relation between elements of two sets, the *source context* and the *target context*. The elements in these contexts are program elements, such as classes or methods. The relation between them is based on a combination of various source code relations, such as *call* or *inheritance* relations, that can be extracted using static analysis. A query can limit each context by selection clauses that impose restrictions to the elements that participate in the relation.

The elements and collaborations relevant to the sort queries are shown in Figure 4.1. These relations are used to pose restrictions in queries, such as “any method *m* that *is_of_type T*”. The type of a method is considered to be its returned type.

4.2.2 Description and Formalization of Sorts

This section covers the set of six crosscutting concerns sorts shown in Table 4.1. Besides these sorts, we discuss a number of additional concern sorts in Section 4.5, where we show how new sorts can be contributed to our list.

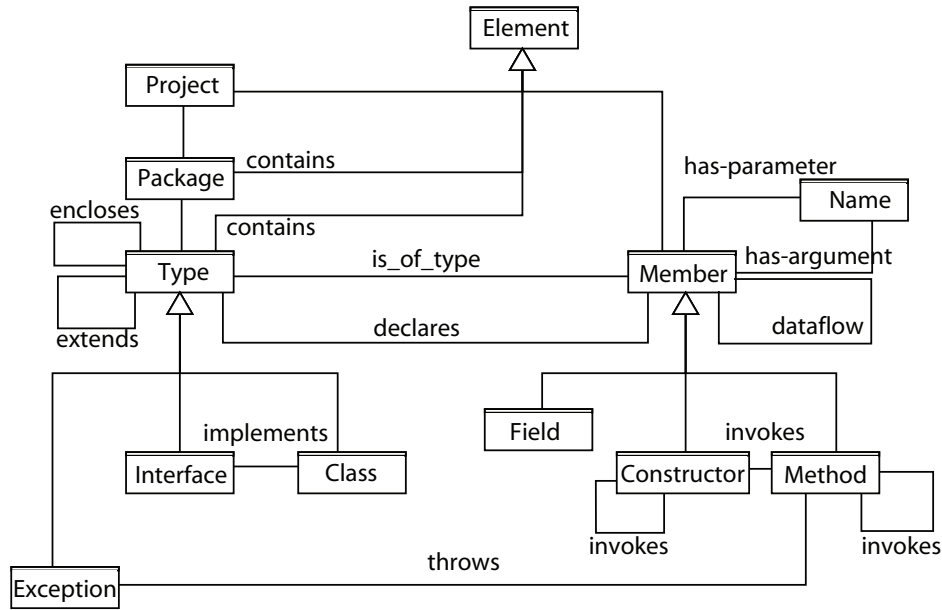


Figure 4.1: Meta-model relevant to sort queries

(Method) Consistent Behavior The crosscutting relation specific to this sort occurs between a set of methods in a defined (source) context and a given action implemented by a method. The methods in the set consistently invoke the action to fulfill a requirement additional to their core functionality.

While the target context is defined through one method, namely the invoked action, the definition of the source context can cover various cases; for example, in the case of a logging concern, we could define the source context as the set of all methods in a Java project. In this example, the definition of the context requires a *project* element. In other cases, the context could cover a type hierarchy, or just the set of methods of a class, etc. Each of these contexts requires a different (type of) element for definition. Our helper function $\text{Context}_{\text{CB}}$ extracts all methods from a given starting point s , which we shall call *context seed*.

We formalize the concern sort and document its instances through a query that takes as input the invoked method and the seed element to define the source context.

$$\text{CB}(\text{Element } s, \text{Method } m) := \{ (m', m) \mid m' \in \text{Method} \cap \text{Context}_{\text{CB}}(s) \wedge m' \text{ invokes } m \}$$

The common idiom to implement instances of this sort in an object-oriented language (particularly Java) consist of scattered method calls (from a defined context) to the method implementing the common action to be executed consistently.

Another example of consistent behavior is the notification mechanism in the Observer pattern: actions that change the state of the Subject have to consistently call the notification method to allow the observers to update their state.

Yet another example, from transaction management, is aimed at maintaining data integrity by ensuring that an operation is committed only when it is fully completed and rolled-back otherwise (e.g., in bank transfers both the debit and credit operations have to succeed to keep the data in a consistent state). Transaction management in Java is supported via JDBC transactions and the Java Transaction API (JTA).¹ A JTA transaction requires that methods implementing the transaction logic consistently invoke dedicated methods of the *javax.transaction.UserTransaction* interface: the *begin* method at the beginning and the *commit* (or *rollback*) at the end to demarcate a JTA transaction. These invocations are instances of the *Consistent behavior* sort.

Other instances of this sort include: logging of exceptions thrown in a system, wrapping service level exceptions of business services into application level exceptions [Marin et al., 2007a], checking credentials as part of authorization mechanisms [Laddad, 2003b], etc.

Redirection layer A redirection layer defines an interfacing layer to an existing object, and acts as a front-end that accepts calls and redirects them to dedicated methods of that object, optionally executing additional functionality. The consistent (yet, method specific) redirection logic crosscuts this layer's methods.

The relation for this sort is between the redirecting layer and the target object, and resides in the consistent redirection of calls between method pairs. The source context is defined by the class acting as a redirector, the target context is the type whose methods receive the redirection. The implementation idiom consists of the identical logic in the methods (of a given class) that redirect their calls to partner methods of a given type.

The query to document instances of this sort is parameterized with the redirecting type and a reference to the object receiving the redirection:

$$\begin{aligned} \text{RL}(\text{Class } c, \text{Member } f) := \{ (m, m') \mid & m, m' \in \text{Method} \wedge \\ & c \text{ declares } m \wedge f \text{ is_of_type } c' \wedge c' \text{ declares } m' \wedge \\ & \text{invokes}(m) \cap \text{methods}(c') = \{m'\} \wedge \\ & \text{invokes}^{-1}(m') \cap \text{methods}(c) = \{m\} \} \end{aligned}$$

where $\text{invokes}(m) = \{m' \mid m \text{ invokes } m'\}$, $\text{invokes}^{-1}(m) = \{m' \mid m' \text{ invokes } m\}$, and $\text{methods}(c) = \{m \mid c \text{ declares } m\}$.

Implementations of the *Decorator* pattern are common examples of instances of this sort. For example, decorators are used in JHOTDRAW to allow to attach elements like borders to *Figure* objects. The decorators for figures extend *DecoratorFigure*, which defines the set of methods to consistently forward their calls to the stored reference of the decorated object. Subclasses of *DecoratorFigure*, like *Border* or *AnimationDecorator*, override its methods to dynamically extend its functionality. Consistent redirection is also common in implementation of patterns like *Adapter* and *Facade*, as well as in wrapper classes [Marin et al., 2007a].

¹<http://java.sun.com/j2ee/1.4/docs/tutorial/doc/>

Expose context (Context passing) Instances of this sort are characterized by methods that are part of a call chain which use an (additional) parameter to pass context along the chain. The caller exposes its context to a callee by passing information to each method in the call stack of that callee. The idiom specific to this sort is the cross-cutting declaration of additional parameters that are used to pass context.

The elements related by this sort are the caller that wishes to propagate the context info and the callees to which the caller passes the argument. We use transitive closure to get a complete description of context passing along the chain.

$$\begin{aligned} \text{EC}(\text{Method } m) := \{ (m_1, n) \mid & (m_1, n) \in \text{Method} \times \text{Name} \wedge \\ & m \text{ has-parameter } n \wedge m_1 = \text{endpoint}(\text{invokes}^+(m)) \wedge \\ & \forall m_2 \in \text{invokes}^+(m, m_1) . \exists n_2 \in \text{Name} . \\ & m_2 \text{ has-argument } n_2 \wedge n \text{ dataflow } n_2 \wedge \\ & m_1 \neq m_2 \leftrightarrow \neg \text{uses}(m_2, n_2) \} \end{aligned}$$

where $\text{endpoint}(\text{invokes}^+(m))$ returns last element of the call-chain started from m , and $\text{uses}(m_2, n_2)$ indicates that m_2 uses the value of n_2 for something else than propagation to its callees (i.e. n_2 occurs in another statement than a call to the next methods on the call-chain towards m_1).

An example instance of this sort is the monitoring of progress for long-running operations, such as in Eclipse applications that employ *IProgressMonitor* objects for this task. These operations are passed a reference to the monitor class through a parameter. They invoke methods of this monitor to indicate progress, like the `worked(int)` method to indicate that a given number of work units of the executing task have been completed. Any sub-operations receive a reference to the same monitor and use it to report their contribution to general progress.

Laddad discusses several other examples of concerns of this sort, as part of transaction management or authorization mechanisms, and proposes an AspectJ solution to improve modularization (the Wormhole pattern) [Laddad, 2003b].

Role superimposition This sort describes the relation between a type, such as a class, and the secondary role(s) implemented by this type. The sort's specific idiom is, therefore, an *implements* relation. Each of the secondary roles corresponds to an additional responsibility attached to the type, for example, due to its participation in multiple collaborations [Riehle and Gross, 1998]. The roles can be defined as distinct interfaces, or just consist of a set of members implemented by the multi-role type.

In order to get those classes that implement an extra role, other than their main one, we specify a seed element for the source context, as well as the role-element. The definition of the context is done similarly to that of the *Consistent behavior* sort. The role-element is specified as an interface or class, or, as we shall see in Section 4.3, as a *virtual interface* if the role does not have a dedicated type.

$$\begin{aligned} \text{RSI}(\text{Element } s, \text{Type } role) := \{ (t, role) \mid & \\ & t \in \text{Type} \cap \text{Context}_{\text{RSI}}(s) \wedge ((role \in \text{Interface} \wedge \\ & t \text{ implements } role) \vee (role \in \text{Class} \wedge t \text{ extends } role)) \} \end{aligned}$$

Common examples of *Role superimposition* occur as part of implementations of design patterns defining specific roles, like the Visitor pattern, or the Observer pattern discussed before.

The implementation of persistence is also possible through role superimposition: the *Figure* elements in JHOTDRAW implement a *Storable* interface which defines the methods for a (figure) object to write and read itself to and from a file. Each figure implements these methods in a specific way to provide persistence and recovery of drawings over work sessions.

Other examples of this sort are based on implementations of multiple interfaces with dedicated, specific roles, e.g., *Serializable*, *Cloneable*, *Undoable*, etc.

Support classes for role superimposition The (object-oriented) mechanism of nesting classes both defines and enforces a relation between the enclosing and the nested class. This allows for superimposition of roles, as a number of elements can share a common role by enclosing support classes of a specific type.

Role superimposition through nested, support classes typically occurs for complex roles and as an alternative to implementation of multiple interfaces: two type hierarchies interact by having elements from one hierarchy as support classes for the elements in the second hierarchy.

The relation between the enclosing and the support class (and their respective roles) can be formalized as:

$$SC(\text{Type } c, \text{Type } role) := \{ (c_1, c_2) \mid c_1, c_2 \in \text{Type} \wedge \\ c_1 \text{ implements } c \wedge c_1 \text{ encloses } c_2 \wedge c_2 \text{ implements } role \}$$

An idiomatic implementation of the sort exhibits interaction of hierarchies through containment of nested classes.

An instance of this sort is present in JHOTDRAW as part of the support for undo functionality. Two type-hierarchies interact through support classes by having the members of one hierarchy enclosing members of the second one. The main hierarchy, *Command*, defines command elements for executing various application-specific activities like, copy and paste, or operations for setting the attributes of a figure, e.g. color or font size. The second hierarchy, *Undoable*, defines operations for undo-ing and redo-ing the results of executing a command. Typically, each Command class encloses its associated Undo class.

Other examples of instances of the sort include implementations of specialized iterators for various Collection types and event dispatcher classes for managing notifications of listeners.

Exception propagation The intent of the sort is described by the consistent propagation of exceptions in a call chain when no appropriate response is available in the callers. Similar to context passing, the relation of the sort applies to a call chain. The callers implement the consistent (enforced) logic of declaring (and re-throwing) exceptions if they are not able to handle them.

To document such a concern, the query needs the method in which the exception originates and the type of exception. Optionally, a context seed can be provided to restrict the set of methods considered for (re-)throwing the exception:

$$\text{EP}(\text{Element } s, \text{Method } m, \text{Exception } e) := \{ (m', e) \mid \\ m, m' \in \text{Method} \cap \text{Context}_{\text{EP}}(s) \wedge m' \text{ invokes+ } m \wedge \\ m \text{ throws } e \wedge m' \text{ throws } e \}$$

The query then returns call relations in a call chain, where each caller re-throws, i.e. declares a *throws* clause for, the exception of its callee.

Common examples of this sort comprise checked exception in Java, such as *IOException* that we shall discuss later in Section 4.4.1. Unhandled checked exceptions need to be declared in a *throws* clause by the users of the method that throws these exceptions.

4.3 Sort-Based Concern Modeling

The queries discussed in the previous section represent atomic concerns. To further describe complex relations and designs in source code, we also need to allow for meaningful compositions of such concerns.

Concern modeling tools support software engineers in modeling their software systems in terms of concerns. Examples of such tools include the Concern Manipulation Environment CME [Harrison et al., 2004], and the Feature Exploration and Analysis Tool FEAT [Robillard and Murphy, 2002]. An empirical study conducted by Robillard and Murphy [2002] suggests that concern models are helpful when performing software change tasks.

Currently, concern modeling tools create rather low level models that are built from concrete source elements from the system that is documented. Some allow for user-defined queries to be attached to these models, although these are rather simple and unstructured queries. We propose the use of *crosscutting concern sorts* to raise the level of abstraction in concern modeling. We integrate sorts into concern models by permitting queries as elements in the concern hierarchy.

As an example, consider Figure 4.2 that illustrates the Observer pattern solution for figure changes in the JHOTDRAW drawing application. The *Figure* elements play the Subject role in the pattern and declare a number of role-specific members, such as the *changed()* method that notifies observers of each change in the Figure's state, the methods to add or remove observer-objects, and the fields in concrete classes to store the reference to the list of observers. Similarly, the observers implement the *FigureChangeListener* interface as an additional, super-imposed role.

The concern model for this design is shown at the bottom of Figure 4.2. The composite *FigureChangeObserver* model groups instances of sorts like *Role superimposition* and *Consistent behavior* to describe the crosscutting concerns in the implementa-

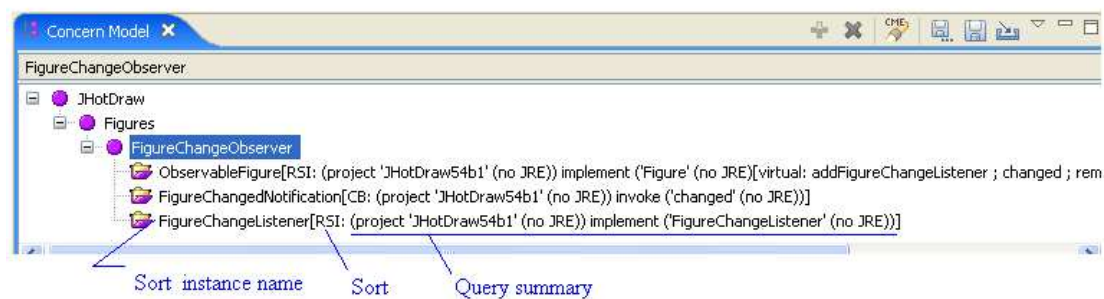
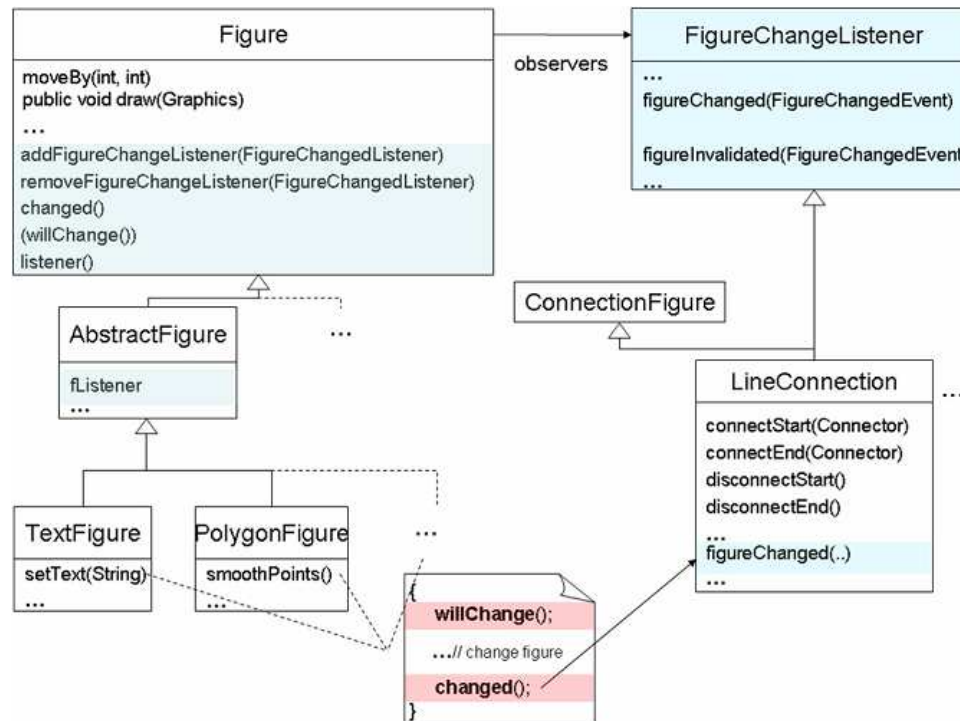


Figure 4.2: Observer for Figure changes and its (partial) sort-based concern model in SoQuET

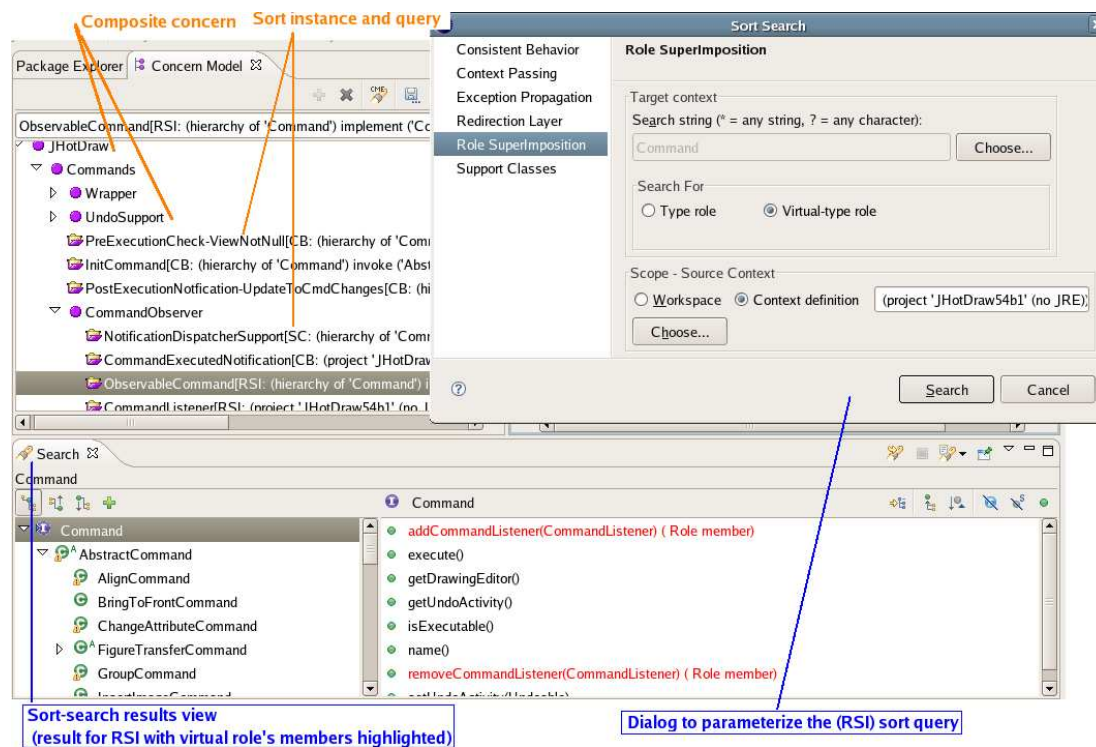


Figure 4.3: SOQUET views and dialogs

tion of the Observer design above. A sort instance is described by a user-defined name and an associated query. A concern model is also described by a given name.

The FigureChanged relation is part of parent, custom-defined relations, like the one grouping all the concerns and design considerations for Figure elements in the JHOTDRAW project. In this case, the project corresponds to the top-level concern model.

4.3.1 SOQUET

To support sort-based concern modeling, we have built an Eclipse plug-in called SOQUET (SORTs QUERY Tool) that is freely available for download.² This tool allows one to describe crosscutting relations in a system based on querying the system's source code for instances of crosscutting concern sorts. These queries can be composed and stored to create persistent, sort-based documentation of concerns in existing code. The tool's main user interfaces are shown in Figure 4.3.

SOQUET assists the user in documenting and/or understanding crosscutting concerns in a system in the following way: First, the user defines a query for a specific sort based on its predefined template. The template guides the user in querying for elements that pertain to concrete sort instances and the user can restrict the query context,

²<http://swerl.tudelft.nl/view/AMR/SoQueT>

for example, by limiting it to a certain inheritance hierarchy.

Next, the results of the query are displayed in the *Sort-search results* view. This view provides a number of options for navigating and investigating the results, like display and organization layouts, sorting and filtering options, links from the query results for source code inspection, etc.

Finally, a *Concern model* view allows one to organize sort instances in composite concerns and describe them by user defined names. The concern model is a tree that defines a view over the system that is complementary to Eclipse's standard *Package Explorer*. The system's sort instances are leaves in this tree and intermediate nodes describe composite concerns. The context menu of an element representing a sort instance includes options to re-run the query documenting that instance and display its updated results. Note that queries can be associated only with sort instances and not to a composite concern. A model can exist at various levels of abstraction and describe complex concerns, system features, or whole projects.

SOQUET introduces the concept of a *virtual interface* to define and describe a role whose definition is tangled within another type and cannot be identified by means of a standard (Java) interface. This mechanism allows the user to create a virtual interface by selecting in a graphical interface those members of the multi-role type, such as methods or fields, that are part of the role of interest.

4.3.2 Documentation of FigureChanged Observer

To build the concern model in SOQUET for the Observer solution summarized in Figure 4.2, we first create a composite concern (*FigureChangedObserver*) to group the crosscutting concerns in the pattern that we like to document. The Subject role for figures changes is defined by a set of methods in the *Figure* interface. The concern is already tangled with the Figure's core functionality and requires the definition of a virtual interface in SOQUET. The interface comprises the set of methods part of the Subject role, like `addFigureChangeListener`, `removeFigureChangeListener`, `willChange`, and `changed`. The seed for the source context is simply the whole JHOT-DRAW project, since all the *Figure* implementations inherit the tangled role. Each sort instance, i.e. atomic concern, is represented by a symbolic name explaining the intent of that concern, as well as a summary of its associated query, in square brackets. The summary of the query consists of a two- or three-letter identifier for the concern's sort, the elements used to define the source and target contexts, and a short description of the sort's specific relation.

Other sort instances document the consistent behavior of notifying listeners of changes occurred in the observed figure and the Observer role defined by the *FigureChangeListener* interface respectively. Furthermore we can include in our documentation the pre-change notification implemented by consistent calls to the `willChange` method, as well as the consistent registration or deregistration of listeners.

4.3.3 SOQUET Support for Software Evolution

The support in SOQUET for persistent documentation of crosscutting concerns, based on systematic queries over the source code, is also aimed at helping the users with software evolution tasks. We can initiate a software change by loading in SOQUET a concern model for the system under investigation and examining the queries in the model. The tool also allows for searching a concern model and displaying only those queries that document concerns associated to a specific element, such as a *Figure*. These queries show that, for instance, Figures are observable elements, and that any modification in their state should be notified by a call to the changed method. Therefore, our changes in the implementation of a *Figure* class, such as adding a method for resizing Figures, need to be consistent with this concern and implement the notification call. While not enforcing the notification call, the documentation in SOQUET allows us to become aware of the notification concern and to understand its implementation by examining the results of the associated query.

4.4 Sorts in Practice

In this section, we look at how sort instances occur in real systems. We describe two cases, totaling around 40,000 non-comment lines of code, from different application domains: JHOTDRAW³ is an open-source drawing application and framework, and PETSTORE⁴ is a sample J2EE e-business application (an on-line shop) developed by SUN. These applications have been regularly used as benchmarks in (collaborative) aspect mining studies, and detailed reports of our findings have been discussed in the previous chapters of this thesis.

The discussion of the first case is structured by the main sorts of concerns encountered in the analyzed system. We discuss a significant number of sort instances to show how “typical” the proposed sorts are and how they occur in practice.⁵ The concern model created for this case can be downloaded from the same web-site as the tool.

The organization of the second case is aimed at showing how well-known crosscutting features and mechanisms can be decomposed and captured using concern sorts.

Table 4.3 on page 106 shows the number of identified and documented sort instances in the two systems.

³ <http://jhotdraw.org/>, version 5.4b1

⁴ <http://java.sun.com/blueprints/>, Java PetStore v. 1.3.2.

⁵ An additional discussion of the occurrences of concern sorts in JHOTDRAW, structured by the various design patterns implementations in this system, is available in Marin [2006a].

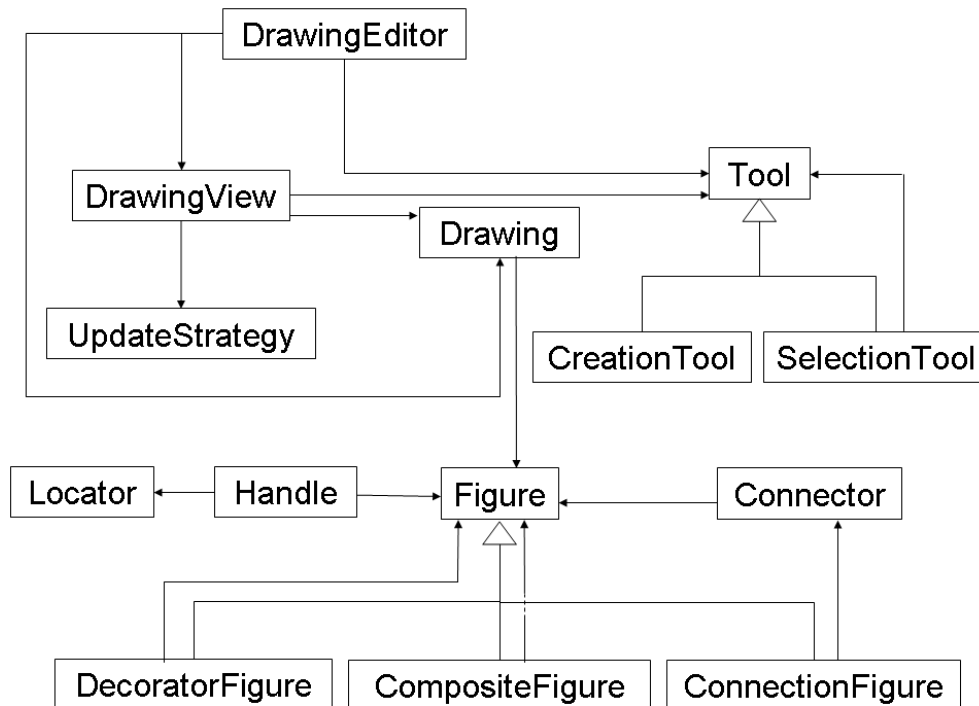


Figure 4.4: Collaborations in JHotDraw

4.4.1 JHOTDRAW

Figure 4.4 shows the core components of JHOTDRAW’s architecture and their collaborations. The *Figure* type generalizes the notion of geometrical and text figures in the application. Figure elements support core operations like drawing and management of their display box. On top of these responsibilities, Figures participate in collaborations that require implementation of multiple roles, such as *observability* for changes, *composability*, and *visitability* for insertion or deletion of figures in composites. Moreover, figures and their enclosing drawings are *persistent* and implement specialized (read and write) methods defined by the *Storable* interface.

JHOTDRAW’s user interface contains menus to execute operations like storing drawings and manipulating figures. It uses a dedicated *Command* hierarchy of around 40 elements to implement these operations. Figures can also be manipulated using *Tool* elements, such as a selection or copy tool. Tools access figures via a common interface realized by *Handle* elements, which act as Figure adapters.

Operations on figures notify listeners, such as drawing views, of changes and support *undo* functionality of such changes. Below we give an overview of the crosscutting concern sorts encountered in JHOTDRAW together with their concrete instances.

Role superimposition We document the multiple responsibilities in the *Figure* elements as distinct instances of the *Role superimposition* sort. Roles like observing or manipulating parts of a composite figure require the use of virtual interfaces since their elements are tangled within the main *Figure* interface. Concerns such as persistence simply require to pass the *Storable* interface as a parameter to instantiate the sort-query. A similar case holds for drawing persistence.

Other instances of *Role superimposition* describe listeners (i.e., observers) for Command and Tool events. These listeners typically implement a dedicated, secondary interface. However, the *Observable* role is tangled with the definition of the core concerns in the top interfaces for the Command and Tool hierarchies respectively. These interfaces also include elements to support such roles as Undo functionality, thus increasing the challenge of distinguishing between the various roles. To document these secondary roles, we define a virtual interface in SOQUET for each of them.

The Command, Tool and Handle elements participate in various design patterns, like Command, State, and Adapter, respectively. We use sort instances to distinctively document each of the roles associated with these patterns in the three elements.

Other elements of our documentation are drawing views and editors that are common event and change listeners.

Consistent behavior and Contract enforcement A large number of concern instances documented for JHOTDRAW belong to the *Consistent behavior* sort. Some of these implement notification mechanisms for the various Observer designs, like drawing and figure changes, tool state changes, etc. Others crosscut the *Command* hierarchy, such as consistently checking that a view is active before command execution, or refreshing that view after execution. The same (named) commands initialize the reference to the associated undo activity prior to their execution, and save the set of figures to be affected by the execution of the command. Similar instances are present in the Tool and Handle implementations.

Several other concerns cut across the Undo activities that consistently conduct a number of checks before execution, like checking the state of the action to be undone.

Consistent behavior is also present in several constructors, for example, those of *Commands* and *Tools*, and in mouse or key handling actions, that implement a shared functionality by means of *super* calls. Each of these concerns crosscuts specific type hierarchies.

Exception propagation The operations to implement persistence, like reading Figure objects from input streams, are designed to throw *IOExceptions* if not successful. This exception is propagated upward in the call chain to the method handling the drawing-recovery command triggered by the user actions, which catches the exception and prints an error message. We document the mechanism in SOQUET through an instance of the *Exception propagation* sort: The query starts from the method generating the exception of the given type, and displays recursively those callers re-throwing the

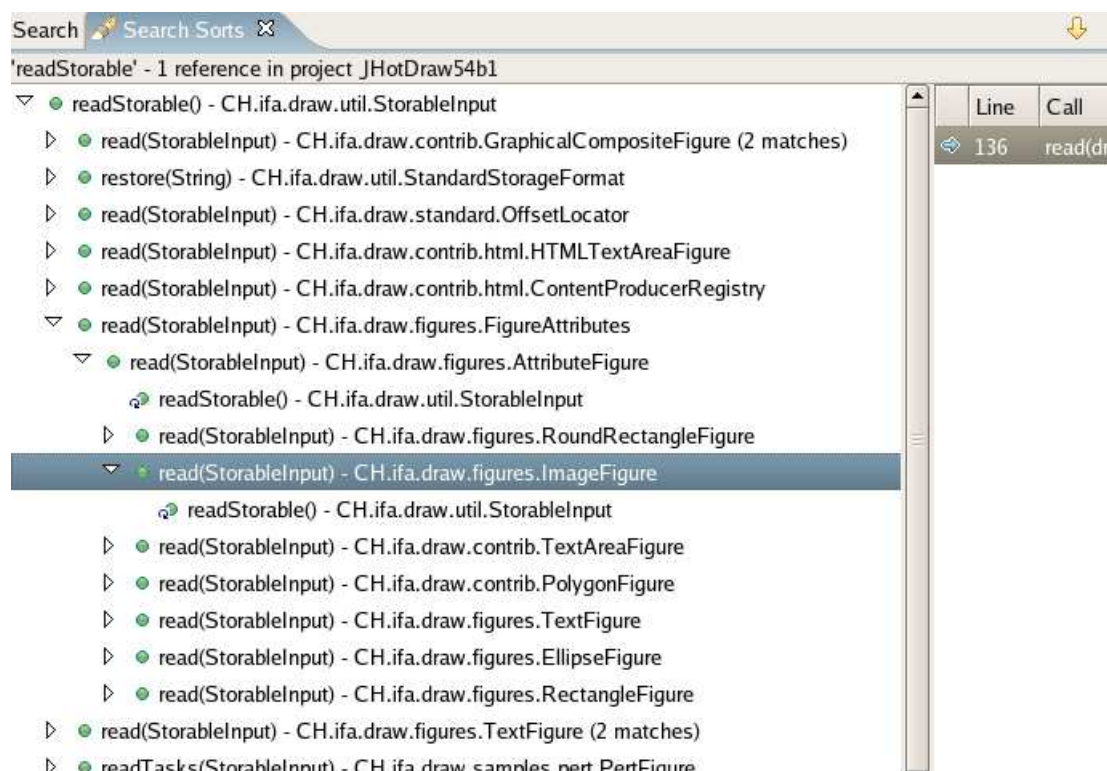


Figure 4.5: SoQuET view for Exception Propagation

exception (by declaring a *throws* clause).

Figure 4.5 shows the results of the query in the SOQUET view; The nodes for the callers re-throwing the exception can be expanded to display their own callers that propagate the same exception further in the call chain.

Redirection layer Besides the figure decorators that were previously discussed, JHOT-DRAW contains a number of redirector instances for event handling and action delegation.

Command invokers implement the *ActionListener* interface whose only method, *actionPerformed*, consistently invokes the *execute* method of the associated command. We document this action delegation for command execution as an instance of the Redirection layer sort. Similar Redirection instances occur in key and mouse listeners which forward the handling of the captured events to associated tools.

Finally, commands that can be undone are wrapped by an *UndoableCommand*, which redirects requests to the wrapped command. We document this concern using a sort query that reports those methods that consistently redirect from the wrapper to the *Command* reference. A similar wrapper is used for *Tool* elements. Redirection layers are also used to reverse undo/redo activities, as in the *UndoRedoActivity* class.

Support classes Commands use instances of the *Support classes* sort to implement undo functionality. Similarly, instances of the same sort are present in the undo support for Tools as well as for Handles.

Other instances of this sort document *EventDispatchers* nested classes that implement more complex notification mechanisms for various Observers, like for *Abstract-Tool*. The support class stores the association between a tool and its list of observers, and notifies the observers of various events.

We use the query for *Support classes* to document these instances, by passing as arguments the defining types of the hierarchies for the nested and the enclosing classes respectively. For the last of our examples of concerns above, these are the *EventDispatcher* and the *Tool* types respectively.

4.4.2 Enterprise Applications

Several mechanisms commonly encountered in enterprise (J2EE) application development, such as transaction management, persistence or component lookup are well-known to be crosscutting and amenable to aspect-oriented solutions [Laddad, 2003b; Colyer et al., 2005]. To elaborate on the coverage of real-life crosscutting concerns by crosscutting concern *sorts*, we discuss a number of these mechanisms as encountered in our second case, PETSTORE. The case is a reference Java web application.

Resource lookup: Service locator and caching, Business delegate, and exception wrapping and handling PETSTORE uses *service locators* [Alur et al., 2003] to provide single access points for resource lookup. The (singleton) implementation of the service locator in the web tier includes a caching mechanism to hold references to enterprise bean home objects or Java Message Service (JMS) resources for re-use. The locator is used by a business delegate (*AdminRequestBD*) to lookup business components. The delegate handles the distributed component lookup, decoupling the business services from their (presentation-tier) clients, and is responsible for catching exceptions thrown by the underlying implementation and converting them to application exceptions. We refer to this mechanism as exception wrapping [Marin et al., 2007a].

The documentation of the caching mechanism is based on two crosscutting concern sorts: First, the caching support in the service locator is a secondary role and an instance of *Role superimposition*. The role elements comprise the map structure used for caching. Second, the component lookup uses *Consistent behavior* to first check the cache for the searched component, and then insert the key in the cache if not present.

The exception wrapping mechanisms, present in both the locator and the business delegate, are instances of *Consistent behavior*, very common in enterprise applications. The concern implementation consists of invoking the constructor of a specific exception type to wrap a caught exception. The delegate class also implements *Consistent behavior* for logging the caught (and wrapped) exceptions.

Persistence strategy PETSTORE allows online purchases from a list of items whose details are stored in an external catalog that is accessed only for creating these lists. The application uses Data Access Objects (DAOs) to read the catalog and keep the data access mechanism hidden.

Client access to the catalog is done through a stateless session bean (*CatalogEJB*), which gets the data (*Item*, *Product*, or *Category*) from its wrapped DAO object (*CatalogDAO*): The data-access methods of the bean forward their invocations to the business methods of the DAO object. The DAO object manages persistence of the (*Serializable*) catalog entries by accessing the Enterprise Information System (EIS).

The serialization of the catalog entries is an instance of *Role superimposition*. To further document the wrapping of the DAO object in the stateless session bean, we need the query for the *Redirection layer* sort. Both the DAO and the session bean elements implement exception wrapping, by catching specific type of exception thrown by their business methods, and re-throwing a different exception type; these concerns are instances of the *Consistent behavior* sort.

The DAO objects managing access to the persistent storage exhibit several other instances of *Consistent behavior*: the business methods consistently require a (JDBC) connection before executing the specific query, and after execution, the connection is closed using a specific method invocation.

Similar mechanisms and concerns are present in other J2EE persistence examples described in literature, such as the use of Hibernate⁶ for persistence by Colyer et al. [2005].

Transaction management Programmatic transaction management is often acknowledged as crosscutting due to the consistent calls to Java Transaction API (JTA) for demarcation of the transaction, which has to execute completely or not at all [Laddad, 2003b]. The transaction mechanism in PETSTORE is similar to the one discussed by [Laddad, 2003b]: by design, the application's web-tier does not benefit from automatic (declarative) transaction management and hence implements it through JTA calls (e.g., the *TemplateServlet* servlet).

Transaction control implies several instances of *Consistent behavior*: first, a lookup action provides a *UserTransaction* object that can be used to begin the transaction by invoking the object's *begin* method. Next, the execution of the operation is followed by a commit if no exception occurred, or by a rollback otherwise. These actions invoke specialized methods on the transaction object.

PETSTORE also uses instances of *Consistent behavior* for (simple) exception logging. Laddad's example exhibits an instance of the same sort to wrap the exception when the lookup operation for the transaction object fails.

The *ServletTemplate* class, which implements a templating service for composing multiple views in one page, dispatches specific requests to an appropriate template

⁶hibernate.org

component by passing its request and response parameters as arguments. We document this mechanism as an instance of the *Expose context* sort.

Order processing center: Transition delegates Transition delegates are part of an asynchronous messaging system implemented by the application for processing customer orders. After an order is received, a number of activities execute specific operations in a predefined sequence; these include sending emails to customers to acknowledge orders, sending order documents to suppliers, or updating orders based on invoice information. At the end of its execution, an activity asynchronously passes a message to the next activity by using a dedicated transition delegate. The delegate knows the successive activity in the workflow to be notified.

The activities are implemented as message-driven beans and trigger the notification by first setting-up their related transition delegate and then invoking the delegate's `doTransition` method at the end of their work. This notification occurs as a distinct concern from the main logic of the activity sending it, and hence we document it as a sort instance, in this case as *Consistent behavior*.

4.5 Sorts in Design Patterns

Crosscuttingness in design patterns has been commonly acknowledged and discussed by various authors, such as Hannemann and Kiczales [2002] who compared Java and AspectJ implementations for the design patterns described by Gamma et al. [1994]. The report on this comparison shows that AspectJ can improve modularity for the implementation of a number of patterns, which we summarize in Table 4.2. The summarized patterns are also covered in this section.

Our analysis of crosscuttingness in design patterns investigates the use of sorts for describing each pattern as a composition of those elements and relations that make the pattern's structure crosscutting, and hence make it more difficult to understand and recognize in source code. We show that instances of the same concern sorts occur in various patterns. This motivates a clear distinction between patterns and concern sorts, where the latter ones describe idiomatic implementations of concerns at a consistent level of granularity that can be shared, among others, by patterns.

Furthermore, we investigate in this section how new sorts can be added to our list, provided that they describe a common implementation idiom, which is distinct from those of the other sorts in the list. While most of the concern models for the discussed patterns can be described by compositions of the sorts shown in Table 4.1, a few of them point us to new sorts that we shall discuss next.

Design pattern	Composition of sort instances
Adapter	<code>Adapter = RSI(contextElem, Adaptee) + RL(Adapter, adapteeReference);</code>
State	<code>State = RSI(contextElem, Context) + CB(contextElemStateChanger, Context.changeState(State)) + RL(Context, stateReference);</code>
Decorator	<code>Decorator = RL(Decorator, componentReference);</code>
Proxy	<code>Proxy = RL(Proxy, fieldRefRealSubject);</code> <i>Protection proxies:</i> <i>document the consistent behavior of checking credentials:</i> <code>CB(contextElem, checkAccessPermission());</code>
Visitor	<code>Visitor = RSI(contextElem, VisitableElement);</code> <i>Specific implementations:</i> <code>Visitor = AV(VisitableElement);</code>
Command	<code>Command = RSI(contextElem1, Receiver) + RSI(contextElem2, Invoker) + IL(Invoker, commandReference);</code> <i>Particular implementations using Command for method objects:</i> <code>AV(Command);</code>
Composite	<code>Composite = RSI(contextElem, Composite);</code> <i>RSI(contextElem2, Leaf)) - not crosscutting</i>
Iterator	<code>Iterator = RSI(contextElem, Aggregate);</code>
Flyweight	<code>Flyweight = RSI(contextElem1, Flyweight) + CB(contextElem2, FlyweightFactory.getFlyweight));</code>
Memento	<code>Memento = RSI(contextElem1, Originator) + CB(careTakerContextElem1, Originator.createMemento));</code>
Strategy	<code>Strategy = RSI(contextElem, Context);</code> <i>sometimes, we could also have:</i> <code>RSI(contextElem1, Strategy);</code>
Mediator	<code>Mediator = RSI(contextElem, Colleague) + CB(contextElem, notifyMediator));</code>
Chain of Responsibility	<code>ChainOfResponsibility = RSI(contextElem1, Handler) + CB(Handler+, Handler.next());</code>
Prototype	<code>Prototype = RSI(contextElem, Prototype);</code> <i>In some languages, like C++, copy constructors are required:</i> <code>DE(contextCloneableObjs, CloneableType.new(const CloneableType&));</code> <i>A similar instance can be used for requiring implementation of the Object.clone method in Java</i>
Singleton	<code>Singleton = RSI(contextElem1, Singleton) + DE(contextElemSingleton, private Singleton.new(..)) + CB(contextElem2, Singleton.instance());</code>
Observer	<code>Observer = RSI(contextElem1, Observer) + RSI(contextElem2, Subject) + CB(contextElem3, notify)+ CB(contextElem1, attachObserver)+ CB(contextElem1, dettachObserver);</code>

Table 4.2: Design patterns as composition of sort-instances.

4.5.1 Interfacing Commands and *Adding variability* to Commands and Visitors

One of the patterns that points to new concern sorts is the *Command* pattern. A number of atomic concerns in this design pattern, can be documented as instances of *Role Superimposition*, like the roles to define the participants in the pattern. These roles include Invokers of the command's action and Receivers that carry out the request, although, for particular implementations, these roles might not be superimposed, nor declare specific members.

Another concern, however, occurs in common implementations of command invokers, like (Java Swing) graphical user interface elements, which store a reference to their associated command and delegate requests to this reference. These invokers “mirror” the state of the command object through their own state, for example, a button element in the user interface that is enabled only if its corresponding command can be executed with the current configuration of the application. This particular wrapping and mirroring of the command's state shows a high (logical) coupling between the graphical button element and the command object, which is due to design considerations.

The relation described above resembles the *Redirection layer* sort by having the graphical element passing requests to its command; however, the discussed concern exhibits a different general intent than that of *Redirection layer*, which is aimed at enhancing functionality dynamically. In this case, the invoker turns into a visual representation of the command's state, in addition to dealing with concerns like graphical display or user action handling. To describe the relation between invoker and command, we introduce a new sort, namely the *Interfacing layer* sort. The sort's query reports all references from the interfacing layer, such as the command invoker, to the object to which it is coupled, such as the command object:

$$\text{IL}(\text{Type } t, \text{Member } m) := \{ (m', t') \mid \\ m' \in \text{Member} \wedge t \text{ declares } m' \wedge m \text{ is_of_type } t' \wedge m' \text{ refers } t' \}$$

Another interesting set of implementations of Commands, particularly those that do not require a persistent state, show a new kind of concerns that cannot be mapped onto any sort in our initial list: *Adding Variability* describes a contract between client-callers and server-callees that make use of method-objects as a substitute for passing references to methods. Instances of the sort implement a consistent mechanism of building and passing method-objects as method arguments. Method-objects are (typically) objects of a type declaring one method. The methods expecting arguments of this type only need and invoke the specific method for the passed object. Languages like Java use this mechanism, which is also referred as *closures* or *functors* or *function objects*, to achieve a behavior similar to the use of callback functions [Bloch, 2001].

Instances of this sort occur in other contexts as well: *Component* elements (like Swing objects), for example, need to execute in a specific thread, i.e. the event dispatching thread, to avoid deadlocks during painting the graphical components. Two

Java dedicated methods, `invokeLater` and `invokeAndWait`, ensure that these components execute in the special thread. The two methods expect an argument of type *Runnable* whose (only) `run` method contains the code accessing functionality of the graphical (Swing) component to be executed. Other examples of instances are discussed in Marin [2006a] as well as in Laddad [2003b]. The latter one proposes a *worker object* solution in AspectJ to address asynchronous method execution or authorization using Java Authentication and Authorization Service (JAAS) API.

In the list of design patterns in Table 4.1, we can recognize instances of this sort among implementations of the *Visitor* pattern, namely for the *Visitable* participants.

4.5.2 Design enforcement in Singleton and Prototype

One crosscutting element that occurs in typical implementations of the *Prototype* pattern is due to the super-imposition of the *Prototype* role. The role declares a specific *clone* method that enables objects to copy themselves. In some languages, like C++, the *Prototype* must declare a copy constructor for cloning.⁷

In Java, the cloning is realized through the *clone* method in the *Object* class, which is a superclass for all the other classes. The class overriding the default `clone` method has to implement the *Cloneable* interface to indicate to the *clone* method that it is legal to make copies of the fields of the *Cloneable* class.

This sort of *Design enforcement* also occurs in *Singleton* classes that have special requirements, most notably, they have to declare the constructor as *private* for not allowing constructor calls from outside the class.

The *Design enforcement* sort discussed above targets declaration of type members required for compliance with design considerations. Its instances can be documented using a query that searches for and emphasizes those declarations of interest, such as *private* or copy constructors. Note that neither Java nor AspectJ can specify *Design enforcement* in other way than by comments.

More examples of instances of this sort include the design of (Java) *bean* objects that are required to declare no-arguments constructors.

In addition to the *Design enforcement* concern, the access to singletons implies an instance of *Consistent behavior*: singletons define an access method to the sole instance of the singleton class, which has to be used by clients instead of calling the constructor.

4.5.3 Other Patterns

The discussion of the rest of the patterns is aimed at investigating how our sorts are able to capture the crosscutting relations that occur in these various designs.

Implementations of the *Adapter* pattern could use either multiple roles or object composition to adapt a class to an interface expected by clients. In the first case, the

⁷A copy constructor receives as (single) parameter a constant reference to the object to be cloned.

Adaptee role is super-imposed to the class implementing the Adapter functionality. The Adapter class implements both a Target interface and (extends) the Adaptee, which is an instance of the *Role superimposition* sort.

The solution relying on object composition would typically use delegation from the Adapter to a stored reference of the Adaptee object. This is an instance of the *Redirection layer* sort.

The *State* pattern comprises a number of crosscutting elements: The *Context* role is super-imposed and has specific members for maintaining a reference to the object defining the current state; second, the notification of changes of the current state to be stored in the Context object is an instance of the *Consistent behavior* sort. The third element is an instance of the *Redirection layer*: the Context object forwards the received calls to the methods of the object storing the current state.

