Extracting Architectural Information Using Different Levels of Collaboration

Diplomarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Tobias Aebi

2003

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz
Prof. Dr. Roel Wuyts

Institut für Informatik und angewandte Mathematik

Further information about this work, the used tools and an *online* version of this document can be found at:
`http://www.iam.unibe.ch/~scg/`

The address of the author:

Tobias Aebi
Schweizerhubelstrasse 13
3052 Zollikofen

or

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
`aebi@iam.unibe.ch`
`http://www.iam.unibe.ch/~aebi/`

Abstract

In software reengineering one of the main problems is missing or out-of-date documentation of a system. To solve this problem, tools exist to recover information from the source-code and to build high-level models of a system.

High-level models extracted by tools usually consist of boxes and arrows between them. With the abstraction from the source code we get an overview of the current system and gain a high-level model of it. What is missing in current approaches is that they only consider collaboration between components and do not address collaboration inside the components.

Considering not only the collaboration between the components of the high-level model but also the collaboration *within* the components improves considerably the value of information the extracted model provides. Our approach extracts dynamic and static collaboration information of a system and shows different levels of collaboration in one single view without loosing the architectural overview of the system.

We validate the benefits of our approach by comparing the high-level models represented by collaboration-views to strict high-level models based on structural information. Our case studies show that we do not only reach better understanding of the system but additionally detect meaningful collaboration patterns and possible unfavorable dependencies in the system.

Acknowledgments

On the way getting to this thesis I got kind support of many people. First I wish to thank my supervisor Roel Wuyts for his excellent guidance, and the head of the group Oscar Nierstrasz for giving the opportunity to work in this group.

I want to thank all the other people of the Software Composition Group for their support, for always answering questions, reading and giving wise comments. Especially I want to thank Michele Lanza for having nice discussions and a lot of fun, Gabriela Arevalo for her sense of humor and her social competence, Stephane Ducasse reading the work giving useful comments.

Thanks to all the students from the pool making possible to survive a cold winter and a hot summer - without you, life would have been hard. Those guys enlightening life, exchanging ideas and spending nice times together: Frank Buchli, David Vogel, Daniele Talerico, Calogero Butera, Tom Bühler.

Giving very useful feedback on a presentation that had a direct impact on rest of the work I want to thank Bernhard Rytz. For a final proof-read and some lucky hours, thanks to Andreas Schlapbach.

For life beyond university I want to thank my best friend Patrick and his wife for always being there and last but not least I want to thank my parents for the support and the encouragement they give to me and all of my brothers whenever one of us gets in pain.

Tobias Aebi
September 2003

Contents

Abstract **i**

Acknowledgments **ii**

1 Introduction **1**

1.1 Contributions 3

1.2 Thesis Overview 3

2 Object-Oriented Reengineering **5**

2.1 Reengineering and Reverse Engineering 6

2.2 General approaches of Design Recovery 7

2.3 Definition of Terms 8

2.4 Existing Approaches of defining High-Level Models 8

2.4.1 Software Reflexion Models 8

2.4.2 Declaratively Codifying Software Architecture 9

2.4.3 Recovering Behavioral Design Views 10

2.5 An Approach for Visualizing Behavior between Classes 11

2.6 Discussion 11

2.7 Our Approach 12

2.7.1 The Goals 12

2.7.2 Our Solution 12

2.8 Conclusion 13

3 Collaboration-Views 14

3.1 Defining High-Level Models 14

3.1.1 Defining Entity Groups 15

3.1.2 Defining the Relations 16

3.2 Visualisation 17

3.2.1 Browsing-View 18

3.2.2 Collaboration-Matrix 19

3.2.3 Collaboration Patterns 22

4 Tool Support 26

4.1 MOOSE: using Software Models 26

4.2 CodeCrawler: a Software Visualization Tool 27

4.3 Collaboration Sniffer 28

4.3.1 Architecture of Collaboration Sniffer 28

4.3.2 Information Repositories 30

4.3.3 Representation of the High-Level Model 30

5 Validation: Case Studies 32

5.1 Collaboration inside CodeCrawler 32

5.1.1 Coarse Overview 34

5.1.2 Collaboration by Local Variables 35

5.1.3 Collaboration by Instance Variables 37

5.1.4 Collaboration at Runtime (Method Level) 38

5.1.5 Collaboration at Runtime (Instance Level) 40

5.1.6 Collaboration summarized 41

5.2 Comparing Application Scenarios with Application Structure 42

5.3 Collaboration in an Average-Sized System: Jun 43

5.3.1 Facilities and Limits 43
5.4 Conclusion 44

6 Extracting Collaboration Information 46
6.1 Calculating types in an dynamically typed language 46
6.1.1 Heuristic Approach 47
6.1.2 Implementation of a Type Annotator 49
6.2 Extracting runtime collaboration 53