The crosscuttingness occurring in the implementation of the *Decorator* pattern is described by the *Redirection layer* sort. The methods in the decorator class consistently redirect their calls to dedicated methods in the decorated class via a reference stored in the decorator object. A simple decorator is a typical example of a *Redirection layer* instance, and also of a pattern mapping into a single sort. More complex implementations of the pattern, however, require multiple sorts, and hence a composite concern model to document them.

The crosscutting element of the *Proxy* pattern resides in the consistent forwarding of the calls to the reference of the real subject class, stored by the Proxy object. Another crosscutting concern occurs in *protection proxies* as an instance of *Consistent behavior*: this consists of a method call that checks the access permissions before executing the forwarding operation. Some implementations also consistently check if the proxy's subject has been initialized. This check is part of the method for accessing the reference to the subject, which is invoked by the actions in the proxy that forward their calls.

The *Visitor* and *Composite* patterns are often used in combination [Gamma et al., 1994; Hannemann and Kiczales, 2002]. Both patterns define roles that in various implementations are super-imposed, like the *Visitable* and *Composite* roles. The roles, which we document by sort instances, typically define role-specific members.

Certain implementations idioms make use of method objects to allow the methods of the visitor to access the *Visitable* object, and hence its accept method. Such implementations exhibit instances of the *Add variability* sort, as discussed in the previous sections.

A crosscutting element occurring in the implementation of the *Iterator* pattern is the super-imposed *Aggregate* role. The role defines the `CreateIterator()` method to create an iterator object for traversing the elements of the aggregate (structure).

The concerns documented for the *Flyweight* pattern comprise a *Role superimposition* instance for the *Flyweight* role, and a *Consistent behavior* for obtaining references to a (new) flyweight object. This behavior consists of calling the accessor method in the factory class for the flyweight instances, instead of attempting to build new flyweight objects. This behavior is similar to accessing *Singleton* objects via a dedicated method, which we also documented as *Consistent behavior*.

The refactoring of *Memento* pattern to AspectJ discussed by Hannemann and Kiczales [2002] uses the introduction mechanism for superimposing the *Originator* role. In addition to this, we document a *Consistent behavior* instance, namely acquiring a memento object before performing the operation that changes the state.

The *Strategy* pattern defines two roles, namely the Strategy object and the (Strategy)Context. Most commonly, the Context is a super-imposed role, maintaining a reference to the Strategy object (and defining methods to access that reference). In some cases, the Context object delegates the requests from its clients to the Strategy reference.

The *Mediator* pattern implies a super-imposed role (*Colleague*) to store and access the reference to the *Mediator* class. Moreover, each change in the colleague class results in a consistent notification of the mediator for coordinating the other colleague-classes. In some implementations, the *Mediator* role could also be super-imposed.

The participation in the responsibility chain requires the implementation of a Handler role. This role defines the method for handling specific requests and the member to store the reference to the next Handler in the chain. The handler-methods check the request and consistently pass it to the next handler in the chain, according to the *Consistent behavior* concern implemented by all handlers.

The *Observer* pattern is documented as a composition of *Consistent behavior* and *Role superimposition* instances. In addition to the consistent behavior of notifying changes in the Subject's state, we also document the mechanisms for registration and deregistration of observers.

4.6 Discussion

4.6.1 Coverage of the Crosscutting Concerns by Sorts

The list of sorts is open-ended, i.e. new sorts can be added to it, following the rules of the proposed catalog for formalizing concerns, if their relations cannot be covered by the existing sorts. The present catalog covers all concerns that we are aware of in real-life systems, like PETSTORE and JHOTDRAW, as discussed in this chapter. The analysis carried out and the examples accompanying the description of sorts also show that these sorts cover well many of the crosscutting concerns described in the literature on aspect-oriented programming.

It is important to notice that the concerns described by sorts are meaningful on their own, although they can also occur in more complex compositions, like a transaction management mechanism or an Observer design. In fact, common refactoring solutions typically address concern sorts, like introduction of roles, or advice for consistent behavior, which are only presented in a larger context of a specific feature or design [Laddad, 2003b; Hannemann and Kiczales, 2002]. The classification in sorts helps us to describe those crosscutting concerns at a consistent granularity level.

Crosscutting concerns as relations The sort-based approach to crosscutting concerns proposed in this chapter looks at such concerns as implicit relations between two sets of elements (i.e., the source and the target contexts respectively). Each sort is described by a distinctive relation captured by the sort’s specific query. While the relation is common to all instances of the sort, the definition of the context can vary from instance to instance: for example, a `log` method (the target context) can be invoked by, and hence be crosscutting for, all the methods in a system (the source context), while a notification action (target) is invoked by the set of methods changing the state of an Observable object (source). Each of these instances of the *Consistent behavior* sort has particular contexts.

The definition of contexts for describing a concern is a research topic on its own, similar to the definition of *pointcuts* in aspect-oriented languages. Current aspect languages cover definitions based on naming conventions or some structural relations (e.g., type hierarchies), but do not support specification of a context like “all elements changing the state of a Figure object”. SOQUET allows for a number of context definitions, such as class, project, package, or hierarchy, as well as for simple collections of elements. Although the last option is very flexible, permitting any selection of elements, generally, a concise definition based on shared properties of the context’s elements is more relevant for capturing the intent of the crosscutting concern.

Tool usage SOQUET can typically be used from two perspectives, namely, (1) as a tool for consistently *creating* crosscutting concern documentation for a system, and (2) as a tool for *exploring* query-based crosscutting concern documentation that was defined earlier for the system under investigation. In the first scenario, the user is assumed to be acquainted with the concerns to be documented. An example is a developer that wants to explicitly document some relations that are otherwise “hidden” by the object-based decomposition of a given system.

In the second scenario, the user explores a given system by loading (pre-existing) sort-based documentation of application into SOQUET in order to locate and better understand certain crosscutting concerns in the implementation. This documentation highlights policies and contracts in the code that are relevant for software evolution tasks and migration towards aspect solutions.

The main challenges with describing and documenting crosscutting concerns stem from to the flexibility of the tool for defining contexts, as discussed above. SOQUET could be improved by supporting set theoretic operations, such as the union of type hierarchies. In addition, defining contexts using pattern matching on names (e.g., all `set*` methods) is not implemented at the moment.

Adding new sort queries in the current version of the tool is fairly complex, as it relies on the “extension points” mechanism in Eclipse. We are exploring how we can prototype our queries using tools that support more direct source code queries, as discussed in the next section. However, this support is still limited at the moment.

Sort	JHotDraw	PetStore
<i>(Method) Consistent behavior</i>	34	15
<i>Contract enforcement</i>	5	1
<i>Redirection layer</i>	15	2
<i>Expose context (Context passing)</i>	1	1
<i>Role superimposition</i>	32	2
<i>Support classes for role superimposition</i>	5	0
<i>Exception propagation (Declare throws clause)</i>	11	5
TOTAL	103	26

Table 4.3: Number of sort instances.

4.6.2 Using Sorts in Aspect Mining and Refactoring

In the next chapter we shall see how the crosscutting concern sorts are used to define a common framework for aspect mining. The framework uses the sort specific idioms as a starting point for the design of aspect mining techniques and as a reference for defining mining results representation that can ensure consistent comparison of techniques and results. The definition and description of sorts allows us to conduct *idiom-driven aspect mining*, and to engineer techniques that target specific idioms and hence sort instances.

Sort instances also allow us to group elements participating in relevant crosscutting relations, which are not explicit in source code. In this respect, the concern sorts are modular units comparable with aspects. Sorts are mainly aimed at supporting crosscutting concern comprehension by describing atomic elements in a standard, consistent way. However, sorts can be associated with template refactorings and sort queries can help instantiate such refactorings by selecting those program elements that participate in a crosscutting implementation. This can help in refactoring concerns to aspect solutions, as we shall discuss in Chapter 6.

The number of sort instances Table 4.3 shows the number of identified and documented sort instances for each of the two analyzed applications. One observation about the data shown in the table regards the number of sort instances in JHOTDRAW case compared to the PETSTORE one. A reason for this difference is the nature of the two applications: PETSTORE is a J2EE application and a number of (potential) crosscutting concerns can be dealt with by the container, such as declarative transaction management. Therefore, these concerns do not occur in the (Java) source code.

Another observation is that some sorts are (typically) more common than others; examples include *Consistent behavior* and *Role superimposition*. This suggests that aspect language mechanisms aimed at refactoring instances of these sorts could address most of the encountered crosscuttingness. However, as previously mentioned, a major difficulty in dealing with these concerns resides in the ability to define contexts for the sorts relations. This is similar to the challenge of having a flexible and expressive

pointcut definition in an aspect-oriented language, as discussed earlier.

4.7 Related Work

Our approach differs from related work by identifying typical implementation idioms of crosscutting concerns which we formalize as concern sorts. The sorts define a system to consistently describe crosscutting implementation of concerns, which helps us to recognize and document such concerns in source code. Furthermore, the approach aims at emphasizing relations rather than program elements as crosscutting, which, we believe, is a more intuitive way of describing and understanding concerns.

A number of tools support source code querying and exploration for concern understanding. FEAT organizes program elements that implement a concern in *concern graphs* [Robillard and Murphy, 2002]. The user can add elements to a concern graph by investigating the incoming and outgoing relations to and from an element that is part of the concern implementation. The elements in a concern graph are classes, methods or fields connected by a *call*, *read*, *write*, *check*, *create*, *declare*, or *superclass* relation.

Although the tool allows one to add relations to the graph describing a concern, the focus is on the elements participating in the implementation of the concern. The navigation for understanding a concern and incrementally building its graph representation is from a root (class) element to other elements in the relation chain. That is, a concern is described by its elements, and an element is connected to other elements via relations. Unlike FEAT, the sorts-based approach uses relations as the main representation of a concern and builds concern models based on these relations. Moreover, FEAT has no built-in support for describing typical crosscutting relations, focusing on code browsing and organization instead.

The Concern Manipulation Environment (CME) [Harrison et al., 2004] also allows for code querying, and, furthermore, for restricting the query domain similar to context definitions in SOQUET. The CME concern model is persistent and the queries, written in its own (pattern-matching) language Panther [Tarr et al., 2004], can be saved over work sessions. However, neither CME nor FEAT allow for complex queries like the ones we used to describe redirections, (exception) propagations or support classes.

We analyzed the possibility to use CME's query language and its internal code representation for implementing our sort queries. As yet, the syntax of the language is not completely defined and is not fully implemented. Informal communication with CME developers revealed that they see queries as available in SOQUET, as a desired extension for CME.

Alike CME, JQuery is a code browser developed as an Eclipse plugin [Janzen and Volder, 2003]. JQuery uses a logic query language (TyRuBa) similar to Prolog [Deransart et al., 1996]. The TyRuBa predicates supported by JQuery cover all relationships defined by FEAT and include additional ones, such as checking the type of an argument. It also supports additional source relations with respect to FEAT and Panther, such as thrown exceptions.

Despite being more flexible than CME for querying code, JQuery does not allow to save and then re-load a concern model of choice for a given project. The tool is also not suitable for large systems due to performance issues.

Such performance improvements have motivated the work on CodeQuest, a software query tool using Datalog as a query language, implemented on top of a relational database system [Hajiyev et al., 2006]. CodeQuest has recently evolved in SemmleCode⁸, a tool that we plan to experiment with for expressing the relations of the concern sorts presented in this chapter.

Sextant is another tool similar to JQuery, which allows querying different kinds of system artifacts [Eichberg et al., 2005]. The tool represents these artifacts in XML and uses the XQuery⁹ language to query this representation. Our focus so far has been on describing crosscutting relations in source code.

Other approaches to documentation of source code include *intensional views*, which are queries based on logic meta-programming [Mens et al., 2003, 2006]. The work on intensional views also identifies and abstracts a number of (five) use-cases, called “usage patterns”, for which definition of views could be helpful in program development and maintenance. These “patterns” are rather general, such as defining views to verify the use of coding conventions or the coverage of the unit tests. By comparison with the concern sort, they do not denote categories of (crosscutting) concerns or typical idiomatic implementations of concerns. Moreover, the granularity of the views is not defined and lies with the user.

Concern sorts can also be conceptually (and by the level of abstraction) compared to the Java micro patterns discussed by Gil and Maman [2005]. The latter ones aim at capturing traces of design, whereas our sorts aim at capturing traces of crosscutting concerns. Gil and Maman show how many systems are made up of their micro patterns: we show how the crosscutting concerns known to exist in, e.g., JHOTDRAW can be composed from our catalog of sorts.

4.8 Conclusions

This chapter proposes a model for addressing crosscutting functionality in source code based on crosscutting concern sorts. Such a model can provide consistency and coherence for referring to, and describing crosscutting concerns. As a result, sorts are useful in program comprehension and areas like aspect mining and refactoring.

We have described crosscutting concern sorts as relations between sets of program elements and formalized these relations as queries over source representations. We have discussed a selection of sorts in detail and presented the SOQUET tool for documenting sort instances using queries. Last but not least, we have used sorts to analyze crosscutting relations present in systems from two different application domains.

⁸<http://semmle.com/>

⁹www.w3.org/TR/xquery

Chapter 5

A Framework for Evaluating and Combining Aspect Mining Techniques

The increasing number of aspect mining techniques proposed in literature calls for a methodological way of comparing and combining them in order to assess, and improve on, their quality. This chapter addresses this challenge by proposing a common framework based on crosscutting concern sorts which allows for consistent assessment, comparison and combination of aspect mining techniques. The framework identifies a set of requirements that ensure homogeneity in formulating the mining goals, presenting the results and evaluating their quality.

We demonstrate feasibility of the approach by retrofitting an existing aspect mining technique to the framework, and by using it to design and implement two new mining techniques. We apply the three techniques to a known aspect mining benchmark and show how they can be consistently assessed and combined to increase the quality of the results. Furthermore, we position a range of existing aspect mining approaches into our framework, allowing software engineers to interpret and compare the results of these approaches.

5.1 Introduction

Aspect mining research aims at providing techniques and tools that support the identification of crosscutting concerns in existing code. Such concerns are of interest as they are particularly difficult to manage and understand due to their specific lack of modularization and locality. The aspect mining results provide us with first insights into policies and designs whose implementation is crosscutting, and hence challenging for software evolution tasks that have to ensure compliance with these policies.

With a growing number and variety of mining techniques proposed in literature, it becomes increasingly important to aim at consistency and compatibility between these techniques and their results. Such properties would allow for a systematic evaluation of the techniques, an assessment of results and the combination of techniques to improve

quality.

However, most mining techniques rely on non-uniform descriptions of the cross-cutting concerns they aim to identify and of the steps to be taken to map their results onto potentially associated concerns. In some cases, the description of the discovered concerns is specific to the context into which they were encountered, and explained through other, better known, examples of crosscutting functionality (e.g., CORBA Portable Interceptors¹ are described as “observer style entities” [Zhang and Jacobsen, 2003]). Quite often, the mining techniques focus on generic symptoms of crosscuttingness, like tangling or scattering, instead of exploiting specific characteristics of the particular types of concerns they aim to identify. In addition, there is little consistency in describing results and concerns, which makes it hard to compare or combine the results.

Previous experiments aimed at comparing and combining aspect mining techniques [Ceccato et al., 2006] show that a significant challenge rises from the lack of a sound definition of crosscutting concerns. This leads to the following (hypothetical but likely) evaluation scenario: One technique describes its results through the participants in an implementation of the Observer pattern that are crosscut by the super-imposed roles of Subject and Observer [Gamma et al., 1994]. A second technique reports results related to the same instance of the pattern, but identified through the elements implementing the crosscutting mechanism of the observers-notification (that is, the methods changing the state of the Subject object consistently invoke a notification method). Human analyzers interpret the results and agree on ad-hoc convergence rules of them: the Observer pattern instance is counted as a common finding based on the valid results from both techniques, and the argument that the pattern is a well known example of crosscuttingness. Each technique can further explain how the implementation of the Observer is related to its own identification mechanism.

The problems with the sketched scenario are apparent: the convergence relies on an inconsistent level of granularity for the reported findings, as the Observer implementation comprises distinct (atomic) crosscutting concerns that the techniques identify. The results require a tedious manual correlation effort as they do not (always) overlap directly but are related by the design decisions they implement. Moreover, the approach requires that, despite their inconsistency, detailed descriptions of results and associated concerns are present. In practice, however, such descriptions are often not available.

To address these issues, we identify a set of requirements for systematic aspect mining aimed at ensuring consistency and compatibility in identification of crosscutting concerns and description of the mining results. These requirements form the basis of a common framework for aspect mining. They comprise a clearly defined *search-goal* for the mining technique, descriptions of the rules for mapping the mining results onto the description of the concerns targeted by the technique, and objective metrics for assessment.

Contributions of this chapter can be summarized as follows:

¹ Object Management Group - CORBA v3.0.3 specification

- We present a common framework that defines a systematic approach to aspect mining (Section 5.2);
- We introduce two new aspect mining techniques and show how these and a previously proposed technique conform to the proposed framework (Section 5.3);
- We provide tool support for the techniques and their combination (Section 5.5 and 5.4).
- We apply the three techniques to a common benchmark, both individually and in combination, and assess the results and the approach (Sections 5.6);
- We present a survey of existing aspect mining techniques and discuss their compliance with the proposed framework (Section 5.7).

The next section introduces the proposed framework and its elements, such as the metrics to assess aspect mining techniques. Then, we present a set of three techniques that consists of a previous contribution and two new techniques, and show how the framework is used to retrofit and, respectively, design aspect mining techniques. In Section 5.4, we discuss a number of combinations of the mining techniques aimed at improving the quality of their individual results. The tool support for each of the three techniques as well as for their combination is discussed in Section 5.5. We use the tool to conduct idiom-driven aspect mining on a common benchmark application and report on the setup and results of our case study in Section 5.6. Section 5.8 gives an overview of the existing aspect mining techniques and discusses their conformance with the framework, followed by a discussion of the results and lessons learned. The final sections of the chapter present related work, draw conclusions and discuss future work.

5.2 A Common Framework for Aspect Mining

The focus of this work is on systematic aspect mining for *generative* techniques: approaches that identify program elements which participate in a crosscutting concern based on source code characteristics, without using domain knowledge about the system that is analyzed. The identified elements are known as crosscutting concern *seeds*.

Most generative aspect mining techniques contain an automatic step in the analysis. The results of this step are *candidate-seeds* (or *candidates*): results which are proposed as seeds by the tool, but still require human inspection and validation. Rejected candidates are *non-seeds* (false positives).

To ensure consistent and systematic aspect mining, we identify a number of requirements for aspect mining techniques. One of these requirements is to define the targeted categories of crosscutting concerns; that is, the *search-goal* of the technique. We propose to define search-goals using the classification of crosscutting concerns in sorts defined in Chapter 4.

Sort	Intent	Object-oriented Idiom	Relation	Instances
<i>Consistent Behavior</i>	Implement consistent behavior as a controlled step in the execution of a number of methods that can be captured by a natural pointcut	Method calls to the desired functionality	Set of methods <i>invoke</i> a specific action	Log exception throwing events in a system; Wrap/Translate business service exceptions [Marin et al., 2007a]; Notify and register listeners; Authorization;
<i>Redirection Layer</i>	Define an interfacing layer to an object (add functionality or change the context) and forward the calls to the object	Declare a routing layer (wrapper/decorator/adaptor), and have methods in this layer to forward the calls	Set of methods in class <i>forward calls</i> to pair methods in a receiver type	Decorator (pattern), Adapter (pattern) [Hannemann and Kiczales, 2002]; Local calls redirection to remote instances (RMI) [Soares et al., 2002];
<i>Role superimposition</i>	Implement a specific secondary role or responsibility	Interface implementation, or direct implementation of methods that could be abstracted into an interface definition	Set of types <i>implement</i> secondary role	Roles specific to design patterns: Observer, Command, Visitor, etc.; Persistence [Marin et al., 2007a];
<i>Expose context (Context passing)</i>	Expose the caller's context to a callee by passing information to each method in the call stack to that callee (aka Wormhole [Laddad, 2003b])	Add arguments to each method in the call stack	Method (<i>declares and passes</i> parameter as argument (to callee)	Transaction management, Authorization [Laddad, 2003b].

Table 5.1: Sorts of crosscuttingness.

5.2.1 Crosscutting Concern Sorts

An important limitation of aspect mining comes from the lack of a clear definition of crosscutting concerns. For example, Filman et al. [2005] refer crosscutting concerns as “systematic behavior” whose implementation is “scattered throughout the rest of an implementation”, while Kiczales et al. [1997] defines such concerns as “properties” that “cannot be cleanly encapsulated in a generalized procedure”. Unfortunately, these definitions do not allow to clearly specify the search-goals of a technique and the mapping between these goals and the actual results. Without a clear definition, aspect mining techniques have to resort to ad-hoc descriptions of their goals and output and sometimes even omit a detailed specification of their findings and the associated crosscuttingness.

A first step towards overcoming this limitation is a consistent system for addressing and describing crosscutting concerns. To this end, we propose the use of *crosscutting concerns sorts*, a classification system for crosscutting functionality presented in the previous chapter.

Crosscutting concern sorts are categories of *atomic* crosscutting concerns (i.e., concerns that cannot be naturally decomposed into smaller, yet meaningful concerns). They are characterized by a number of properties common to all the instances of the sort, such as a generic description of the sort (i.e., the sort’s *intent*), and a specific implementation idiom of the sort’s instances in a non-aspect-oriented language (i.e., the sort’s specific *symptom*).

Table 5.1 shows a selection of four crosscutting concern sorts. They are described by their defining properties and by a number of concrete instances. For example, the roles super-imposed to participants in a typical implementation of the Observer pattern (the concrete Subject and the Observer roles) are instances of the *Role superimposition* sort. Similarly, the mechanism of consistently notifying Observer objects of changes in the Subject’s state by invoking a notification-method is an instance of the *Consistent behavior* sort.

The classification of crosscutting concerns based on sorts ensures a number of important properties for consistent aspect mining: first, the atomicity of the sorts ensures a consistent granularity level for the mining results; second, sorts describe the relation between concrete instances and the associated crosscutting functionality; third, sorts provide a common language for referring to typical crosscuttingness, and hence for defining the search-goals of an aspect mining technique.

5.2.2 Defining the Common Framework

We propose a common framework for aspect mining that defines a systematic approach to identify crosscutting concerns. The framework is aimed at ensuring consistency of the mining process and compatibility of results. This compatibility would further allow for assessment and combination of mining techniques and results. The framework insists on the following four steps to be taken by the developer of an aspect mining

technique:

Step 1. Define the search-goal of the mining technique. An aspect mining technique has to define its search-goal in terms of kinds of crosscutting concerns that the technique aims to identify. Thus, we use the classification system based on crosscutting concern sorts to define search-goals.

For example, we can define the search goal of our aspect mining techniques as instances of the *Consistent behavior* sort. The underlying relation of the sort consists of method invocations, as shown at the left of Figure 5.1. The same figure (in the middle) also shows two techniques, based on identification of scattered calls and clone detection respectively, targeting concerns of this sort, as we discuss next.

Step 2. Describe the representation of the mining results. An aspect mining technique has to define and describe the format for presenting the results of the automatic mining process (i.e., the source code elements that will constitute the candidate-seeds). Such a common format would typically resemble the specific implementation of the crosscutting concerns targeted by the mining technique (i.e., the sort's implementation idiom).

Step 3. Define a mapping between the mining results and the goal. The mining technique has to define how the candidate-seeds map onto the targeted crosscutting concerns (i.e., the implementation idiom of the targeted sort). This mapping forms the relation between mining results and potentially associated concerns. Furthermore, it describes how we should understand and reason about the candidate-seeds, and how we can expand them into complete crosscutting concern implementations.

Candidate-seeds that cannot be mapped to actual crosscutting concerns are rejected.

Considering again the example in Figure 5.1, the first mining technique, which searches for scattered calls and reports results as call relations, maps the callee in the mining result onto the crosscutting element, and the callers onto the crosscut elements. The second technique assumes that the identified cloned code is extractable into a method that crosscuts its call sites, i.e., the methods enclosing the detected clone.

Step 4. Define metrics to assess mining techniques and results. We distinguish three metrics: (1) *precision*, (2) *absolute recall*, and (3) *seed-quality* metric. The first metric evaluates the quality of the whole set of candidate-seeds generated by the mining technique. The value of the metric is given by the percentage of correctly identified seeds in the whole set of candidates reported by the technique.

The second metric counts the absolute number of identified concern seeds. We use this metric instead of the standard recall because the total number of concerns of a certain sort in a reasonably large system is typically impossible to determine.

The last metric operates at the level of individual seeds rather than at the level of a full technique. It characterizes each seed by a percentage, providing a measure of the

effort required for reasoning about the candidate. The metric was proposed by Marin [2006b] and used by Ceccato et al. [2006] to compare three aspect mining techniques. The quality metric shows what percentage of the (program) elements covered by a mining result belong to the concern associated with that candidate. For example, in Figure 5.1, we assume that our candidate-seed is reported as method call relations, and only four of the six method invocations in our candidate turn out to implement the same (crosscutting) concern (in this case, a condition check realized by an invocation to the *super* method that cuts across the *Command* type hierarchy). The quality of this result is therefore 67%. Candidates with a low value of the quality metric will typically be dismissed.

To generalize the seed-quality metric at the level of the mining technique, we can consider an *average seed-quality* for all the seeds identified by the technique; the metric indicates the level of confidence in the concern seeds identified by a particular technique.

These three metrics form the core set that is used for assessment; however, this set can be extended with other metrics provided that they are generally applicable.

Optionally (but recommended) the mining technique should provide *guidelines for improving the metric values*. Such improvements can be made, for instance, through combinations of techniques. For example, precision can be improved by combining techniques with the same search-goal: same results reported by two or more different techniques are more likely to correspond to valid seeds. Absolute recall can be improved by combining techniques with different goals, which would produce complementary sets of results of different sorts. Seed-quality would typically be improved by generating results that better overlap with the implementation of their associated crosscutting concerns, and hence have a higher confidence level.

5.3 Three Aspect Mining Techniques

In this section, we describe three techniques for identifying crosscutting concern seeds and how they conform with our framework for systematic and consistent aspect mining. One of these techniques, Fan-in analysis, is a previous contribution, while the other two techniques are new. This shows how an existing technique can be retrofitted to the framework and how new techniques can be designed based on the framework's structure. Furthermore, it shows how the framework allows us to reason about the impact (be it positive or negative) that adjusting certain thresholds in any of these techniques has. Experiments that apply these techniques to a common case are discussed in Section 5.6.

5.3.1 Fan-in Analysis

Fan-in analysis is a mining technique aimed at identifying crosscutting concerns whose implementation consists of a large number of scattered invocations of specific function-

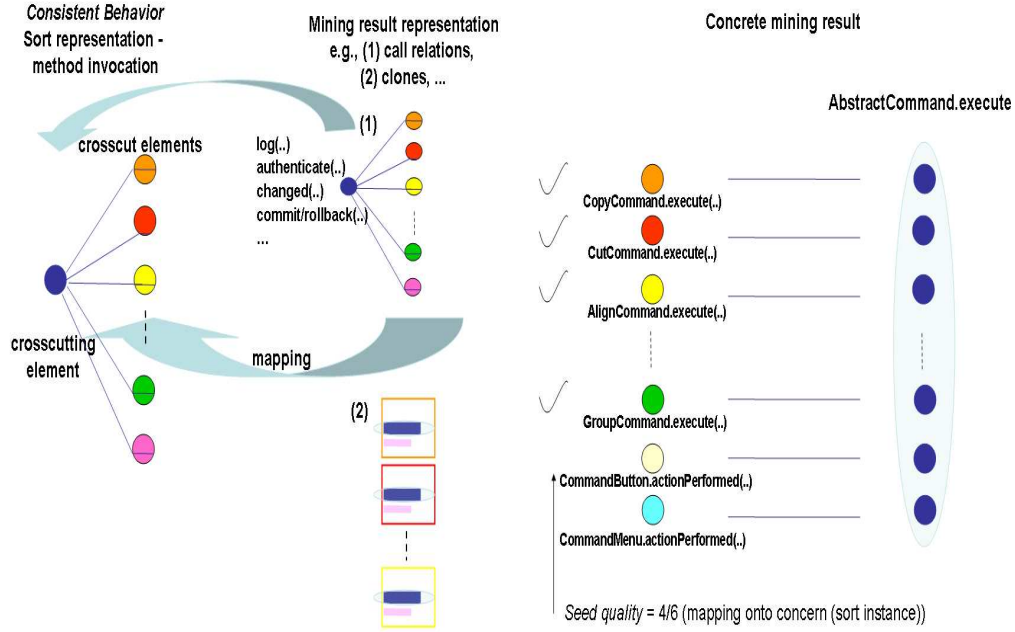


Figure 5.1: Framework elements

ality implemented by a method, as described in Chapter 2 of this thesis. The number of distinct call sites gives the *fan-in* metric of the method invoked. The analysis reports methods with large values of their fan-in metric as candidate-seeds. The seeds found using Fan-in analysis typically correspond to crosscutting concerns refactorable by an aspect-oriented *pointcut* and *advice* mechanism that exists, for example, in AspectJ: the aspect solution captures the call sites in a pointcut definition and triggers the automatic execution of the method with a high fan-in value at these call sites.

Fan-in analysis can identify a number of crosscutting concern sorts. The typical one is *Consistent behavior*, such as events notification in Observer pattern implementations, consistent logging or tracing operations, exception handling and wrapping, credentials checks, etc. Another type of concern that can be identified by Fan-in analysis is *Role superimposition*: for example, the implementation of a secondary role, such as persistence, across a set of classes might consist of methods that invoke a particular helper method; The calls in the persistence methods lead to a high fan-in value for the helper one, which can be recognized by our technique.

To improve assessability, we will differentiate between various Fan-in analyses based on the concern sort(s) that are actually targeted by a particular analysis. This will allow us to distinguish between intended and unintended discoveries.

In this chapter we will focus on Fan-in analysis aimed at identifying *Consistent behavior*. When we describe properties particular to this analysis, we will refer to it as Fan-in_{CC}.

Figure 5.2 describes Fan-in_{CC} in terms of our framework. It lists the goals, and

Search goal Instances of the *Consistent behavior* sort.

Presentation Results are call relations, described by a callee and a set of callers.

Mapping The method with a high fan-in value (the callee) maps onto the method implementing the crosscutting functionality, and the callers of the method correspond to the crosscut elements.

Metrics We consider three metrics for assessment:

- precision: the percentage of seeds for instances of Consistent behavior in the whole set of reported candidates;
- absolute recall: number of identified seeds (i.e., validated candidates);
- seed-quality: the percentage of callers in the reported call-relation that match elements crosscut by the consistent invocation of the method with a high fan-in value. Callers that increase the metric value are those that are validated as participants in the implementation of the associated crosscutting concern.

Figure 5.2: Fan-in analysis represented in the aspect mining framework

provides metrics that can be used to assess the effectiveness of fan-in analysis.

In our previous work we discuss a number of properties that affect the seed-quality for Fan-in analysis [Marin, 2006b]. These include, for instance, structural or call position relations between the callers of a method with a high fan-in value. High quality candidates contain mostly elements that participate in the implementation of a crosscutting concern, and hence are relevant for reasoning about a candidate.

Recall is likely to improve for lower threshold values of the fan-in metric; however, this is also likely to reduce precision.

5.3.2 Grouped calls Analysis

Our next aspect mining technique is based on the observation that the implementation of different crosscutting concerns can be closely related, or that a single concern can be implemented by a number of related method calls. Examples include pre- and post-operation notifications, consistent initialization and clean-up of resources, and multi-step set-up operations. Such concerns typically share their intent and crosscut the same elements. We can identify them by looking for groups of methods that consistently invoke a shared set of callees.

Thus, we propose a new aspect mining technique that we call Grouped calls Analysis. It works by applying formal concept analysis [Ganter and Wille, 1997; Lindig,

Search goal Instances of the *Consistent behavior* sort.

Presentation The results are concepts, where the grouped callees are the attributes and the callers are the objects in the concept.

Mapping The attributes in the concept (i.e., the callees) map onto methods implementing crosscutting functionality, and the objects in the concept (i.e., the callers) match the crosscut elements.

Metrics We consider the same three metrics for assessment as for Fan-in analysis:

- precision: the percentage of seeds for *Consistent behavior* instances in the whole set of reported candidates;
- absolute recall: number of identified seeds;
- seed-quality: is given by two partial measures: (1) the percentage of callers that are indeed crosscut by a consistent call to a specific functionality and (2) the percentage of callees that are part of the crosscutting concern implementation as assessed by an human analyzer. The value of the metric is obtained by multiplying the partial measures.

Figure 5.3: Grouped calls analysis represented in the aspect mining framework

2000] to all calls in the analyzed system in order to find maximal groups of callees that are invoked by the same callers.

The positioning of this technique into our framework is shown in Figure 5.3.

Improving the seed-quality for this analysis can target the set of callers for a reported group of callees, similar to Fan-in analysis, as well as the set of grouped callees, by selecting only those callees that are relevant for a potentially associated crosscutting concern.

5.3.3 Redirections finder

Our third mining technique, Redirections finder, looks for classes whose methods consistently redirect their callers to dedicated methods in another class. Typical examples include implementations of wrapper types, such as in the Decorator pattern [Gamma et al., 1994]: The Decorator class' methods receive calls, optionally add extra functionality, and then redirect the calls to specific methods in the Decorated class.

To detect such a consistent, yet method-specific, redirection concern, the technique looks for classes (C) whose methods (m) invoke specific methods from another class D (D.n). The automatic selection rule is:

C.m calls D.n and only n from D and

Search goal Instances of *Redirection layer*.

Presentation Redirection relations described by a set of pair methods from two different classes, related by one-to-one call relations.

Mapping The callers in the reported set match the methods executing the redirection, while their pair callees receive the redirection.

Metrics We consider three metrics for assessment:

- precision: the percentage of *Redirection layer* seeds in the set of reported candidates;
- absolute recall: number of identified seeds;
- seed-quality: the percentage of redirectors in the reported candidate.

Figure 5.4: Redirections finder analysis represented in the aspect mining framework

D.n is called only by m from C.

Class C and its redirector methods are reported by the technique if the *number* of methods in C complying with these conditions is above a chosen threshold, and if the *percentage* of methods in C complying with the conditions with respect to the total set of methods of C is higher than a second threshold.

To further improve the seed-quality, we can add a filter that checks for matching names between the callers and callees. This is a common practice for implementing redirectors, although it could also introduce false negatives, and hence reduce (absolute) recall.

The representation of this technique in terms of our framework is shown in Figure 5.4.

5.4 Combining Techniques

Having used our framework to describe three techniques, we next can use the framework to reason about *combination* of these techniques. In this section, we discuss the three metrics in our framework, indicating how their values are affected through different combinations of the three techniques just presented.

5.4.1 Improving Precision

Precision is measured by the percentage of crosscutting concern seeds in the complete set of candidates reported by the (automatic) mining technique. A straightforward

combination of two aspect mining techniques that increases precision is achieved by intersecting their results (i.e., the set of candidates). However, this can be done only when the techniques target the same crosscutting concern sorts, with compatible representations of the results.

Two techniques that satisfy this condition are, for example, Fan-in and Grouped calls analyses. To combine them, we select those results of Fan-in analysis whose callees occur as callees in at least one of the Grouped calls candidates.

5.4.2 Improving Absolute Recall

To improve absolute recall, we can simply consider the union of the results of different mining techniques. For techniques that target different concern sorts, this union will not contain overlap in the individual results, and the number of seeds for the combination is the sum of the seeds for each technique.

As argued before, another way of improving the absolute recall is by being less selective, i.e., by lowering the thresholds. However, this is likely to reduce precision. For Fan-in and Grouped calls analyses, precision can be restored by combining these two techniques with the same search-goal, and taking the intersection of their results. The lower thresholds allow for new candidates to be reported and the intersection filters the results so the precision does not drop significantly.

5.4.3 Improving the Seed-Quality

Like precision, the seed-quality metric can be improved by combining techniques targeting the same sort. For example, we can consider the intersection of the results for Fan-in and Grouped calls analyses, selecting the common callees and the common callers of these callees. Thus, for the same value of the threshold for the number of callers, we consider only callees reported by both techniques, and the callers reported by Grouped calls analysis.

Because Grouped calls analysis is the most restrictive of both techniques, the number of callers for a callee is typically lower than for Fan-in analysis. Moreover, the quality of the combined results will be higher than for Grouped calls analysis alone because the combination takes only one callee, and hence we have no false positives grouped together with seeds.

It may be the case that one result of Fan-in analysis occurs in multiple groups of callees reported by Grouped calls analysis: in this case, we select the Grouped calls result for which the callers set has the largest overlap with the set of callers for the Fan-in candidate.

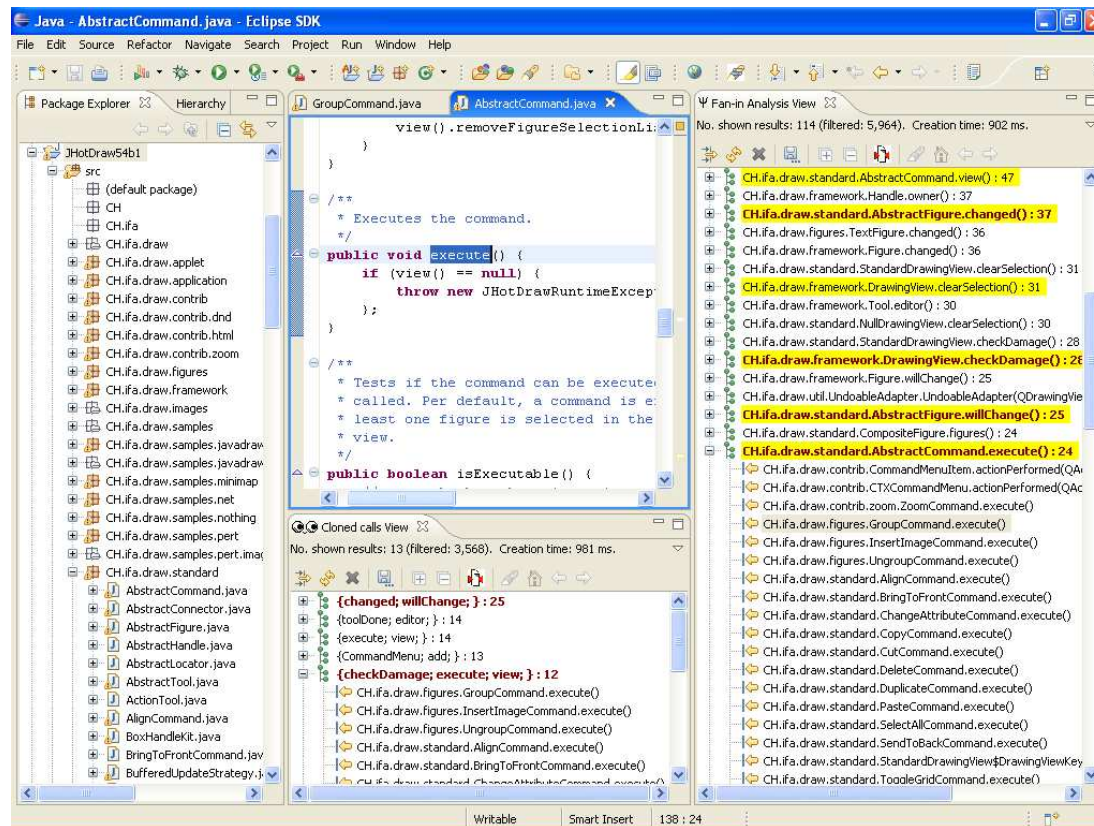


Figure 5.5: FINT views for Fan-in (at the right) and Grouped calls (in the middle, at the bottom) analysis.

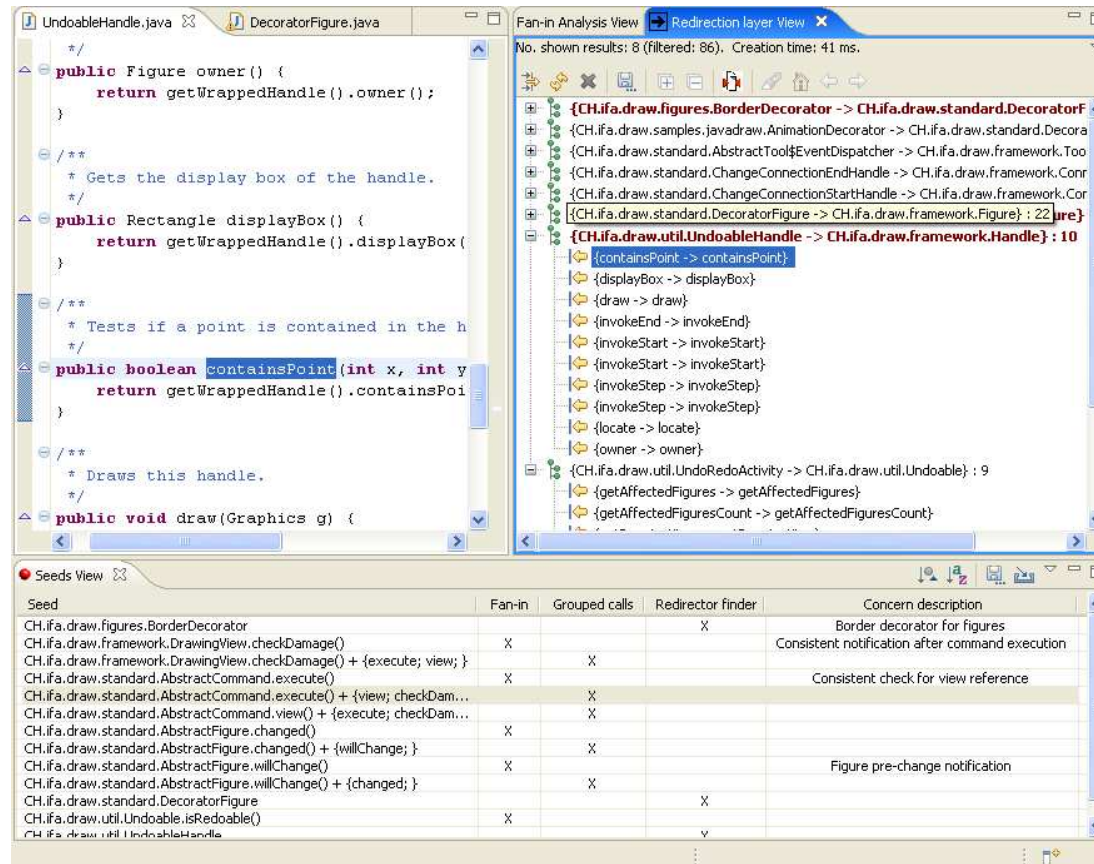


Figure 5.6: FINT view for Redirection finder and the Seeds view.

5.5 Tool Support

To experiment with the ideas laid out in this chapter, we have extended our free aspect mining tool FINT² (see Chapter 2 and Marin et al. [2007a]) to include automatic support for the three techniques and their various combinations. FINT is available as an Eclipse³ plug-in. Figures 5.5 and 5.6 show part of the functionality provided by the tool. Figure 5.5 displays the views to inspect and manage the results of Fan-in (on the right side) and Grouped calls (at the bottom) analysis. The similar view for the results of the Redirections finder technique is shown on the right side of Figure 5.6.

The results for each technique are displayed by following the representation described in Section 5.3. The views allow for various sorting operations and code inspection from the elements selected by the user in the view. The user can further open and inspect each candidate in a new view, and run a number of analyses for improving the quality of the candidate. These analyses include inspection of various structural relationships between the elements describing a candidate.

Support for combining techniques is available, for example, through intersection of the sets of results of two techniques: The views showing the results can be synchronized so common findings are highlighted in the views. For example, the highlighted elements in the Fan-in Analysis View of Figure 5.5 correspond to methods that are also present in at least one group reported by the Grouped calls analysis. The bold colored elements show candidates marked as seeds by the user. These elements are also shown in the Seeds view.

Each technique allows for a number of specific, automatic filters, like filters for *utility* elements or *accessor* methods. Utility elements are those that the user considers as irrelevant for analysis. To filter them, the user is presented with the hierarchical structure of the top-level Java element selected for analysis (e.g., a Java project) in which the elements to be ignored can be selected (e.g., all the elements in packages containing JUnit tests). The accessor methods, that is getter and setter methods, are filtered by automatic inspection of either the signature of the methods or their implementation.

5.6 Experiment

In this section, we apply the mining techniques described above to JHOTDRAW⁴, which has been proposed and used as common benchmark for aspect mining [Marin et al., 2007a; Ceccato et al., 2006] as discussed in Chapters 2 and 3. JHOTDRAW is an open-source framework for bi-dimensional drawings editors. The distribution (v 5.4b1) comes with a default drawing application that we also analyze. The system is also a show-case for applying design pattern solutions in a Java implementation. Its

² <http://swierl.tudelft.nl/view/AMR/FINT> (v0.6)

³ <http://www.eclipse.org/>

⁴ <http://www.jhotdraw.org/>

	Fan-in _{CC}	Grouped calls	Redirection
Targeted sorts	<i>Consistent behavior</i>	<i>Consistent behavior</i>	<i>Redirection layer</i>
Utility filters	Collection wrappers and test classes	Collection wrappers and test classes	Test classes
Accessor filters	Accessors by name and implementation	Accessors by name and implementation	-
Threshold filters	No. callers: 10	No. callers: (1:) 10 and (2:) 7; No. grouped callees: 2	No. redirectors: 3; % redirectors: 50
Accepted Seed quality	$\geq 50\%$	$\geq 50\%$	$\geq 50\%$

Table 5.2: Selection conditions applied for the aspect mining experiment.

size is approximatively 20,000 non-comment, non-blank lines of code.⁵

One goal of this section is to conduct idiom-driven aspect mining and to report on identified concerns in a relevant application. A second and more important goal of this section is to show how the proposed framework allows for consistent assessment of the results of the three aspect mining techniques, as well as of two proposed combinations. To this end, we present and compare the quality measures for each technique and for the results of their combinations.

Detailed results of the experiments discussed in this section and quality metric values are available on-line.⁶ Next, we discuss the setup of the experiment.

5.6.1 Applied Filters

Table 5.2 shows the filters applied for conducting the mining experiments on the selected case-study. For all techniques, we filter out the (JUnit) test classes delivered with the application; i.e., the methods from the test classes do not occur among the reported candidates of any technique, and methods from these classes do not contribute to the fan-in metric of a method.

Collection wrappers, like *IteratorWrapper* or *SetWrapper*, are also marked as utilities to be filtered from the set of candidates. Similar to the test classes, these wrappers are typically part of dedicated packages (*CH.ifa.draw.util.collections.**). Collection elements tend to be frequently used in an application. In most cases, however, they are not part of a consistent mechanism associated with crosscutting functionality. Filtering these elements is likely to reduce the number of candidates without introducing false negatives.

For Fan-in and Grouped calls analysis, we also filter accessor methods from the set of candidates. The filters check both the signatures of methods (*set** and *get** names) and their implementation (i.e., only set a field or return a reference).

A number of threshold values are specific to each case and can be varied by the user to refine the results:

⁵ SLOCCount: <http://www.dwheeler.com/sloccount/>

⁶ <http://swertl.tudelft.nl/view/AMR/CombinationResults>

- Fan-in_{CC}: the threshold value for the number of callers of a candidate is set to 10, following considerations from previous experiments discussed in Chapter 2;
- Grouped calls: the first experiment uses a threshold of 10 for the number of callers, which is lowered to 7 for the second experiment; the threshold for the minimal number of callees to be grouped by a candidate is always set to 2;
- Redirections finder: the technique uses two threshold values, the first sets the minimal number of redirector methods in a class to 3, and the second sets the minimal percentage of methods in the class executing the redirection to 50%. Thus candidates reported by this technique will have at least 3 redirector methods, and at least 50% of all their methods execute the required redirection.

Candidates are marked as seeds if they correspond to a crosscutting concern according to the mapping rules of each technique, and if the seed-quality of the candidate is at least 50%. This metric value shows that most elements in the make-up of a seed belong to the (implementation of the) crosscutting concern identified by that seed; hence, we estimate that concerns associated to such seeds are fairly recognizable by a simple inspection of the seed. We believe that in practice only seeds with a relevant high quality value will be recognized by the user in the set of mining results produced by a technique. This is due to the fact that the goal of aspect mining is to provide a quick insight into the (crosscutting) concerns of a system, and so it is a time-constrained activity, aimed at minimizing the effort of understanding a system. Therefore, the precision of a technique should be measured by a certain standard of the quality of its results. Note, however, that lower values of the seed-quality metric are likely to increase the number of concerns identified by a technique at the cost of increased effort required to analyze each mining result (i.e., each reported candidate).

5.6.2 Results

This section shows a number of typical results and metric values for each of the three techniques, as well as for their combination. The calculated metrics are summarized in Table 5.3.

Fan-in analysis Before discussing the metric values for this technique, we first look at an example of identified seeds and how we select them from the set of candidates. A number of seeds recognized by Fan-in_{CC} implement concerns that crosscut the *Command* hierarchy in JHOTDRAW. *Command* classes follow the design described by the pattern with the same name; they implement an `execute` method that carries out specific activities in response to, for instance, user actions like menu-items selection.

Figure 5.7 shows the method for executing cut operations in a drawing editor. The method starts with a precondition check implemented by the command's super-class

```

//CutCommand.execute()
public void execute() {
    // perform check whether view() isn't null.
    super.execute();

    // prepare for undo
    setUndoActivity(createUndoActivity());
    getUndoActivity().setAffectedFigures(view().selection());

    // key logic: cut == copy + delete.
    copyFigures(view().selection(), view().selectionCount());
    deleteFigures(view().selection());

    // refresh view if necessary.
    view().checkDamage();
}

```

Figure 5.7: (Simplified) execute method in JHOTDRAW's Command hierarchy.

(*AbstractCommand*). Similarly, the (around 20) methods overriding the *AbstractCommand*'s execute method in non-anonymous classes check this condition. The commands then conclude with a notification of the editor's view.

The check- and notification-actions implement two crosscutting concerns scattered over a large number of methods that invoke these actions, and hence increase the value of their fan-in metric. These invocations are typical seeds for instances of the *Consistent behavior* sort.

To calculate the quality of these results, we have to consider all the callers reported by Fan-in_{CC} for each of the invoked actions. Not all the callers, however, belong to the context of the *Command* hierarchy crosscut by the concerns of the two candidate-seeds. For instance, one of the calls to the execute method originates from an action-event handler in a *MenuItem* class. The quality of the candidate for the execute method is given by the proportion of the 18 methods crosscut by the reported call, to the whole set of 24 callers. This value is 75%, above the set threshold of 50% for selecting a candidate as a seed.

Returning to Table 5.3, the *Consistent behavior* seeds identified through Fan-in_{CC} analysis count 33 methods in the total set of 109 candidates reported. This indicates a precision of around 30% for the targeted sorts, as shown in Table 5.3. Despite a lower precision value when compared to the other techniques that we discuss next, we notice that Fan-in_{CC} analysis identifies the largest number of seeds, and hence, of concerns.

Grouped calls analysis As can be seen from Table 5.3, Grouped calls analysis (GC₁) yields fewer candidates and seeds than Fan-in_{CC}, but with a higher precision and qual-

⁷ Results for Fan-in_{CC} in this work are exclusively for the targeted sorts. They differ from results reported in Chapter 2 (and in Marin et al. [2007a]) because in that case more concern sorts were targeted. All results are documented on the experiment's web-page mentioned earlier (footnote 6).

Technique	# Candidates	Absolute recall (# Seeds)	Precision	Average Seed-quality
Fan-in _{CC} (FI) ⁷	109	33	30% (33/109)	77.4%
Grouped calls (GC ₁)	11	6	55% (6/11)	91.6%
Grouped calls (GC ₂)	22	12	55% (12/22)	87%
Redirection finder (RF)	13	12	92% (12/13)	93.5%
FI + GC ₁	17	7	41% (7/17)	97.6%
FI + GC ₂ + RF	-	51	-	

Table 5.3: Metric values for individual and combined techniques.

ity. To understand this, consider the precondition checking and notification concerns just discussed. These two candidates share the largest part of their callers, and hence are also among the results reported by the Grouped calls analysis. Although the two concerns are distinct, they are related by the set of elements they crosscut (i.e., the *Command* hierarchy). The Grouped calls analysis does not separate the two concerns, but instead allows to put them in a single, shared context.

One of the candidates reported by this technique groups the *view* and *execute* methods in the set of callees, together with 14 common callers. Another candidate groups the same two methods, but also the *checkDamage* method, together with 12 common callers. In the first case, the *execute* method is the relevant element for the crosscutting concern associated with the reported candidate. The *view* method has no relevance to this concern, and hence it decreases the quality of the candidate. However, we can still select this candidate as all the callers participate in the associated concern and hence the overall quality (for the callees and callers groups) is 50%.

On the second case, each invocation of the *view* method occurs together with a call to the *checkDamage* method, which is a seed for the previously discussed instance of the *Consistent behavior* sort. In this case, the reported *view* method is relevant for the crosscutting concern associated with the reported candidate and contributes to the quality metric. For this candidate, the quality metric is 100% as all the grouped callees and callers belong to the implementation of the related crosscutting concern.

In comparison with Fan-in analysis, the number of results and seeds for this technique is lower for the same threshold for the number of callers. This is to be expected, as this technique has more restrictive selection rules for the candidates: a callee should not only have a large number of callers, but it also has to be called together with at least one other same method.

For a lower threshold, namely 7, the number of seeds is almost double, but is still lower than the one reported for Fan-in analysis. The results of this experiment are labeled with GC₂ in Table 5.3. This experiment allows to consider callees that are potentially missed by fan-in analysis due to its higher threshold filter. Most of the results and seeds of the Grouped calls analysis with a lower threshold overlap with the results of Fan-in analysis, although, we also observe a number of new seeds.

```

public abstract class DecoratorFigure {
    // ...
    private Figure myDecoratedFigure;

    public TextHolder getTextHolder() {
        return getDecoratedFigure().getTextHolder();
    }

    public Rectangle displayBox() {
        return getDecoratedFigure().displayBox();
    }

    // Forwards draw to its contained figure.
    public void draw(Graphics g) {
        getDecoratedFigure().draw(g);
    }

    // ...
}

```

Figure 5.8: (Part of) DecoratorFigure - super-class for Figure objects decorators.

Redirections finder analysis We observe in Table 5.3 that this technique has the highest precision of all the techniques employed by our experiment. The particular high precision has also been confirmed by experiments on other case-studies, such as Tomcat⁸ and JBoss⁹.

A typical example of concerns found through Redirections finder is the Decorator. A number of classes in JHOTDRAW, like *Border-* or *Animation-Decorator*, extend the *DecoratorFigure* class shown in Figure 5.8, which provides the basic functionality to forward calls to a decorated *Figure* object. This example is a typical instance of the *Decorator* pattern: Methods in the Decorator classes consistently redirect their callers to dedicated methods of a target object, before or after (optionally) providing additional functionality.

The Redirections finder candidate for this concern consists of 22 call relations, 3 of which correspond to the methods shown in the figure. Since all reported results implement the redirection concern, the seed-quality of this candidate is 100%.

Combination of Fan-in and Grouped calls analysis Table 5.3 also shows that the combination of the two techniques (FI and GC₁) targeting instances of the same (*Consistent behavior*) sort leads to improved precision when compared with the results of the individual technique (Fan-in_{CC}). However, this comes at the cost of a significantly lower absolute recall, of only 7 for the combination.

The quality for each of these 7 seeds is listed in Table 5.4. As expected due to

⁸ <http://tomcat.apache.org/>

⁹ <http://jboss.org/>

Candidate	FI quality	GC ₁ quality	FI+GC ₁ quality
framework.DrawingView.checkDamage	64%	100%	100%
framework.DrawingView.clearSelection	55%	100%	100%
framework.DrawingView.selectionCount	63%	83%	83%
standard.AbstractCommand.execute	71%	100%	100%
standard.AbstractFigure.changed	100%	100%	100%
standard.AbstractFigure.willChange	100%	100%	100%
util.UndoableAdapter.undo	92%	100%	100%

Table 5.4: Values of the quality metric for individual and combined techniques.

the more restrictive selection rule of the candidates, Grouped calls analysis achieves a better seed-quality, which is also reflected in the results of the combination. These results show improved values of the quality metric, typically by retaining the higher value of the Grouped calls analysis results.

All three techniques The combination of all the three techniques is aimed at getting the largest possible set of seeds. Therefore, we select for combination the results of Grouped calls analysis obtained by applying the lower value of the threshold for the number of shared callers, i.e., 7. The result of the combination is given by the union of the sets of seeds identified by the three techniques. This union consists of 51 distinct seeds for various concerns in the analyzed system. The other metric values are not relevant for such combinations of techniques that target different concern sorts, as the combination is applied after selection of concern seeds for each targeted sort.

5.7 Retrofitting Existing Techniques

In this section we investigate how existing aspect mining techniques can be retrofitted to the proposed framework. This survey helps in:

- (1) the interpretation of the results of an aspect mining technique;
- (2) clarifying how seeds are translated to crosscutting concerns;
- (3) investigating new combinations of aspect mining techniques.

Below, we discuss for each sort the most important aspect mining techniques that target crosscutting concerns of that sort. The techniques we discuss are summarized in Table 5.5.

5.7.1 Role Superimposition

For detecting role superimposition, two techniques have been proposed: one employs static analysis, while the other one is based on dynamic analysis. The technique using

Technique	Sort search goal	Presentation	Mapping
Aspectizable interfaces [Tonella and Ceccato, 2004b]	<i>Role superimposition</i>	Inheritance relations described by groups of methods that belong to or can be abstracted into an interface definition, and their implementing types.	The reported interface and its members map onto elements that crosscut the types implementing them.
Concepts in traces [Tonella and Ceccato, 2004a; Ceccato et al., 2006]	<i>Role superimposition</i>	Set of methods in a type hierarchy defining the superimposed role, and their implementing, crosscut classes.	The methods map onto the members of the superimposed type and cut across their implementing classes.
Clone detection [Shepherd et al., 2005a; Bruntink et al., 2004]	<i>Consistent behavior</i>	Set of relations (and statements) grouped by a code fragment that is duplicated in multiple method bodies (and that is refactorable by a method extraction).	The method to extract the cloned fragment maps onto the crosscutting element; the methods containing the cloned code fragment map onto the elements being crosscut.
Execution patterns (dynamic [Breu and Krinke, 2004] and static [Krinke, 2006])	<i>Consistent behavior</i>	Call relations between a set of methods (i.e. callers) and identical(ly positioned) sequence of other methods.	The recurrent sequence of method invocations maps onto the elements crosscutting the callers in the relation.
Fan-in analysis [Marin et al., 2007a; Gybels and Kellens, 2005]	<i>Consistent behavior</i>	See Section 5.3.1	See Section 5.3.1
Grouped calls	<i>Consistent behavior</i>	See Section 5.3.2	See Section 5.3.2
History-based mining [Breu and Zimmermann, 2006]	<i>Consistent behavior</i>	Call relations between two sets of methods, where each method in the callers set calls all methods in the callees set (i.e., similar to Grouped calls analysis).	The invoked methods map onto the elements crosscutting their reported callers.
Context flow mining [Seiter, 2006]	<i>Context passing</i>	Call chain sequence annotated with the position of the parameter passed by each caller to its callee in the chain.	The caller in each invocation in the chain maps onto the method passing the context through the mapping parameter.
Redirection finder	<i>Redirection layer</i>	See Section 5.3.3	See Section 5.3.3
Name-based mining (Identifier analysis [Tourwé and Mens, 2004], Language clues [Shepherd et al., 2005b])	<i>(Role Superimposition)</i>	-	-

Table 5.5: Retrofitting existing techniques to the framework.

static analysis aims at the detection of interfaces, or type members that can be abstracted into an interface definition, that crosscut their implementing classes [Tonella and Ceccato, 2004b]. The analyses employed by this technique check the names of interfaces to recognize common naming conventions like a “-ble” suffix, or attempt to use clustering for grouping the members of a type that might belong to a secondary role. The technique fits naturally into our framework and we can define its search goal as instances of *Role superimposition*.

The technique for recognizing role superimposition based on dynamic analysis was proposed Tonella and Ceccato [2004a]; Ceccato et al. [2006]. This technique uses formal concept analysis to group execution traces obtained under certain use-case scenarios, with methods executed in these traces. The resulting concepts are selected by two rules, namely, (1) the methods grouped by a use-case specific concept (i.e., containing traces of only that use-case scenario) belong to more than one class, and (2) the methods in that concept occur in more than one use-case specific concept. The technique, and the second rule in particular, is aimed at finding classes that implement more than one functionality, i.e., more than one role. The concepts filtered by the above rules are reported by the technique as seed candidates. However, one more (manual) step is required to actually mine the potentially secondary role from the set of methods in the concept, a step that is not explicitly described by Tonella and Ceccato [2004a]. From the reports of the experiments with the technique [Tonella and Ceccato, 2004a], we derive the following additional step, consisting of a rule for turning mining results into (meaningful) concerns: methods in the same type hierarchy whose implementations occur multiple times among the methods grouped by a selected concept are considered to be part of a superimposed role.

We therefore propose that the last step is integrated with the technique and the presentation of the results is a set of methods of the same type. These methods map onto the methods of the secondary role, and the percentage of methods mapping correctly gives us the quality of the seed. All the implementations of the role correspond to the crosscut elements.

5.7.2 Consistent Behavior

In order to identify instances of *Consistent behavior* sort, we can consider employing techniques based on clone detection. A number of aspect mining experiments have been carried out using clone detection [Bruntink et al., 2005; Shepherd et al., 2005a]. Shepherd et al. [2005a], for example, propose to examine clones for same method invocations. Related research has explored how well clone detection can detect several specific idiomatic implementations of crosscutting concerns, with best results reported for such concerns as tracing and checking against NULL values [Bruntink et al., 2005]. Both concerns are clear instances of the *Consistent behavior* sort. There is, however, no report to our knowledge of a complete analysis of all the results produced by clone detection-based techniques, and of their total precision and absolute recall.

The Grouped calls technique resembles mining based on clone detection but only

in some respects: by not considering the position of the calls in the body of the callers, the technique allows to identify related calls that would typically be missed by classical clone detection tools. On the other hand, it can introduce false positives that probably will not be present in standard clone detection.

The mining for recurring execution patterns in program traces [Breu and Krinke, 2004] and control flow graphs [Krinke, 2006] is particularly suited for identification of instances of the *Consistent behavior* sort. The techniques search dynamically and statically respectively for patterns such as methods whose execution always follows the completion of another, specific method, or methods that are always executed first/last in the body of their callers. In terms of our framework, the technique results in a specific call relation. The callee is mapped to the crosscutting functionality, and the callees are mapped to crosscut elements.

The results of the latter technique, based on static analysis, are compared by its author to our own results reported for Fan-in analysis on the JHOTDRAW case [Marin et al., 2007a]. The comparison and the overlapping results confirm the compatibility of the two techniques, as they share the search goal, and provide a clear, intuitive mapping of their results representation into the targeted sort's description.

HAM (history-based aspect mining) is a technique that resembles our own Fan-in and Grouped calls analysis [Breu and Zimmermann, 2006]. The technique searches (CVS) version archives for addition of method calls, and selects those groups of methods with a large (threshold-based) number of common callers. Additional filters of the results consider the number of transactions in which the calls were added, the time and the authorship of the transactions. The presentation of HAM's results is the same as for Grouped calls analysis, and hence fits naturally in our framework.

A few other techniques apply similar recognition criteria of crosscutting concerns as the ones already discussed. For example, the Unique methods by Gybels and Kellens propose several selection conditions for methods with high fan-in values, such as a void return type [Gybels and Kellens, 2005]. Other similar filters include selection of methods with no parameters [Shepherd et al., 2005a].

5.7.3 Context Passing

The *Context passing* sort describes concerns that cut across a call chain, and that are implemented by adding new parameters to the methods in the chain in order to pass specific context information along the call chain. This sort is targeted by the control flow mining technique proposed by Seiter [2006]. The technique looks for sequences of method invocations in a call chain, where each method in the chain passes a specific parameter (identified by its position) as an argument to its callee in the chain. The results are reported as a sequence of methods annotated by the position of the passed parameter.

In terms of our framework, the results are mapped to crosscutting concerns in the following way: for each invocation in the chain, the caller maps onto the method passing the context information in the chain, and the identified parameter maps into the

parameter passed as an argument for carrying the information. The seed-quality is given by the number of invocations in the reported sequence that indeed correspond to context passing, with respect to the total number of invocations in the chain.

5.7.4 Name-Based Approaches

Several approaches to aspect mining rely on naming conventions for the identification of crosscutting code. As an example, *Identifier analysis* applies formal concept analysis to group program elements, like classes and methods, by words occurring in their identifiers [Tourwé and Mens, 2004; Ceccato et al., 2006]. The technique does not propose an interpretation of the results to recognize crosscutting concerns in the generated concepts. However, the search for related program elements employed by this analysis suggests that elements grouped in a concept belong to the same role, which may be crosscutting. The search goal can then be defined as instances of the *Role superimposition* sort.

Previous experiments have shown that the inspection of results for Identifier analysis for recognizing crosscutting concerns is difficult and time consuming [Ceccato et al., 2006]. The technique works better as an enhancing technique that starts from known crosscutting concern seeds and provides us with related program elements using names as association criteria.

Another approach to name-based aspect mining uses lexical chaining; that is, semantically related words recognized in fields, methods and class names as well as in comments are grouped together [Shepherd et al., 2005b]. The output of the technique consists of groups of related words (i.e., chains) that occur in many different source files, and pointers to source code entities that correspond to the words in the chain. However, to understand the relation between these results and potentially associated concerns we still need to turn the list of words into a meaningful concern representation. Therefore, we propose to consider the set of program elements associated with these words as the results of this aspect mining technique. These elements would typically represent members of a potentially superimposed role. In terms of our framework, the technique thus targets instances of *Role superimposition*, and each reported result proposes a set of elements as possible members of the superimposed role. The percentage of correctly identified members of the superimposed role then gives the value of the seed quality metric.

5.8 Discussion

Multiple search-goals As mentioned before, the relation between a technique and its search-goal is not exclusive: one technique can target instances of different sorts if different mappings are defined and applied.

Earlier, our focus for Fan-in analysis was at identifying *Consistent behavior*. However, if we were to employ Fan-in (or Grouped calls) analysis to identify instances

of the Role superimposition sort (Fan-in_{RS}), we could define the following mapping: the callers of the high fan-in method belong to the implementation of a crosscutting, super-imposed role, and the reported method with a high fan-in value implements functionality dedicated to and accessed from the scattered places implementing the role.

Several instances of *Role superimposition* are present in JHOTDRAW. A typical example is the persistence concern: The *Figure* elements implement the *Storable* interface that defines (read and write) methods to (re-)store a figure from/to a file. The scattered implementations of these methods for persistence invoke functionality from classes (*StorableInput* and *StorableOutput*) that are specialized in reading/writing specific types of data. The candidates reported by the technique are the methods in the specialized classes together with their callers in the *Storable* hierarchy.

The persistence candidates for *Role superimposition* instances add to the total number of seeds identified for the analyzed system. However, since these results are not compatible with the *Consistent behavior* instances discussed earlier, they should be addressed distinctly if the technique is to be compared with another one. This is achieved by explicitly specifying the search-goal, as is done with Fan-in_{CC} and Fan-in_{RS}.

Another observation regards two sorts that share their implementation idiom, namely *Consistent behavior* and *Contract enforcement*. The *Contract enforcement* sort describes concerns implementing condition checks for design by contract. However, the specific crosscutting symptom for this sort is the same as for *Consistent behavior*, with the only difference consisting in the *intent* of their instances. This allows for aspect mining techniques to target instances of both sorts together. However, automatic distinction of the sort for a certain instance is difficult, and thus this step requires human analysis. Yet, for the purpose of the analysis presented in this chapter, the distinction between the two sorts is not relevant.

Filters and results extension The Grouped calls analysis builds concepts of callees×callers from the complete (i.e., un-filtered) set of methods in the analyzed system. To these concepts, we then apply our filters, such as those for accessor methods, which eliminate getters and setters from the sets of grouped callees. Finally, we reason about the filtered results and decide whether a candidate is a valid seed or not. However, our filters might eliminate methods relevant to a concern implementation, particularly from those concepts marked as seeds. A simple solution to missing relevant methods is to extend the concept of our seeds to their full representation by removing the filters for each of the concepts marked as seeds. This way, we are still able to eliminate the most unlikely candidates by using filters prior to the manual inspection of the results, but also to reduce the number of false negatives by investigating all the (remaining) elements in the *extended* representation of our seeds. For example, by extending the concept for the Grouped calls seed discussed in Section 5.6.2, which groups the *execute*, *view*, and *checkDamage* methods (see Figure 5.7), will also uncover a set of three methods dealing with setting up the undo support for a command, namely *setUndoActivity*, *getUndoActivity*, and *setAffectedFigure*.

These methods point us to the undo concern in the body of the commands' execute methods.

Similarly, we can use such extensions for the results of the combination of Fan-in and Grouped calls analyses. In order to achieve better seed-quality, the set of callers for these results consists of only those callers from the result of Grouped calls analysis. However, this selection of callers might eliminate relevant callers from the typically more extensive Fan-in result being combined. A recommended practice in this case is to use the combination results for selecting the seeds, and then to extend the identified seeds with the other callers reported by Fan-in analysis.

Results of previous experiments The framework rules and the survey in Section 5.7 confirm the results of our previous experiments reported in Chapter 3. In that chapter, we found by manual examination of results (and without using consistent comparison criteria based on sorts) that Fan-in analysis and the search for concepts in traces (i.e., Dynamic analysis) have a small overlap and are mainly complementary. This is explained by the different main search goals of the two techniques that consist of instances of different sorts.

As we have shown above, Fan-in analysis can be employed to some extent to search for instances of *Role superimposition* (Fan-in_{RS}); This explains the overlap between some of the results of the two techniques. Another explanation of this overlap lies with the description of those results in terms of complex features, like Undo or Persistence support. Such features typically involve more than one crosscutting concern and the distinction between the various concerns proved difficult in the absence of a framework like the one proposed in this chapter. Different concerns (i.e., sort instances) part of the same crosscutting feature were then counted as a common finding.

Tool performance Although this work's focus is less on each individual mining technique and more on the common framework to consistently assess and possibly combine mining techniques, we briefly discuss the performance of our tool FINT. The analysis of the whole JHOTDRAW system for Fan-in analysis requires around 30 seconds on our test machines (Pentium 4 - 2.66 GHz, with 1GB of RAM) running Eclipse 3.1.x under either Linux or Windows OS.

The Grouped calls analysis requires the model built for the Fan-in analysis and takes around 5 minutes to examine all the call relations (approximately 6000×6000 elements). This analysis will not scale up very well to systems like Tomcat or JBoss, which have up to 35,000 elements. However, trying to understand such large systems in one iteration is hardly advisable due to the cognitive complexity. We would suggest dividing them into sub-systems comparable in size with JHOTDRAW and gather understanding for each of these systems. Actually, to ensure maintainability, the architecture of the two aforementioned systems is already conveniently split in components that are suitable for analysis in isolation.

The Redirection finder uses the Fan-in model and requires only a few seconds for

execution.

Reproducibility To allow for reproducibility of the experiments described in this chapter, we provide both the tool and detailed setup elements and results sets on the tool's and experiment's web-pages, indicated in footnotes 2 and 6.

5.9 Related Work

Several authors have proposed (and taken) steps towards the comparison and combination of aspect mining techniques [Marin et al., 2004; Ceccato et al., 2006; Shepherd et al., 2005a]. We are not aware of related work on providing a common framework for systematic aspect mining, and consistent combination and assessment of mining techniques.

Shepherd et al. [2005a] report on machine learning techniques for combining aspect mining analyses. Their approach learns from annotated code and they compare the results of their combination to results of Fan-in analysis [Marin et al., 2004]. A drawback is the required annotation of crosscutting concerns on some significant system, which is needed for training the tool. The techniques considered for combination include filters for accessor or utility methods, as also used in FINT. However, the authors of the experiment do not describe their findings in detail nor do they provide rules to consistently associate results of different representations to crosscutting concerns.

In the collaborative AIRCO effort, described in Chapter 3, three aspect mining techniques are compared and investigated from the perspective of combination. The techniques include fan-in analysis, dynamic analysis of execution traces, and analysis of shared identifiers in signatures of program elements. Major difficulties in this experiment were caused by heterogeneity in the search-goals of the three techniques and in the representation of results. Such experiments require a tedious effort from the participants in the experiment to bring individual results to comparable levels of granularity. Due to such issues, the experiment could focus only on a limited selection of common findings. This formed one the motivations for the work presented here.

The survey of the existing aspect mining techniques extends the work of Mens et al. [2007] that describes a number of aspect mining techniques by their implementation characteristics. By comparison, we investigate how existing aspect techniques can be used to identify instances of typical crosscutting concerns. We try to answer the question of when to use these techniques, i.e., for what sorts of concerns, and how to consistently interpret their results and turn them into meaningful crosscutting concerns.

5.10 Conclusions

With a growing number of aspect mining techniques and approaches, it is increasingly difficult to consistently assess, compare, and combine mining results.

This chapter addresses this challenge by proposing a common framework to define systematic aspect mining based on crosscutting concern sorts. The framework allows for consistent assessment, comparison, and combination of compliant aspect mining techniques. It identifies a set of requirements that ensure homogeneity in formulating the mining search-goals, presenting the results, and evaluating their quality.

We demonstrate the feasibility of the approach by retrofitting an existing aspect mining technique to the framework, and by using it to guide the design and implementation of two new mining techniques in our FINT aspect mining infrastructure. Our application of the three techniques to an aspect mining benchmark known from literature shows how they can be consistently assessed and combined to increase the quality of the results. Furthermore, our table containing a mapping of more than ten existing aspect mining techniques in our framework demonstrates the wide applicability of this framework.

As future work, we would like to extend FINT with new aspect mining techniques, and particularly with techniques that target other concern sorts than the ones currently supported. We shall also use such new techniques to further validate our framework.

Another direction to explore is on elaborating our metrics suite that is part of the framework. For example, we would like to measure how much of a concern's extent is covered by a particular identified seed – the *seed-coverage*. This metric for seeds complements the seed-quality one, for which we can investigate new techniques for improving the obtained values.

Chapter 6

An Integrated Strategy for Migrating Crosscutting Concerns

In this chapter we propose a systematic strategy for migrating crosscutting concerns in existing object-oriented systems to aspect-based solutions. The proposed strategy consists of four steps: mining, exploration, documentation and refactoring of crosscutting concerns. We discuss in detail a new approach to aspect refactoring that is fully integrated with our strategy, and apply the whole strategy to an object-oriented system, namely the JHOTDRAW framework. The result of this migration is made available as an open-source project, which is the largest aspect refactoring available to date. We report on our experiences with conducting this case study and reflect on the success and challenges of the migration process, as well as on the feasibility of automatic aspect refactoring.

6.1 Introduction

The tangling and scattering that results from implementing crosscutting concerns in a software system using traditional object-oriented programming is a known challenge to program comprehension and software evolution. One approach to mitigate these issues is to migrate the system to aspect-oriented programming (AOP) and transform the crosscutting concerns into aspects, a process known as *aspect refactoring*.

Despite significant research efforts on various parts of the refactoring of crosscutting concerns from existing systems, to date there exists no compelling show-case for such a complete migration. One of the main causes for this void is the fact that there is no clearly defined, coherent migration strategy detailing the steps to be taken to perform this process.

Successful migration requires a strategy comprising steps like identification of the concerns (i.e., aspect mining), description of the concerns to be refactored, and consistent refactoring solutions to be applied. Moreover, such a strategy requires *integrated*

migration steps, so that aspect mining results, for example, can be consistently mapped onto concerns in code, and further refactored by general aspect solutions. The present state of the art prevents developers and practitioners from experimenting with a complete migration process and assessing the benefits of migrating to AOP.

In this chapter, we propose such an integrated strategy for migrating crosscutting concerns to aspects, which consists of four main steps: (1) idiom-driven identification of crosscutting concerns in an existing system (aspect mining); (2) exploration of (the context of) the concerns identified in the previous step; (3) query-based modeling and documentation of crosscutting concerns in the system; (4) template-based refactoring of the object-oriented idioms into AOP solutions.

Our strategy builds upon the classification and decomposition of crosscutting concerns in so-called *crosscutting concern sorts* that we proposed earlier in Chapter 4. Each sort describes the typical implementation idiom and relation of crosscutting concerns. Sorts act as glue between the successive steps of the migration: The mining step in our strategy uses the sort-specific idioms to define search-goals for identifying crosscutting concerns that belong to a specific sort (i.e., *sort instances*). To support the exploration and documentation steps, we have formalized the concern sorts using queries over source code and implemented these in a tool for browsing and modeling crosscutting concerns, as described in detail in Chapter 4.

While the first three steps of our approach have been covered in our earlier work, this chapter focuses on the fourth step and its connection with the three preceding steps. In particular, we define template solutions for the aspect refactoring of our sorts (to AspectJ). Furthermore, we describe a case study in which we apply the whole migration strategy to JHOTDRAW,¹ an object-oriented application used in other aspect mining and refactoring studies as well [Marin et al., 2007a; Ceccato et al., 2006; Marin et al., 2006a; Binkley et al., 2006]. The results of our migration are available under version control as an open-source project on sourceforge called AJHOTDRAW, which is also the largest aspect refactoring publicly available to date that we are aware of.

The remainder of the chapter is organized as follows. In next section, we recall the notion of crosscutting concerns sorts. We describe the migration strategy and elaborate on the first three steps in Section 6.3. The sort-based aspect refactoring approach that we introduce for the fourth step is presented in Section 6.4. Section 6.5 covers our experiences with migrating crosscutting concerns in JHOTDRAW to aspect solutions. Section 6.6 discusses the results and outlines a number of lessons learned. We conclude with an overview of related work and recommendations for future research.

6.2 Crosscutting Concern Sorts

A systematic migration strategy requires a consistent way to address crosscutting concerns in source code. To this end, we distinguish a number of *atomic* crosscutting

¹ <http://jhotdraw.org>

concerns (i.e., concerns that cannot be split into smaller, still meaningful concerns) that share properties like their implementation idioms and relations. We group concerns that share such properties in categories called *crosscutting concern sorts* [Marin et al., 2005a]. These sorts can be used on their own, but can also be composed to construct more complex crosscutting designs, for example, the *Observer* pattern, often used as a typical example of crosscuttingness.

The first column of Table 6.1 describes the identified sorts and Table 6.2 shows several examples of instances (the other columns of Table 6.1 will be introduced in later sections). *Consistent behavior*, for instance, groups concerns whose implementation consists of scattered calls to a specific method implementing the crosscutting functionality. Instances of this sort include, for example, a logging concern, a simple authentication or authorization concern implemented as a call to a method checking credentials, or a mechanism for updating observers using calls to a notification method.

Similarly, the idiom for implementation of secondary roles, common in design patterns like *Observer* or *Visitor*, as well as in mechanisms for persistence, is described by the *Role superimposition* sort.

Composite crosscutting designs exhibit multiple sort instances in their implementation: the aforementioned *Observer* pattern, for example, comprises two instances of *Role superimposition*, for the Subject and the Observer role respectively. Furthermore, it comprises instances of *Consistent behavior*, like the concern for notification of observers, or the one for observers registration. Instances of our sorts are therefore *building blocks* for modeling and describing crosscutting functionality.

6.3 An Integrated Migration Strategy

In this section, we define an integrated strategy for migrating crosscutting concerns in existing systems to aspect-based solutions. The strategy consists of four steps:

Step 1. Idiom-driven crosscutting concern identification (also known as *aspect mining*).

Step 2. Concern exploration.

Step 3. Query-based concern modeling and documentation.

Step 4. Sort-based aspect refactoring.

The remainder of this section discusses the first three steps in more detail and the next section presents the fourth step. We show how the steps are integrated via crosscutting concern sorts using examples from our JHOTDRAW to AJHOTDRAW migration experience.

Sort and Intent	Idiom	Template aspect solution
<i>(Method) Consistent Behavior (CB)</i> : A set of methods consistently invoke a specific action as a step in their execution.	Method invocations from set of methods.	Pointcut and advice mechanisms. <pre> around(..) : callersContext(..){ invokeCB(..); //before proceed(); // or after: invokeCB(..); } </pre>
<i>Redirection Layer (RL)</i> : A type acts as a front-end interface having its methods responsible for receiving calls and redirecting them to dedicated methods of a specific reference, optionally executing additional functionality.	Redirector type whose methods consistently forward calls to pair methods in receiver.	Pointcut and around advice to replace each redirection. <pre> around(..) : call Receiver.m(..) && filteredCallers(..) { addBehavior1(); proceed(..); //redirection addBehavior2(); } </pre>
<i>Expose Context (EC)</i> : Context Passing: Methods in a call chain consistently use parameter(s) to propagate context information along the chain.	Method in chain passes parameter as argument to callee.	Pointcut and advice, where the point cut collects the context to be passed - Wormhole [Laddad, 2003b] <pre> around(<caller context>, <callee context>): cflow(callerSpace(<caller context>)) && calleeSpace(<callee context>){ // ... advice body } </pre>
<i>Role Superimposition (RSI)</i> : Types extend their core functionality through the implementation of a secondary role.	Set of types (declare and) implement member roles (which are possibly declared by a distinct interface).	Introduction mechanisms. <pre> declare parents : Type implements SecondaryRole; Modifiers Type Type.roleField; Modifiers Type Type.roleMethod(..){ ...//original implementation }; </pre>
<i>Support Classes for Role Superimposition (SC)</i> : Types implement secondary roles by enclosing nested support classes. The nesting enforces (and explicates) the relation between the enclosing and the support class.	Set of types (in hierarchy) implement Role using nested classes.	The desired solution, introduction for nested classes, is not supported by AspectJ. Our solution is to move the support classes to the aspect.
<i>Exception Propagation (EP)</i> : methods in call chain consistently (re-)throw exceptions from their callees in the absence of an appropriate answer.	Method in call chain re-throws exception to caller.	Softening exceptions mechanisms. <pre> declare soft : ExceptionType : (call(* rootException(..) throws ExceptionType)); </pre> Capture SoftException at top of the call chain.

Table 6.1: Crosscutting concern sorts.

Sort	Examples
<i>(Method) Consistent Behavior (CB)</i>	Logging of exception events in system; Wrapping business service exceptions and re-throwing them as new exception type [Marin et al., 2007a]; Notification of Figure change events.
<i>Redirection Layer (RL)</i>	Border decorations for Figure elements (Decorator pattern); Command wrapper for undo support.
<i>Expose Context (EC)</i>	Transaction management [Laddad, 2003b]; Credentials passing for authorization; Progress monitor for long-running operations.
<i>Role Superimposition (RSI)</i>	Figure elements observed by views for changes (Subject role); Visitable elements (Visitor pattern); Storable figures (Persistence) [Marin et al., 2007a].
<i>Support Classes for Role Superimposition (SC)</i>	Undo support for Command elements; Event dispatcher for observers' notification.
<i>Exception Propagation (EP)</i>	IOException thrown if Figure elements recovery fails; Checked SQLException thrown from methods in the JDBC API.

Table 6.2: Examples of sorts instances.

6.3.1 Aspect Mining

In our earlier work we have proposed and implemented an idiom-driven approach to aspect mining based on crosscutting concern sorts [Marin et al., 2006a]. The approach supports the design of aspect mining techniques that target instances of a specific sort by searching for the sort's implementation idiom.

The third column in Table 6.1 shows the implementation idioms associated with each of the sorts. Consider for example the commands in a drawing application, like JHOTDRAW, that carry out tasks in response to user actions. Each command concludes its execution with a call to the `checkDamage` method in the drawing view, which updates the view with changes triggered by the command. The notification concern is an instance of *Consistent behavior* whose implementation idiom is invocation of a specific method from a (large) set of methods. Aspect mining techniques such as Fan-in analysis [Marin et al., 2007a] or Grouped calls analysis [Marin et al., 2006a] exploit idioms such as this one in their search process. The techniques and their implementation have been discussed in Chapters 2 and 5.

We have implemented the two mining techniques mentioned above and an additional technique that targets instances of *Redirection layer* in our aspect mining tool FINT². The techniques and their implementation have been discussed in Chapters 2 and 5. The results of applying FINT to JHOTDRAW are the starting point of our migration case study.

² Available from <http://swertl.tudelft.nl/view/AMR/FINT> and discussed in detail in Chapter 2 and 5

Like the notification mechanism above, we have found the *Consistent behavior* idiom in multiple concerns implementing support for commands and undo operations. Examples include consistently checking the reference to the active view before execution of each command, consistent initialization of Command objects by means of super calls, or consistent checks implemented by all actions to undo a command. Our search for idioms of the *Redirection layer* pointed us to wrapper objects for undo-able commands: methods in the wrapper delegate calls to their wrapped command object.

6.3.2 Concern Exploration

Aspect mining often does not yield complete crosscutting concern instances, but just concern *seeds*: (possibly incomplete) sets of program elements that belong to a particular crosscutting concern.

The second step of our strategy, concern exploration, aims at expanding mining results (i.e., concern seeds) to the complete implementation of the associated concerns. In this step, we start from the discovered seeds and use the specific relation of the sort for the seed's concern to identify all the participants in the concern implementation.

In our *Consistent behavior* example, this means looking at all call relations directed to the method `checkDamage` (or another method, depending on the particular concern targeted). As it turns out, not all of the 28 calls to this method that we found are part of the concern of interest, but around two-thirds of them, namely those from *Command* classes. Similarly, the *Grouped calls* mining technique, which applies a more conservative search, covers only partially the set of calls participating in the concern.

Our aspect mining tool, FINT, integrates support for seeds exploration and expansion to full concerns, such as detection of structural relations or similar call positions for the callers of a method. A number of other tools also provide (partial) support for querying source code and exploring concern sort relations: Eclipse IDE, the Concern Manipulation Environment (CME) [Tarr et al., 2004], FEAT [Robillard and Murphy, 2002], JQuery [Janzen and Volder, 2003], CodeQuest [Hajiyev et al., 2006], or SO-QUET [Marin et al., 2007d]. The same tools can be used to further understand the context enclosing the discovered crosscutting concern. At this step, we can see, for example, how the identified sort instances in command and undo support relate to each other: commands that can be undone enclose a specialized *UndoActivity* class that knows how to revert the effects of the command's execution. Two of our mined sort instances cover the key methods of the two classes: the `execute` method in a command, and the `undo` one in the enclosed undo activity.

6.3.3 Concern Modeling and Documentation

Most approaches to concern modeling and their tool support do not enforce consistency across the representation of crosscutting concerns. The decision of what is crosscutting in a system, and how to best represent that, lies with the user of these concern modeling

tools. Such a concern model can contain ad-hoc collections of program elements, such as methods and classes, that participate in a concern's implementation.

However, to ensure generally applicable solutions for concern migration, we need a coherent way to describe similar concerns and their common properties. To this end, we have defined queries for each of our crosscutting concern sorts which search for the sort's specific relation between source code elements. For more information on these sort queries, we refer to Chapter 4, which formalizes these queries using relation calculus over source models extracted from the system's source code.

We have implemented support for this third migration step in our concern modeling tool SoQueT³ [Marin et al., 2007d]. Figure 6.1 shows two of the main views of the tool. The *Concern model* view allows us to organize concerns hierarchically, with sort instances and their associated queries as leaf-elements and composite concerns describing more complex crosscutting designs as parents. The user can select a sort instance in the concern model and execute its query; The results of the query are displayed in the *Search (Sorts Result)* view, from where they can be navigated to their source code implementation. To add a new sort instance to the model, the user launches the dialog providing the query templates for each sort, and parameterizes the query for a given crosscutting concern. For example, to document our *Consistent behavior* instance for notification of views, we use the knowledge gained at the previous steps and search for all the calls to the `checkDamage` method from methods in the *Command* hierarchy. The method and the hierarchy are our input parameters to the query. The instances can then be added to the model from the results view.

A part of the concern model built to document concerns in JHOTDRAW is shown in Figure 6.1. The model is available for download at the same web-site as the tool and covers over 100 sort instances.³ In Section 6.5, we use this documentation to guide our refactoring and configure the aspect solutions.

6.4 Aspect Refactoring

We employ a sort-based, idiom-driven approach to aspect refactoring that allows for consistent integration with the previous steps of our migration strategy. Furthermore, we define template aspect solutions for each of our concern sorts that we can instantiate to refactor an occurrence of that sort. Like the previous steps, the refactoring approach addresses crosscuttingness at the level of atomic concerns, which provides the optimal trade-off between complexity of the refactoring and comprehensibility of the refactored element.

The template aspect refactorings for each sort are summarized in the last column of Table 6.1. A solution basically consist of one aspect language mechanism. At the moment, however, some sorts do not have an equivalent mechanism in AspectJ (or any other aspect language existing at this moment). Support classes, for example, cannot

³ Available from <http://swertl.tudelft.nl/view/AMR/SoQueT>

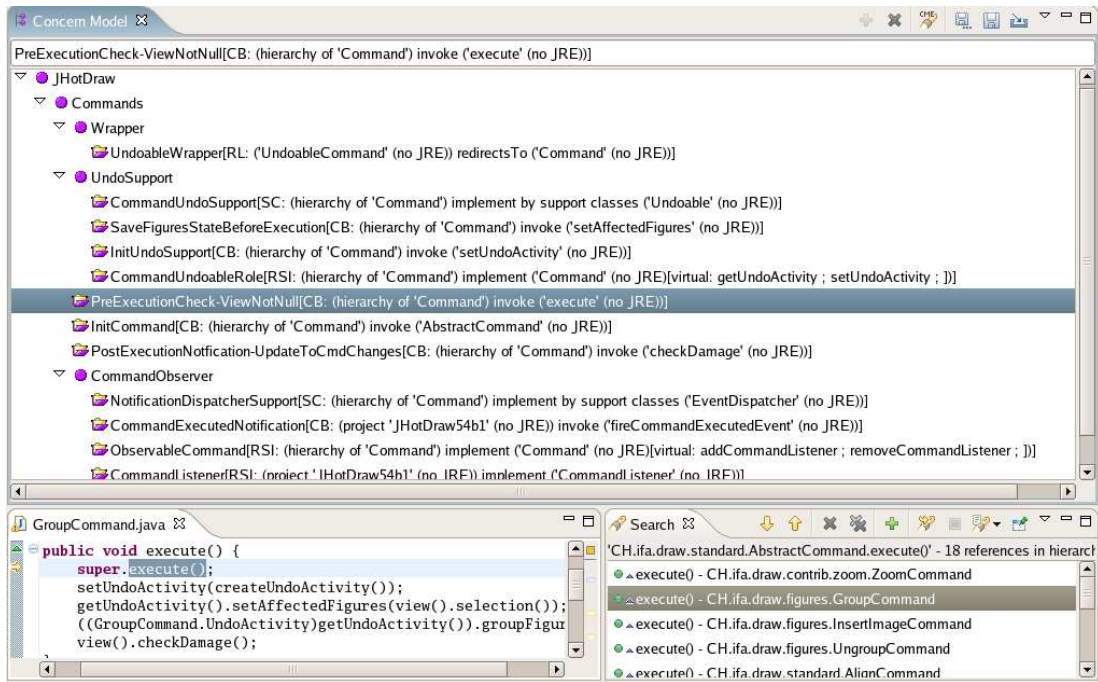


Figure 6.1: SOQUET documentation of the concerns for Command support in JHOT-DRAW.

be introduced similarly to role members, although, as we shall see in Section 6.6, this would be a desired refactoring.

To refactor a sort instance, we start from its query-based documentation (in SOQUET). The query points us to the elements participating in the concern, which we use to configure the template aspect solution. For example, the query for a *Consistent behavior* instance indicates the callers to be captured by a pointcut definition (the source context) and the action to be introduced by the advice (the target context). Other configurable elements, such as the type of advice to introduce the crosscutting call (e.g., before, after, after throwing, etc.), are decided at the refactoring time.

The solution described in Table 6.1 for the *Redirection layer* sort is a common approach to refactoring implementations of the *Decorator* pattern [Hannemann and Kiczales, 2002; Lesiecki, 2005]. This consists of replacing the redirector class by an aspect that intercepts (relevant) calls to the methods receiving the redirection, and then adds the redirector's functionality by means of an advice.

The aspect solution for *Expose context* instances is discussed by Laddad as the *Wormhole* pattern Laddad [2003b]: the extra parameter used to pass context is replaced by using a pointcut to obtain the context from the caller and an advice that makes the context available to the caller's control flow.

Solutions for static crosscutting, like *introduction* and *declare soft* mechanisms in AspectJ, apply to two of the sorts in the list, *Role Superimposition* and *Exception prop-*

agation respectively. The elements to instantiate these aspect templates are again available through the sort-based documentation of the concerns: they indicate the members of a type's secondary role to be moved to and then introduced from an aspect, or the checked exception to be turned into an AspectJ soft exception. Soft exceptions, unlike checked ones, do not need to be caught or re-thrown. This allows us to remove the *throws* clauses from the (transitive) callers of the method initiating the exception propagation. The method at the top of the call chain that deals with the exception has now to catch the soft exception that wraps the original checked one. The top method assumes knowledge of the wrapped exception that it has to extract and cast. The code to handle the (cast) exception requires no modifications.

6.5 Aspect Refactoring of JHOTDRAW

We have used the sort-based migration strategy to refactor a number of crosscutting concerns in JHOTDRAW towards an aspect-oriented solution. Based on these experiments, we would like to obtain answers to the following questions:

1. Are the template aspect solutions proposed in Section 6.4 applicable in practice?
2. What are the risks and benefits of adopting refactoring strategy that is sort-based?
3. What level of automation of all four steps and the fourth refactoring step in particular is feasible?
4. Do the refactorings carried out lead to a better separation of concerns?

In the present section, we report our observations and experiences regarding the migration of specific crosscutting concerns towards aspects in JHOTDRAW. In the next section, we return to our questions, and try to formulate answers to them based on the findings presented here.

6.5.1 AJHOTDRAW

We share the refactored version of JHOTDRAW as an open-source project on sourceforge⁴: AJHOTDRAW is, to our knowledge, the largest migration to aspects available to date. A transparent, gradual migration process is important for building confidence in the aspect-oriented solution. Therefore, our refactorings aim at maintaining the conceptual integrity and stay close to the original design. In addition, by publishing the refactoring steps in a versioned repository, we provide insight in the migration process and enable traceability, making the refactored system easier to understand.

We focus our next discussion on the refactoring of sort instances contained in the implementation of the command and undo functionality, which we also used in Section 6.3 to explain the first three steps of the approach. We use the organization of

⁴ <http://sourceforge.net/projects/ajhotdraw/>

```
public class AbstractCommand implements Command {
    ...
    public void execute() {
        if (view() == null) {
            throw new JHotDrawRuntimeException(
                "execute_should_NOT_be_getting_called_when_view()==null");
        }
    }
}

public class PasteCommand extends AbstractCommand {
    ...
    public void execute() {
        super.execute();
        ...
    }
}
```

Figure 6.2: Consistent check - super method idiom.

concerns in the concern model initiating the refactoring to design the package and type structure of our aspect solutions. The solutions discussed below have been integrated with the source code available on the public repository.

6.5.2 Consistent Behavior in Command

JHOTDRAW makes use of the *Command* design pattern in order to separate the user interface from the underlying model, and to support such features as undoing and redoing user commands. Each command has to realize the *Command* interface, for which a default implementation is provided in the *AbstractCommand* class. The key method is *execute*, which takes care of actually carrying out the command (such as pasting text, duplicating a figure, inserting an image, etc.).

A typical implementation of a command is highly crosscutting, with the *Command* top interface defining three different roles: besides their core functionality, commands are undo-able as well as observable elements. The support for the secondary roles counts for half of the *Command*'s members. Similarly, the *execute* method in a typical concrete command implements multiple concerns.

Each *execute* method should start with a consistency check verifying that the underlying “view” exists. Therefore, each concrete implementation of *execute* starts with a call to the *execute* implementation in the superclass, which is always the one from the *AbstractCommand*. This is illustrated in Figure 6.2.

We apply a *Consistent behavior* refactoring template from the last column in Table 6.1 using a pointcut capturing all *execute* methods, and putting the check itself in the advice. Observe that mimicking the implementation where the check is in a super

```

pointcut cmdExecute(AbstractCommand aCommand) :
    this(aCommand)
    && execution(void AbstractCommand+.execute())
    && !within(*..DrawApplication.*);

before(AbstractCommand aCommand) : cmdExecute(aCommand) {
    if (aCommand.view() == null) {
        throw new JHotDrawRuntimeException("...");
    }
}

```

Figure 6.3: Enforcing consistency using advice.

method is not possible in AspectJ: super methods cannot be accessed when advising a method. The resulting solution is shown in Figure 6.3.