7 Future Work 57
7.1 Refactoring Component Architecture 57

8 Conclusion 60
8.1 Lessons Learned 60

List of Figures

3.1 High-Level Models 15

3.2 Browsing-View 19

3.3 Collaboration Matrix 20

3.4 Clean Layered Structure 22

3.5 Collaboration Patterns 24

4.1 Famix Core 28

4.2 Architecture of *Collaboration Sniffer* 29

5.1 Collaboration-Matrix of collaboration in CodeCrawler version 4.274 by runtime, local variables and instance variables 34

5.2 Collaboration-Matrix of collaboration in CodeCrawler version 4.274 by local variables 35

5.3 Collaboration-Matrix of collaboration in CodeCrawler version 4.274 by instance variables 37

5.4 Collaboration-Matrix of collaboration in CodeCrawler version 4.274 at runtime (method level) 39

5.5 Collaboration-Matrix of collaboration in CodeCrawler version 4.274 at runtime (instance level) 41

5.6 Collaboration-Matrix of collaboration in CodeCrawler version 4.274 at runtime 42

5.7 Collaboration of static information: Jun version 501 45

6.1 Collaboration and Dependency Scenario 53
6.2 Collaboration and Dependency Scenario (cyclic dependency) 54

7.1 Visual Refactorings in Moose 58

List of Tables

5.1 CodeCrawler 33
5.2 Jun version 501 44

6.1 Results of Type Inference 52
6.2 Class Collaboration vs. Component Collaboration 54
6.3 Method-Level Collaboration vs. Instance-Level Collaboration 54

7.1 Classes moved between the Packages in Moose 59

Chapter 1

Introduction

> Software development typically means software enhancement. And to successfully enhance software first requires understanding it [JERD 97].

Documentation should address the need of a software engineer to understand the system, but it falls short of this in practice. The reason is that documentation is often not up to date or is even missing at all [RICH 99]. Since in the current state of software development one-man projects have become rare [LANZ 01], getting an overview of parts of foreign code is an often used requirement. Before a software system can be reengineered and/or maintained the system must be reverse engineered, e.g., the engineer needs to build a mental model of the software which allows him to take informed decisions [LANZ 03a].

To build mental models of software systems can be summarized as to gather information of the source and/or the execution of the system, to introduce abstractions and to display it in a readable form for the user. This leads to the following main questions which have to be addressed:

- Which information is useful and how can this information be extracted?
- What kind of abstractions can be built?
- How can the mental model be visualized?

Answers to these questions can look very differently depending on the perspective and the level of detail one chooses and on the properties one intends to make visible.

In the approach of CodeCrawler [LANZ 03a] a "Coarse-grained Software Visualization" is presented. The focus lies in providing scalable overviews of the structure that is explicit in the source code like classes, class hierarchies and methods. In the work about *Software Reflexion Models* [RICH 02] the information to base the mental model on is extracted from program execution traces. The focus lies in providing a high-level model of defined parts of the systems. To get an initial overview of the complete system knowledge about the system is used.

In this thesis we look at reverse-engineering of *object-oriented* legacy systems. We set the perspective building the highl-level models based on collaborations between the objects and address the problem of how different levels of detail can be visualized.

Our research questions can be refined as follows:

- How can collaboration information be extracted?
- How can different levels of details be visualized consistently?

To extract collaboration information we use two different methods in our approach. We extract *static* collaboration information of the source and we extract *dynamic* collaboration information by running the system and collecting traces.

We introduce abstractions to group the classes into subsystems. These subsystems can either be mapped automatically from existing abstraction in the source like packages in Smalltalk or can be arbitrary be generated by the engineer. This is the process of building the structure of the high-level model.

To visualize the collaboration model we introduce two different views, a *browsing-view* and a *collaboration-matrix*. Those views visualize different levels of detail: the collaborations on subsystem-, class-, and method-level.