The only surprise in this code may be the `within` clause in the pointcut. In the exploration step, we learned that *anonymous* subclasses of *AbstractCommand* do not implement the consistency check. Such classes are used for simple commands like printing, saving, and exiting the application. Since AspectJ does not provide a direct way to exclude anonymous classes in a pointcut, we used the `within` operator to exclude executions occurring in the context of the top level object creating the full user interface. One can also argue that the anonymous classes should include this check (in which case the exclusion can be omitted from the pointcut), but, as stated before, we focus on keeping the behavior as it was, not on modifying it.

Besides the separation of the consistency check from the core logic of the commands, another benefit of the aspect approach is that consistency checks cannot be forgotten. This is illustrated by a number of the anonymous classes, but also by one non-anonymous command,⁵ which does not extend the *AbstractCommand* default implementation. Consequently, it cannot reuse the consistency check using a supercall. Inspection of the `execute` implementation, however, clearly shows that the code exits with a null pointer exception in case the check fails. This suggests that the aspect that we are looking for should implement the check not only for the *AbstractCommand* class, but for all the *Command* implementations.

6.5.3 Undo Functionality

Support for “undo” functionality was added in JHOTDRAW version 5.4. As can be imagined, it is a concern that cuts across many different classes. More than 30 elements of the JHOTDRAW framework, comprising *commands*, *tools* and *handles*, have associated undo constructs to revert the changes spawned by their underlying activities. The *commands* group is the largest in terms of defined undo activities.

⁵Namely, the *UndoableCommand*.

The participants of the “Undo” functionality have the following responsibilities:

- Each command is associated with one *undo activity*, whose method *undo* can be invoked to revert the command. The undo activity is implemented in a nested class of the command, which is instantiated using a factory method called `createUndoActivity`.
- Prior to the execution of the command’s core logic, the command saves a reference to its associated undo activity, by calling a dedicated setter method.
- The primary abstraction in the undo activity is the list of affected figures: when the command’s `execute` method is invoked, the relevant state of the affected figures is stored in the undo activity.
- Undo activities are maintained on a stack by the undo manager.

Support classes for role superimposition

The refactoring that we propose for Undo consists of associating a dedicated undo-aspect to each undo-able command. The aspect implements the entire undo functionality for the given command, while the associated command class remains oblivious to its secondary (undo) concern.

We use naming conventions to relate the aspect to its supported command class. In a successive step, we refactor each of the sort instances in the undo support. The command’s nested *UndoActivity* class belongs to a *Support classes* instance. In the absence of introduction mechanisms for nested classes in AspectJ, our aspect solution consists of moving the *UndoActivity* class into the aspect.

The factory methods for the undo activities (`createUndoActivity()`), as well as the members for managing the reference to the command’s undo activity belong to an instance of *Role superimposition*. The role members move to the aspect, from where they are introduced back into the associated command classes using inter-type declarations. The design, however, suffers modifications as the visibility of the undo factory methods has been altered: ASPECTJ cannot be used to introduce the required factory method as *protected*.

Consistent behavior

The invocations in the `execute` method that are responsible for setting up the undo activity implement *Consistent behavior* concerns: the calls are taken out of the `execute` method, and woven into it by means of advice. In some cases the corresponding pointcut simply needs to capture all `execute` method calls. However, in other cases the pointcut is more complex, depending on the way the undo code is mixed with the regular code.

As an example to illustrate that automating such refactorings is not at all straightforward, consider the `paste-command`, whose `execute` method consists of retrieving the

```

public class PasteCommand extends FigureTransferCommand {
    public void execute() {
        ...
        FigureSelection selection = (FigureSelection)
            Clipboard.getClipboard().getContents();
        if (selection != null) {
            setUndoActivity(createUndoActivity());
            ... //core command logic and other undo setup
            FigureEnumeration fe = insertFigures(...);
            getUndoActivity().setAffectedFigures(fe);
            ...
        }
    }
}

```

Figure 6.4: The original PasteCommand class.

selected figures from the clipboard, inserting them into the current view, and clearing the clipboard. All this is done in a single method, using local variables and if-then-else statements to deal with situations like pasting from an empty clipboard. The undo aspect will require the same conditional logic, and access to the same data in the same order. The following alternatives are possible for aspect refactoring:

- if all getters are side effect free, an approach is to setup the undo activity in a simple before advice. In JHOTDRAW, however, this is not the case, for example because of figure enumerators that have an internal state.
- an alternative is to intercept relevant getters, keep track of the data locally in the advice as well, and inject advice after all data has been collected. This is the approach we follow, but some of the pointcuts are somewhat artificial. Figure 6.5 shows such a pointcut in the undo aspect for the *PasteCommand*, refactored from Figure 6.4. The *clipboardGetContents()* pointcut captures the call that sets the reference to be checked by both the command's core logic and the undo functionality in the aspect.
- The last possibility is to refactor the long *execute* method into smaller steps using non-private methods. The extra method calls can be intercepted allowing smooth extension with setting up the undo activity, at the cost of creating a larger interface and breaking encapsulation. Moreover, we would still introduce artificial pointcuts, as our intention is to enhance the behavior of the *execute* method, and not of various steps created for supporting advice introduction.

Redirection layer

The design of undo in JHOTDRAW uses wrapper objects to associate undo-able commands to menu items and buttons in the user interface (UI). The wrappers share their top level interface with regular commands, so they can connect to UI elements and

```
public aspect PasteCommandUndo {
    //store the Clipboard's contents - common condition
    FigureSelection selection;

    pointcut clipboardGetContents() :
        call(Object Clipboard.getContents()) &&
        withincode(void PasteCommand.execute());

    after() returning(Object select):clipboardGetContents(){
        selection = (FigureSelection)select;
    }
    ...

    pointcut executePasteCommand(PasteCommand cmd) :
        this(cmd) && execution(void PasteCommand.execute());

    // Execute undo setup
    void after(PasteCommand cmd):executePasteCommand(cmd) {
        // the same condition as in the advised method
        if(selection != null) {
            cmd.setUndoActivity(cmd.createUndoActivity());
            ...
            cmd.getUndoActivity().setAffectedFigures(...);
        }
    }
}
```

Figure 6.5: The undo aspect for PasteCommand.

Sort	Limitations and risks
Consistent Behavior	Advice constructs in a privileged aspect can break encapsulation; High degree of tangling might prevent (automatic) refactoring; Anonymous classes cannot be referred to consistently, preventing generic pointcuts; Calls to super class functionality cannot be migrated into advice; Modular reasoning affected by need to keep track of data set in the advised method; Check required that omissions are not on purpose; Sophisticated pointcuts needed to intercept all relevant state modifications in the advised methods; Check required that advice (position) does not change precedence;
Redirection layer	The repetitive logic of redirection for the redirector's methods is not eliminated – the aspect solution addresses the redirection at method level and not at type level; New redirector methods are not (automatically) covered by the solution; The aspect solution is not dynamic (dynamic reordering of redirectors) [Hannemann and Kiczales, 2002]; The aspect solution replaces the redirector (wrapper) and hence changes the public interface of the application to test against; The calls (to the receiver) to be advised for redirection need to be detected;
Role superimposition	Visibility affected since protected (/non-public) methods cannot be introduced.
Support classes for role superimposition	Not supported; Nesting the support class in the aspect breaks dependencies (thus forcing the enclosing class to make more of its interface public) and weakens the relation with the enclosing class;
Exception propagation	Type of thrown exception is lost; Refactoring <i>throws</i> clauses in inheritance hierarchy.

Table 6.3: Risks and possible limitations of the aspect solution.

receive user actions. While most commands are undo-able and wrapped by an *UndoableCommand* object, there are a few exceptions, such as, *CopyCommand*.

Wrappers are instances of *Redirection layer*. The refactoring of such instances raises several important issues: first, we need to identify those commands that are wrapped by an *UndoableCommand* object and accessed through this object; second, we need to check if all clients of a command access its functionality via the wrapper. Only those calls from command clients that are received by a wrapper in the original implementation need to be captured by the aspect solution to attach the wrapper's functionality by means of advice.

Further complications that limit feasibility of automated refactoring have to do with the multiple roles in *UndoableCommand*: since the aspect solution completely replaces the wrapper class, this means that introduction of roles is no longer possible. Some of the original roles in the system are implemented by the wrapper only to comply with the top interface of the wrapped element and add no specific functionality, such as the *Observable* role of *Commands*. The aspect solution can safely omit these roles. For other roles however, this is not desired and refactoring requires customized redirector solutions.

6.6 Discussion

6.6.1 Applicability in Practice

The proposed template aspect solutions proved suitable for refactoring concrete sort instances in the JHOTDRAW case and for separating the crosscutting code from the core system. However, the difficulty of implementing the aspect solution and the quality of the result will vary from case to case. One of the issues is pointcut definitions: Ideally, we would like to use pointcut definitions that describe a set of elements by formalizing a common property instead of a brittle enumeration of the elements in the set. In practice, such definitions will not always be feasible, either because of limitations in the aspect language, or due to irregularities in the code under investigation.

Desired functionality included for example a pointcut to capture calls from “all *Command* classes, except all anonymous classes”, or a pointcut for “all objects interested in command events”. Irregularity in the code might require that for certain methods the advice executes only if a specific condition holds. This is the case for a few commands in JHOTDRAW that send notifications of their execution only if the clipboard's content is not empty. In such a situation, one has to make a trade-off between a generic pointcut definition that captures all commands, but ignores the particular condition, and a definition that enumerates all appropriate elements. The former solution would execute the code in the advice in spite of its void effect; however, the latter pointcut definition needs to be updated (manually) for every new element added to the set of interest (i.e., every new command).

Similar observations can be made about the definition of advices: sometimes we

need to modify the original control flow of a method-to-be-migrated in order to introduce an action to it by means of advice. Although the refactoring may have no effect on the observable behavior of the method, the original flow could be more natural or comprehensible.

6.6.2 Benefits and Risks

In comparison with refactoring approaches proposed by others, our sort-based migration strategy gives a clear definition of the input required for refactoring (i.e., an atomic concern) and describes it consistently using queries. This allows for the definition of reusable solutions and improves comprehension of refactoring by addressing meaningful concerns instead of code fragments [Binkley et al., 2006; Monteiro and Fernandes, 2005]. Moreover, the concern queries allow us to describe the context cut across by a concern, and hence the concern's intent. This gives a better insight into the concern and its aspect solution than the simple enumeration of joinpoints common with most previous refactoring approaches. We believe that a clearly specified input for a refactoring solution is a necessary condition for ensuring consistent migration of concerns.

Among the main risks of refactoring, we identify the high level of coupling and complex dependencies between the base code and the crosscutting concern. We anticipate that any non-trivial aspect refactoring will require object-oriented refactorings, before the crosscutting concern can be taken out of the available system.

The issue with coupling is that, before migration, concern code can freely access certain parts of the core code that may have limited visibility after the migration. Possible risks in such a case are weakening the visibility restrictions of those members or violating encapsulation by declaring the aspect *privileged*. Other risks include code duplication in advice and the advised method or definition of artificial pointcuts to capture return values of calls from the advised method; this could be the case when some control logic is required by both aspect and the advised method.

We encountered several complex dependencies while refactoring instances of *Exception propagation* in JHOTDRAW (see Section 4.4.1 for a description of *Exception-Propagation* in JHOTDRAW). One example is the propagation of the *IOException* rooted in the set of methods to read drawings from file. The methods in the call chain re-throwing the exception override other methods, whose declared thrown exceptions might only serve for compliance with the method to be refactored. In this case, we also need to address their *throws* clause within our refactoring. Moreover, the overriding elements of a method in the chain that throw the same exception need to be refactored as well, as their exception declaration is no longer allowed.

Table 6.3 summarizes the above risks and limitations in refactoring to aspects. Note that many of these limitations are independent of the strategy employed for refactoring. In spite of that, we are not aware of other papers in the area of refactoring to aspects that discuss these limitations.

6.6.3 Automation

The refactoring step in our strategy currently has the least automation of all steps in our approach. However, the other (tool supported) steps give us many of the elements needed for refactoring, such as the crosscutting element and the context it cuts across which are captured in the concern documentation as repeatable queries. Moreover, the description of these elements by the sort-queries is similar in many respects to the definition of pointcuts for a possible aspect solution.

We believe that the case study presented in this chapter is a required step before setting out to design (automated) aspect refactoring tooling. The study gives us insight into the complexity of each refactoring and the trade-offs to be made. The challenges and limitations discussed in the previous sections also indicate that completely automated aspect refactoring is unfeasible in any practical situation, since the process requires a significant level of interaction with the users to guide the system through the right decisions.

A particularly challenging automatic refactoring would be the one for *Redirection layer* instances: the original, dynamic solution uses a common interface for both redirectors and potential receivers. This interface hides the identity of the object for which a call is made; However, the refactoring of redirectors requires to know which calls are meant for a redirector and so need to be attached an advice introducing the functionality of the refactored redirector.

6.6.4 Separation of Concerns

Our case study had a satisfactory outcome in achieving a better separation and modularization of concerns in the targeted application. As we were able to notice, the crosscutting code is an important part of the refactored elements, in some cases, such as the *Command* elements, over 50%.

We appreciate that the core code is easier to understand in the absence of the migrated crosscutting concerns. To understand the aspect code, on the other hand, one typically also needs to understand the base code that it advises. This is exaggerated further by (high) coupling between the aspect and the base code, like for aspects that intercept calls from advised methods to reuse the values returned by such calls.

While refactored, crosscutting-free code is easier to comprehend, modifications to such code would still require awareness of the advice that applies to it. For instance, aspects might assume a certain order of the calls from an advised method, which has to be preserved to correctly introduce additional behavior.

Keeping track of the order of different advice in an aspect solution and preventing accidental changes might prove difficult, particularly when the number of aspects increases. The support from present development environments would not provide much insight into violations of such ordering, or into the ordering itself. This becomes more of an issue when the order is set using name-based wildcards, and new aspects match an existing rule for aspect precedence that should not apply to them. A similar situation

might occur when changing an aspect solution that is already covered by a precedence rule, and the changes would not be compliant with that rule. Changing the position of an advice definition in an aspect could also modify precedence, if multiple advices in the aspect apply to the same joinpoints. Unspecified precedence could also lead to interference between new advices introduced by refactoring and existing ones Storzer and Forster [2006]. Automatic refactoring needs to be aware of these issues when adding advices to an aspect source file.

Some concerns might be crosscutting for advices, similarly to the way they are crosscutting for methods. For instance, the re-use of specialized enumerations in JHOTDRAW requires to reset them after each iteration. Such enumerations are used by some advices in the aspect solutions. Applying aspect solutions to aspects might prove challenging for both tool support and comprehensibility.

6.7 Related Work

While each step in the migration of crosscutting concerns has been addressed by related research, we are not aware of an integrated strategy like the one proposed in this chapter.

The present approaches to aspect refactoring can generally be distinguished by their granularity. Laddad's set of refactorings cover both low level ones, such as *extract method calls into aspects* or *extract interface implementation*, as well as more complex refactorings, like design patterns, transactions management, or business rules [Laddad, 2003b,a]. Although the latter subset typically involves multiple concerns to be refactored, there is no categorization of these various concerns or their refactorings.

Hannemann et al. propose an approach to the aspect refactoring of design patterns based on a library of abstract roles [Hannemann et al., 2005; Hannemann and Kiczales, 2002]. The role-based refactoring requires one to map a pattern's implementation onto the predefined roles describing the pattern, and then applies a set of instructions to refactor the implementation to aspects. The approach is a step further towards generic, abstract solutions to typical problems that involve crosscutting functionality. However, as we have already seen, these patterns typically have a complex (and variable) structure in source code, which exhibits multiple (atomic) crosscutting concerns. The refactoring of a whole pattern in one step might prevent the comprehension of the concerns involved. Moreover, our experience suggests that pattern implementations can vary significantly from a standard description and one-step refactoring could be hampered by complex dependencies. We cannot make a full assessment of this approach as the implementation and the experimental results are not available, but we believe that all the limitations discussed in this chapter would equally apply to it.

Finer-grained refactorings have been proposed in the form of code transformations catalogs [Monteiro and Fernandes, 2005] and AspectJ laws [Cole and Borba, 2005]. These transformations can occur as steps in the aspect refactoring of an (atomic) crosscutting concern, but remain oblivious to the refactored concern. They describe the

mechanics of migrating Java specific units to AspectJ ones (e.g., *Extract Fragment into Advice*, *Move Method/Field from Class to Inter-type*). Similar approaches have been proposed Hanenberg et al. [2003], and Ettinger and Verbaere [2004], who employ program slicing for refactoring to aspects. Such small step transformations might benefit the implementation of automatic refactorings by preventing complex dependencies and ensuring behavior preservation as discussed by Cole and Borba [2005]. However, more effort is required to assess their general applicability: for example, the case-study used for the refactoring in Monteiro and Fernandes [2005], is an *Observer* pattern implemented in a demonstrative application, which lacks the complexity of a real system like JHOTDRAW.

In comparison to the work on fine-grained refactorings, the sort-based approach presented in this chapter emphasizes concerns and identifies common properties at a consistent granularity level. This allows us to design a complete migration strategy, where the refactoring is integrated with steps for concern identification and comprehension.

Similar observations also apply to the comparison with the refactoring approach by Binkley et al. [2006]. Their emphasis is on full automation, and they offer an Eclipse plugin for conducting six elementary refactorings. They focus on our fourth step only, and assume aspect mining has resulted in `@begin-aspect` and `@end-aspect` annotations in the code. As an example, one of their six refactorings moves individual calls to separate aspects, after which a (non-trivial) pointcut abstraction step is needed to merge the results. Our approach eliminates the need for this complex abstraction step, thanks to the sort-based integration between aspect mining and refactoring (refactoring is based on a full concern model in our case). Like us, they use JHOTDRAW as one of their case studies. Somewhat surprisingly, they do not report any of the limitations that we identified, although their results exhibit the same limitations.

6.8 Concluding Remarks

In this chapter, we proposed an integrated strategy for migrating crosscutting concerns to aspect-oriented programming. We presented in detail the refactoring step of our strategy, and applied the entire migration process to concerns in an open-source application. Furthermore, we discussed the challenges of refactoring crosscutting concerns to aspects and how these could impact the design and implementation of automatic aspect refactoring.

The contributions of this work can be summarized as:

- An integrated strategy for migrating crosscutting concerns to AOP solutions;
- An aspect refactoring approach based on crosscutting concern sorts and a set of refactoring templates;
- A report on our experience with migrating concerns in a real system to aspects and the challenges of this process. This report is useful for assessing the present

support for refactoring and the feasibility of automatic aspect refactoring for various categories (that is, sorts) of crosscutting concerns.

- AJHOTDRAW, a show-case for aspect refactoring in an open-source implementation that can be further used by researchers and practitioners to evaluate aspect-based solutions to crosscutting concerns.

AJHOTDRAW provides a code base for related research to measure code improvements due to aspect code. Furthermore, this work provides us with the hands-on experience for designing and implementing sort-based aspect refactoring. We plan to extend our tool support for concern documentation, SOQUET, with aspect refactoring options. The refactoring would apply to each query documenting a sort instance, and hence benefit from the description of the concerns available by the query results. We appreciate that a significant effort would go into the design and implementation of wizards to deal with the various reported challenges.

Crosscutting concerns are a main challenge to program comprehension, and hence to evolution of existing software systems. Their scattered and tangled implementation makes it difficult to locate and understand these concerns, to change their implementation, and to extend a system consistently with its various concerns.

This thesis has focused on better understanding how crosscutting concerns are implemented in existing systems, and how we can support effective software evolution in the presence of such concerns. Particularly, we have addressed three main challenges in managing crosscutting concerns in source code:

- *Crosscutting concern identification;*
- *(Query-based) Crosscutting concern documentation and modeling; and*
- *Concern refactoring to aspects.*

7.1 Summary of Contributions

Each of these challenges is a research topic on its own and we can summarize our contributions in several research areas of software evolution:

- *Program comprehension:* We provide a detailed picture of various (implementations of) crosscutting concerns in source code by analyzing and reporting on a number of open-source Java systems, from different application domains. Our findings also include a number of concerns not previously covered in literature, such as *Undo* support.

This study of the crosscutting concerns concludes with a classification of *atomic* concerns in *sorts* based on their distinctive properties, such as specific relations and implementation idioms. The classification allows for consistency in describing and addressing concerns at source code level. We describe the concern sorts

using source code queries that we implement in SOQUET, our tool supporting persistent, query-based documentation and modeling of concerns. Documentation of concerns allows us to build common benchmarks for concern analysis.

- *Reverse engineering*: One contribution of this thesis is the set of three aspect mining techniques implemented in a freely available tool called FINT, which supports (semi-)automatic identification of concerns in source code. Besides each individual technique, we discuss criteria and challenges to comparison and combination of techniques. The discussion includes reports on a joint-effort with two other research groups, and motivates the need for a common framework and coherent criteria to assess mining techniques and to support combinations aimed at improved quality of the mining results.

We propose such a framework for design, assessment and combination of mining approaches and show how new techniques can be built on top of it. Moreover, we cover in a survey existing approaches to aspect mining and discuss how these can be retrofitted to this framework.

- *Program transformation – Aspect-oriented refactoring*: The refactoring solutions that we propose ensure a number of important properties: (1) the refactorings address meaningful concerns, which allows for transparency of the refactored concerns; (2) the solutions address concerns at a consistent level of granularity, so solutions do not overlap but complement each other; (3) the solutions permit for a high level of flexibility so they can be applied to various implementations of concerns in source code.

Besides the refactoring approach itself, we implement AJHOTDRAW, the largest openly available refactoring to aspects. This open source project is a show-case for our approach, and also provides a comparative implementation of crosscutting concerns in Java and AspectJ respectively.

The feedback from the refactoring case presented in this thesis can also be useful to researchers working on design and implementation of aspect-oriented languages: we report about main challenges in refactoring to aspects due to language limitations, and present a number of considerations on whether the aspect-oriented code improves comprehensibility and evolvability with respect to an object-oriented one.

These contributions summarize our achievements in answering the research question that we proposed to address in the beginning of this thesis: *How can we consistently manage, i.e. identify, model, document and possibly migrate, crosscutting concerns in existing systems in order to better support program comprehension and effective software evolution?*

7.2 Discussion and Evaluation

7.2.1 Revisiting Thesis Objectives

In the beginning of this thesis, we endeavored to meet a set of objectives with our solution for enhancing management of crosscutting concerns in source code. We believe that the aforementioned contributions ensure that we have successfully achieved each of these objectives.

First, the categorization system based on sorts allows us to address crosscutting concerns in a systematic way. We use distinctive properties, such as implementation idioms and underlying relations, to describe concerns and to design solutions for their identification, modeling, and refactoring.

Second, our quest to better understand crosscutting concerns and how they occur in practice resulted in an openly available set of tools for aspect mining and concern documentation. Moreover, we produced comprehensible reports on our analysis of crosscutting concerns in three relevant open source systems, which are already used as common benchmarks (in aspect mining).

In Chapter 5, we used the concern sorts to develop a common framework for evaluation of aspect mining techniques. Conformance with this framework ensures our third objective, namely to consistently assess, compare, and combine mining techniques.

Concern sorts also act as a glue between the various steps towards migration of crosscutting concerns to aspects. This allows for integration of these steps in a coherent solution for porting object-oriented, crosscutting implementation of concerns to more modular solutions.

Finally, we achieve flexibility and re-usability of our concern management solutions by raising the abstraction level of addressing crosscutting concerns to categories of concerns.

7.2.2 Independent and Integrated Migration Steps

Identification, documentation and modeling, or refactoring of concerns to aspects show different steps towards migration of concerns in source code. One way of using the contributions of this thesis is to apply each step independently to conduct software engineering tasks. For instance, aspect mining results point us to relevant program elements and relations that show important code characteristics, but also give a quick insight into the (design and features of the) analyzed code. Besides the case-studies that we have covered in detail in this thesis, we obtained similar such results for other systems as well. The results of applying *Fan-in* analysis to the JBOSS server application, for example, show that logging for debugging operations is a main, widely scattered concern: the four methods with top fan-in values, from 444 to 966, implement this concern. The next top values point us to the server component for registration and management of bean (*MBean*) objects, which is a core element in the (Java Management Extension (JMX)) architecture of the analyzed system.

The metric not only helps us to recognize these concerns in the code, but it also gives us a good estimate of the impact and costs of changing them.

The search for other concern idioms, like redirections as implemented by wrapper objects, helps us to recognize design decisions and the extent of their implementation in the code. The presence of a wrapper, for instance, could indicate that a given (wrapped) object should be used via its wrapper instead of accessing it directly.

Documentation of concerns is aimed at a (persistent) representation of relevant relations in source code and of design decisions that are not transparent in the dominant decomposition of the system under investigation. A main goal of our approach consist of ensuring a structured, coherent understanding and representation of crosscutting concerns. This allows developers to consistently describe their design decisions by creating concern-based models of the source code that complement the main decomposition of concerns, which is based on modularization mechanisms of the employed language. New developers can use this documentation to investigate existing (crosscutting) concerns and designs, and check whether changes or additions to a code element conform to its concerns.

The refactoring approach we propose follows similar coherence considerations as outlined for documentation of concerns. By these considerations we distinguish from other approaches, most of which overlook an explicit representation of the concerns used as inputs for refactoring to aspects: For example, fine-grained refactorings, which describe how class members or relations can be migrated to aspects declarations, assume a priori understanding of the concerns and of the program elements that implement them. Then the refactoring solutions need to be applied to each of these elements. Other approaches consist of solutions for heterogeneous examples of concerns, which require one to map the concerns to be refactored onto these particular examples.

Overall, our refactorings based on concern sorts look to provide flexible solutions for replacing the crosscutting implementation of concerns with aspects, and hence for improving modularity and separation of concerns in the code by means of aspect languages mechanisms.

We observe that while useful on their own, as discussed above, the three different steps in migration of concerns assume inputs (such as concerns to be documented or refactored), which would typically be provided by the other steps (like the mining one). Our classification of concerns in sorts allows for consistency for each of these steps, but also for their integration. The integration of the three steps in a migration strategy, as proposed in this thesis, and the application of the strategy to the AJHOTDRAW case are the first attempts of their kind.

7.2.3 Queries versus Aspects

Both the query-based documentation and the refactoring to aspects are aimed at enhancing comprehensibility of crosscutting concerns by making certain relations in source code explicit or by improving modularity. The two migration steps can be regarded as complementary, but also as alternatives to adoption of concern-driven design

and aspect-oriented programming.

Our experience, reported in this thesis, shows a number of relevant differences between the two steps, which may impact on the choice of the adoption strategy for approaches to crosscutting concerns. One such difference consists of the modification required to the code base: while queries simply report on existing relations, without requiring code restructuring, aspects introduce behavior and need to modify the code. We can say that queries are a *passive* addition to the code, by comparison with the aspects that are *active* elements. This difference has both advantages and disadvantages on each side: Because their active nature, aspects can enforce rules in the code so that they cannot be forgotten, such as having a consistent behavior attached to new elements that are covered by a defined pointcut. On the other side, such behavior could be added erroneously modifying the correct behavior of a system. Preventing such errors requires advanced tool support for aspect programming and exhaustive regression testing. Therefore, developers might find adoption of a query-based approach safer.

Documentation of crosscutting concerns, on the other hand, preserves an “optional” characteristic. Many reports suggest that code often lacks (updated) documentation, and, as a code can function correctly without documentation, there is no strict enforcement on providing it. Yet, we believe that queries have several advantages over traditional, textual documentation that might ease the task of developers of describing the design and the concerns in their code: the sort queries, for instance, allow for a structured, systematic way of producing documentation, which is supported by query templates. Moreover, queries help developers to keep track of their design, and to describe concerns in a consistent way, without the modularization limitations of a certain programming paradigm. A disadvantage here is the possible limited expressiveness of a query, but tool support for concern queries can easily integrate textual descriptions attached to a query.

Query-based documentation of concerns contributes to code comprehensibility by providing hints to relevant relations between program elements. This documentation can be simply assessed as a complement to the available source code for understanding a program. For aspects, however, assessing improvement in code comprehensibility due to their use is yet an open issue. AJHOTDRAW is just a first step towards such an assessment that allows us to compare different implementation of a same design in a software application.

One possible advantage of queries over aspects consist of the involved complexity for ensuring comprehensible, expressive descriptions of concerns. The queries remove part of this complexity as they do not need an equivalent of the advice construct. For example, the query for a concern implemented by method calls, only needs to specify the rule of selecting those call sites that are relevant to the concern implementation. The aspect solution needs to define a pointcut whose definition is similar to the selection rule in the query, but also the advice to introduce the call implementing the refactored concern. The complexity of extracting an advice for refactoring depends on the level of tangling of the concern at the call sites, and it is often not a trivial task as the call to be extracted is not always a simple *before/after* one.

It is important to notice that despite being a key challenge to describing concerns, the problem of expressive pointcuts is typically overlooked by most of the approaches to refactoring, and even modeling.

7.3 Opportunities for Future Research

Each of the chapters of this thesis discusses a number of open issues in the various steps of the management of crosscutting concerns in source code. Some of the main issues include:

- The need for further extensive reports on the coverage of various crosscutting concerns by sorts including, for example, other domain specific concerns.
- Investigating additional metrics for the assessment of mining techniques including, for instance, measures for concern coverage of an aspect mining result.
- Integration of the refactoring to aspects into software development environments.

This section suggests several directions for future work covering the issues above.

7.3.1 Aspect Mining

In the discussion of our proposed common framework for aspect mining we have covered existing techniques and showed how they can fit into the framework. Actual implementation of these techniques and experiments on common benchmarks would allow us to assess their results and experiment with combinations.

Available tool support and publicly shared, detailed results are a key element to better understanding crosscutting concerns and recognizing typical implementation idioms. However, this support is not readily available, and the reports in this thesis are (among) the most comprehensive ones to date. A larger variety of examples from different application domains would also be helpful to assess the feasibility of existing aspect-oriented languages, as well as of our sorts, to cope with various crosscuttingness.

To better understand the sorts of crosscuttingness, we would like to have at least one aspect mining technique targeting each of the sorts. The description of concerns by sorts could be particularly useful here, as new mining techniques can be designed to search for a sort's relation, such as call, implement, redirection, or parameter passing relations. We have found particularly insightful to experiment with the various techniques proposed in this thesis and to discover a significant number of instances for the targeted sorts. Such techniques give more empirical evidence on the use of idioms to implement concerns.

Combination of techniques is still in its infancy and this thesis covers the largest part of the efforts made up to date. The promising results we obtained suggest that this

is an interesting direction to further explore for improving quality of aspect mining results.

7.3.2 Crosscutting Concern Documentation and Modeling

Besides extending the list of our sorts and better understanding what other typical concerns we encounter in source code, we would also like to see how a query-based approach is able to support description of new sorts of concerns. The feasibility of this approach depends on mainly two elements: the ability to query for a sort's specific relation, and the flexibility to formalize contexts, i.e., to restrict the endpoints of a relation to relevant elements, similarly to defining pointcuts in aspect-oriented languages. Furthermore, a sort and its query have to be able to provide an abstract representation of all its instances.

The assessment of the approach to concern documentation based on queries requires effective tool support. We identify a number of desirable extensions to SOQUET, which include improved integration with traditional refactoring operations. For example, changes in the source code, like renaming or removal of elements, should update the definition of the queries in the concern model accordingly.

Another desirable extension consists of improved support for querying a concern model (that documents a system) to find relevant queries attached to a given program element. For example, when changing a *Figure* class in a drawing application by adding a new method, it is relevant to know that figures participate, for instance, in an Observer design and any modification to the figure's state has to be notified to the figure's display. In this case, we would like to search in the concern model of our whole application for those queries that document concerns for *Figure* elements. This search feature is only partially implemented in SOQUET at the moment: the search for concerns attached to the *Figure* class returns all queries for which *Figure* is an endpoint of the crosscutting relation documented by the query. One such result shows us that figures are *Observable* elements, and hence implement multiple roles. However, the query documenting the call to notify the figure's display of changes will not be reported, since the *Figure* class is not an endpoint of the call relation, but only one of its methods. Yet, for this particular case, the notification concern is relevant for the intended change operation.

The search for relevant concerns in concern models requires clearly defined rules to associate queries to program elements. The rules should specify when a particular program element is considered related to a query, so the query is reported as relevant for the relations of that element. Such extensions to SOQUET can be summarized as desired support for querying queries.

7.3.3 Refactoring to Aspect-Oriented Programming

A natural extension of the work presented in this thesis is to build tool support for (semi-)automatic refactoring based on the solutions proposed for concern sorts. Pre-

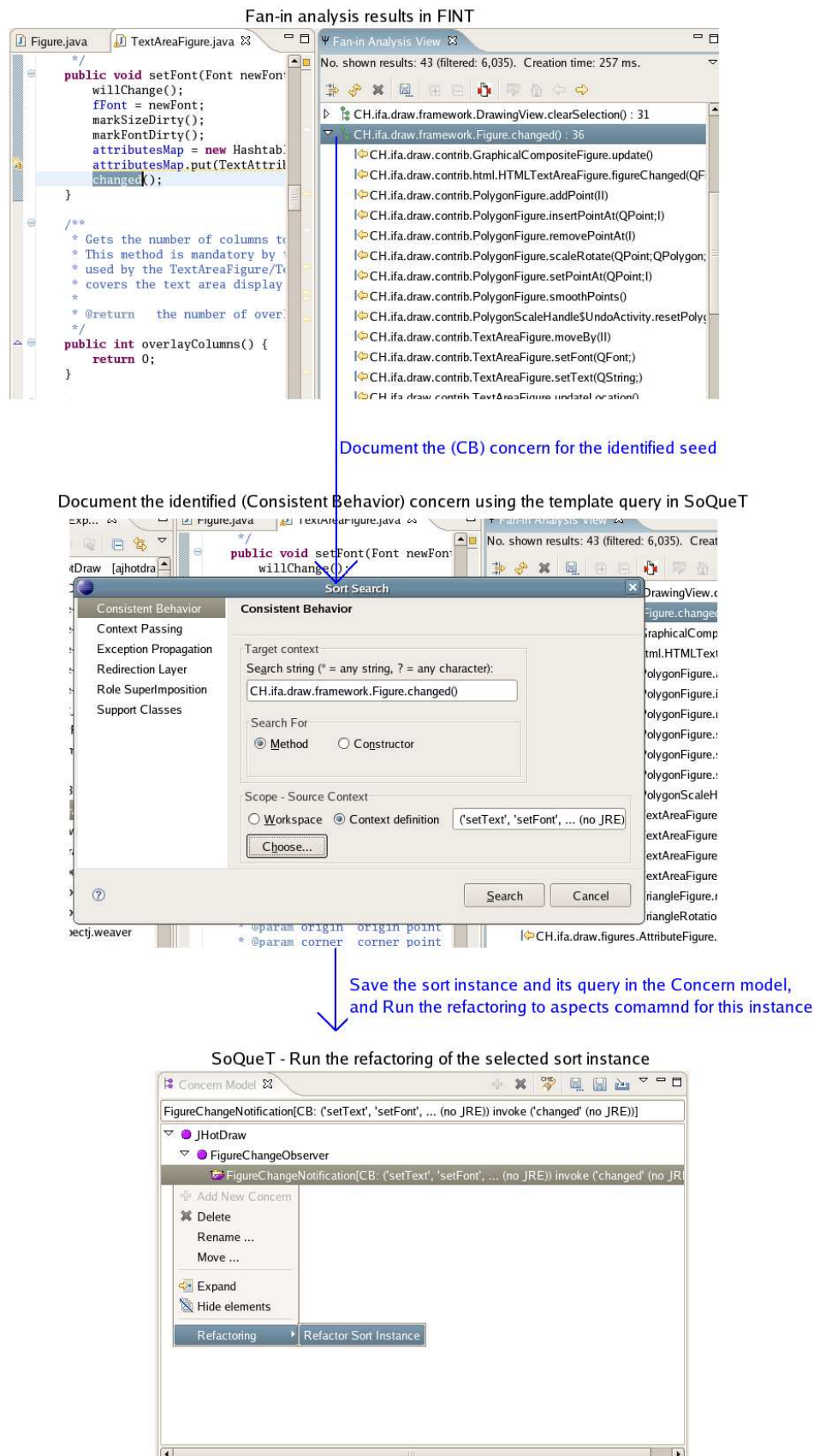


Figure 7.1: Integration of migration steps in FINT and SoQUET.

liminary experiments show promising results, mainly because the granularity of the sorts allows for increased flexibility of the refactorings. However, more work is required on this tool support implementation.

Another interesting direction to investigate comprises testing strategies to ensure behavior preservation of the refactoring to aspects. We would like to attach to each of our refactorings a testing component that automatically checks for faults after the execution of the refactoring.

7.3.4 Integration of Migration Steps

FINT and SOQUET provide us with a proper infrastructure for an integrated migration strategy. A simple extension that we plan to add to FINT would allow us to turn mining results into input parameters for the query templates in SOQUET. For instance, the mining results of *Fan-in* analysis, which are displayed by FINT as call relations, could be turned (automatically) into set of elements that describe the contexts for the *Consistent behavior* sort query, as shown in Figure 7.1.

Furthermore, SOQUET implements support for extending the query-based documentation of a sort instance in the concern model with options for refactoring it to aspects. This support includes mechanisms for reusing the information in the query documenting a concern to configure the refactoring solution for the concern's sort. The tool-supported integration of these migration steps is briefly outlined in Figure 7.1.

7.4 Closing Remarks

The work presented in this thesis advances the state-of-the-art in the management of crosscutting concerns in source code by means of a significant set of techniques, detailed case-studies reports and tool support. Moreover, we presented the first integrated approach to management and migration of concerns. These contributions are aimed at helping software engineers to better deal with the complexity of existing systems and with the tasks of evolving such systems.

In the last chapter of this thesis, we identify a number of research topics that, we believe, point to relevant and interesting challenges in the area of software engineering and concern management in particular.

Bibliography

- Alur, D., Crupi, J., and Malks, D. *Core J2EE Patterns*. Sun Microsystems, Inc., USA [2003].
- The AspectJ Team. *The AspectJ Programming Guide*. Palo Alto Research Center [2003]. Version 1.2.
- Baldwin, C.Y. and Clark, K.B. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA [1999].
- Beck, K. *Smalltalk: best practice patterns*. Prentice-Hall [1997].
- Bergmans, L. and Aksit, M. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57 [2001].
- Biggerstaff, T.J., Mitbender, B.G., and Webster, D.E. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82 [1994].
- Binkley, D., Ceccato, M., Harman, M., Ricca, F., and Tonella, P. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering*, 32(9):698–717 [2006].
- Binkley, D., Ceccato, M., Harman, M., Ricca, F., and Tonella, P. Automated refactoring of object oriented code into aspects. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pages 27–36. IEEE Computer Society, Washington, DC, USA [2005].
- Bloch, J. *Effective Java programming language guide*. Sun Microsystems, Inc., Mountain View, CA, USA [2001].
- Breu, S. and Krinke, J. Aspect mining using event traces. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE '04)*, pages 310–315. IEEE Computer Society, Washington, DC, USA [2004].

- Breu, S. and Zimmermann, T. Mining aspects from version history. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE '06)*, pages 221–230. IEEE Computer Society, Washington, DC, USA [2006].
- Briand, L.C., Daly, J.W., and Wüst, J.K. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121 [1999].
- Bruntink, M., van Deursen, A., van Engelen, R., and Tourwé, T. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM '04)*, pages 200–209. IEEE Computer Society, Los Alamitos, CA [2004].
- Bruntink, M., van Deursen, A., van Engelen, R., and Tourwé, T. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818 [2005].
- Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., and Tourwe, T. A qualitative comparison of three aspect mining techniques. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05)*, pages 13–22. IEEE Computer Society, Washington, DC, USA [2005].
- Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., and Tourwé, T. Applying and combining three different aspect mining techniques. *Software Quality Journal*, 14(3):209–231 [2006]. Included as Chapter 3 of this thesis.
- Cole, L. and Borba, P. Deriving refactorings for AspectJ. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 123–134. ACM Press, New York, NY, USA [2005].
- Colyer, A., Clement, A., Harley, G., and Webster, M. *Eclipse AspectJ*. Pearson Education, Inc., NJ [2005].
- Deransart, P., Ed-Dali, A., and Cervoni, L. *Prolog, The Standard : Reference Manual*. Springer Verlag [1996].
- van Deursen, A., Quilici, A., and Woods, S. Program plan recognition for year 2000 tools. *Science of Computer Programming*, 36:303–324 [2000].
- Dijkstra, E.W. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA [1997].
- Eichberg, M., Haupt, M., Mezini, M., and Schäfer, T. Comprehensive software understanding with Sextant. In *Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pages 315–324. IEEE Computer Society [2005].

- Eick, S.G., Steffen, J.L., and Eric E. Sumner, J. Seesoft-A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968 [1992].
- Eisenbarth, T., Koschke, R., and Simon, D. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):195–209 [2003].
- Erlikh, L. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23 [2000].
- Ettinger, R. and Verbaere, M. Untangling: a slice extraction refactoring. In *Proceedings of the 3rd International Conference on Aspect-Oriented software Development (AOSD '04)*, pages 93–101. ACM Press, New York, NY, USA [2004].
- Fabry, J. *Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code*. Ph.D. thesis, Vrije Universiteit Brussel [2005].
- Ferenc, R., Beszédes, Á., Fulop, L., and Lele, J. Design pattern mining enhanced by machine learning. In *Proceedings 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pages 295–304. IEEE Computer Society, Los Alamitos [2005].
- Filman, R.E., Elrad, T., Clarke, S., and Akşit, M., editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston [2005].
- Fowler, M. Inversion of control containers and the dependency injection pattern. <http://martinfowler.com/articles/injection.html> [2004].
- Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA [1999].
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA [1994].
- Ganter, B. and Wille, R. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag [1997].
- Gil, J.Y. and Maman, I. Micro patterns in java code. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA '05)*, pages 97–116. ACM Press, New York, NY, USA [2005].
- Goetz, B. Garbage collection and performance. IBM developersWorks articles [2004]. www-136.ibm.com/developerworks/java/.

- Gradecki, J.D. and Lesiecki, N. *Mastering AspectJ - Aspect Oriented Programming in Java*. Wiley Publishing, Inc., Indianapolis, Indiana [2003].
- Griswold, W.G., Yuan, J.J., and Kato, Y. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, pages 265–274. IEEE Computer Society, Washington, DC, USA [2001].
- Gybels, K. and Kellens, A. Experiences with identifying aspects in Smalltalk using unique methods. In *Proceedings of the 1st Workshop on Linking Aspect Technology and Evolution (LATE '05) at AOSD* [2005].
- Hajiyev, E., Verbaere, M., and de Moor, O. CodeQuest: Scalable source code queries with Datalog. In D. Thomas, editor, *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP '06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer [2006].
- Hanenbergh, S., Oberschulte, C., and Unland, R. Refactoring of aspect-oriented software. In *Proceedings of the 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 19–35 [2003].
- Hannemann, J. and Kiczales, G. Overcoming the prevalent decomposition of legacy code. In *Workshop on Advanced Separation of Concerns at ICSE* [2001].
- Hannemann, J. and Kiczales, G. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, pages 161–173. ACM Press, Boston, MA [2002].
- Hannemann, J., Murphy, G.C., and Kiczales, G. Role-based refactoring of crosscutting concerns. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 135–146. ACM Press, New York, NY, USA [2005].
- Harrison, W., Ossher, H., Jr., S.M.S., and Tarr, P. Concern modeling in the concern manipulation environment. In *IBM Research Report RC23344*. IBM Thomas J. Watson Research Center, Yorktown Heights, NY [2004].
- Henderson-Sellers, B., Constantine, L.L., and Graham, I.M. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3:143–158 [1996].
- Henderson-Sellers, B. *Object-oriented metrics : measures of complexity*. Prentice-Hall, Inc. [1996].

- Henry, S. and Kafura, K. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518 [1981].
- Janzen, D. and Volder, K.. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 178–187. ACM Press, New York, NY, USA [2003].
- Johnson, R. *J2EE Design and Development*. Wiley Publishing, Indianapolis, IN [2003].
- Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., and Irwin, J. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York [1997].
- Kiczales, G. and Mezini, M. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 49–58. ACM Press, New York, NY, USA [2005a].
- Kiczales, G. and Mezini, M. Separation of concerns with procedures, annotations, advice and pointcuts. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP '05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer [2005b].
- Kitchenham, B., Pickard, L., and Pfleeger, S.L. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62 [1995].
- Koschke, R. and Quante, J. On dynamic feature location. In *Proceedings 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, pages 86–95. ACM Press, Boston, MA [2005].
- Krinke, J. Mining control flow graphs for crosscutting concerns. In *Proceedings of 13th Working Conference on Reverse Engineering (the 9th ASTReNet Workshop)*, pages 334–342. IEEE Computer Society, Washington, DC, USA [2006].
- Laddad, R. Aspect-oriented refactoring. www.theserverside.com [2003a].
- Laddad, R. *AspectJ in Action - Practical Aspect Oriented Programming*. Manning Publications Co., Greenwich, CT [2003b].
- Lesiecki, N. Aop@work: Enhance design patterns with AspectJ. www-128.ibm.com/developerworks [2005].
- Lindig, C. Fast concept analysis. In *Working with Conceptual Structures - Contributions to ICCS 2000*, pages 152–161. Shaker Verlag [2000].

- Lippert, M. and Lopes, C. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 418–427. ACM Press, Boston, MA [2000].
- Marin, M. Refactoring JHOTDRAW's undo concern to AspectJ. In *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE '04) at WCRE.*, pages 24–30. CWI Report SEN-E0502, Amsterdam, The Netherlands [2004].
- Marin, M. Formalizing typical crosscutting concerns. Technical Report TUD-SERG-2006-010, Delft University of Technology [2006a].
- Marin, M., van Deursen, A., and Moonen, L. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE '04)*, pages 132–141. IEEE Computer Society, Los Alamitos, CA [2004].
- Marin, M., van Deursen, A., and Moonen, L. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–37 [2007a]. Included as Chapter 2 of this thesis.
- Marin, M., Moonen, L., and van Deursen, A. A classification of crosscutting concerns. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pages 673–677. IEEE Computer Society, Los Alamitos [2005a]. (Partially) covered by Chapter 4 of this thesis.
- Marin, M., Moonen, L., and van Deursen, A. A systematic aspect-oriented testing and refactoring process, and its application to JHOTDRAW. Technical Report SEN-R0507, CWI [2005b].
- Marin, M., Moonen, L., and van Deursen, A. A common framework for aspect mining based on crosscutting concern sorts. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pages 29–38. IEEE Computer Society, Washington, DC, USA [2006a]. Extended version included as Chapter 5 of this thesis.
- Marin, M., Moonen, L., and van Deursen, A. FINT: Tool support for aspect mining. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pages 299–300. IEEE Computer Society, Washington, DC, USA [2006b]. (Partially) covered by Chapters 2, 3 and 5 of this thesis.
- Marin, M. Reasoning about assessing and improving the seed quality of a generative aspect mining technique. In *Proceedings of the 2nd Workshop on Linking Aspect Technology and Evolution (LATE '06) at AOSD*, pages 23–27. CWI Report SEN-E0604 [2006b]. (Partially) covered by Chapters 3 and 5 of this thesis.
- Marin, M., Moonen, L., and van Deursen, A. An approach to aspect refactoring based on crosscutting concern types. *SIGSOFT Software Engineering Notes*, 30(4):1–5 [2005c]. (Partially) covered by Chapter 4 of this thesis.

- Marin, M., Moonen, L., and van Deursen, A. Documenting typical crosscutting concerns. In *Proceedings of the 14th IEEE Conference on Reverse Engineering (WCRE '07)*. IEEE Computer Society, Washington, DC, USA [2007b]. Covered by Chapter 4 of this thesis.
- Marin, M., Moonen, L., and van Deursen, A. An integrated strategy to crosscutting concern migration and its application to JHOTDRAW. In *Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '07)*. IEEE Computer Society, Washington, DC, USA [2007c]. Covered by Chapter 6 of this thesis.
- Marin, M., Moonen, L., and van Deursen, A. SOQUET: Query-based documentation of crosscutting concerns. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 758–761. IEEE Computer Society, Washington, DC, USA [2007d]. (Partially) covered by Chapter 4 of this thesis.
- Mens, K. and Tourwé, T. Delving source-code with formal concept analysis. *Elsevier Journal on Computer Languages, Systems & Structures*, 31(3–4):183–198 [2005].
- Mens, K., Kellens, A., Pluquet, F., and Wuyts, R. Co-evolving code and design with intensional views: A case study. *Computer Languages, Systems & Structures*, 32(2–3):140–156 [2006].
- Mens, K., Kellens, A., and Tonella, P. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, 4(Special Issue on Software Evolution):145–164 [2007].
- Mens, K., Poll, B., and González, S. Using intentional source-code views to aid software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM '03)*, pages 169–178. IEEE Computer Society, Washington, DC, USA [2003].
- Mesbah, A. and van Deursen, A. Crosscutting concerns in J2EE applications. In *Proceedings of the 7th International Symposium on Web Site Evolution*, pages 14–21. IEEE Computer Society, Los Alamitos, CA [2005].
- Monteiro, M. Catalogue of refactorings for AspectJ. Technical Report UM-DI-GECS-200401, Universidade do Minho [2004].
- Monteiro, M. and Fernandes, J. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 111–122. ACM Press, New York, NY, USA [2005].
- Moodie, M. *Pro Jakarta Tomcat 5*. Apress, Berkely, CA [2005].

- Murali, T., Pawlak, R., and Younessi, H. Applying aspect orientation to J2EE business tier patterns. In Y. Coady and D. Lorenz, editors, *Proceedings of the 3rd Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) at AOSD*, pages 55–61. University of Victoria, Victoria, Canada [2004].
- Murphy, G.C., Griswold, W.G., Robillard, M.P., Hannemann, J., and Leong, W. Design recommendations for concern elaboration tools. In R.E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 507–530. Addison-Wesley, Boston [2005].
- Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058 [1972].
- Pigoski, T.M. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., New York, NY, USA [1996].
- Porter, M. An algorithm for suffix stripping. *Program*, 14(3):130–137 [1980].
- Rich, C. and Wills, L.M. Recognizing a program’s design: A graph-parsing approach. *IEEE Software*, 7(1):82–89 [1990].
- Riehle, D. and Gross, T. Role model based framework design and integration. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA ’98)*, pages 117–133. ACM Press [1998].
- Robillard, M.P. and Murphy, G.C. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE ’02)*, pages 406–416. ACM Press, New York, NY, USA [2002].
- Robillard, M.P. and Murphy, G.C. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):3 [2007].
- Seiter, L. Automatic mining of context passing in java programs. In *Proceedings of the Workshop Towards Evolution of Aspect Mining (TEAM) at ECOOP*, pages 9–13. Delft University of Technology Report TUD-SERG-2006-012 [2006].
- Shepherd, D., Gibson, E., and Pollock, L. Design and evaluation of an automated aspect mining tool. In *Software Engineering Research and Practice*, pages 601–607. CSREA Press, Las Vegas, NV [2004].
- Shepherd, D., Palm, J., Pollock, L., and Chu-Carroll, M. Timna: a framework for automatically combining aspect mining analyses. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE ’05)*, pages 184–193. ACM Press, New York, NY, USA [2005a].

- Shepherd, D., Pollock, L., and Tourwé, T. Using language clues to discover crosscutting concerns. In *Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software (MACS '05) at ICSE*, pages 1–6. ACM Press, New York, NY, USA [2005b].
- Soares, S., Laureano, E., and Borba, P. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, pages 174–190. ACM Press [2002].
- Sommerville, I. *Software Engineering*. Pearson, NJ, 7th edition [2004].
- Storzer, M. and Forster, F. Detecting precedence-related advice interference. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE '06)*, pages 317–322. IEEE Computer Society, Washington, DC, USA [2006].
- Sutton, S.M. and Rouvellou, I. Concern modeling for aspect-oriented software development. In R.E. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*, chapter 21, pages 479–505. Addison-Wesley, Boston [2005].
- Tarr, P., Harrison, W., and Ossher, H. Pervasive query support in the Concern Manipulation Environment. Technical Report RC23343 (W0409-135), IBM TJ Watson Research Research Center, Yorktown Heights, NY [2004].
- Tarr, P., Ossher, H., Harrison, W., and Stanley M. Sutton, J. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 107–119. IEEE Computer Society Press, Los Alamitos, CA, USA [1999].
- Tonella, P. and Ceccato, M. Aspect mining through the formal concept analysis of execution traces. In *Proceedings 11th Working Conference on Reverse Engineering (WCRE '04)*. IEEE Computer Society, Los Alamitos, CA [2004a].
- Tonella, P. and Ceccato, M. Migrating interface implementation to aspect-oriented programming. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 220–229. IEEE Computer Society, Los Alamitos, CA [2004b].
- Tourwé, T. and Mens, K. Mining aspectual views using formal concept analysis. In *Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '04)*. IEEE Computer Society, Chicago, Illinois, USA [2004].
- Vickers, P. Why finalizers should (and can) be avoided. IBM developersWorks articles [2002]. www-136.ibm.com/developerworks/java/.

- Wilde, N. and Scully, M.C. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62 [1995].
- Wills, L.M. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45(1–2):113–171 [1990].
- Xie, X., Poshyvanyk, D., and Marcus, A. 3D visualization for concept location in source code. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pages 839–842. ACM Press, Boston, MA [2006].
- Yin, R.K. *Case Study Research: Design and Methods*. Sage Publications, USA, 3rd edition [2003].
- Zhang, C. and Jacobsen, H.A. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 130–139. ACM Press, Boston, MA [2003].
- Zhang, C. and Jacobsen, H.A. PRISM is research in aspect mining. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–21. ACM Press, Boston, MA [2004].

This appendix contains the FINT user manual. The underlying ideas are described in Chapters 2 and 5 of this thesis.

FINT is a (Java) source code analysis tool for detecting smells of crosscutting implementation of concerns. The tool is available as a plug-in for the Eclipse IDE (v.3.0.x – v.3.3).¹ The source code of FINT consists of 12,500 NCLOC.²

A.1 Installation

The installation procedure simply requires to download and save the “jar” distribution of the tool into the “plugins” directory of Eclipse and then (re-)start the IDE.

A.2 User manual

FINT implements three code analysis techniques that we shall see next in action:

- *Fan-in analysis* looks for crosscutting concerns by investigating all the method call relations in a system, and allowing the user to select those methods that are called from many different places, possibly from similar calling contexts. For example, a tracer for all the method executions in a system might consist of calls to a tracing method attached at the beginning of each of the system’s methods. The tracer method will have a large number of callers, and hence a high fan-in metric value, which makes it likely to be identified by our technique.

A typical refactoring to aspect-oriented programming (AOP) of the concerns identified by fan-in analysis consists of replacing the scattered method calls identified by this technique with pointcut and advice constructs.

¹Some of the figures may be more difficult to read on paper. We refer the reader to the FINT web site for the on-line version of this manual, in which the figures are available in high resolution.

²Metrics plug-in, v.1.3.6 – <http://metrics.sourceforge.net/>. Note that the Metrics tool does not count lines of code in interfaces.

- *Grouped calls analysis* targets similar code smells and concerns as the previous technique; however, instead of single method invocations, this technique is looking for groups of (at least two) methods that are called by the same callers. Examples of crosscutting concerns following this implementation idiom include programmatic (JTA) transaction management, where the transaction demarcation is realized by calls to methods such as `begin`, `commit`, and `rollback`.
- *Redirections finder* aims at identifying wrapper classes (such as decorators) by analyzing the methods of all the classes in the system under investigation for exclusive one-to-one call relations with methods of another class.

Each technique has a dedicated view for investigating and further refining the results. A fourth view allows us to collect and save the results that participate in, and hence point us to, crosscutting concerns; these results are crosscutting concern *seeds*. The views can be opened from Eclipse's "Window/Show View/Other..." menu, under the FINT group, as shown in Figure A.1.

A.2.1 Fan-in analysis

Fan-in is the default analysis in the tool. To run the analysis, the user chooses the program elements to be analyzed in the *Package Explorer* view of Eclipse and selects from the context menu of these elements (right click) the *Fan-in Analysis* option, as illustrated in Figure A.2.

The tool first parses the source code of the selected elements, and builds an internal, in-memory model of the source code that will be used by all the analyses available in FINT (Figure A.3). The model building (for fan-in analysis) takes about 30 seconds for a system of 20,000 non-comment lines of code (NCLOC) and around 5 minutes for over 360,000 NCLOC, on a Pentium 4 machine (2.66 GHz).

Fan-in analysis looks at all call relations in the system under investigation and displays the results in a dedicated view, as shown in Figure A.4. In this view, each callee method is the tree root of its callers and has attached to its name the number of callers. The fan-in value of each callee is indicated next to its name.

As shown in the detailed Figure A.5 the view presents the user with a number of options, like:

- Sorting the results by their name or by their fan-in value;
- Showing/Hiding the library-methods in the view (that is, methods that are called from the analyzed element but not declared in this element. Such elements include, for instance, the JDK libraries);
- Showing/Hiding the accessor-methods in the view. The tool checks accessor-methods by their name (`get*` and `set*` methods) or by implementation (methods that simply return a reference or set the value of a field).

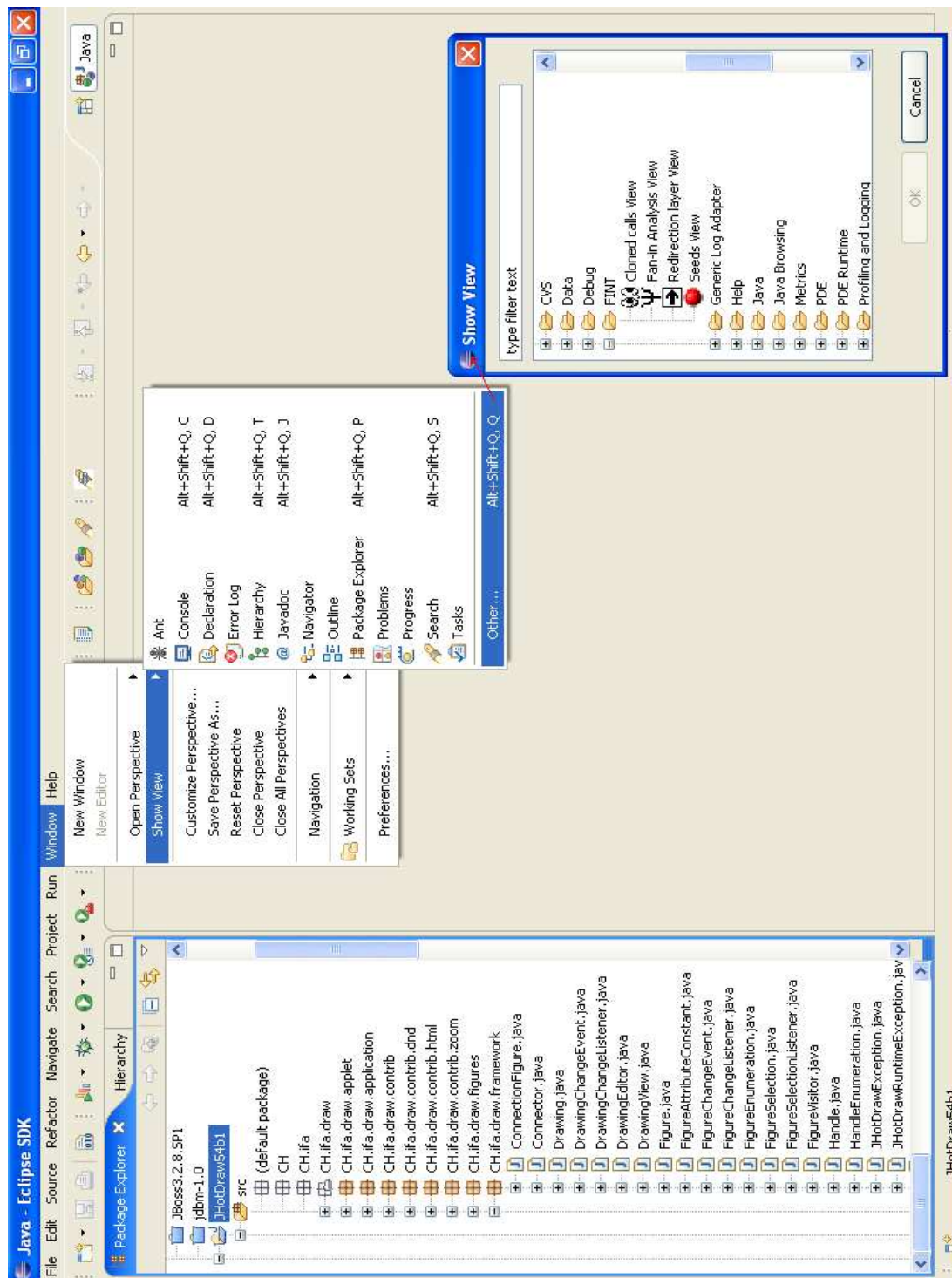


Figure A.1: The FINT views.

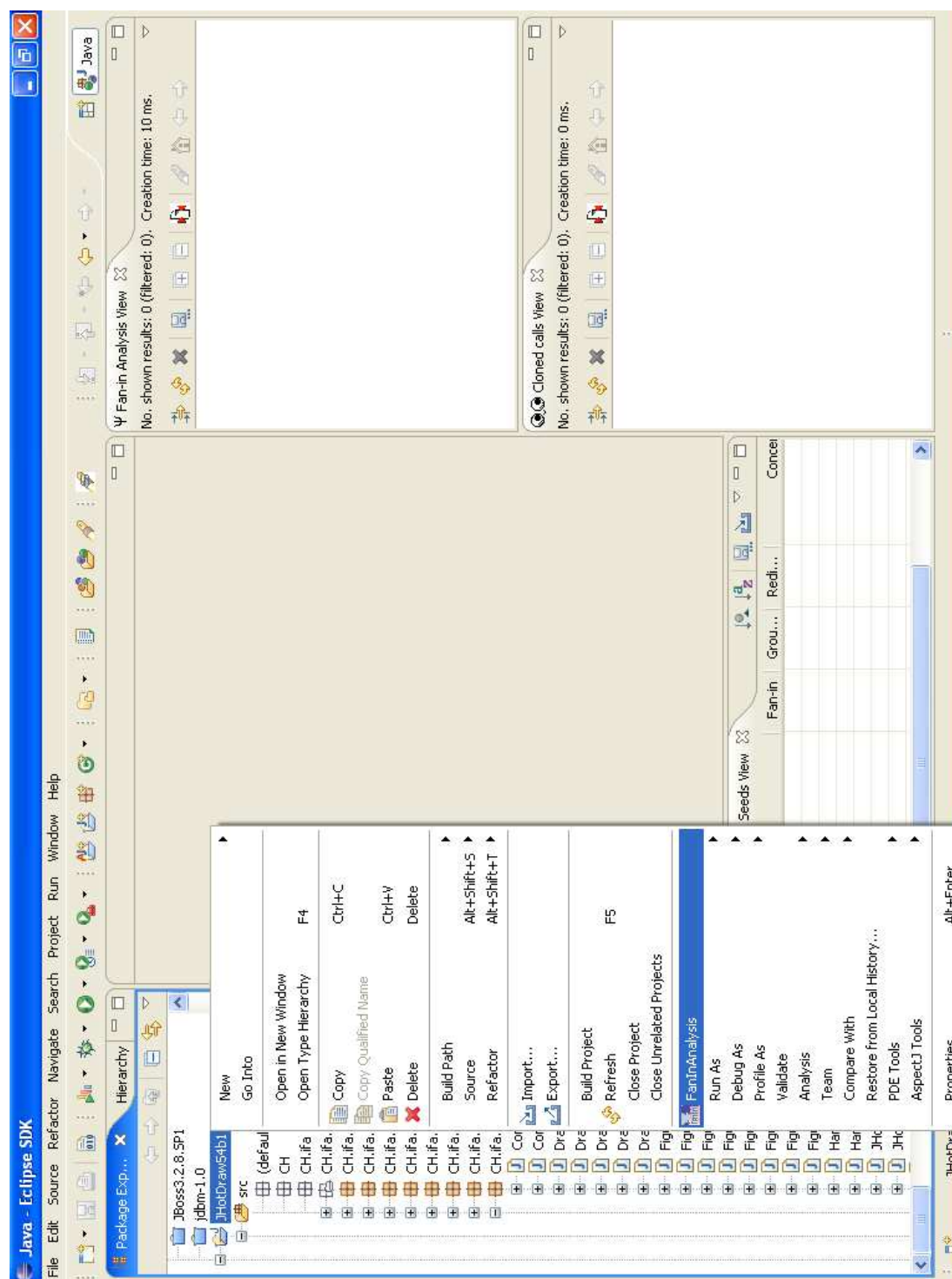


Figure A.2: Run the analysis from the context menu.

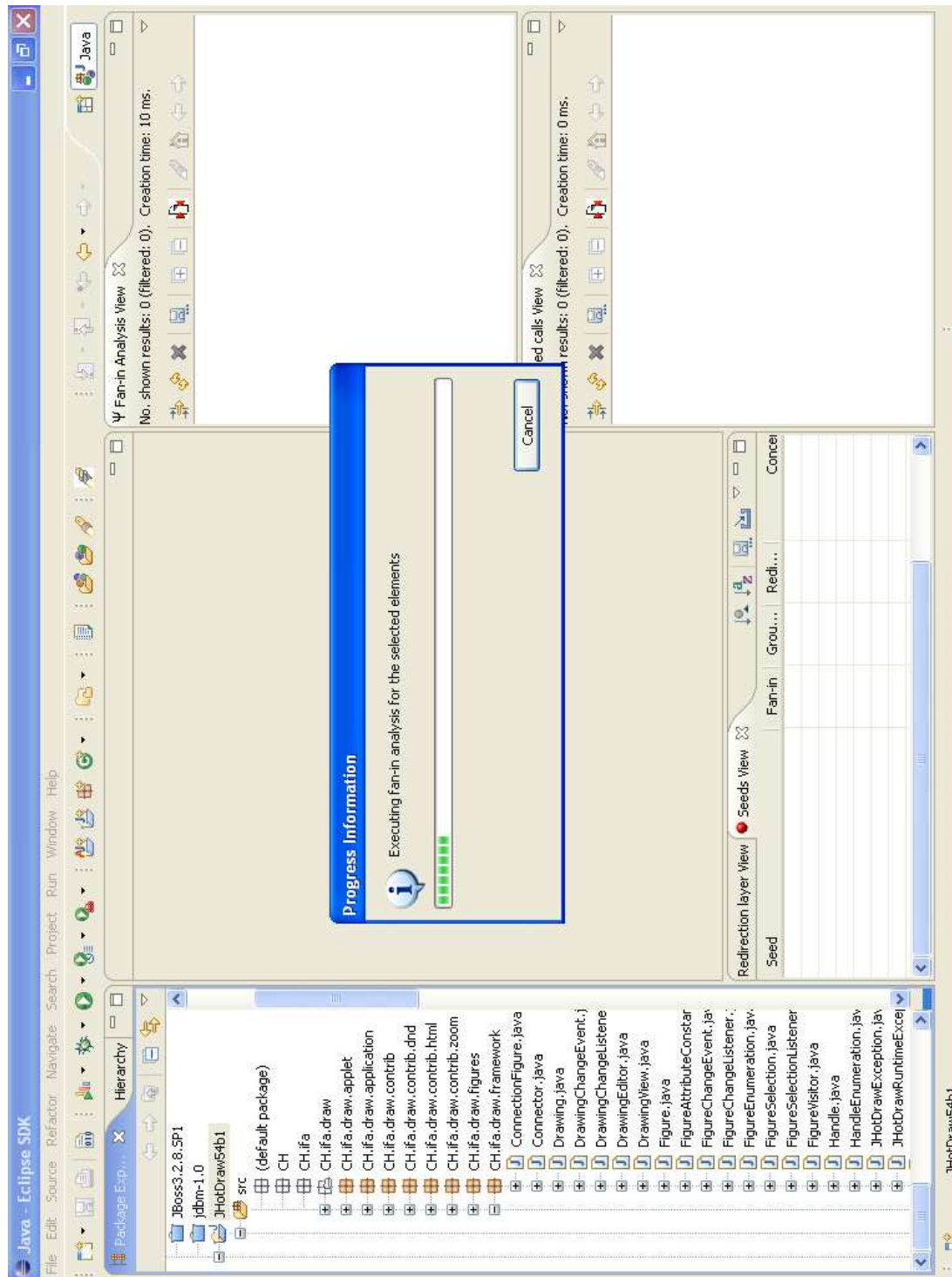


Figure A.3: A progress bar shows the time left to complete the internal model and to execute the fan-in analysis.

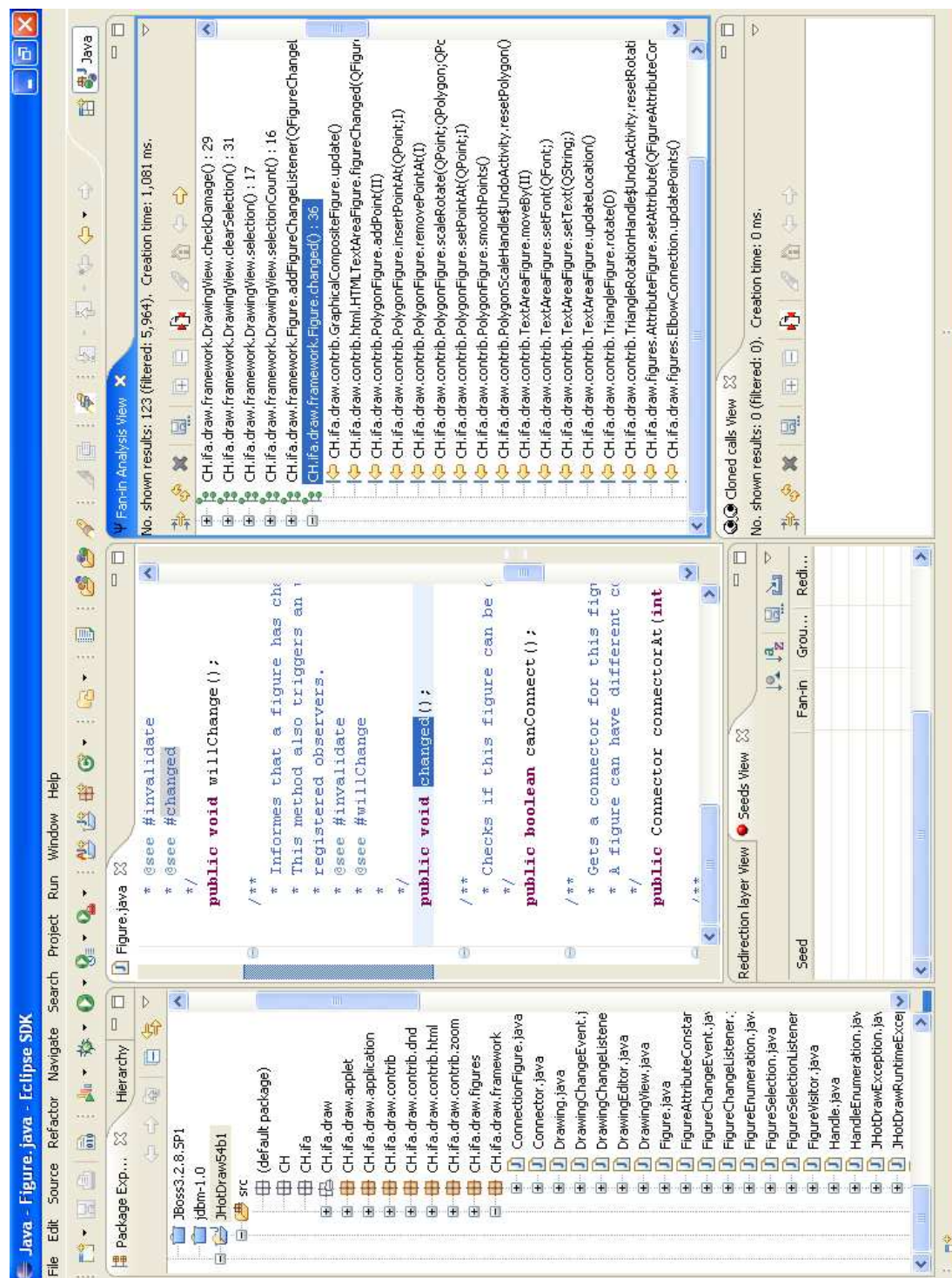


Figure A.4: The results of fan-in analysis displayed in the dedicated view.

Note that the filter for accessors also checks methods in interfaces and eliminates those methods for which all implementations are accessors.

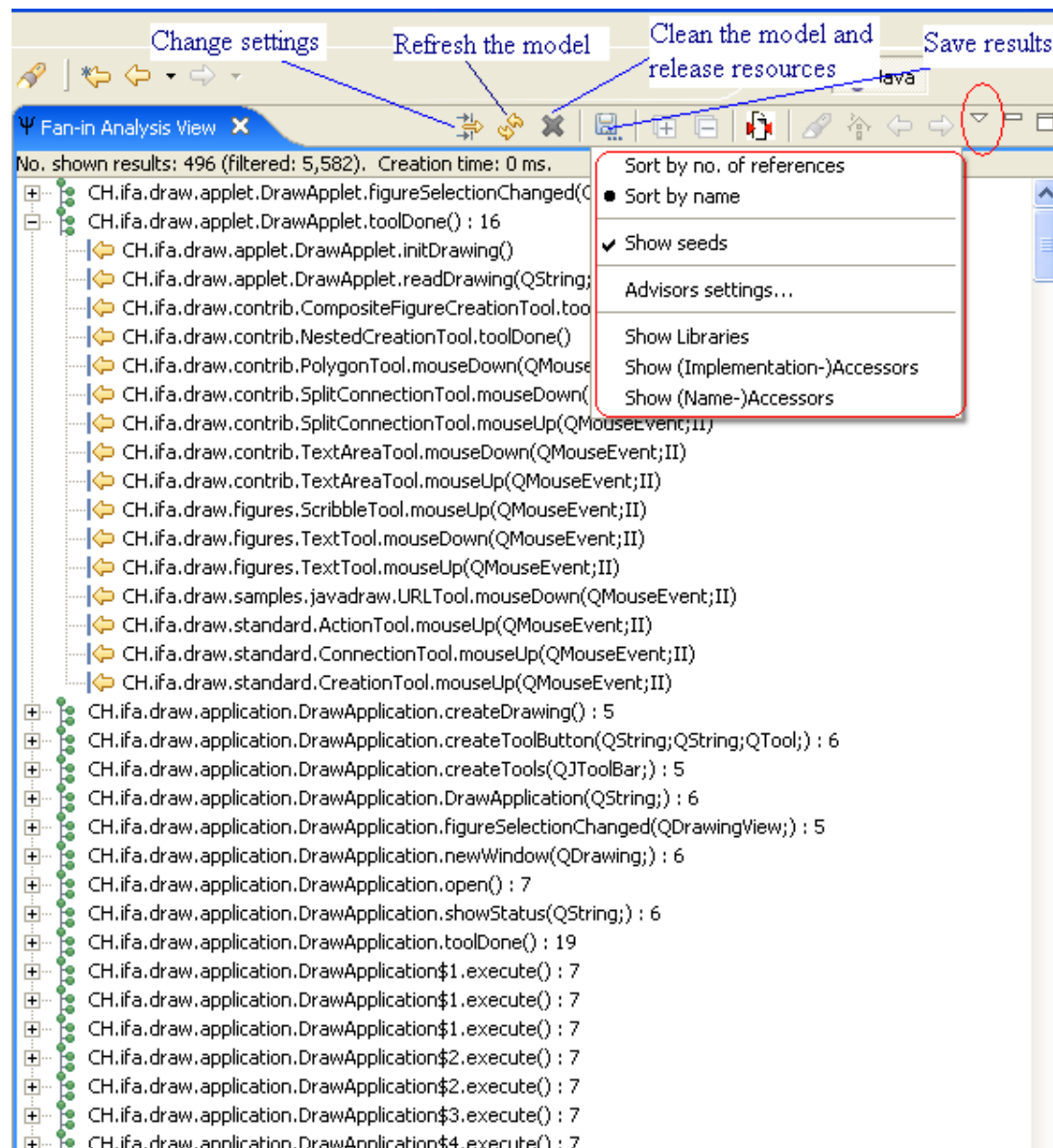


Figure A.5: The Fan-in analysis view and the menus for various options.

By double-clicking any element in the view, the user can inspect the code for that element. Further on, the menus in the view allow us to:

- Change the settings of the analysis, like the fan-in threshold value;
- Refresh the model by re-analyzing (/re-building the model for) the last analyzed element;

- Clean the model built for the analyzed element and release memory;
- Save the results to file.

Besides the filters for the accessor and library methods, also shown in Figure A.6, the set of filters include:

- Callees filters
 - Fan-in threshold: Methods with a fan-in value below the chosen threshold are not shown in the view;
 - "Utility" methods: methods that the user chooses to ignore and that will not be shown in the view;
- Callers filters
 - Methods that should not contribute to the fan-in value of their callees.

To select "utility" elements, the user is presented with the Java element hierarchy of the analyzed Java element. The user can check the "utility" elements in the dialog window of the Fan-in Analysis Setup. Such utility elements could include, for instance, (JUnit) test packages, as in the example shown in Figure A.6.

Reasoning about a candidate

The filtered callee-methods are our *candidates* for crosscutting concern seeds. To reason about a candidate, we select it in the view and choose the "Go Into" option from its context menu, as illustrated in Figure A.7. This command opens the list of its callers and activates the toolbar button for launching various analyses for the callers.

FINT assists the user through several analyses to decide whether a (high fan-in) method is a concern seed. These analyses can be accessed from the menu of the Fan-in analysis view, as illustrated in Figure A.8.

The option for analyzing the hierarchies of the callers will check the top-level declaring type (i.e., interfaces/classes) of each caller, and highlight with the same color those methods that are declared by the same type (see Figure A.9).

The option for analyzing the position of the calls to the (analyzed) method with a high fan-in value opens a window that shows all the callers of the method and the position of the calls to this method. The positions are relative to the caller's body. This analysis is illustrated in Figure A.10.

A similar analysis is shown in Figure A.11: in this case, we look at all the call relations for the callers of our method to see whether these callers have other callees in common besides the method we analyze.

If we decide that a method with a high fan-in value is part of a crosscutting concern implementation (i.e., it is a concern *seed*), we can mark it as such by selecting the Mark

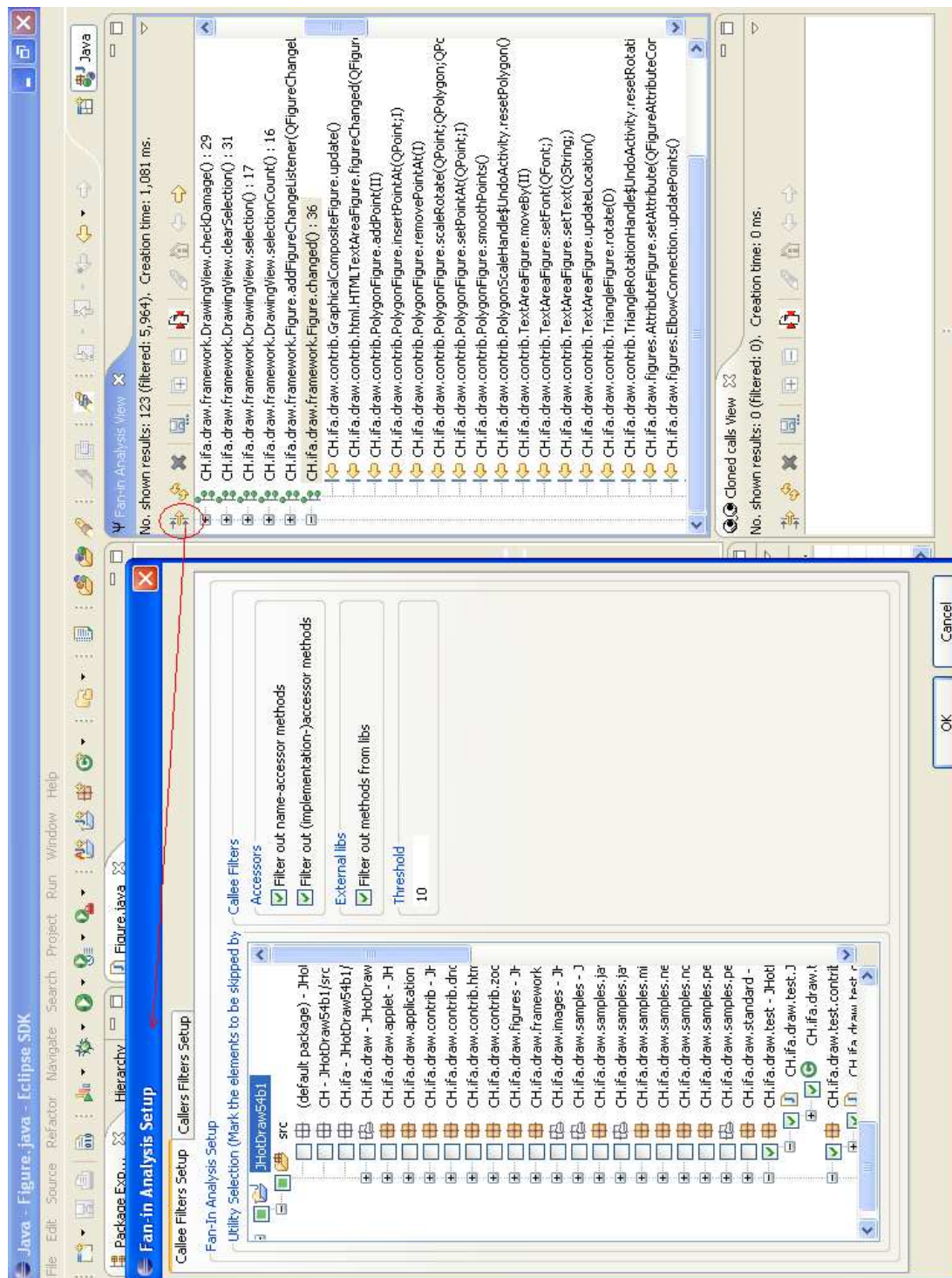


Figure A.6: The dialog to set the filters for the fan-in analysis results.

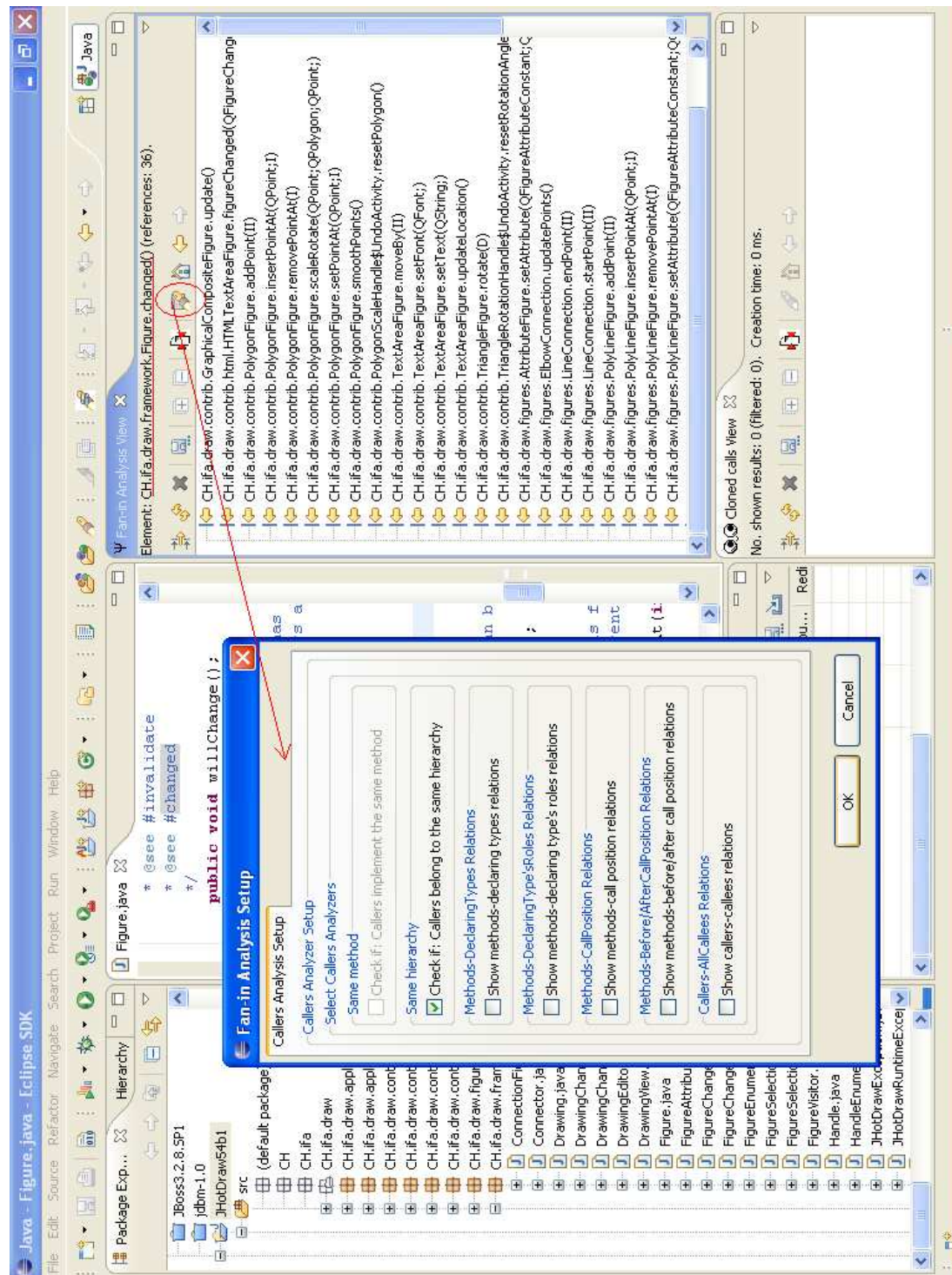


Figure A.8: The dialog to select analyses for the callers of a method (with a high fan-in value).

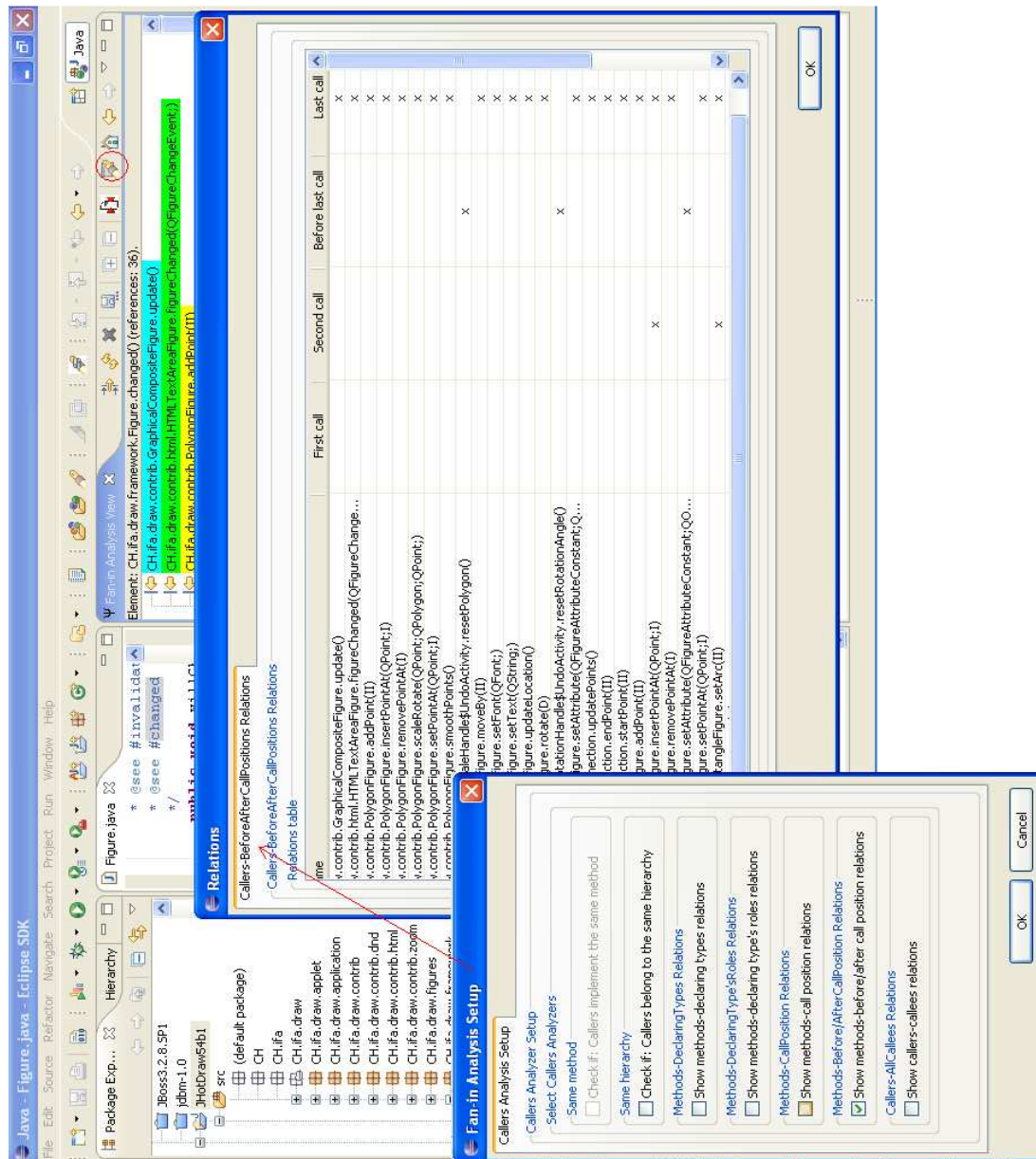


Figure A.10: Analyzing the position of the calls in the caller-method's body.

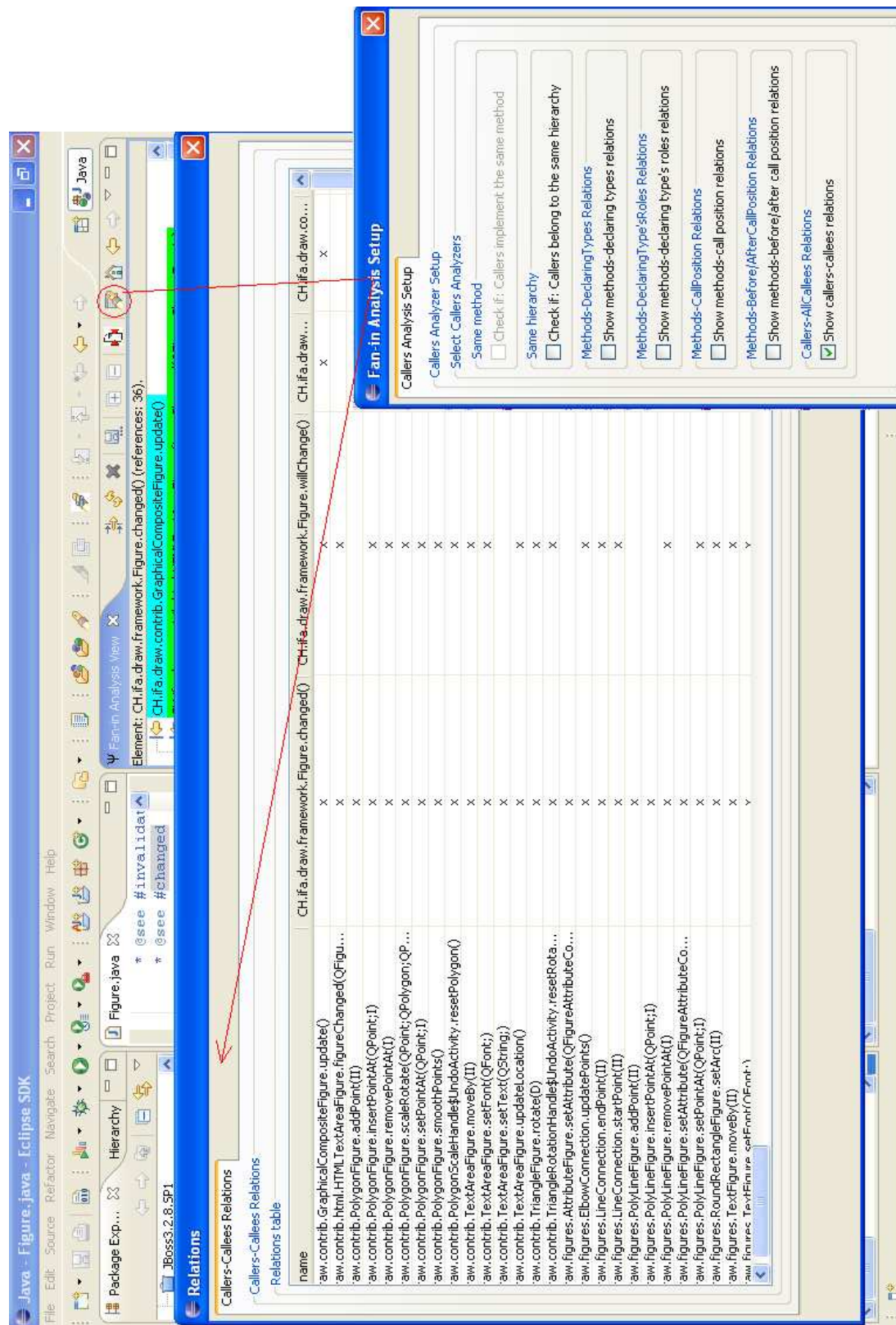


Figure A.11: Callers-callees analysis: Display all the callees for all the callers of the method with the a high fan-in value.

Seed option from the context menu of the candidate (right-click the candidate in the view). The method will be marked distinctively and displayed in the Seeds view, as in Figure A.12.

A.2.2 Grouped calls analysis

Grouped calls analysis requires the model previously built by fan-in analysis. This new analysis can be run from the Grouped calls view, as shown in Figure A.13.

The candidate-seeds consist of groups of methods that share their callers. The candidates are displayed in the view as a tree hierarchy, with each group of methods at the root of the list of their common callers, as illustrated in Figure A.14.

The results of Grouped calls analysis can be sorted and filtered similarly to Fan-in analysis. These filters include checking for setters/getters, checking for libraries methods, as well as for “utilities”.

Besides the threshold for the minimum number of callers of a candidate, we can also set the minimum number of grouped methods that share their callers. All these filters are shown in Figure A.15.

The filters for the callers are again similar to Fan-in analysis and allow the user to ignore certain calls in the analysis, such as, for example, those from unit tests.

Marking a seed for this analysis proceeds again as described for the previous technique. Each of the methods grouped by this analysis is shown in the Seeds view, together with the other methods in the same group, as illustrated in Figure A.16.

A.2.3 Redirections finder

Redirections finder requires too the model built by Fan-in analysis. To run the search for redirections in the code, the user needs to select the Refresh button in the Redirections finder view, which is marked with a circle in Figure A.17.

The same figure shows the results of the analysis: the redirector class and the receiver of the redirection are shown as the root of the set of methods from each class related by an exclusive one-to-one relationship. Such a relationship means that a redirector method calls only one method in the class receiving the redirection, and that the receiver method is not called by any other method in the redirector class.

The filters that can be applied to this analysis are shown in Figure A.18. In this dialog, we can select the minimum number of redirector methods in a class, according to the previously defined rule, as well as the minimum percentage of redirector-methods. In our example, a candidate-redirector class has to have at least 3 methods implementing a redirection and then these methods count for at least 50% of all the methods in that class.

The utility filter is based on the same considerations as the techniques described earlier.