The browsing-view is a organized as an often used "box-and-arrow" sketch with boxes representing the entities (subsystems or classes) and arrows representing the collaborations. The view allows the user to *navigate* through different levels of collaboration by interactively opening a child view on a subsystem to see the inside collaboration. Because the collaborations (arrows) keep the link with the source code they can be inspected and the involved methods are displayed.

The collaboration-matrix provides facilities to extract certain roles of classes and subsystems in respect to collaboration. This view is organized by listing the classes on the x and y axes and always marking a point when there is a collaboration between the two involved classes. Important for this view is the facility to zoom and like for the

browsing-view the connection to the involved methods in a collaboration. We define collaboration-patterns which aids the engineer to identify core components and classes in this view.

1.1 Contributions

The contributions of this thesis can be summarized as follows:

- Main Contributions: Extract collaboration information to build views of software architecture.
- Extract collaboration based on static and dynamic information and combine them.
- Provide views to detect collaboration patterns.
- Make collaborations explicit that do not conform to a layered architecture.
- Show that valuable information is gained by combining different levels of granularity.
- Extract collaborations out of static information in a dynamically typed language.

1.2 Thesis Overview

- Chapter 2: This chapter presents the terminology used in the context of software reengineering, reverse engineering and design recovery. Additionally we discuss existing approaches.
- Chapter 3: This chapter introduces our high-level model and the collaboration-views. We present the organisation and the facilities of two specific collaboration-views, the *browsing-view* and the *collaboration-matrix*.
- Chapter 4: In this chapter the approach is applied on software systems and the results are presented and discussed.
- Chapter 5: This chapter describes the extraction of collaboration information based on static and dynamic information.

- Chapter 6: This chapter gives an outlook on future work in this field of research.
- Chapter 7: The final chapter summarizes the main contributions of our work.

Chapter 2

Object-Oriented Reengineering

Lehman and Les Belady [LEHM 85] expressed in their laws of software evolution that a program that is used in a real-world environment must change or become less useful in that environment. Lately these laws became even more apparent, with systems getting more and more complex.

One of the reasons that most of todays software systems are built using object-oriented techniques is that the quality of the software is expected to be superior because of better data abstraction and better information hiding. As objects often help to bridge the gap between real world objects and software structure it is expected to promote a better understanding between software engineers. It is also expected that a better quality and understandability of the source code would simplify the process of software maintenance and to decrease costs. The real-world experience of software maintenance shows that those expectations are not met:

> Object-oriented software needs maintenance, just like conventional software. The maintenance phase will likely be the most costly part of the system life cycle.[WILD 93]

Despite the choice of object oriented languages to divide the world in objects which communicate with each other [LANZ 99] and thereby raising the level of abstraction, the complexity problems of todays software systems are not yet solved. Compared to the size of large software systems even the abstraction achieved using objects remains at a very low level. Thinking at higher levels of abstraction has become vital in software engineering because of the enormous complexity of current systems [LANZ 99]. Since objects are not able to provide the required level of abstraction there is a need to

introduce even higher levels of abstraction to get an adequate understanding of complex systems.

An important impact on the understanding of an object-oriented software system lies right at the core of its characteristics. Traditional software structuring techniques concentrate first on *function* - the function of a program, its sub-functions, their sub-sub-functions... But human cognition often works the other way, recognizing *things* first, and the function that connect them afterwards [LIU 96]. Even if the focus on objects first corresponds much better to the human way of thinking, the communication between the objects, represented by their functions, remains essential. Introducing structural abstractions does not make up for losses of the focus on functionality. Therefore the problem of understanding an object-oriented system is usually not to extract the structure of the system because the structure itself is explicit, but the complex dependencies and collaboration between the objects. Only inserting higher levels of structural abstraction, does not improve the understanding of an object-oriented system where the main problem is to understand data flow and collaboration. To insert abstractions of collaboration however improves the understanding considerably.

Reengineering and in special reverse engineering which is part of it are the main tasks of software maintenance. In the next sections we focus on the context of reengineering and reverse engineering - and after discussing general approaches of design recovery - we discuss existing approaches that extract high-level models. We show the weak points of those solutions and collect them in a list of unsolved aspects. The rest of the thesis focusses on addressing these aspects.

2.1 Reengineering and Reverse Engineering

Reengineering and reverse engineering and their relationship among each other we define as follows:

Reengineering ...is the examination and alteration of a subject system to reconstitute it in a new form... It generally includes some form of reverse engineering (to achieve a more abstract definition) followed by some form of forward engineering or restructuring [CHIK 90].

Reverse engineering is the process of analyzing a subject system to identify the systems's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction [CHIK 90].

Reverse engineering itself is part of the reengineering process. Reengineering is dependent on certain reverse engineering tasks because in order to do modifications on a software system, understanding it is a precondition. Changing a large and complex system without sufficient knowledge of its inner structure, almost certainly triggers unwanted side effects which can make the system inoperable [LANZ 99].

There are many subareas of reverse engineering [CHIK 90]. The subarea of reverse engineering where our approach focuses on is defined by the term of *design recovery*.

Design recovery recreates design abstraction from a combination of code, existing design documentation (if available), personal experience, and general knowledge about the problem and application domains. [BIGG 89]

The result of design recovery, or reverse engineering in general, is recapturing or recreating the design and deciphering the requirements actually implemented by the system [CHIK 90]. The usage of those results are to get a general understanding and overview or even for problem detection focusing on a specific problem scope.

2.2 General approaches of Design Recovery

The spectrum of how to get a better understanding by recovering design is quite large. A possible way to categorize approaches of design recovery is to take the basic strategy in account [RICH 02]. The two basic strategies are described as *bottom-up* strategy and *top-down* strategy. A third category is taken in account, *visualization tools* which is in a way complementary to the first two. As this categorization emphasis the basic properties of design recovery, which we use in our approach, we introduce them in more detail.

Top-Down approach. In top-down approaches programmers hypothesize an initial model reflecting their current understanding of it, then analyze the code to confirm or reject the hypothesis and use this information to revise their model.

Bottom-Up approach. In bottom-up approaches programmers chunk low-level code artifacts into higher-level abstractions.

Visualization Tools. Visualization Tools display the low-level abstractions or the description of the expected models. Visualization tools can also display metrics associated with software structures.

These categories do not provide a unambiguous classification of design recovery tools. In most cases recovery tools contain elements of each of those categories.

2.3 Definition of Terms

To be clear in the use of certain terms we define how they are used in this thesis.

Mental Model. A mental model is the model what the engineer thinks of a system to be like.

Software Model. A software model is an abstract model of a software system extracted from the source code.

High-Level Model. A high-level model describes a *software model* at a higher-level of abstraction.

2.4 Existing Approaches of defining High-Level Models

Looking at existing approaches delimits the field of research on extracting software architecture. The three approaches we chose to present in more detail summarize a lot of aspects of architectural extraction.

The work of *Software Reflexion Models* [MURP 95] focusses on the iterative refinement of a high-level model. The approach of *Declaratively Codifying Software Architecture* [MENS 99] provides facilities to check conformance between the model and the source. The approach of *Recovering Behavioral Design Views* [RICH 02] integrates ideas of iterative refinement and declarative codifying software architecture.

2.4.1 Software Reflexion Models

One of the main ideas in the work about Software Reflexion Models [MURP 95] is that the first level goal is to compare high-level models with source models rather than to discover high-level models. The engineer has to specify the high-level entities explicitly. Therefore the precondition for an engineer is to have a sufficient knowledge

about the system to be able to define an initial high-level model. Having computed once a reflexion model of a system the engineer has then to interpret it which can either lead to adapt the initial high-level model calculating the reflexion model again till a satisfactory state is reached or to study specific divergences in the source to understand the unexpected differences.

The definition of the high-level model entities is done in this approach by a source-mapping on file level by using regular expressions. The high-level model relation is defined by tuples between high-level model entities. The results are visualized by using a graph-view showing the high-level model entities as boxes and the convergence, divergence and absence relations as different marked arrows.

The use of this approach is to get a better understanding by iteratively modifying the high-level model and to do design conformance checks. The practical usability is shown on several systems. The source mapping which is used to define the high-level models is quite flexible but is limited to systems where the structure is explicit on the level of file systems. This approach does not provide any direct link from the visualization of the collaboration to the source of collaboration.

2.4.2 Declaratively Codifying Software Architecture

Wuyts/Mens [MENS 99] describe a way to codify software architecture declaratively. An architectural description consists of a list of components (virtual classifications) of which the architecture is composed and a list of relationships among the architectural components.