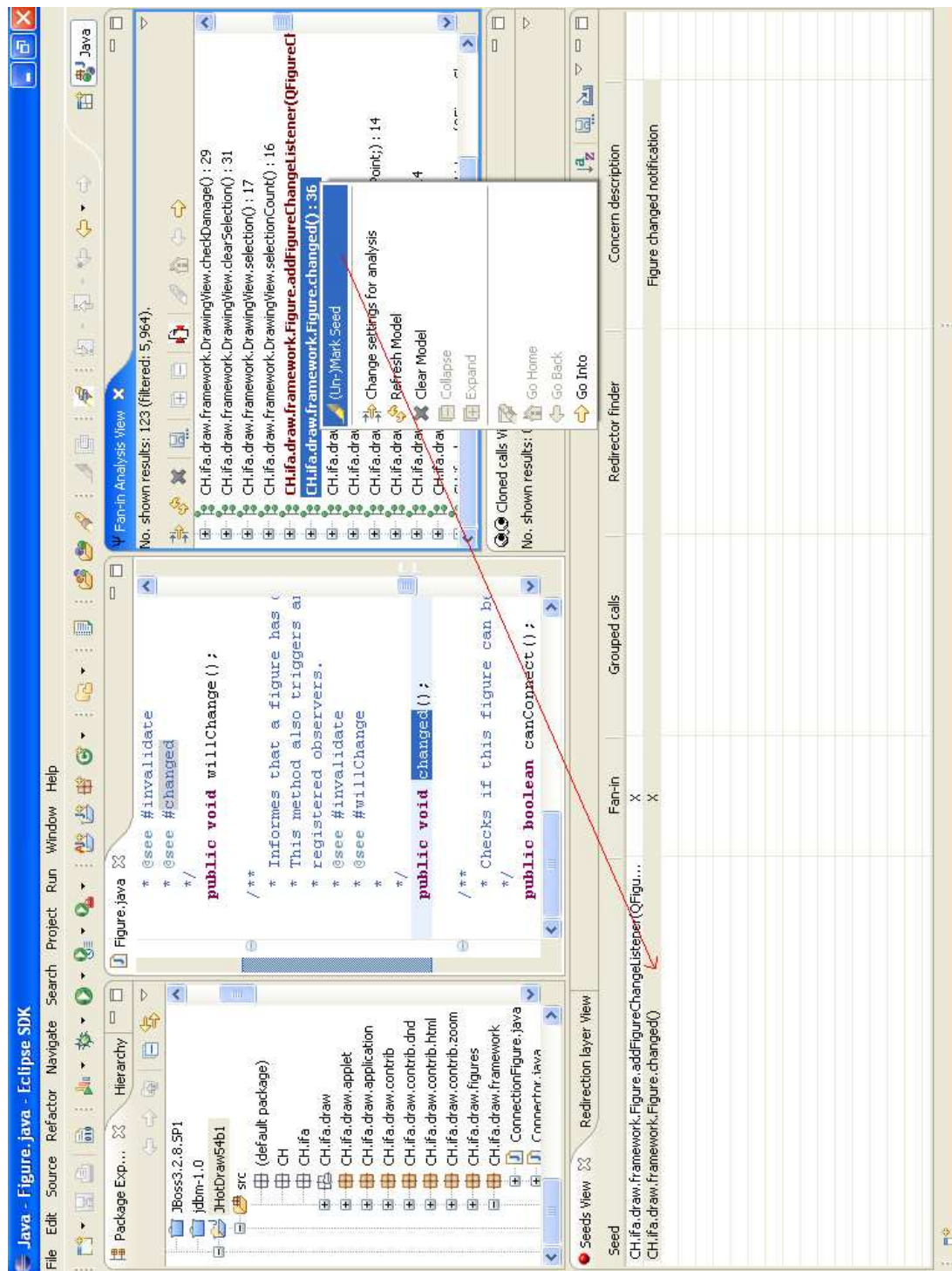


Figure A.12: Marking selected method as a crosscutting concern seed.

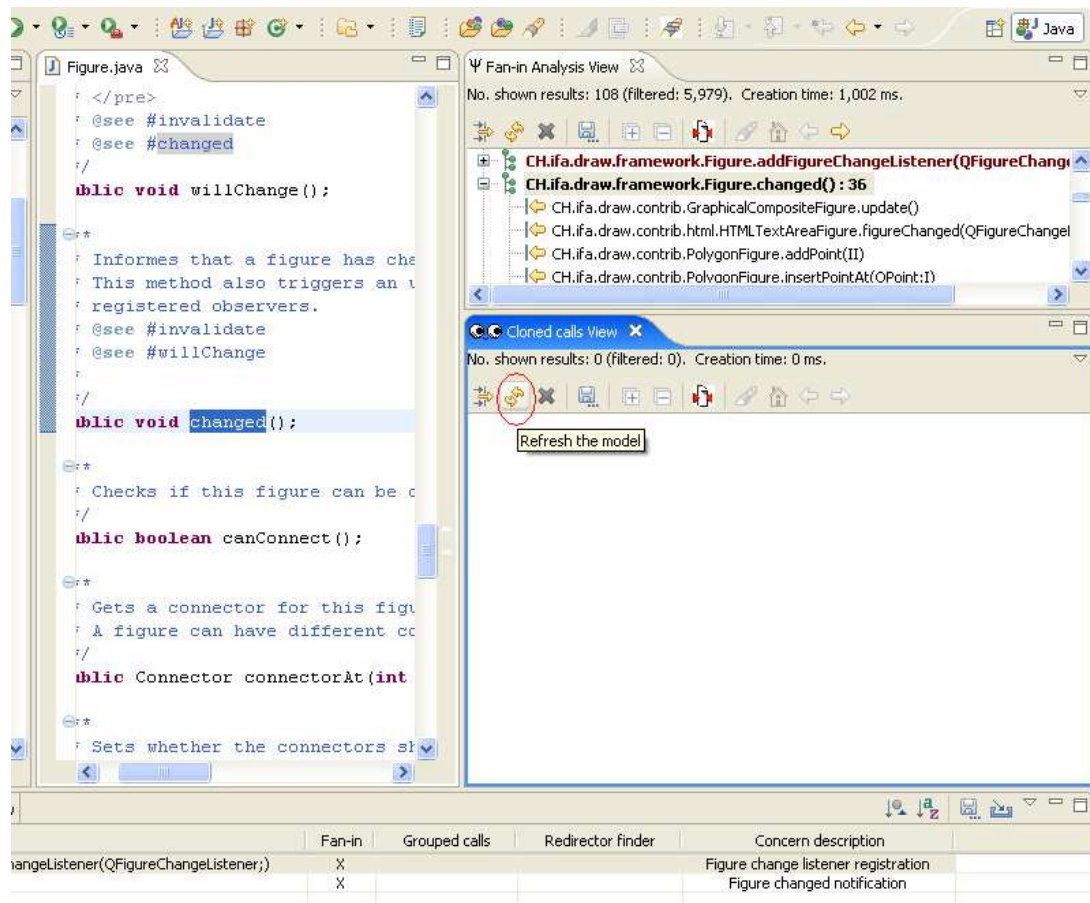


Figure A.13: Running grouped calls analysis.

The screenshot displays the Eclipse IDE interface for a Java project named 'Java - PolygonFigure.java'. The main editor shows the source code of 'PolygonFigure.java'. A red line highlights the 'insertPointAt' method. The right-hand side of the IDE shows the 'Fan-in Analysis View' with a table of grouped calls and a 'Redirection layer View' at the bottom.

Source Code (PolygonFigure.java):

```

/**
 * Insert a node at the given
 */
public void insertPointAt(Point
willChange();
int n = pointCount() + 1;
int[] xs = new int[n];
int[] ys = new int[n];
for (int j = 0; j < i; ++
xs[j] = getInternalPo
ys[j] = getInternalPo
)
xs[i] = p.x;
ys[i] = p.y;
for (int j = i; j < pointC
xs[j + 1] = getIntern
ys[j + 1] = getIntern
)
setInternalPolygon(new Po
changed();

public void removePointAt(int
willChange();
int n = pointCount() - 1;
int[] xs = new int[n];
int[] ys = new int[n];

```

Fan-in Analysis View:

Seeds View	Redirection layer View	Fan-in	Grouped calls	Redirector finder	Concern description
Seed					
CH.f.a.draw.framework.Figure.addFigureChangeListener(QFigureChangeListener);		X			Figure change listener registration
CH.f.a.draw.framework.Figure.changed();		X			Figure changed notification

Redirection layer View:

Seeds View	Redirection layer View	Fan-in	Grouped calls	Redirector finder	Concern description
Seed					
CH.f.a.draw.framework.Figure.addFigureChangeListener(QFigureChangeListener);		X			Figure change listener registration
CH.f.a.draw.framework.Figure.changed();		X			Figure changed notification

Figure A.14: Grouped calls analysis results.

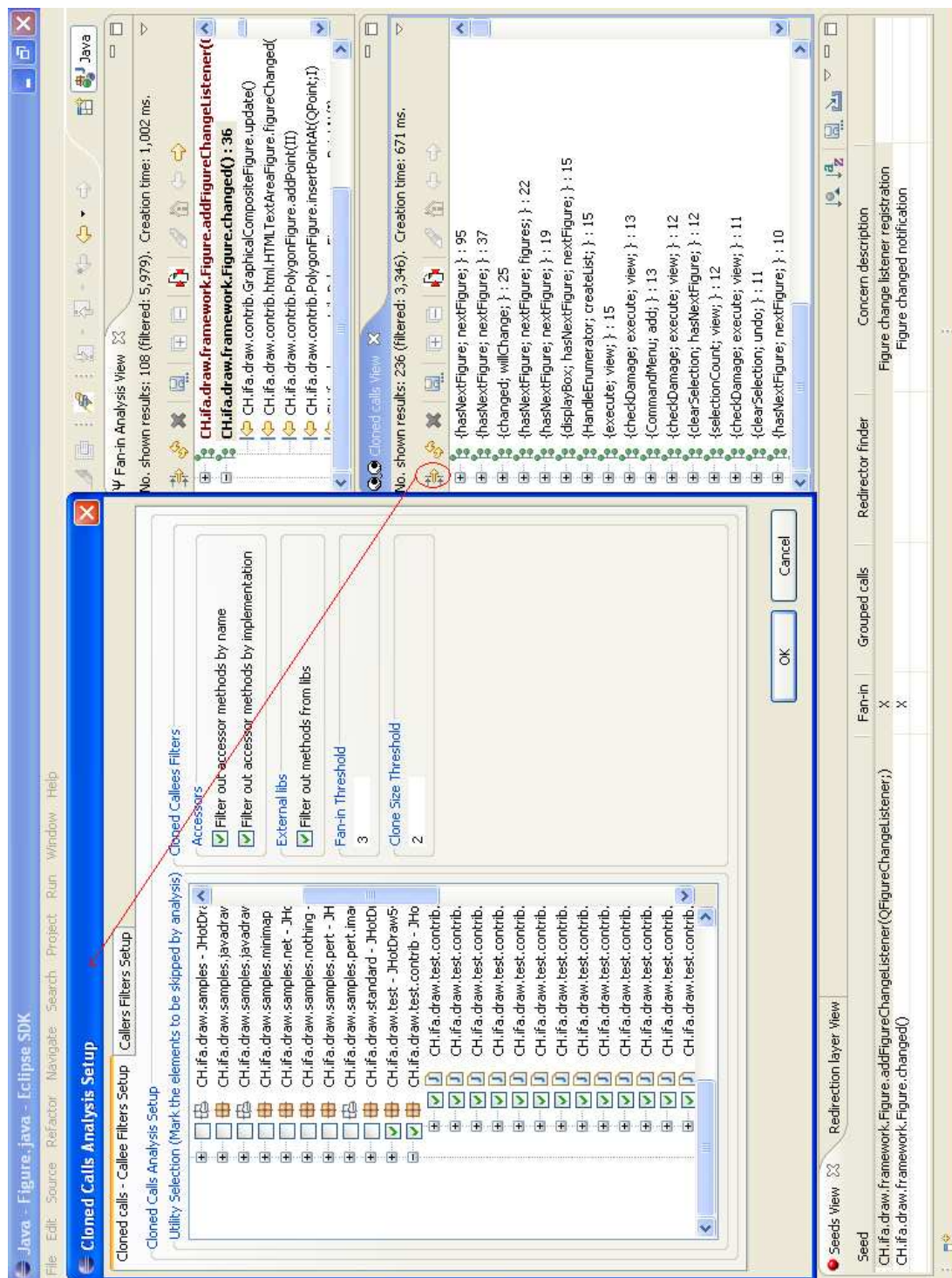


Figure A.15: The filters for Grouped calls analysis.

The screenshot shows the Eclipse IDE with the 'Seeds for Grouped calls analysis' tool. The tool is divided into three main sections:

- Seeds:** A list of Java classes used as seeds for the analysis. The classes include:
 - CH.f.a.draw.framework.Figure.addFigureChangeListener(QFigureChangeListener);
 - CH.f.a.draw.framework.Figure.changed();
 - CH.f.a.draw.standard.AbstractFigure.changed() + {willChange};
 - CH.f.a.draw.standard.AbstractFigure.willChange() + {changed};
- Fan-in Analysis View:** A call graph showing the relationships between the seed classes. It displays a list of methods and their callers, with a filter of 3,560 results. The methods include:
 - CH.f.a.draw.contrib.GraphicalCompositeFigure.update();
 - CH.f.a.draw.contrib.HTMLTextAreaFigure.figureChanged(QFigureChangeListener);
 - CH.f.a.draw.contrib.PolygonFigure.insertPointAt(QPoint);
 - CH.f.a.draw.contrib.PolygonFigure.removePointAt(I);
 - CH.f.a.draw.contrib.PolygonFigure.scaleRotate(QPoint;QPolygon;QPoint);
 - CH.f.a.draw.contrib.PolygonFigure.setPointAt(QPoint;I);
 - CH.f.a.draw.contrib.PolygonFigure.smoothPoints();
 - CH.f.a.draw.contrib.PolygonFigure.updateUndoActivity;resetPolygon();
 - CH.f.a.draw.contrib.TextAreaFigure.moveBy(II);
 - CH.f.a.draw.contrib.TextAreaFigure.setFont(QFont);
 - CH.f.a.draw.contrib.TextAreaFigure.updateLocation();
 - CH.f.a.draw.contrib.TriangleFigure.rotate(D);
 - CH.f.a.draw.contrib.TriangleRotationHandle\$UndoActivity.resetRotationAr...
 - CH.f.a.draw.figures.LineConnection.endPoint(II);
 - CH.f.a.draw.figures.LineConnection.startPoint(II);
 - CH.f.a.draw.figures.PolyLineFigure.removePointAt(I);
 - CH.f.a.draw.figures.PolyLineFigure.setPointAt(QPoint;I);
 - CH.f.a.draw.figures.RoundRectangleFigure.setArc(II);
 - CH.f.a.draw.figures.TextFigure.moveBy(II);
 - CH.f.a.draw.figures.TextFigure.setFont(QFont);
 - CH.f.a.draw.figures.TextFigure.setText(QString);
- Redirection layer View:** A table showing the redirections between the seed classes. The table has columns: 'Seeds', 'Fan-in', 'Grouped calls', 'Redirector finder', and 'Concern description'. The rows show the redirections for the seed classes.

Figure A.16: Seeds for Grouped calls analysis.

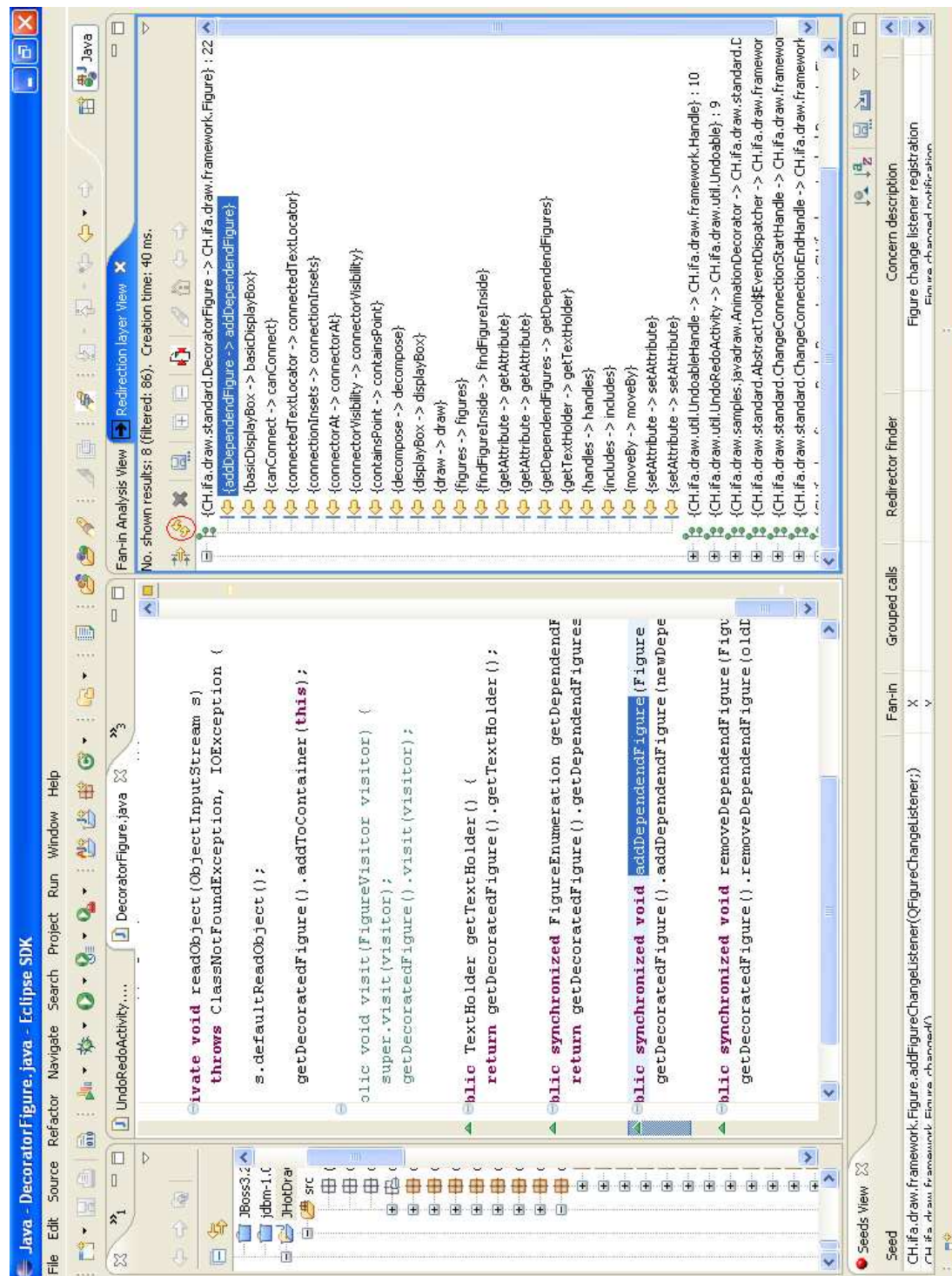


Figure A.17: The view for the Redirections finder analysis.

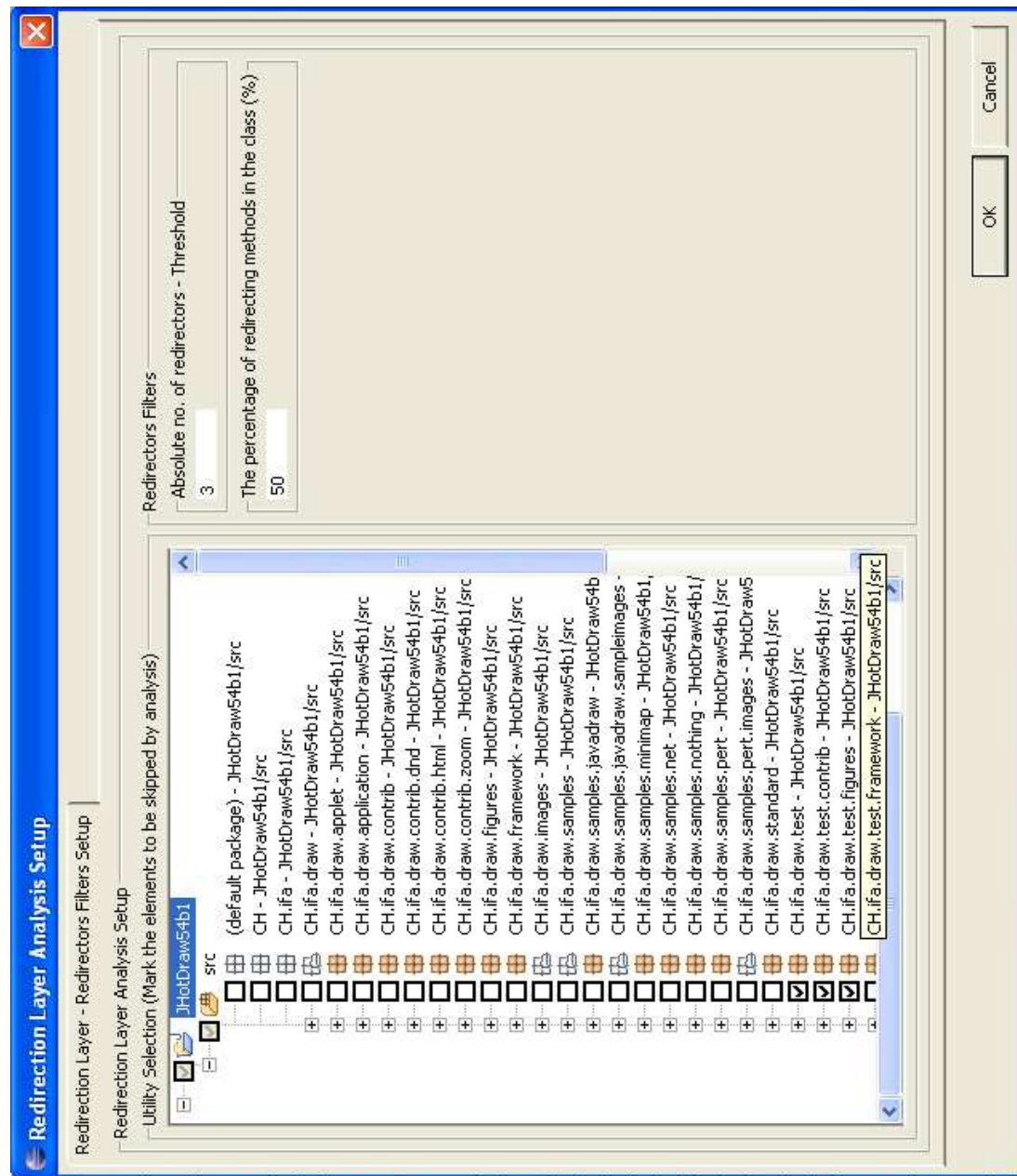


Figure A.18: Redirections finder filters.

A.2.4 Combination of techniques

FINT also allows the user to combine techniques, namely those techniques that target concerns following similar implementation idioms. Two such techniques are Fan-in and Grouped calls analysis, which both analyze method-call relations for crosscutting concerns. The combination consists of searching for the results of one technique among the results of the other one. As Grouped calls analysis makes a stricter selection of the methods with a large number of callers, we can select a lower fan-in threshold for this technique and then look for the grouped methods among the results of Fan-in analysis. For each of these methods, we might find a larger number of callers in Fan-in analysis and hence a better coverage of the concern of that method.

The combination can be launched as illustrated in Figure A.19.

The intersection of results of the two techniques is highlighted in the view of Fan-in analysis, as shown in Figure A.20.

A.2.5 Seeds management

All the seeds marked by the user are collected and displayed in the Seeds view next to the technique that identified them, as shown in Figure A.21. The view also allows the user to add a short description of the concern implemented by each of the seeds, next to each seed. The list of seeds can be saved to or re-loaded from file.

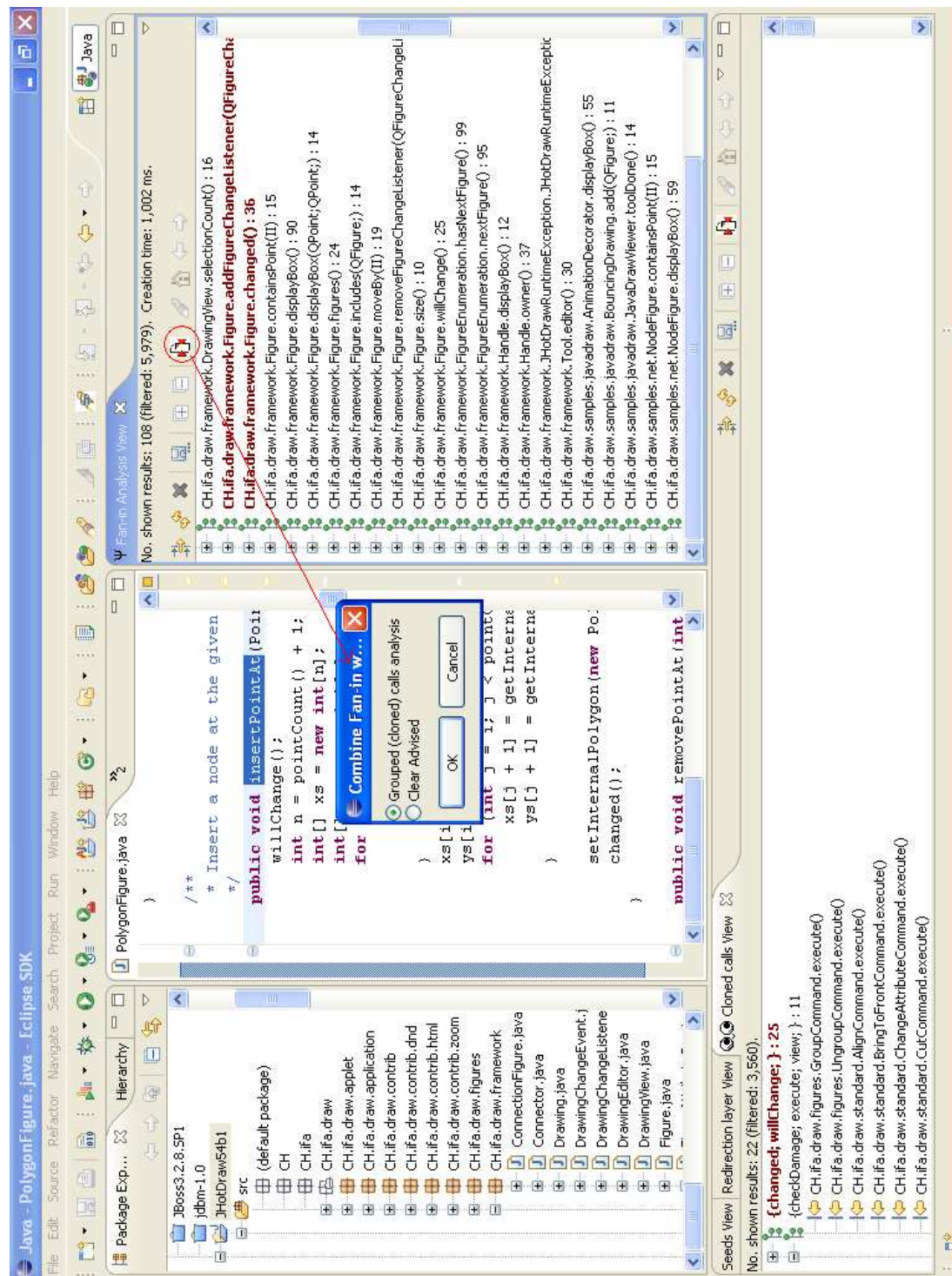


Figure A.19: Combining Fan-in and Grouped calls analysis.

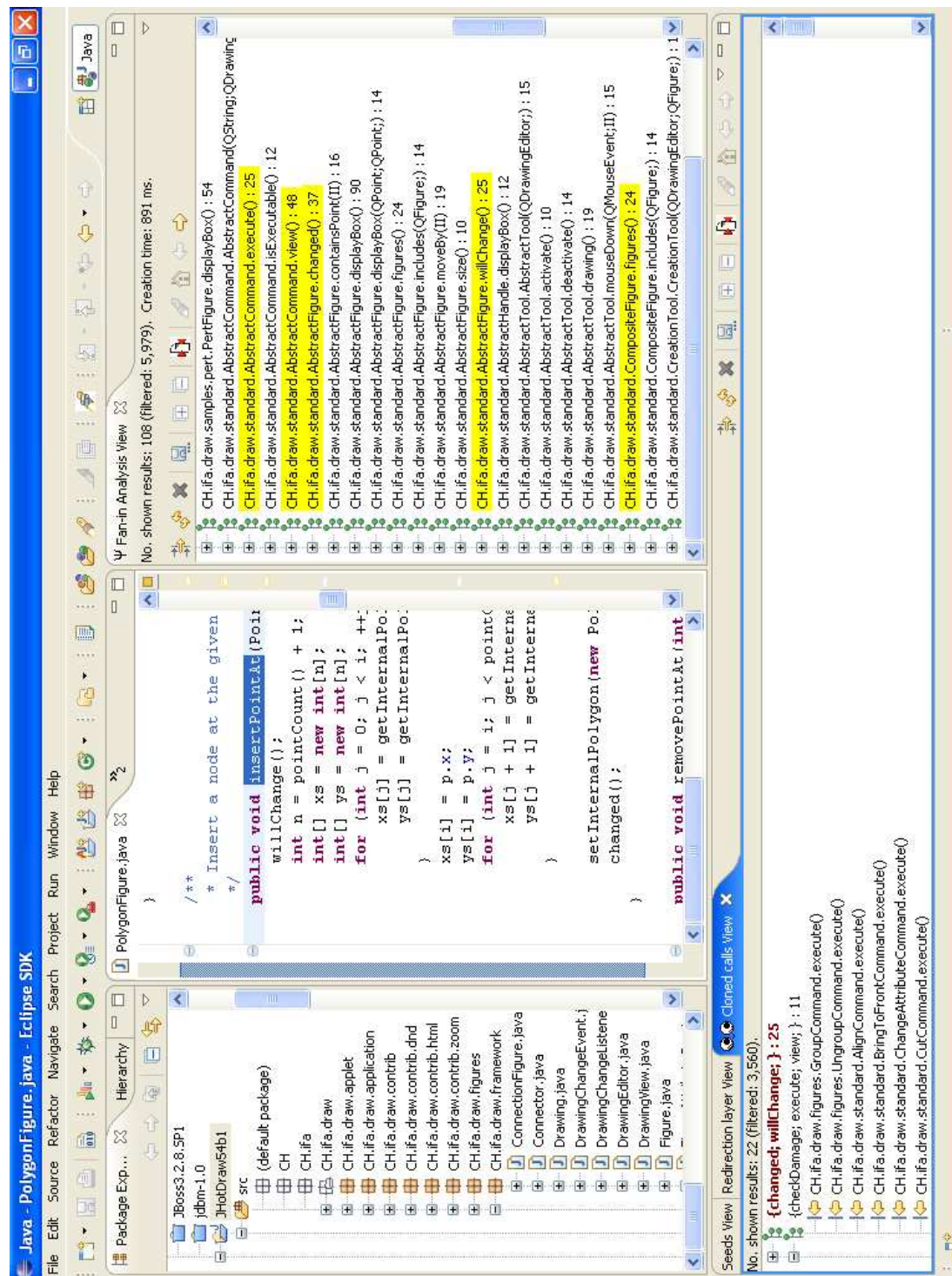


Figure A.20: Combination results.

The screenshot shows the 'Seeds View' window in IntelliJ IDEA. The window has a title bar with 'Seeds View' and a close button. The main area is a table with the following columns: 'Seed', 'Fan-in', 'Grouped calls', 'Redirector finder', and 'Concern description'. The table contains four rows of data. The first row has a seed 'CH.ifa.draw.framework.Figure.addFigureChangeListener(QFigureChangeListener;)' and a fan-in of 'X'. The second row has a seed 'CH.ifa.draw.framework.Figure.changed()' and a fan-in of 'X'. The third row has a seed 'CH.ifa.draw.standard.DecoratorFigure' and a fan-in of 'X'. The fourth row has a seed 'CH.ifa.draw.util.UndoableHandle' and a fan-in of 'X'. The 'Grouped calls' column is empty for all rows. The 'Redirector finder' column has 'X' for the third row. The 'Concern description' column has 'Figure change listener registration' for the first row, 'Figure changed notification' for the second row, 'Figure decorator' for the third row, and 'Handle wrapper for undo' for the fourth row. In the top right corner of the window, there is a toolbar with several icons. One icon, which looks like a document with a checkmark, is circled in red. A red arrow points from this icon to the 'SaveSeedsList' option in the 'Sort' menu. The 'Sort' menu is open, showing options: 'Sort-ConcernDescription', 'Sort-Name', 'SaveSeedsList', and 'ReadSeedsList'. The 'SaveSeedsList' option is highlighted.

Seed	Fan-in	Grouped calls	Redirector finder	Concern description
CH.ifa.draw.framework.Figure.addFigureChangeListener(QFigureChangeListener;)	X			Figure change listener registration
CH.ifa.draw.framework.Figure.changed()	X			Figure changed notification
CH.ifa.draw.standard.DecoratorFigure			X	Figure decorator
CH.ifa.draw.util.UndoableHandle			X	Handle wrapper for undo

Figure A.21: The Seeds view.

Appendix B

SOrrts QUERy Tool (SOQUET)

This appendix contains the SOQUET user manual. The underlying ideas are described in Chapters 4 and (partially) 6 of this thesis.

SOQUET is a query-based (crosscutting) concern modeling and documentation tool distributed as an Eclipse IDE (v.3.2.x – v.3.3) plug-in.¹ The source code of SOQUET consists of 35,453 NCLOC.² In order to improve independence on the Eclipse releases, SOQUET code also includes internal packages of Eclipse, whose implementation and interface is documented to be subject of change between Eclipse releases. This re-used code serves, for instance, to preserve Eclipse’s default look-and-feel for some of the tool’s views.

B.1 Installation

To install the tool, the user needs to download and save the “jar” distribution into the “plugins” directory of Eclipse and then (re-)start the IDE.

B.2 User manual

The documentation and modeling of concerns in SOQUET is based on a categorization of crosscutting concerns in so-called *sorts*. A concern sort describes elementary (atomic) crosscutting concerns that share their typical implementation idiom in an object-oriented language, like Java. For example, logging, authentication and authorization mechanisms, notification of changes in observable objects, etc., are typically implemented in Java by means of scattered calls to dedicated methods, such as `org.apache.log4j.Logger.debug(message)`,

¹Some of the figures may be more difficult to read on paper. We refer the reader to the SOQUET web site for the on-line version of this manual, in which the figures are available in high resolution.

²Metrics plug-in, v.1.3.6 – <http://metrics.sourceforge.net/>. Note that the Metrics tool does not count lines of code in interfaces.

`java.security.AccessController.checkPermission(permission),` or `Subject.notifyObservers()` (see the Observer pattern [Gamma et al., 1994]). We call the sort that describes these concerns and their common idiom, *Consistent Behavior (CB)*, and we call each of the concerns above an *instance* of this sort.

SOQUET provides the user with a set of (6) query templates. Each query template describes the relation specific to the sort of concerns associated to it, such as method call relations or inheritance relations. The sort-queries can be parameterized by the user in a SOQUET dialog window, to define concrete queries. The concrete queries are defined by the user so that their results map onto the source code elements that implement a particular, (atomic) crosscutting concern.

The parameterized sort-queries can be saved by the user in a dedicated SOQUET view to document crosscutting concerns by showing the underlying relations of these concerns as well as the program elements that implement them. Moreover, the queries can be grouped in composite, hierarchical models to show relations and associations between different concerns. Such models are aimed at assisting developers in understanding what concerns exist in a system and how these concerns are implemented.

This user-manual presents two use-case scenarios: in the first scenario, we use SOQUET to document crosscutting concerns in the JHOTDRAW drawing application, particularly in the Observer pattern for figure changes outlined in Figure B.1.

In the second use-case scenario, we should use an existing concern model documenting various concerns in JHOTDRAW to assist us with a software change task.³

B.2.1 Modeling and documenting concerns in SOQUET

Our first use-case scenario assumes the perspective of a developer that is familiar with a particular system, namely JHOTDRAW, and with the crosscutting concerns in this system. These concerns are not visible in the class decomposition of our system due to their crosscutting nature, and hence they are harder to notice and understand. Our goal here is to use SOQUET to document these concerns and make them explicit.

The main elements of the user interface in SOQUET consist of two Eclipse views and a dialog window for the sort-queries templates. One of the views is a customized extension of the default *Search* view in Eclipse. The results of our sort-queries will be displayed in this view.

The second view is the *Concern Model* view, which can be opened as shown in Figure B.2. This view can be used to save the queries documenting concerns and to organize them in hierarchical, composite structures, similar to the ones in Eclipse's *Package Explorer* view, shown on the left-side of Figure B.2.

³This manual uses for exemplification JHotDraw 5.4b1, which can be downloaded from <http://sourceforge.net/projects/jhotdraw/> or from the web-site of SOQUET. To use the concern model built for this application, the project has to be imported in Eclipse and named JHotDraw54b1. The source packages (CH.ifa.draw..) need to be placed in a 'src' folder – see the Package Explorer view in Figures B.2 and B.3.

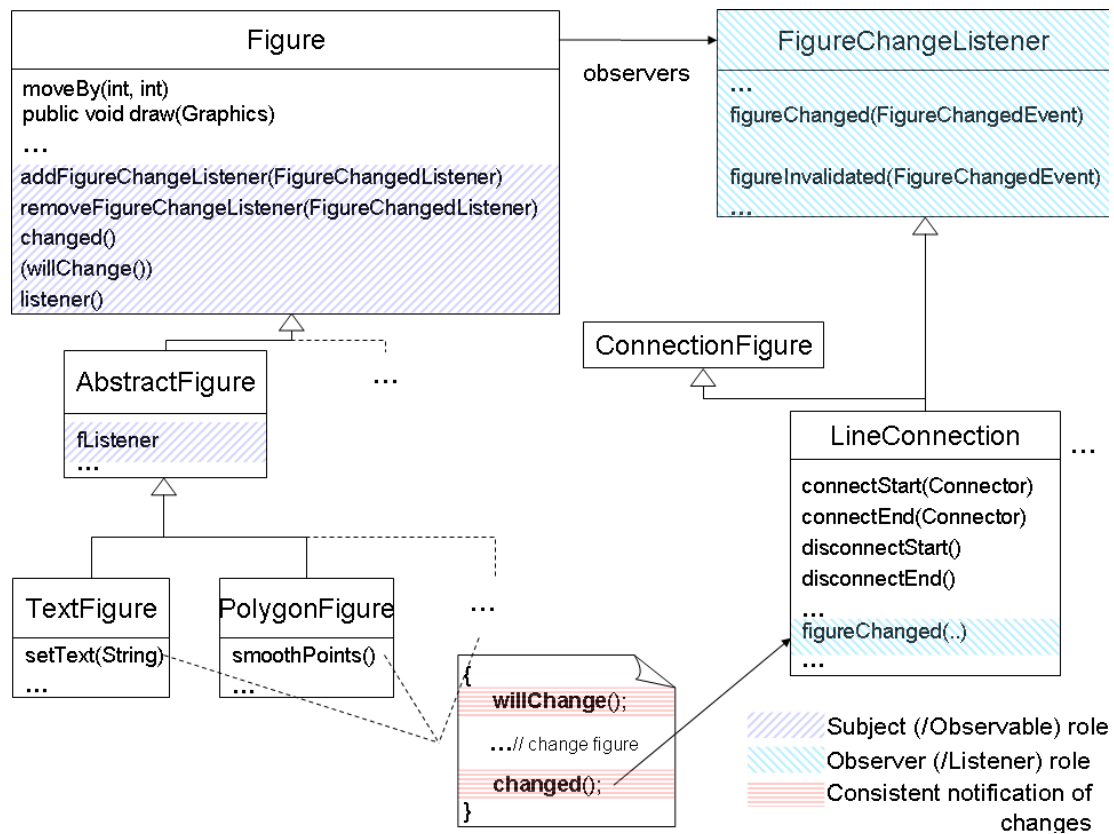


Figure B.1: Observer solution for figure changes in the JHOTDRAW drawing application.

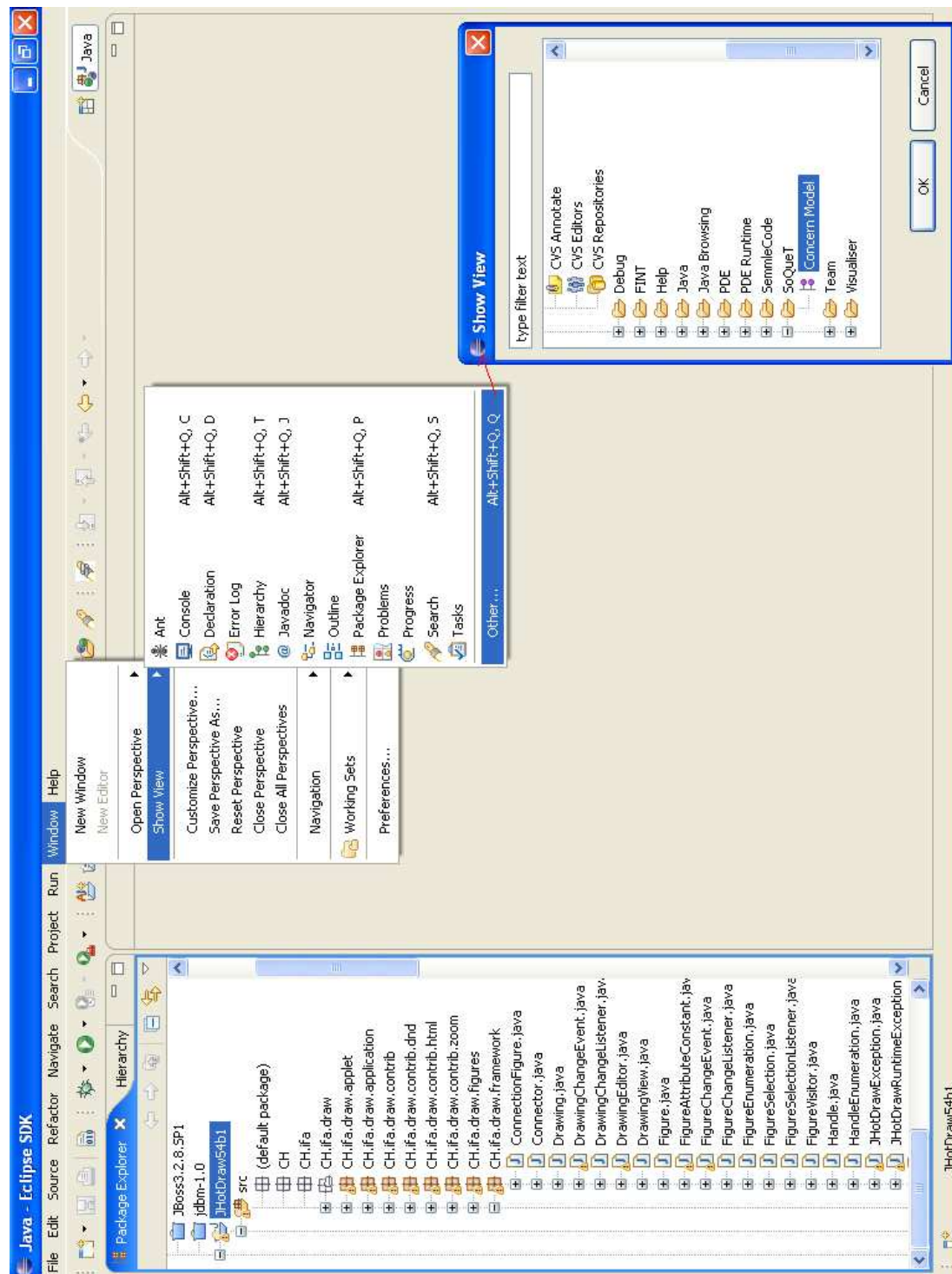


Figure B.2: Opening the Concern Model view.

The sort-search dialog can be accessed from Eclipse's Search menu, as shown in Figure B.3. The dialog window presents the user with six query templates for the six most commonly encountered sorts of concerns. (If the option is not available in the Search menu, check that you are in the Java perspective and select some element in the Package Explorer view.)

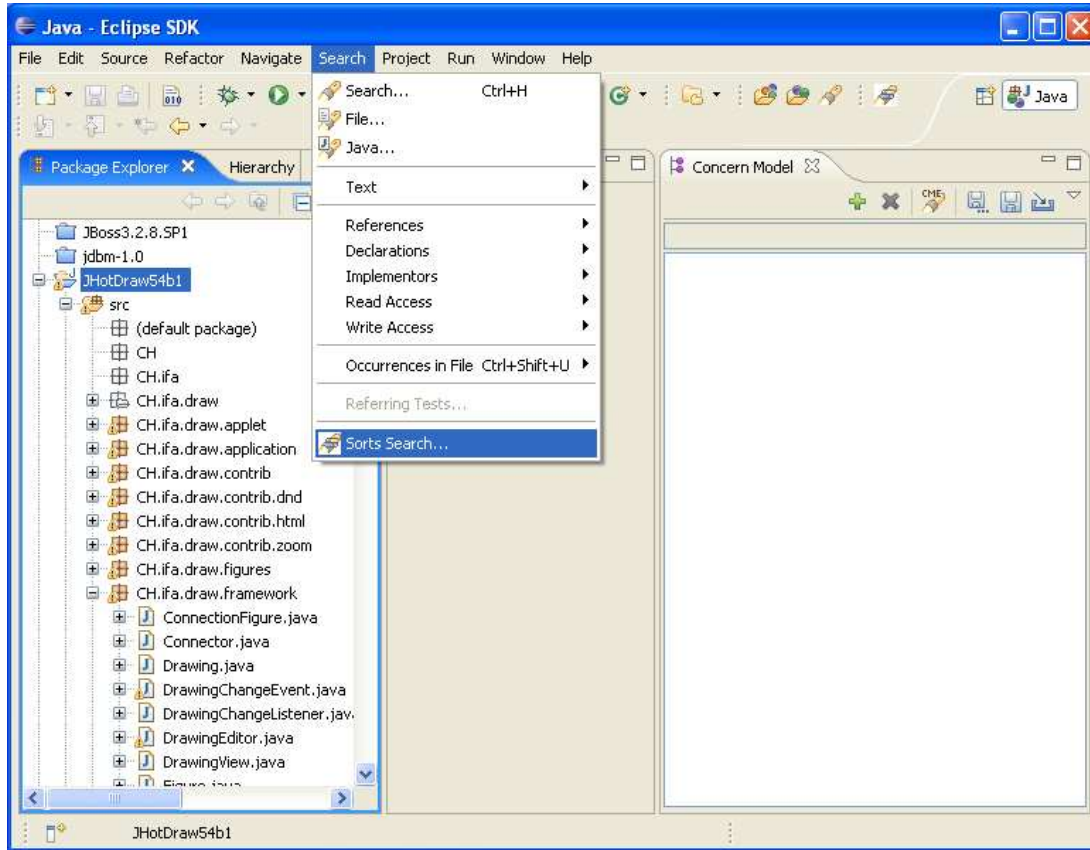


Figure B.3: Opening the sort-search dialog.

Documenting Consistent Behavior (CB)

A first step in documenting our Observer design for figure changes is to recognize those (atomic) crosscutting concerns that occur in this design. In Figure B.1, we notice that each action (method) changing the state of a figure (for instance, moving or resizing a figure), consistently invokes the `willChange` method at the beginning of the execution and the `changed` method after the change is completed. The concerns to notify (pre-) changes in a figure follow the idiom described by the *Consistent Behavior (CB)* sort above, hence we will use the template for the CB query to document these concerns as CB instances.

The template for *Consistent Behavior*, shown in Figure B.4, requires two parameters: the first is the method invoked consistently as part of the crosscutting con-

cern to be documented (the target context). For our notification concern, this is the `Figure.changed()` method.

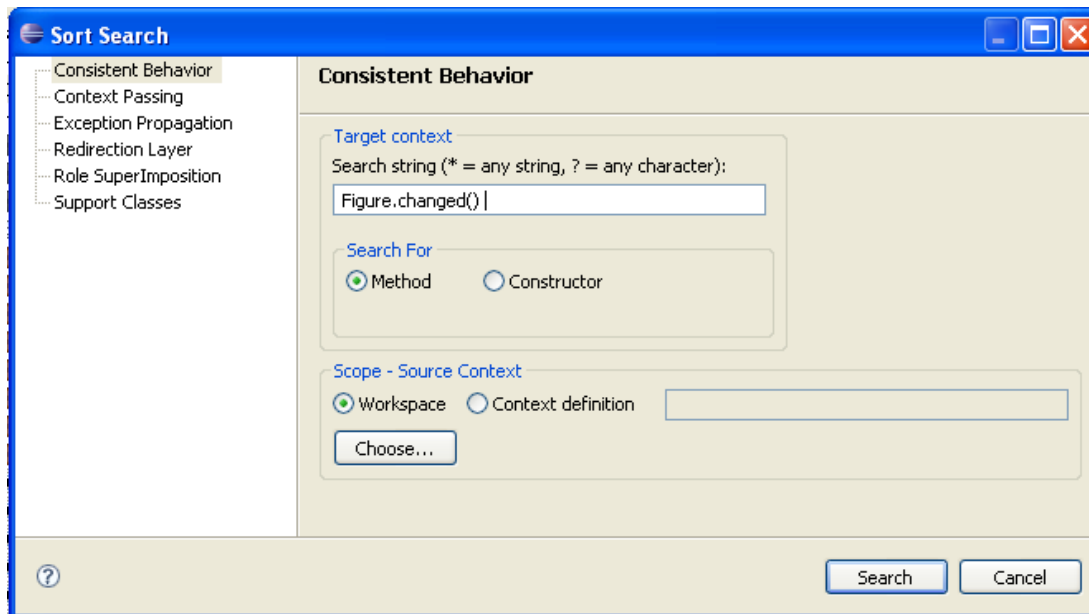


Figure B.4: The dialog to parameterize the Consistent Behavior query.

The second parameter allows us to select from all the callers of the `Figure.changed()` method only those methods that are part of the concern that we want to document. In our case, all the calls to the `changed()` method from elements in the JHotDraw project are crosscutting and part of our notification concern. Therefore, we define our source context by selecting the whole JHotDraw project. Figures B.5 and B.6 illustrate how to do this.

After defining the the two contexts and selecting the Search button in the dialog, the query will run and analyze all the elements of the JHotDraw project to identify calls to the `Figure.changed` method. The results of the search give us the participants in the notification concern. These are displayed in the Search view (see Figure B.7), from where the user can navigate to the source code of these elements, organize them by various layouts, apply different filters, etc.

Saving the query documenting the concern

To save our query capturing the notification concern, we need the *Concern Model* view. A concern model in this view can include (1) atomic concerns, which are associated a concern sort query, as well as (2) composite concerns, which group together multiple sort instances (i.e., atomic concerns) and/or other composite concerns.

We start by creating a composite concern that will group together all the concerns in the JHotDraw project. Figure B.8 shows how to create a new composite concern in the

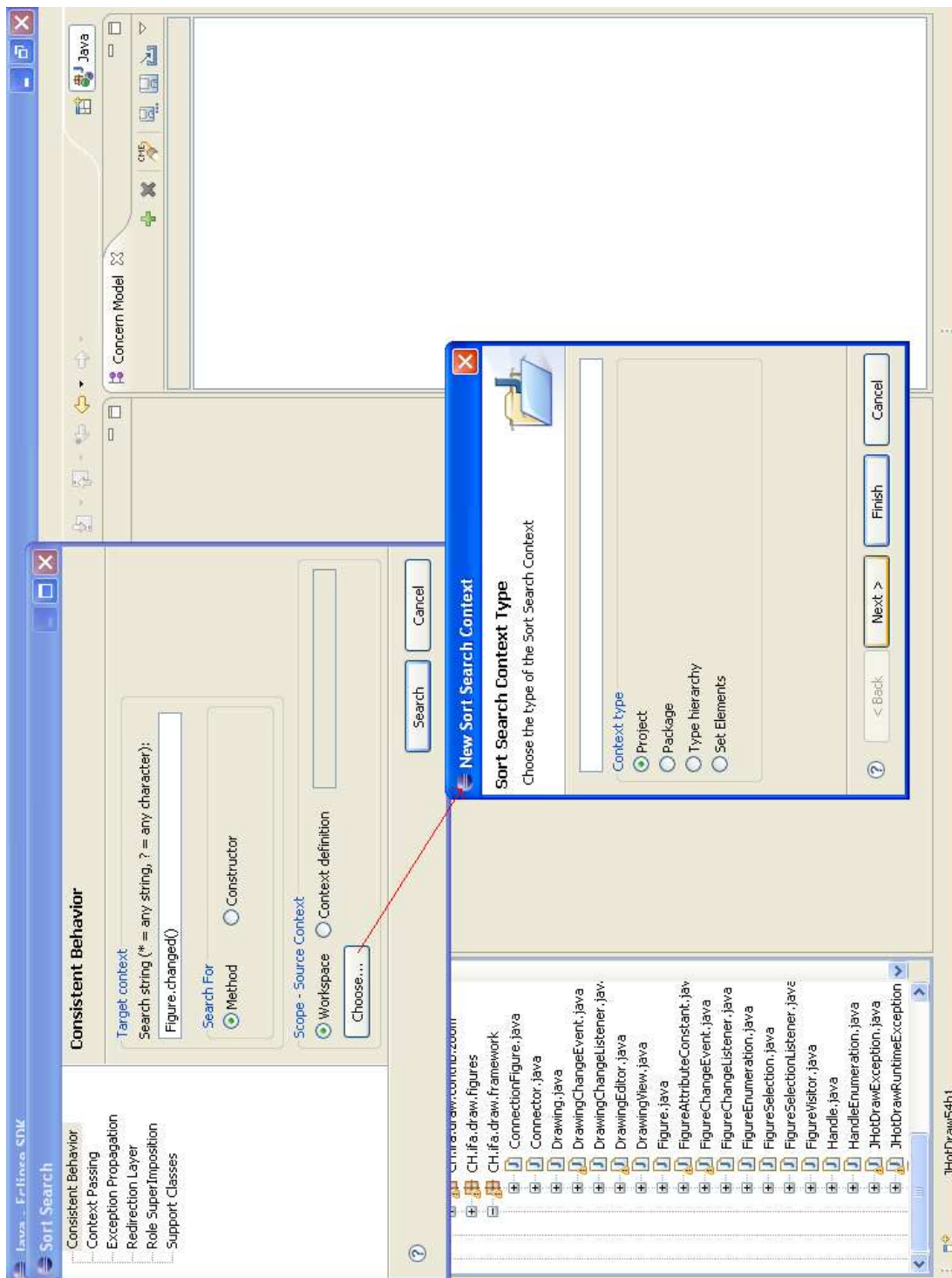


Figure B.5: Defining the source context for the CB query.

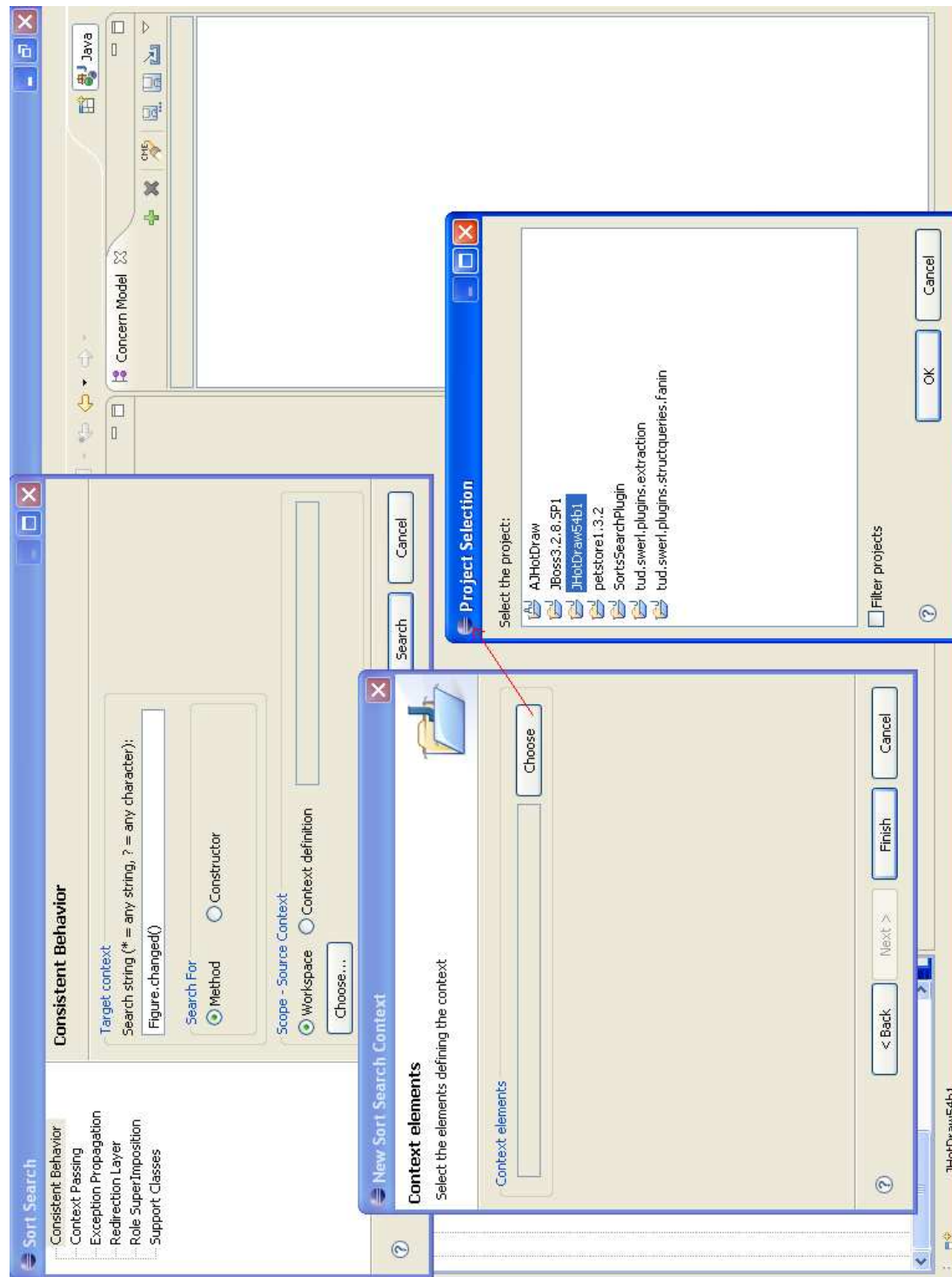


Figure B.6: Select the whole JHotDraw project as the element that comprises all the callers of interest for our notification concern.

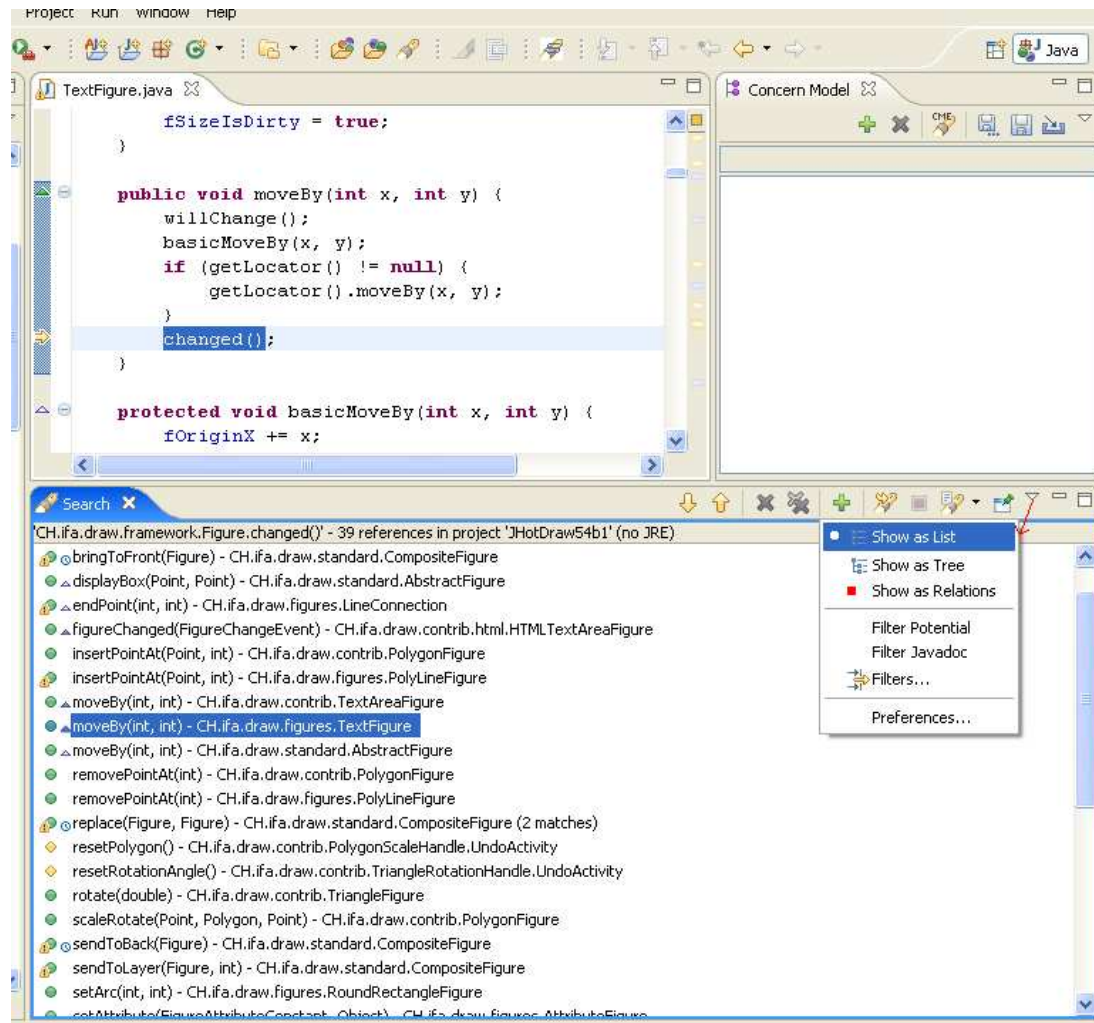


Figure B.7: The results of the CB query that describes the figure-change notification concern as an instance of CB.

Concern Model view: Right click in the view, select *Add New Concern*, and introduce a name for the new concern. In our case, our top composite concern is named by the same name as the JHotDraw project.

We continue with a new composite concern for our Observer pattern, in order to group together all the sort instances in this pattern's implementation. This concern is a child of the composite for the JHotDraw project. The steps to add this new composite concern to the model are shown in Figure B.9.

Now, we can add to our concern model the *Consistent behavior* sort instance documenting the notification concern, and its associated query. Following the steps in Figure B.10, we start in the Search view, which displays the results of our sort query, and select "Add Sort Instance to Concern Model". We then choose the parent concern of our sort instance: *FigureChangeObserver*.

The new concern shows up now in the *Concern Model* view, together with the description of its associated query (see Figure B.11).

In case we added our sort instance to the wrong parent, we can select the Move option in the context menu of the instance (right click), and re-assign the parent. The Expand option in the same menu allows the user to re-run the query associated with the documented sort instance.

Documenting Role SuperImposition (RSI)

A second (atomic) crosscutting concern in our Observer occurs in *Figure* classes, which declare a number of members (on top of their main functionality) to allow listeners to register and receive notifications every time a change occurs in a figure's state. These members define a secondary, crosscutting role implemented by Figures, namely the Subject (or Observable) role. Similarly, the listener elements, such as line connections between figures, have to implement the *FigureChangeListener* role that defines the methods for handling notifications from figures. Both these crosscutting roles follow a similar idiom, namely members declaration (and implementation) to support additional responsibilities. Same idiom can be observed in other well-known concerns as well, such as persistence and (special handling of) serialization in Java (*java.io.Serializable*).

All these concerns are instances of a different sort, namely *Role Superimposition*.

The Listener role is already defined by a distinct interface: *FigureChangeListener*. To document this crosscutting role, we select the *Role SuperImposition* query in the *Sort Search* dialog, and pass as parameter the listener interface defining the role, as illustrated in Figure B.12. The source context allows the user to specify what implementations of this interface are part of the concern to be documented by the query. Once again, we select the whole JHotDraw project, as all the implementations of the Listener interface are of interest.

The results of the query are displayed in the Search view, as shown in Figure B.13. The view shows the *FigureChangeListener* hierarchy and highlights the members of the crosscutting role for each element in the hierarchy selected in the view.

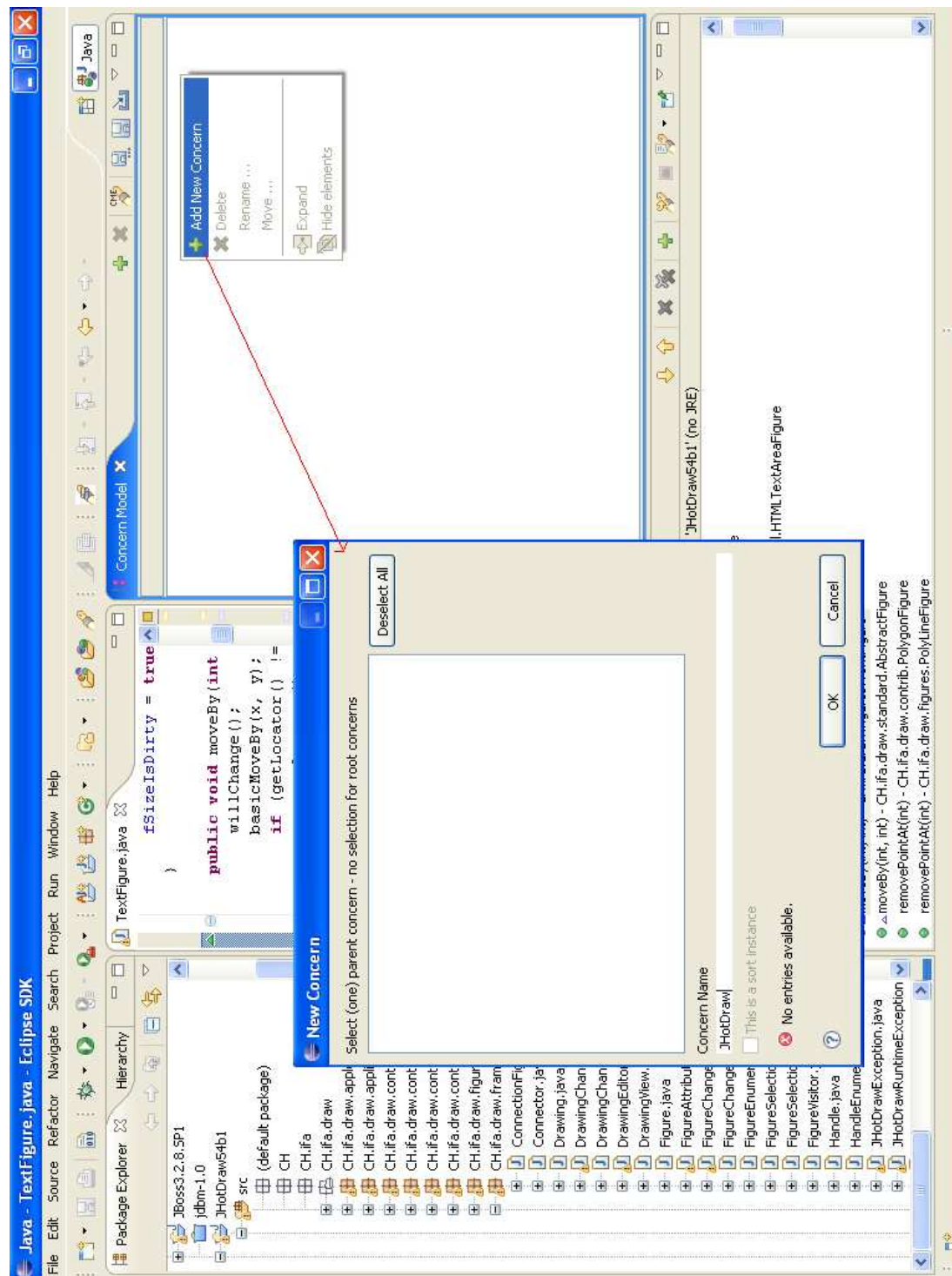


Figure B.8: Create a composite concern for the whole JHotDraw project.

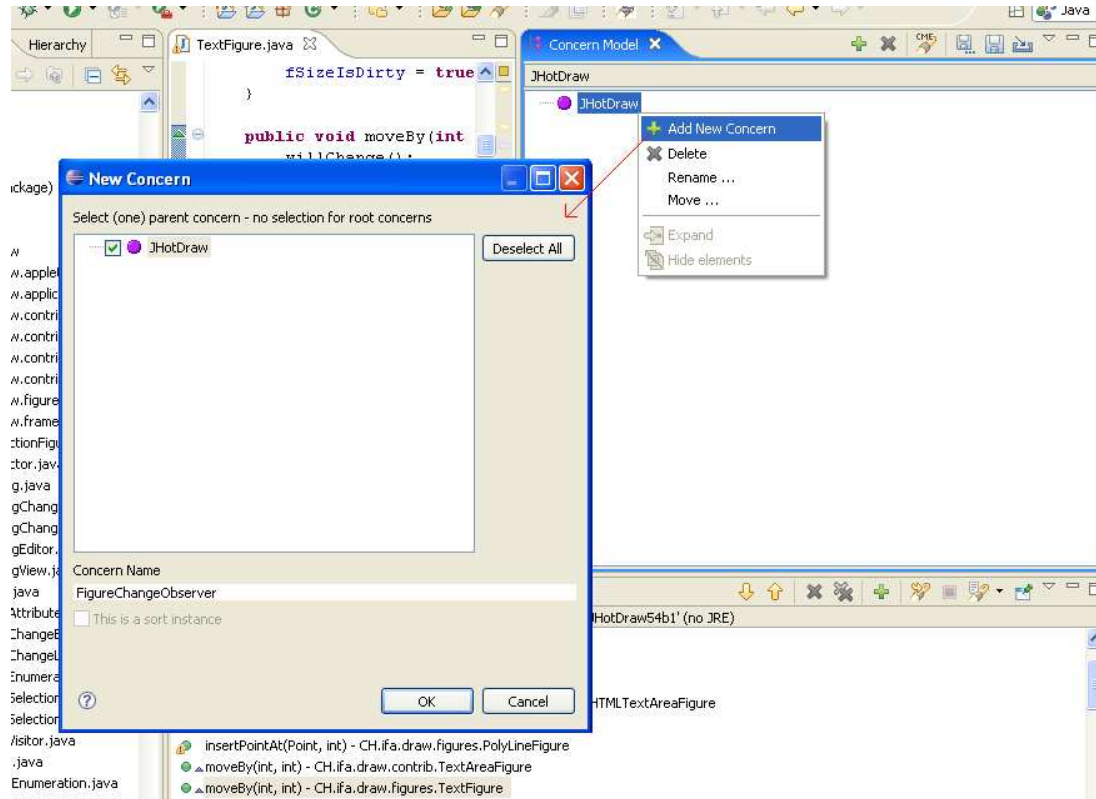


Figure B.9: Create a composite concern for the Observer for figure changes.

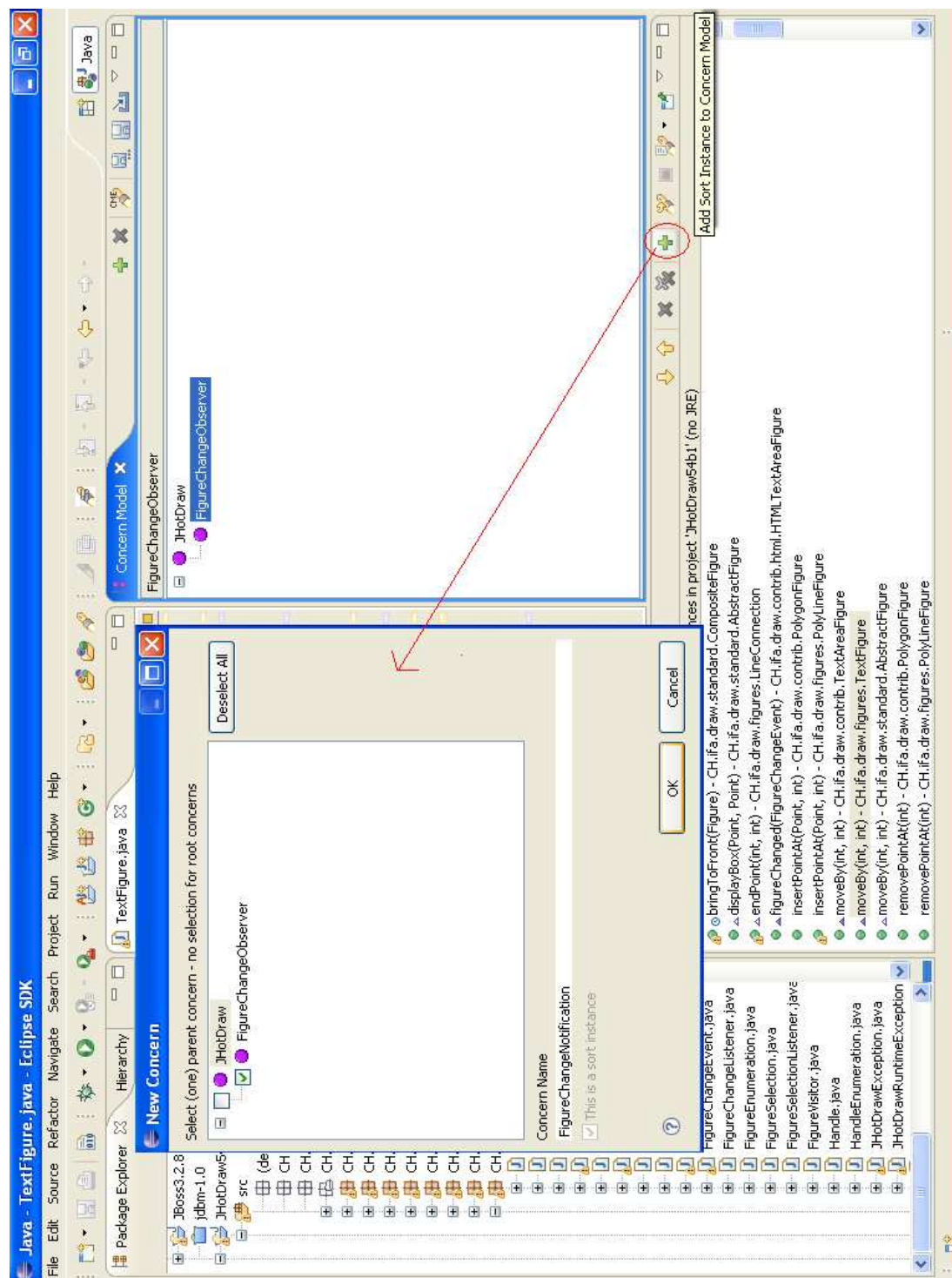


Figure B.10: Add a sort instance to the concern model for the Observer pattern.

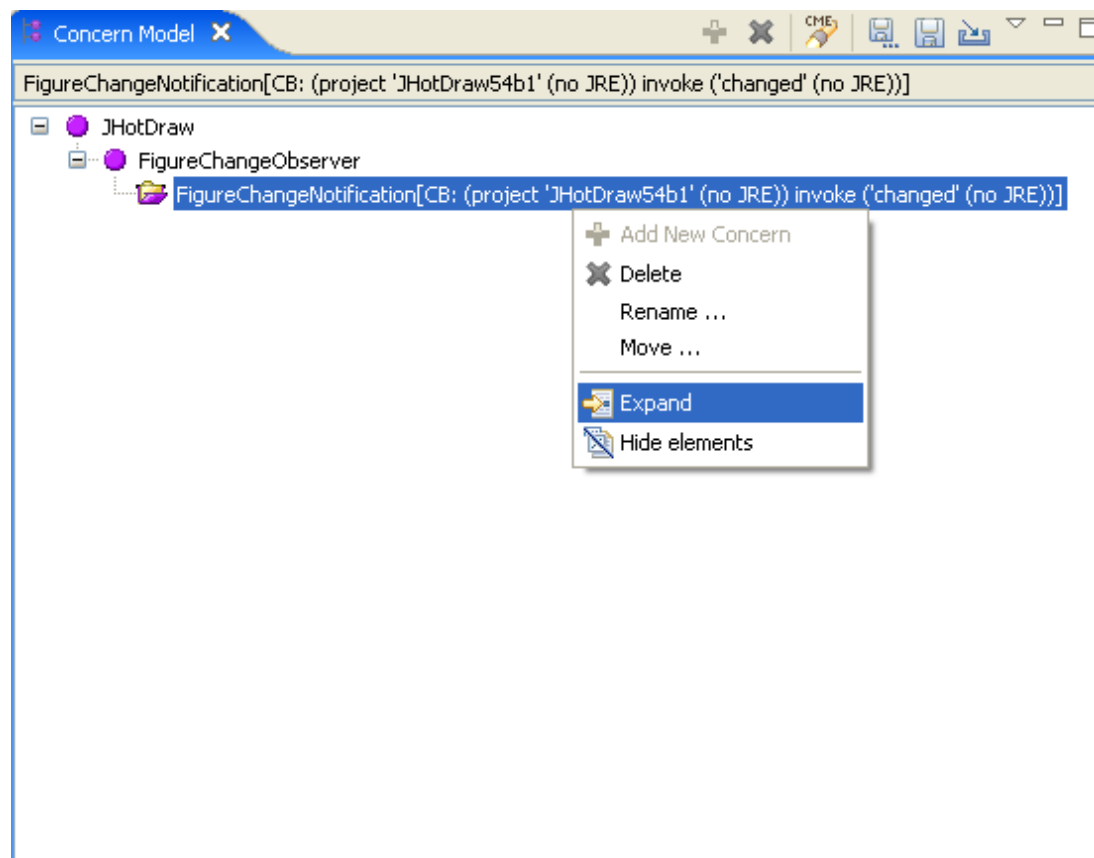


Figure B.11: *Consistent Behavior* instance in Concern Model and its context menu.

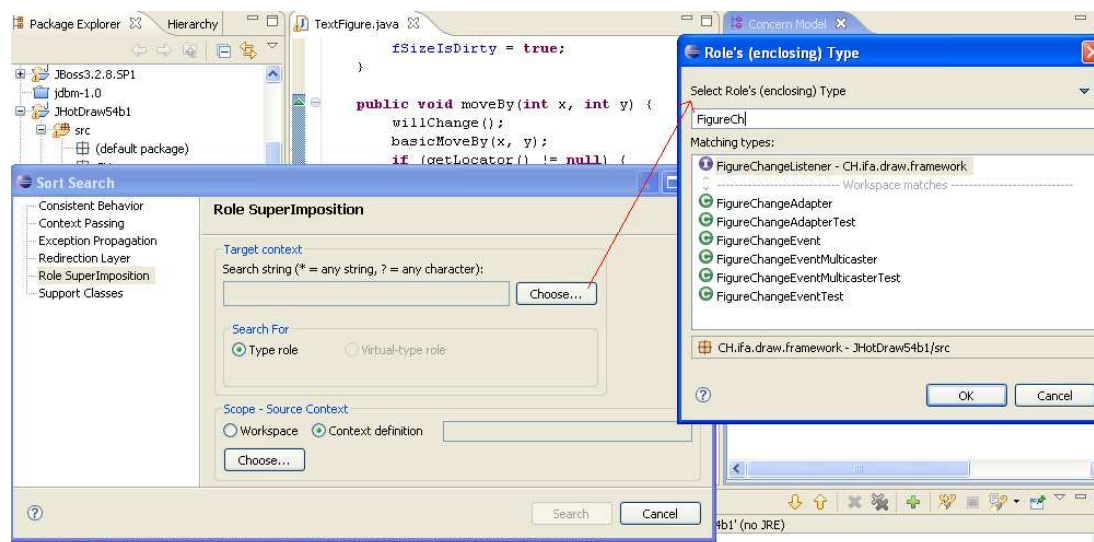


Figure B.12: Selecting the type that defines the crosscutting role.

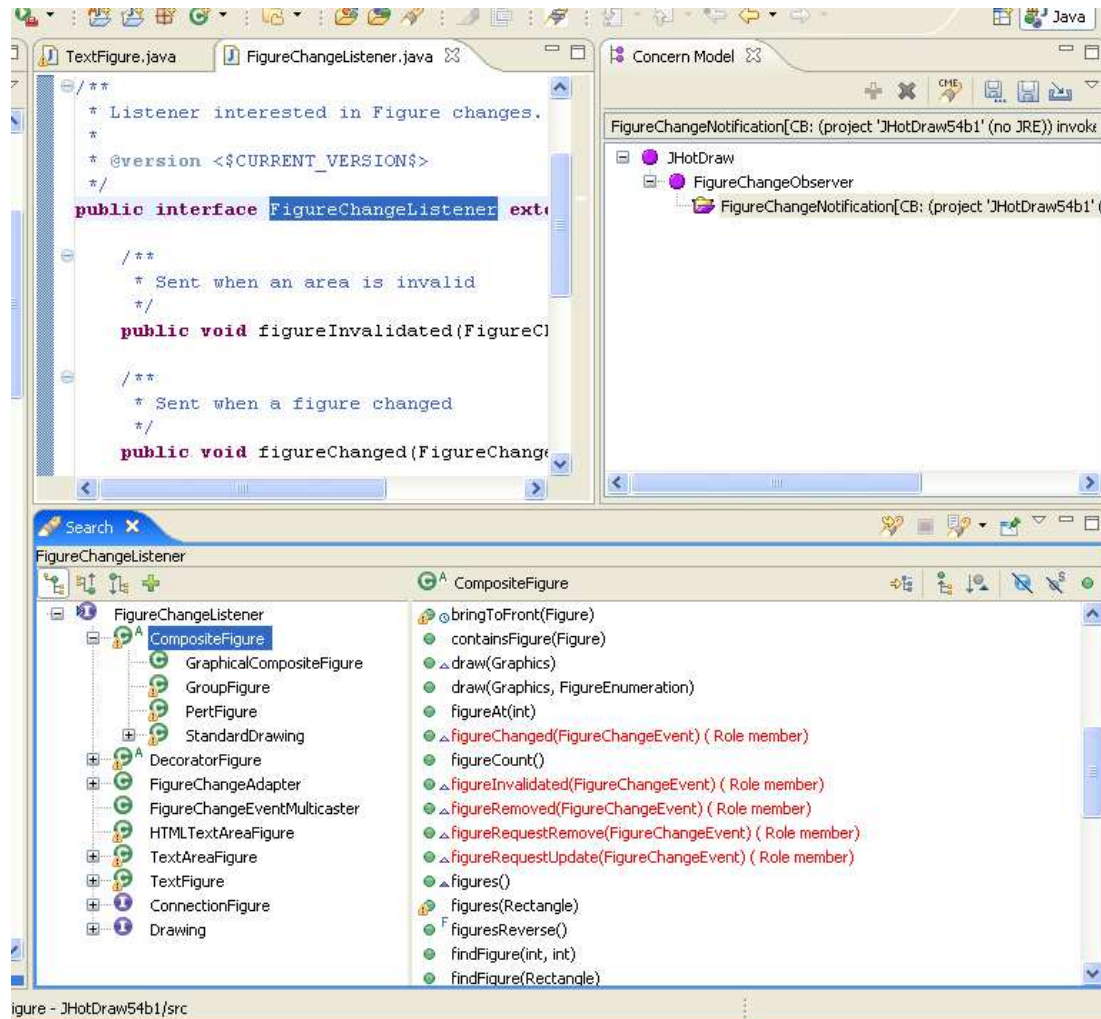


Figure B.13: Results of the query for the *Role SuperImposition* of listeners for figure changes.

We can add our sort instance for the listener role to the concern model for the FigureChange Observer following the same steps as for the notification concern discussed earlier.

Role SuperImposition (RSI) – virtual roles

The second crosscutting role we would like to document is implemented by Figure elements to allow listeners to register for and receive notification of changes. This is the Observable or Subject role in the Observer pattern [Gamma et al., 1994].

Unlike the listener role above, the Subject role in our Observer pattern implementation is not defined by a distinct interface. To document this role in SOQUET, we choose the *Figure* interface for the target context, and then select in the dialog for the *Role SuperImposition* query the Virtual-type role option. The selection opens a window, as shown in Figure B.14, that allows us to select the members of the *Figure* classes that belong to the crosscutting role. The rest of steps are then similar to the documentation of the listener concern previously discussed.

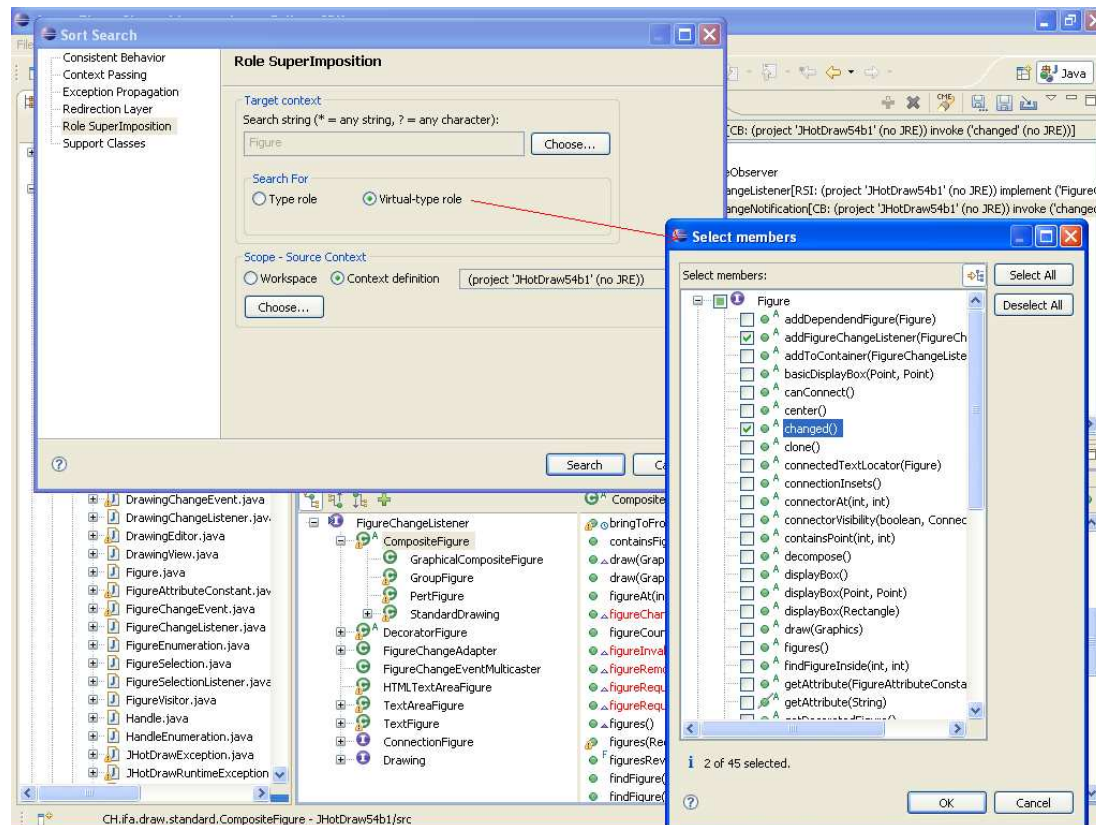


Figure B.14: Virtual roles in SOQUET.

The results of the search and all the concerns we documented so far are shown in Figure B.15. We can now save our model by selecting the Save command in the

Concern Model view.

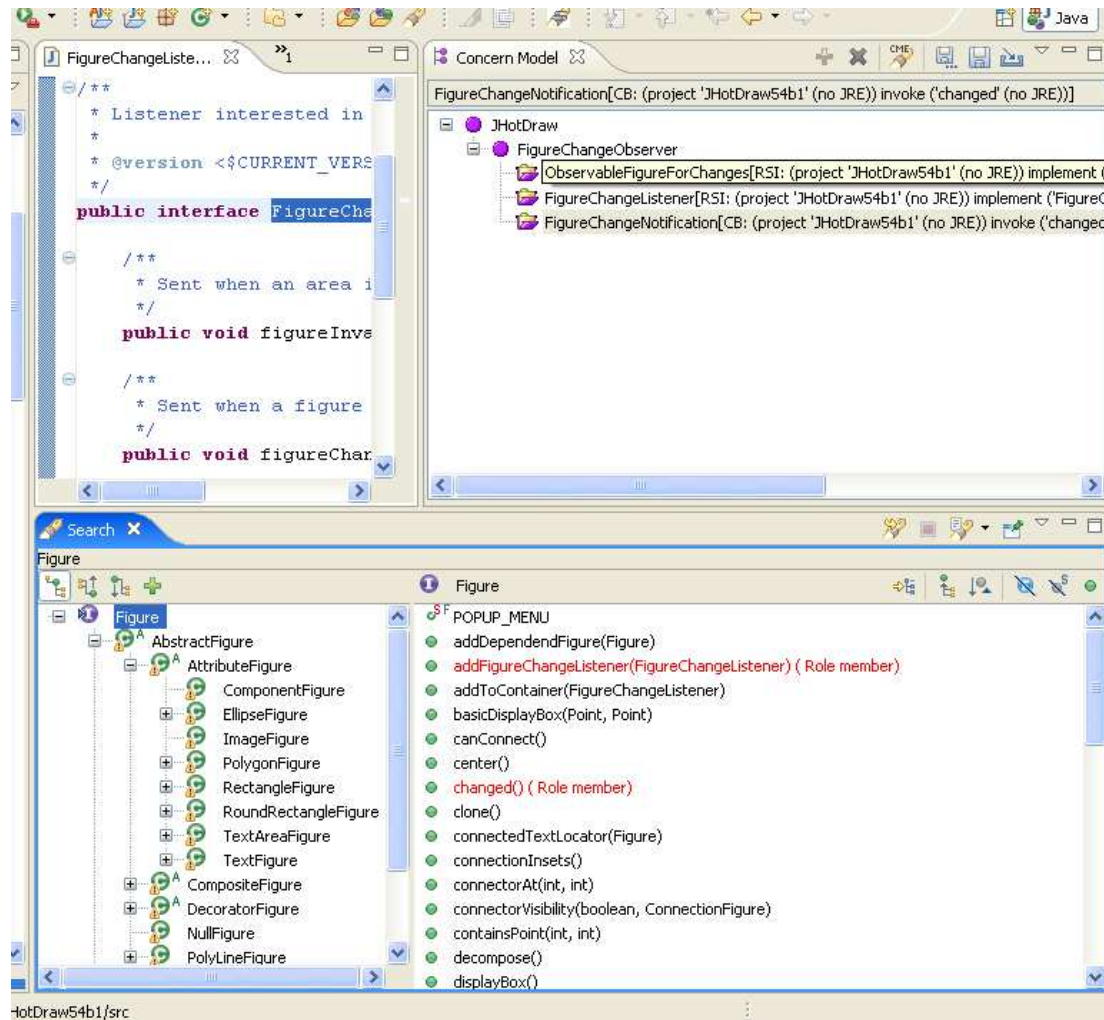


Figure B.15: The concerns documented in the FigureChange Observer.

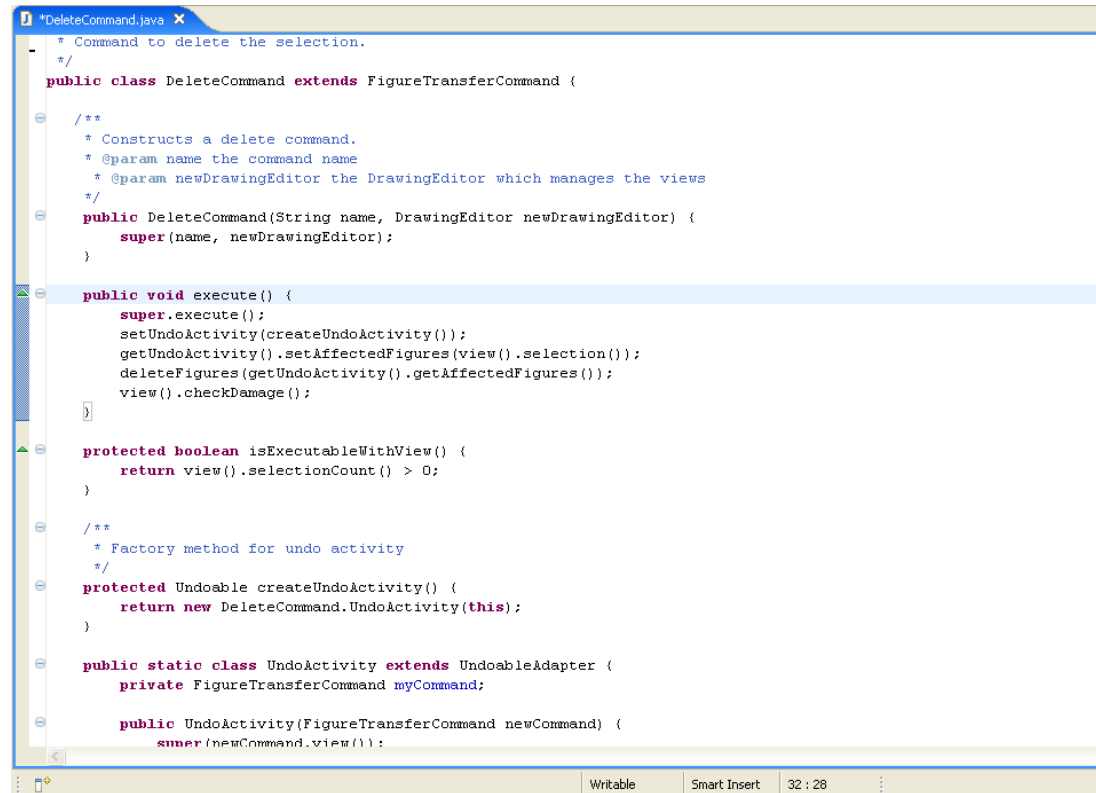
B.2.2 Using SOQUET to aid program comprehension and software change tasks

A second use-case scenario we shall look at consists of using SOQUET and an existing concern model to support us in a software change task. In this case, we assume that JHOTDRAW is a new system into which we have little or no insight. We would like to use a provided concern model in SOQUET to be able to extend JHOTDRAW consistently with concerns already present in this system.

The change we would like to make consists of an extension of the Command support in JHOTDRAW, namely adding a Command for mirroring a selected figure in the drawing view.

Command classes in JHOTDRAW implement actions to be run from the application's menus, such as copying and pasting figures in a drawing view, changing the color of a geometrical figure, etc. Each command implements the *Command* interface and implements the core logic of its action in the *execute* method. As an example, Figure B.16 shows the command for deleting selected figures from a drawing (view).

(Most of the) Commands can also be undone. The logic of undoing a command is implemented by a nested *UndoActivity* class, for each command class. The *UndoActivity* for *DeleteCommand* is also partially visible in Figure B.16.



```

* Command to delete the selection.
*/
public class DeleteCommand extends FigureTransferCommand {

    /**
     * Constructs a delete command.
     * @param name the command name
     * @param newDrawingEditor the DrawingEditor which manages the views
     */
    public DeleteCommand(String name, DrawingEditor newDrawingEditor) {
        super(name, newDrawingEditor);
    }

    public void execute() {
        super.execute();
        setUndoActivity(createUndoActivity());
        getUndoActivity().setAffectedFigures(view().selection());
        deleteFigures(getUndoActivity().getAffectedFigures());
        view().checkDamage();
    }

    protected boolean isExecutableWithView() {
        return view().selectionCount() > 0;
    }

    /**
     * Factory method for undo activity
     */
    protected Undoable createUndoActivity() {
        return new DeleteCommand.UndoActivity(this);
    }

    public static class UndoActivity extends UndoableAdapter {
        private FigureTransferCommand myCommand;

        public UndoActivity(FigureTransferCommand newCommand) {
            super(newCommand.view());
        }
    }
}

```

Figure B.16: DeleteCommand in JHotDraw.

We start our change task by loading in SOQUET an existing concern model that documents various concerns in JHotDraw, including concerns in the Command support. Figure B.17 shows how to load a concern model in the tool.

For a first insight into our drawing system, we can simply explore the concern hierarchy in the *Concern Model* view and use the Expand option to run some of the queries documenting atomic concerns, and navigate the results, as shown in Figure B.18.

For significantly large systems and concern models, SOQUET provides (partial) support to search the concern model for queries documenting concerns that cover a specific program element. Following the example in Figure B.19, we select from the concern model only those concerns whose queries have as one of their end points (context elements) a Command element.

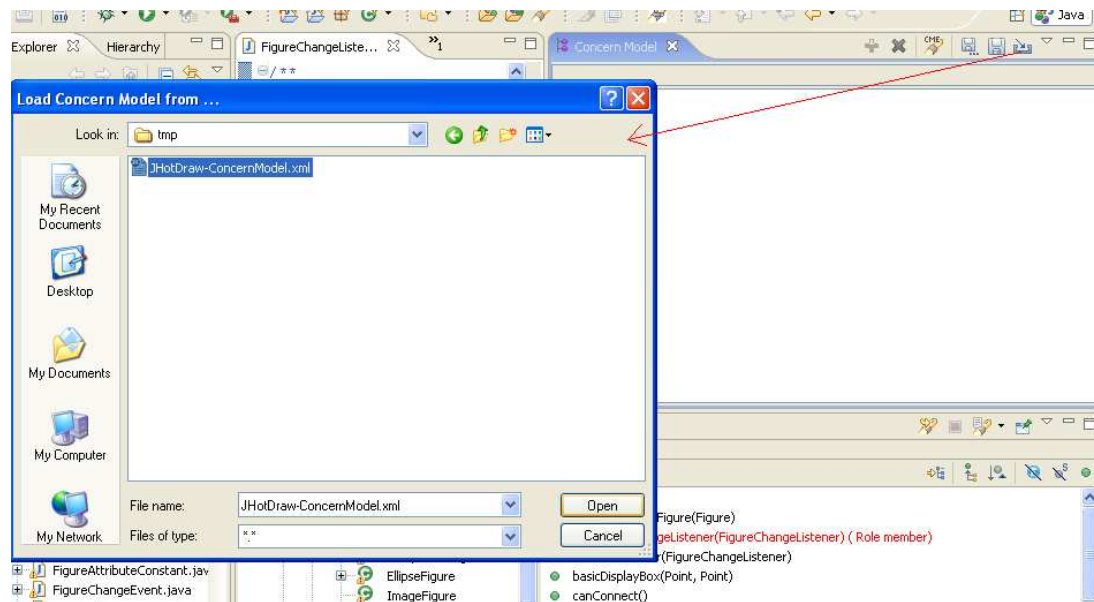


Figure B.17: Loading an existing concern model into SOQUET.

After selecting the Command interface (`CH.ifa.draw.util.Command`) from all possible name matches, the view will show us only the sort instances of interest (see Figure B.20).

By examining the concern in the root of the JHotDraw concern model and expanding its query (Figure B.21), we learn that our commands are grouped in the *Command* hierarchy. This hierarchy is rooted in the *Command* interface and a default abstract command implementation, *AbstractCommand*.

The other sort instances in the view, shown in Figure B.22, document a command as a multi-role element, which implements members to support a Listener role as well as Undoable functionality.

Based on the knowledge gained from exploring the concern model, we can create a stub class for our *MirrorCommand* by extending *AbstractCommand*, and then distinguishing in this class between the different roles. The new class is shown in Figure B.23.

After creating the stub, we can return to the full concern model and show all the concerns, as illustrated in Figure B.24.

We continue with examining some other of the documented concerns in the root of the JHotDraw model to learn more about the implementation of Commands. One such concern is shown in Figure B.25 – its description indicates a common pre-condition check for commands. We see by running the query for this concern and examining the results in the Search view that the concern's implementation consists of method calls from execute methods in various *Command* classes. The documented call is the invocation of the super's method, as shown in Figures B.25 and B.26. This call is aimed at checking a common condition in all commands, so we add it to the execute

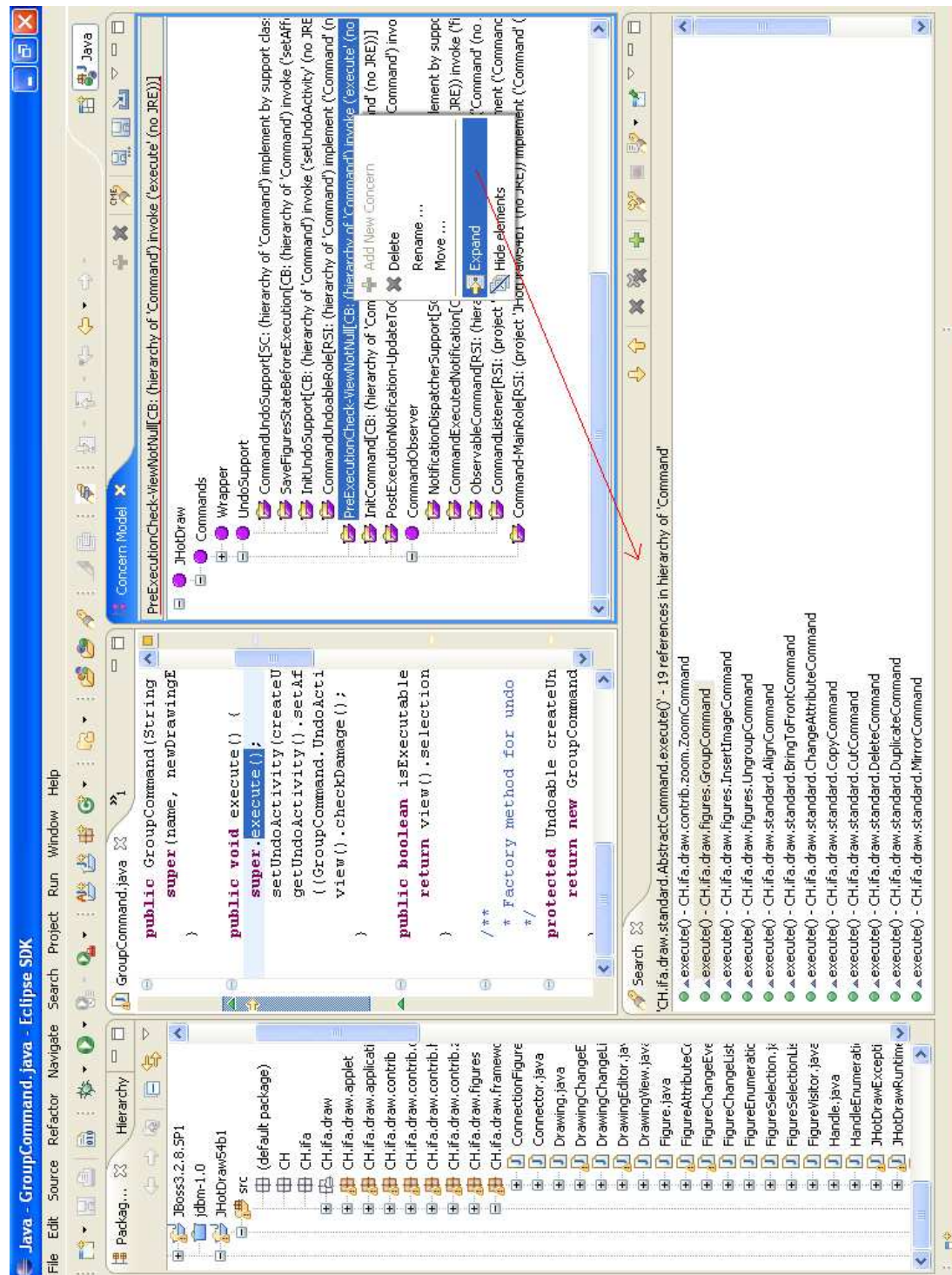


Figure B.18: Exploring the implementations of concerns using the queries in the concern model.

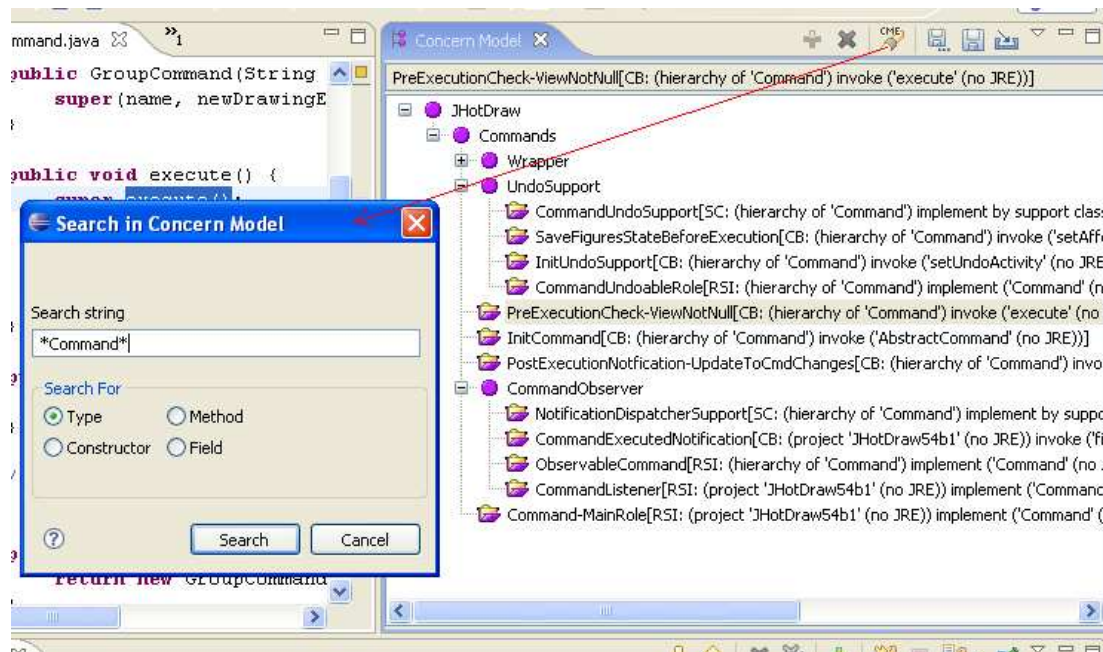


Figure B.19: Searching concerns for a program element in the concern model.

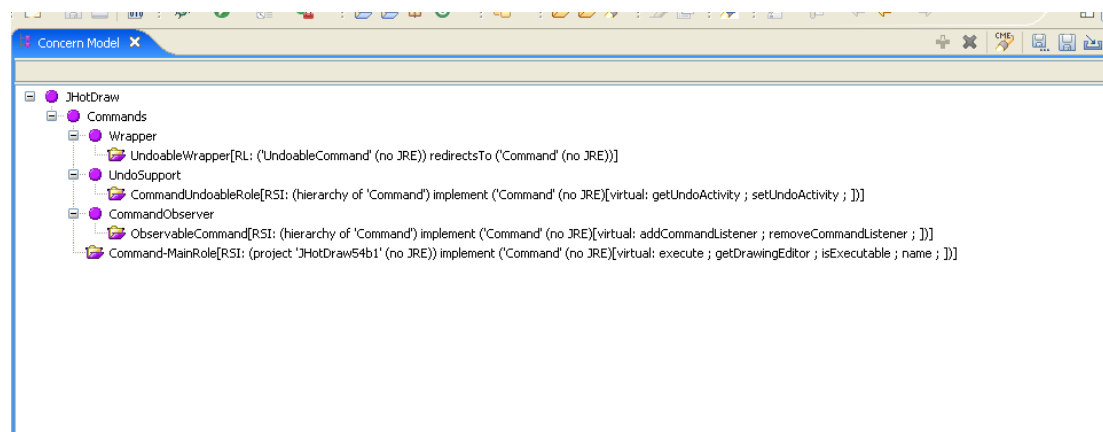


Figure B.20: Filtered concern model.

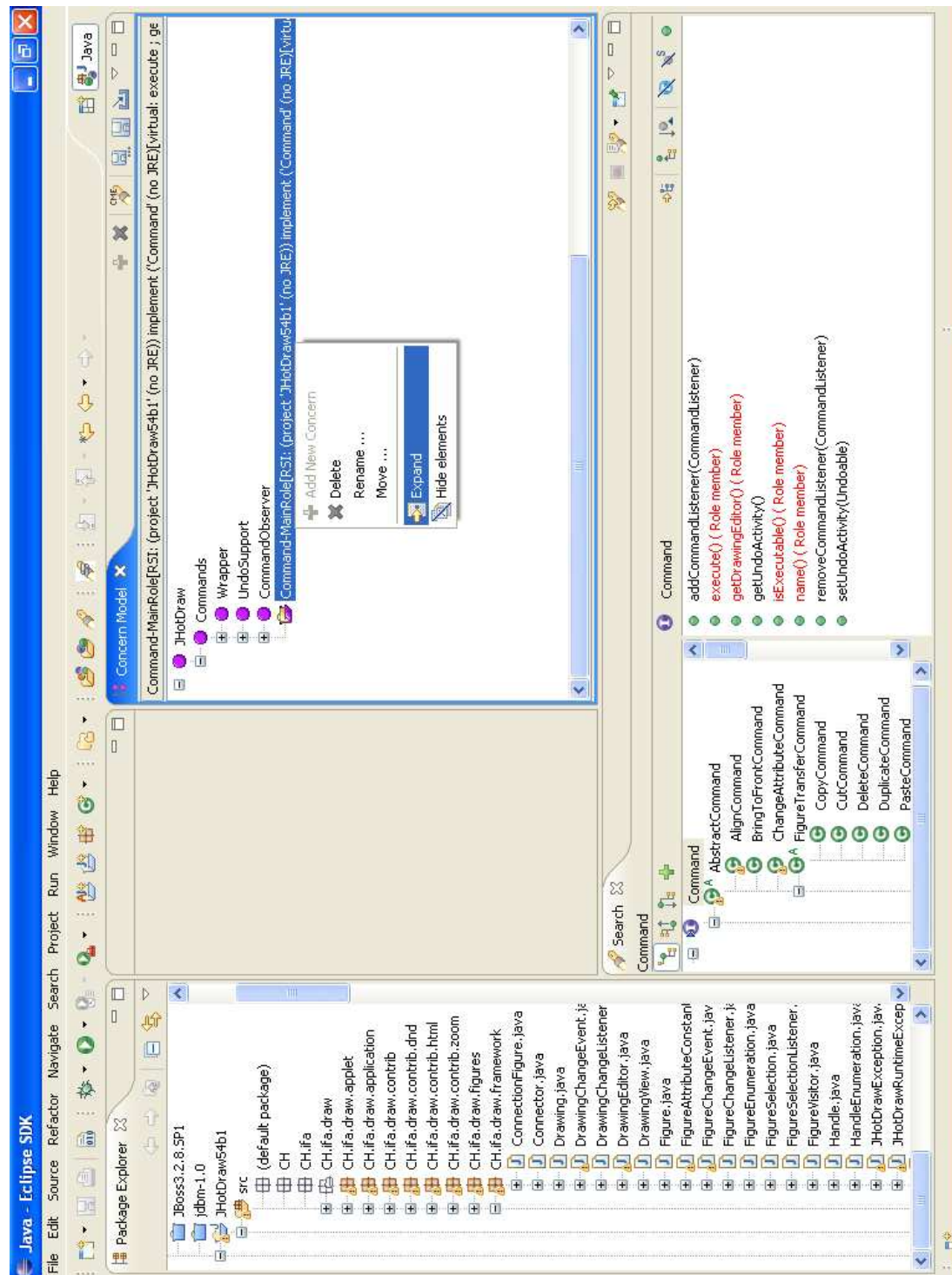


Figure B.21: Exploring the concern model by running the queries documenting concerns/sort instances.

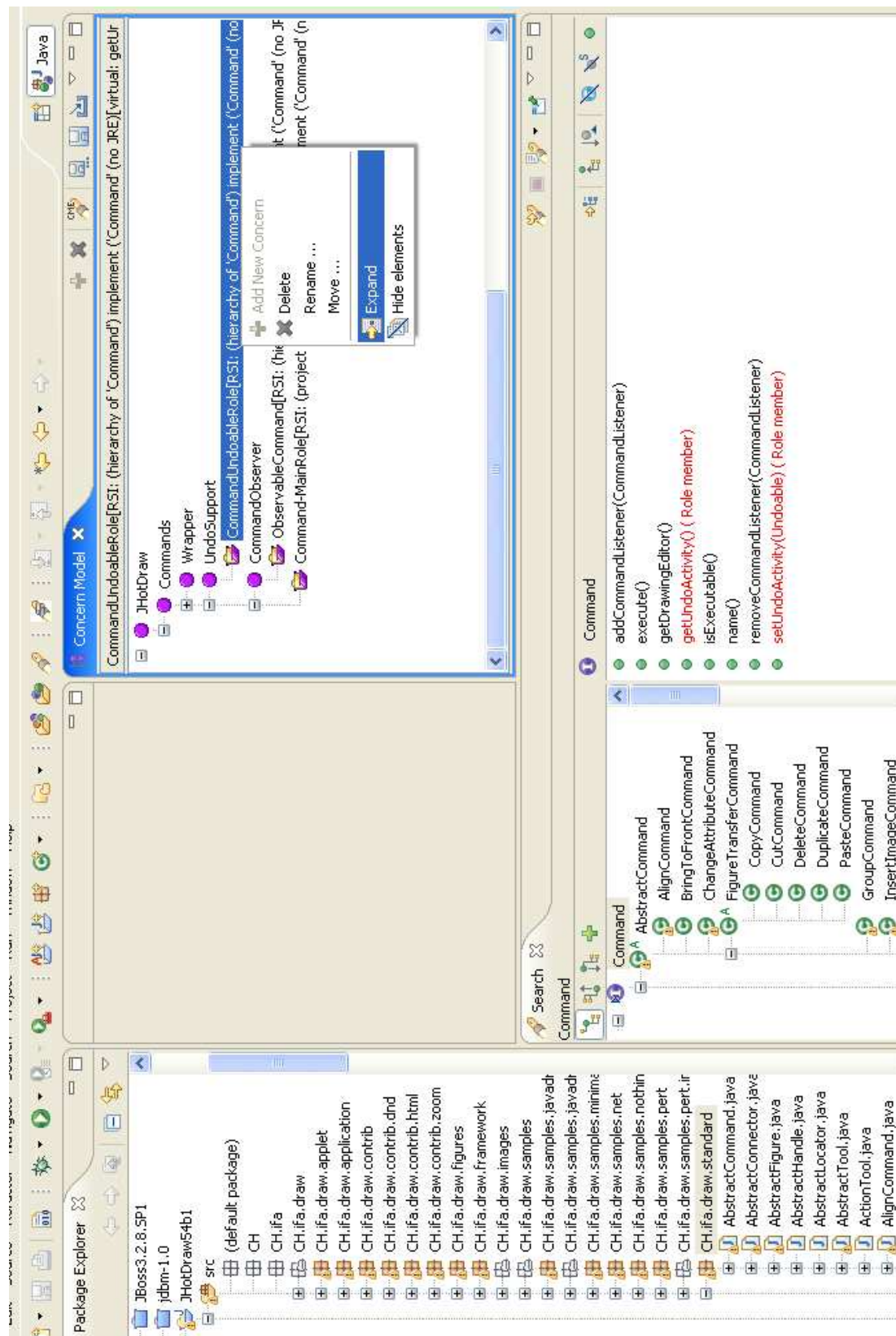


Figure B.22: The Undoable role in the Command classes.

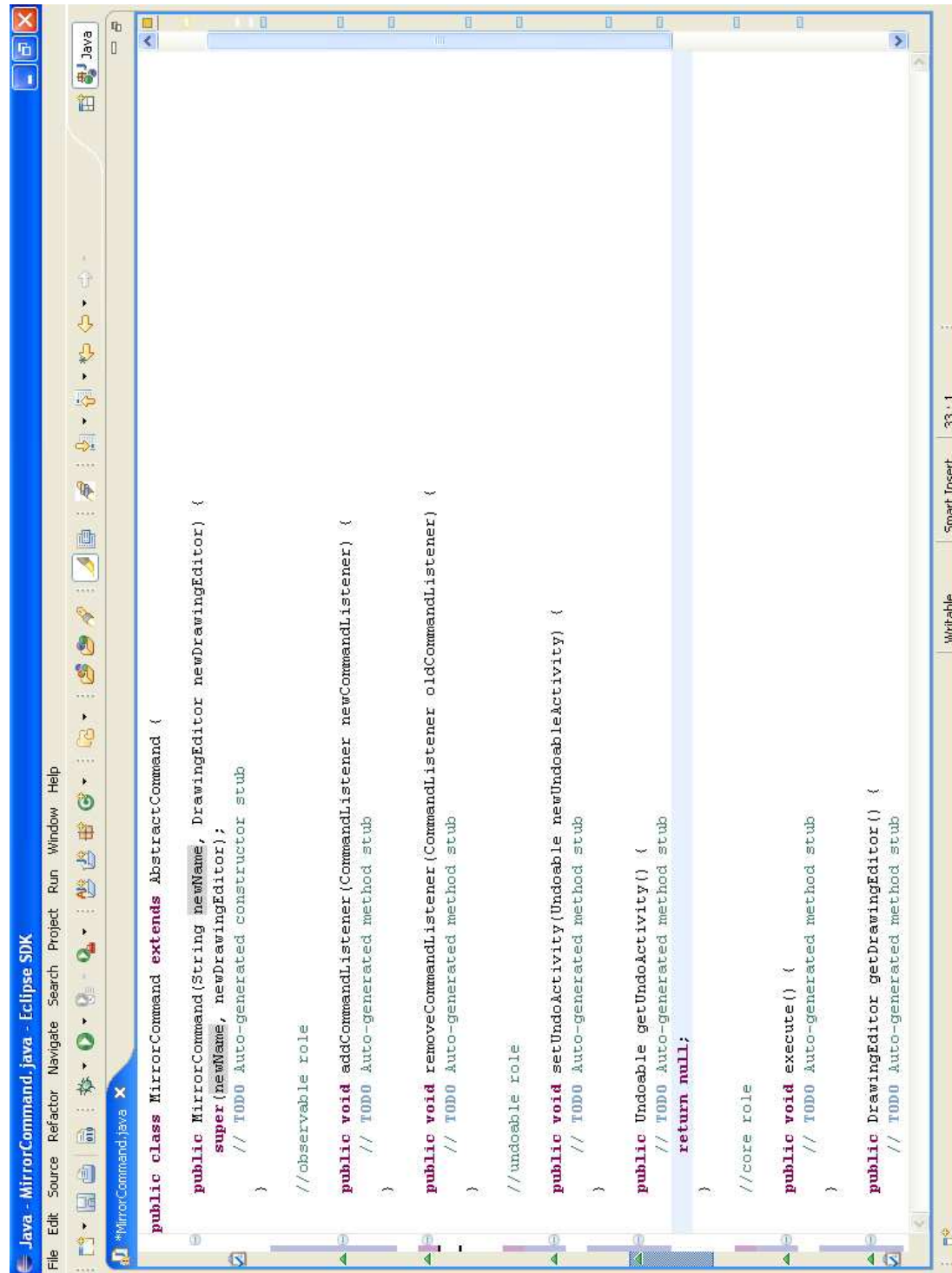


Figure B.23: The MirrorCommand stub distinguishes the multiple roles based on the documentation of concerns in SOQUET.

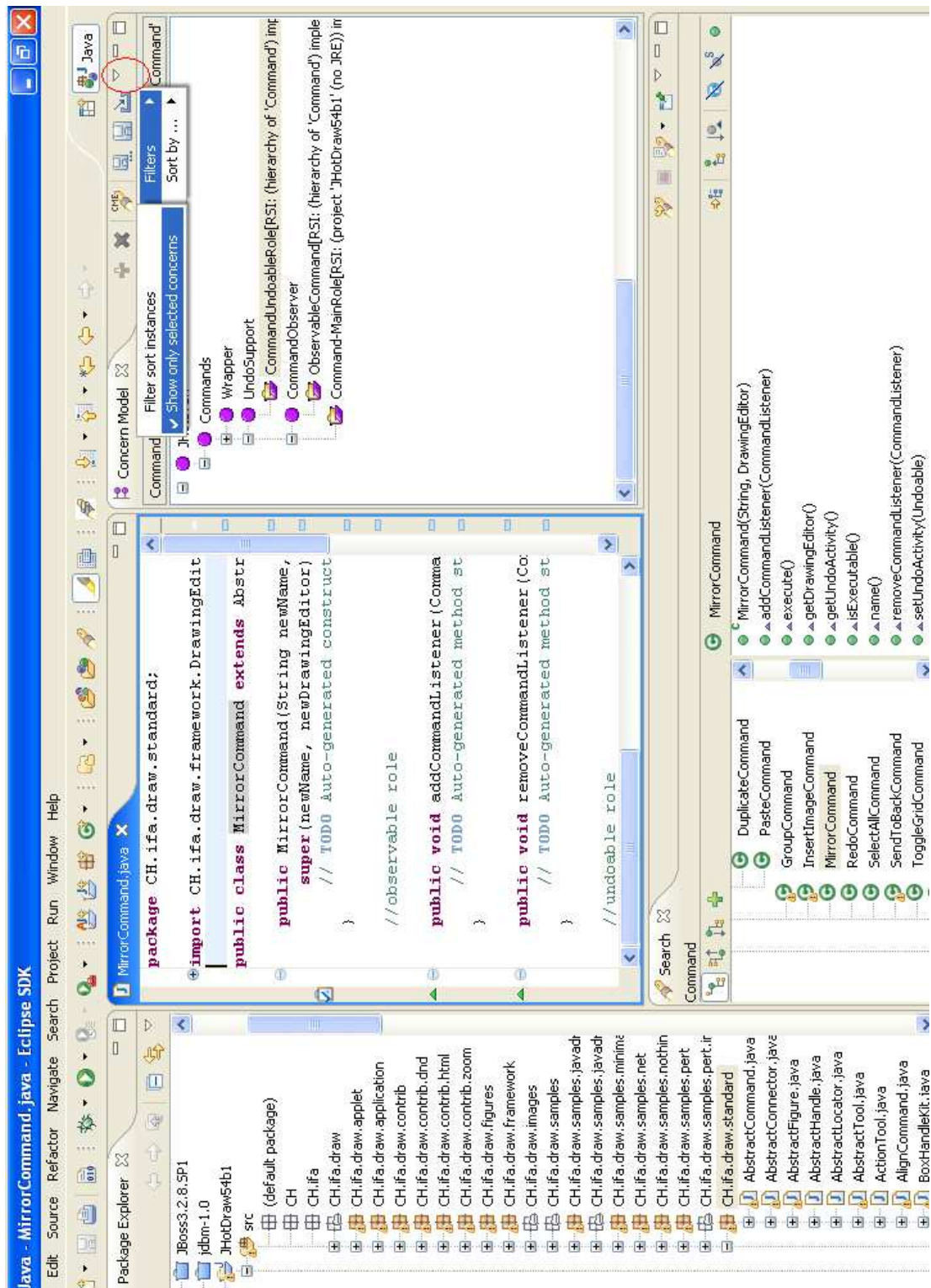


Figure B.24: Filters settings for the concerns to be displayed in the Concern Model view.

method of our new *MirrorCommand* as well.

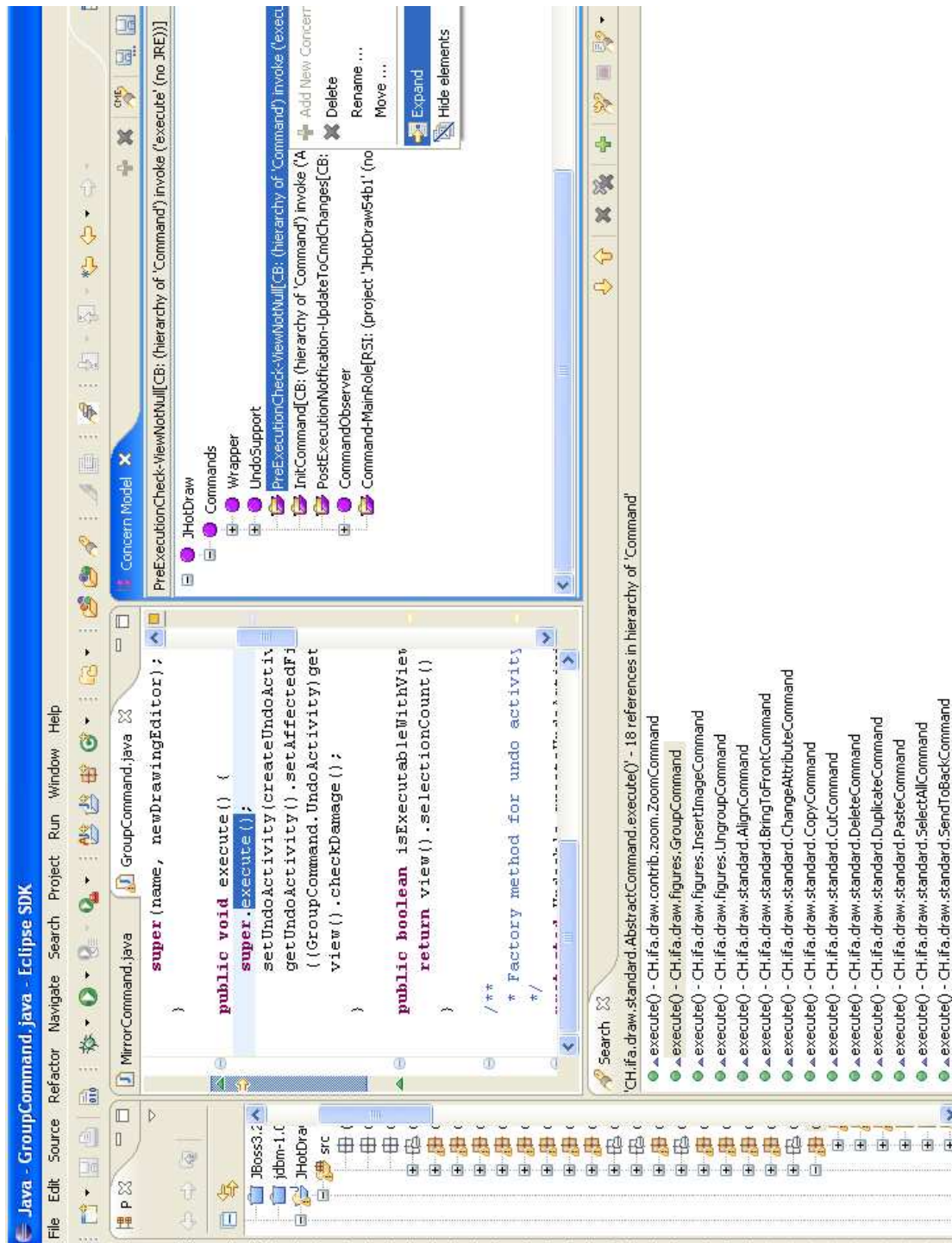
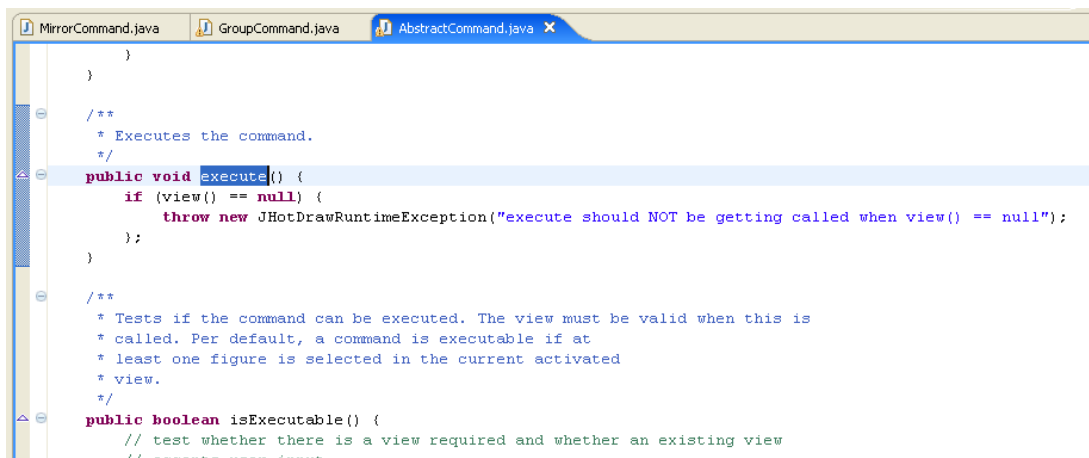


Figure B.25: The concern for pre-condition check before Command executions.

Similarly, we can investigate all the other concerns in the model and ensure that



```

}

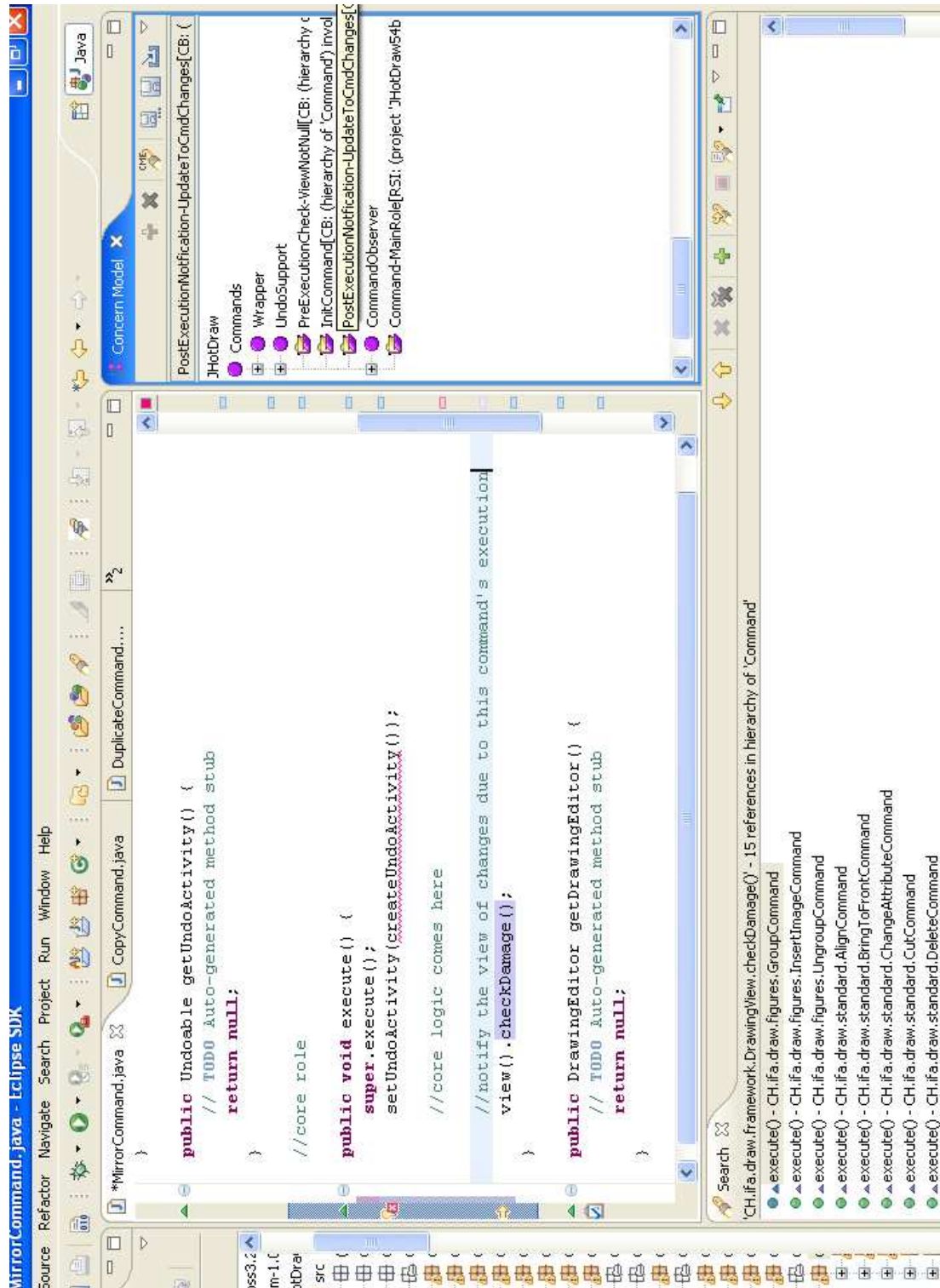
/**
 * Executes the command.
 */
public void execute() {
    if (view() == null) {
        throw new JHotDrawRuntimeException("execute should NOT be getting called when view() == null");
    }
}

/**
 * Tests if the command can be executed. The view must be valid when this is
 * called. Per default, a command is executable if at
 * least one figure is selected in the current activated
 * view.
 */
public boolean isExecutable() {
    // test whether there is a view required and whether an existing view
    // -----

```

Figure B.26: The execute() method in AbstractCommand.

the implementation of our *MirrorCommand* is consistent with all existing policies and rules for Command classes. This exploration of concerns leads us to the stub implementation of the `execute()` method for the *MirrorCommand* shown in Figure B.27. The stub shows the crosscutting concerns in this method and makes it consistent with the other existing Command implementations. Now, we can go ahead with implementing the core logic of our class. This is not a crosscutting concerns so it is up to the user of this manual to do it.

Figure B.27: The crosscutting concerns in the `MirrorCommand.execute()` method.

Samenvatting

Het onderwerp van dit proefschrift is het omgaan met *crosscutting concerns* in broncode. De evolutie van softwaresystemen beslaat het grootste deel van hun levenscyclus en dus ook van hun kosten. Daarom komt het ook veel vaker voor dat software engineers moeten werken aan complexe, reeds bestaande software systemen in plaats van nieuwe systemen te ontwikkelen. Deze reeds bestaande systemen moeten echter eerst goed doorgrond worden, alvorens wijzigingen kunnen worden doorgevoerd. Het begrijpen van deze bestaande systemen vergt inzicht in de verschillende *concerns* (denk aan functionaliteit of ontwerpbeslissingen) die de systemen implementeren. De meeste van deze concerns moeten worden afgeleid uit de broncode. Een bijzondere uitdaging voor de doorgronding van een systeem – en bijgevolg voor software-evolutie – vormen de concerns die doorsnijdend (*crosscutting*) worden genoemd: de implementatie van deze concerns doorsnijden de opsplitsing in modules. Dit heeft als gevolg dat code verspreid (*scattered*) en verweven (*tangled*) wordt.

Het onderzoek dat binnen dit proefschrift wordt gepresenteerd, biedt een geïntegreerde aanpak die consistente doorgronding, identificatie, documentatie en migratie van crosscutting concerns in bestaande systemen mogelijk maakt. Dit werk heeft als doel dat deze crosscutting concerns makkelijker begrijpbaar en beheersbaar worden voor software engineers. Een laatste stap van de aanpak die we voorstellen is een experiment dat crosscutting concerns herfactoriseert naar een aspect-geörienteerde aanpak van programmeren. Hierbij reflecteren we op de ondersteuning die deze nieuwe programmeertechniek biedt voor het verbeteren van de modularisering van concerns.

Inleiding

Moderne software systemen worden steeds complexer, en bestaan uit miljoenen regels broncode die meerdere verantwoordelijkheden implementeren, die ook *concerns* worden genoemd. Een reenvoudig tekenprogramma, bijvoorbeeld, stelt de gebruiker in staat om geometrische figuren te tekenen, te manipuleren en aan te passen; om veranderingen terug te draaien, tekeningen op te slaan in een bestand, of ze in te laden;

om te interacteren met de applicatie door middel van een reeks menu's in een grafische interface, enzovoorts. Om de complexiteit van dergelijke systemen te beheersen maken software engineers in de ontwerpfase gebruik van bekende programmeerwijzen, zoals het toekennen van verantwoordelijkheden in de applicatie aan specifieke programmamodules zoals *klassen* of *methoden*. Deze werkwijze wordt ook wel de *scheiding van verantwoordelijkheden* of *separation of concerns* genoemd [Parnas, 1972; Dijkstra, 1997; Baldwin and Clark, 1999].

Voor niet-triviale software systemen is er echter geen manier om een volledige opdeling van concerns te bewerkstelligen. Het resultaat is dat de implementatie van bepaalde concerns is verspreid over verscheidene modules, en dus wordt vermengd met de voornaamste functionaliteiten van deze modules. Zulke alomtegenwoordige concerns hebben een doorsnijdend karakter, en worden (*crosscutting*) genoemd. Hun karakteristieke implementatie maakt dat ze in de broncode moeizaam zijn te herkennen en te begrijpen, wat ze tot een belangrijke uitdaging maakt bij het aanpassen en evolueren van bestaande software-systemen. Dit proefschrift gaat deze uitdaging aan door een geïntegreerd systeem te introduceren voor het werken met alomtegenwoordige concerns in broncode.

De onderzoeksmethode die in dit proefschrift wordt gehanteerd is gebaseerd op de volgende pijlers:

- Het gebruik van casussen om een beter begrip te verkrijgen van het probleemdoel.
- De ontwikkeling van nieuwe theorieën, concepten, en technieken, zoals een nieuwe techniek voor de identificatie van concerns, een benadering voor het modelleren van concerns, of een nieuwe beschrijving van het fenomeen alomtegenwoordige concerns.
- De ontwikkeling van gereedschap dat de toepassing van de methoden en technieken op bestaande software systemen mogelijk maakt.
- De validatie van de nieuwe methoden en technieken door middel van verkennen de casussen waarin het ontwikkelde gereedschap worden toepast op een reeks “open source” Java systemen.
- Een analytische generalisatie van de casusresultaten, met daarin een kritische discussie van de bevindingen.

Probleemdefinitie

Het omgaan met alomtegenwoordige concerns in broncode omvat meerdere activiteiten, zoals de identificatie, modellering, en documentatie of herconstructie van concerns naar aspect-georiënteerd programmeren. Dit laatste is een nieuwe manier van programmeren waarin crosscutting concerns toch op één lokatie gedefinieerd kunnen worden,

waarna ze met behulp van programma-transformaties in de rest van de code geweven kunnen worden.

Hieronder bespreken we in het kort elk van deze activiteiten.

Aspect-opsporing Aspect-opsporing (*aspect mining*) is een relatief recent onderzoeksgebied waarin de ontwikkeling van (broncode analyse) technieken en ondersteuning voor de (semi-)automatische identificatie van alomtegenwoordige concerns in bestaande systemen centraal staat.

De identificatie van alomtegenwoordige implementaties is een noodzakelijke eerste stap om bewustzijn bij ontwikkelaars te creëren dat het systeem dergelijke concerns implementeert. Dit bewustzijn is nodig bij elke functionele wijziging die een ontwikkelaar doorvoert, daar deze wijziging immers altijd kan interfereren met één of meer alomtegenwoordige eigenschappen.

Bovendien is deze stap belangrijk om de aard te doorgronden van voorkomens van alomtegenwoordige concerns in “echte” applicaties, van hun typische implementaties, en van de specifieke eigenschappen die hen onderscheiden van andere concerns.

Het Modelleren van Concerns De volgende opgave is de representatie van de geïdentificeerde alomtegenwoordige concerns in broncode, om hen eenduidig en systematisch te beschrijven, te modelleren, en te documenteren. Deze *concern modeling* stap stelt ons onder meer in staat om beschrijvingen van ontdekte concerns op te slaan. Bovendien helpt dergelijke documentatie de alomtegenwoordige betrekkingen tussen programma-elementen expliciet te maken, en daarmee het verkrijgen van softwarebegrip en het uitvoeren van evolutietaken te vergemakkelijken.

Aspect-georiënteerd Programmeren en Herconstructie naar Aspecten Aspect-georiënteerd programmeren omvat verscheidene programmeertechnieken die zijn ontworpen om modularisatie van alomtegenwoordige concerns in broncode te ondersteunen door gebruik te maken van nieuwe taalconstructies en compositiemechanismen. De populairste van deze benaderingen is vooralsnog AspectJ⁴ [Kiczales et al., 1997], een Java taaluitbreiding die is gebaseerd op het *joinpoint* model. Dit model stelt de programmeur in staat om bijvoorbeeld reeksen executiepunten in een programma aan te duiden waar bepaalde code wordt uitgevoerd, zoals een methode-aanroep.

Om de modulariteit van concerns in bestaande systemen te verbeteren d.m.v. aspect-georiënteerde technieken, moeten we deze concerns *migreren* door hun implementatie te herstructureren naar een aspect-georiënteerde oplossing.

Uitdagingen en Probleemdefinitie Ondanks een behoorlijke dosis bestaand onderzoek is een aantal belangrijke problemen die betrekking hebben op het beheren van alomtegenwoordige concerns in broncode nog niet voldoende opgelost. De verscheidene

⁴eclipse.org/aspectj/

oplossingen die vooralsnog beschikbaar zijn, zijn doorgaans moeizaam met elkaar te integreren. Ook is het moeilijk onderlinge resultaten te vergelijken, omdat criteria voor een uniforme evaluatie ontbreken. Zelfs binnen één en dezelfde benadering, bijvoorbeeld voor de opsporing van concerns of herstructurering, worden alomtegenwoordige concerns behandeld op verschillende niveaus van granulariteit. Dit bemoeilijkt het vergelijken en combineren van deze oplossingen. Hier komt nog bij dat ondersteuning in de vorm van vrij beschikbaar gereedschap en gedetailleerde casussen schaars zijn.

Dit proefschrift richt zich op alomtegenwoordige concerns in bestaande systemen en streeft ernaar de volgende onderzoeksvraag te beantwoorden:

Hoe kunnen we op consistente wijze alomtegenwoordige concerns in bestaande systemen beheren, dat wil zeggen, identificeren, modelleren, documenteren, en wellicht migreren, om zodoende het verkrijgen van softwarebegrip te ondersteunen en softwareevolutie te verbeteren?

Aanpak en Resultaten

We gebruiken de volgende aanpak om onze onderzoeksvragen te beantwoorden:

1. We beginnen met een studie van crosscutting concerns in bestaande systemen. Om deze studie te ondersteunen gebruiken we een nieuwe aspect mining techniek.
2. Met het begrip van crosscutting concerns dat we hebben opgedaan door bestaande systemen te bestuderen, stellen we een categorisatie voor van concerns in *soorten* die gebaseerd is op typische implementatie idiomen en specifieke relaties.
3. Vervolgens gebruiken we deze soorten van crosscutting concerns om een geïntegreerd systeem te bouwen dat ondersteuning biedt bij het omgaan met crosscutting concerns in broncode. Dit systeem bestaat uit drie componenten, respectievelijk een component voor aspect mining, voor het documenteren en modeleren van concerns en voor het refactoren van concerns naar aspect-georiënteerde oplossingen.

De belangrijkste contributies van dit proefschrift kunnen als volgt samengevat worden:

- Het tot op heden meest uitvoerige rapport over aspect mining resultaten en crosscutting concerns in broncode. We analyseren en rapporteren de resultaten van drie relevante open-source software case studies in de hoofdstukken 2, 3 en 5.
- Een verzameling van drie aspect mining technieken met bijbehorende programma-ondersteuning, inclusief combinaties van deze technieken, die worden besproken in de hoofdstukken 2 en 5.

- Een nieuwe classificatie van crosscutting concerns op basis van onderscheidende eigenschappen en een programma-ondersteunde, vraag-gebaseerde aanpak voor het documenteren en modelleren van concerns, zoals wordt beschreven in hoofdstuk 4.
- Een nieuwe aanpak voor het herfactoriseren van concerns naar een aspect-georiënteerde oplossing gebaseerd op elementaire crosscutting concerns. Voorts stellen we een showcase voor het refactoren naar aspecten voor, die beschikbaar is als een open-source project, genaamd AJHOTDRAW. Dit open-source project is momenteel het grootste publiek beschikbare software systeem dat het resultaat is van een herfactorisering naar een aspect-georiënteerde oplossing. De aanpak en zijn toepassingen worden besproken in hoofdstuk 6.
- Een geïntegreerde migratie-strategie die volgende stappen omvat: aspect mining, concern documentatie en modellering en aspect herfactorisering, wordt voorgesteld in hoofdstuk 6.

Conclusie

Met dit proefschrift hebben we beoogd de stand der techniek te versterken op het gebied van het management van crosscutting concerns in broncode. Dit heeft geleid tot een uitgebreide verzameling van technieken, bijbehorend software-greedschap, en gedetailleerde rapporten van uitgevoerde case studies. Bovendien stellen we een geïntegreerde aanpak voor die toelaat concerns the managemen en te migreren. Deze contributies zijn bedoeld om softare engineers beter te laten omgaan met de complexiteit van bestaande software systemen en met de taken die nodig zijn om dergelijke systemen te laten evolueren.

In het laatste hoofdstuk van deze thesis identificeren we een aantal onderzoeksvragen die in het verlengde van het onderzoek van dit proefschrift liggen, in het bijzonder op het vlak van concern management.

Curriculum Vitae

Marius Marin was born on September 30th, 1976 in Bucharest. From 1995 to 1996 he studied at the Police Academy in Bucharest, after which he transferred to the Technical University of Civil Engineering, Bucharest. He graduated in 2000 with a Diplomat Engineer degree in Civil Engineering – Buildings Services. From 1997 to 2002, he studied at the Academy of Economic Studies, Bucharest, where he graduated with a Licentiate degree in Economics – Economic Informatics (IT). Between 2001 and 2003 he worked as an IT specialist with Nergal, an IT solutions and consultancy company based in Rome.

He started his PhD research in 2003, at Delft University of Technology. Until 2007 he was a member of the Software Engineering group, the Software Evolution and Research Laboratory.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemsen. *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining.* Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μ CRL.* Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand.* Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations.*

Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell.* Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenbergh. *Route Planning Algorithms for Car Navigation.* Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets.* Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics.* Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station.* Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents.* Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures.* Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling.* Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications.* Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures.* Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network*

Reliability. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoetewij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applica-*

tions. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

- L. Brandán Briones.** *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05
- I. Loeb.** *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06
- M.W.A. Streppel.** *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07
- N. Trčka.** *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08
- R. Brinkman.** *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09
- A. van Weelden.** *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10
- J.A.R. Noppen.** *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11
- R. Boumen.** *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12
- A.J. Wijs.** *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13
- C.F.J. Lange.** *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14
- T. van der Storm.** *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-15
- B.S. Graaf.** *Model-Driven Evolution of Software Architecture.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16
- A.H.J. Mathijssen.** *Logical Calculi for Reasoning with Binding.* Faculty of Mathematics and Computer Science, TU/e. 2007-17
- D. Jarnikov.** *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18
- M. A. Abam.** *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19
- W. Pieters.** *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01
- A.L. de Groot.** *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02
- M. Bruntink.** *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03
- A.M. Marin.** *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04