The virtual classifications are defined with logical rules. Besides using straightforward mappings to source-code artifacts the mappings can be cross-cut the system or other architectural entities can be used. The following example defines a virtual classification called *UnifyingQueries* using two logical rules. The first is that the software artifacts have to be in the hierarchy of the class *Query*, the second is that they have to implement the method *unifyQuery:*. [1]

```
classIsClassifiedAs(?Class, UnifyingQueries)
    hierarchy([Query], ?Class),
```

[1] This code extract is presented in SOUL syntax which is very similar to Prolog syntax, with a few notworthy differences. Instead of using square brackets '[' and ']', SOUL lists are delimited by '<' and '>'. In SOUL, '[...]' denotes a re-ification of Smalltalk code in the SOUL language. Finally, SOUL variables start with a '?' instead of with a capital

classImplements(?Class, [#unifyQuery:])

The relationship between the virtual classifications is defined by intuitive connectors such as *uses, creates, accesses* etc. The relationships are not only defined between virtual classifications, they are overloaded at many levels of abstractions. They are translated into relationships between lists of source-code artifacts. In the case of a *uses*-relationship between two methods it is just checked wheter there is a message invocation between the two.

The power of this approach is its intuitive way of defining and refining the architecture with simple rules. The contribution is to provide conformance checks of the declaratively codified architecture and the source. As the main goal is to show the feasibility of codifying architecture declaratively in general the practical usability is not reached yet because of missing performance optimisations. Since the entities as well as the relations are always extracted from the source on-the-fly there is a lack of efficiency. Nevertheless the elegance of codifying architectural descriptions with a declarative formalism is shown. What this approach does not provide are facilities to get a simple overview of the system and to find particular and exceptional entities without knowledge of the domain system.

2.4.3 Recovering Behavioral Design Views

The declarative way of defining software architecture described in the previous section is taken up in the work about Recovering Behaviour Design Views [RICH 02]. The model is built up of static and dynamic aspects of an object-oriented application in terms of logic facts. Both static and dynamic information are stored in a single logic database. On top of these data the approach provides two complementary views, the *concept view recovery* and the *collaboration view recovery*.

With the *concept view recovery* views are extracted showing binary relation between components. Concept view recovery proposes an extensible framework for defining perspectives in terms of components and connectors to obtain a range of views of an application. The components and connectors are defined declaratively using a logic-programming language. A view is the result of applying a perspective to a source model.

The *collaboration view recovery* extracts finer-grained views of collaborations and the roles classes play in them. This kind of views aid for the understanding what is

happening on the level of interacting objects to carry out a certain functionality. There are three basic kinds of operations: querying the current base of dynamic information, editing the base of dynamic information through filtering out information or loading a collaboration instance, and displaying interaction diagrams.

This approach remains on a relatively low level of abstraction and roles of particular artifacts can be extracted properly. To extract information on a higher level the precondition for the user is to have a detailed knowledge about the system.

2.5 An Approach for Visualizing Behavior between Classes

Another approach to visualize behavior in Object-Oriented System is presented by Wim de Pauw [PAUW 93]. This approach does not rely on a high-level model and does just visualize collaborations between the low level entities, the classes. Besides cluster-views and histogram-views an inter-class call matrix is used which on the one hand makes visible particular classes as well as giving a useful overview of a system in respect of collaboration.

2.6 Discussion

Even if the approaches extracting models from collaboration provide a wide spectrum of facilities how to extract roles and architectural information, there are some weak points which are not solved yet:

1. Detailed knowledge about the domain system is required to build a useful high-level model.
2. There is only the level of collaboration taken into account between the high-level entities but there is no link to the level below, the collaboration between classes.

Even if the approach of Visualizing Behaviour (section 2.5) does not rely on a high-level model and does not scale up, it provides some facilities which the approaches relying on a high-level model described above lack:

1. It gives an overview of the system.

2. There is no detailed knowledge of the system required by the user.

Three important points which are valuable to find a solution for the problems mentioned above are:

1. To keep scalability it is useful to rely on a high-level model.
2. The ideas of the work of W. de Pauw (section 2.5), especially those of the inter-class call matrix, can help considerably to get a valuable overview of a system.
3. It is helpful to add user-interaction to the views to achieve links between high-level entities and the classes as well as between the collaborations in the view and the collaborations in the source.

2.7 Our Approach

In this thesis we present an approach to build high-level models relying on collaboration information between subsystems and classes.

2.7.1 The Goals

We settle our goals in the context of design recovery as follows:

- Gain an overview of the system in terms of collaboration between subsystems and between classes.
- Identify the core subsystems and the core classes in terms of collaboration.
- Identify the exceptional subsystems and classes in terms of collaboration.
- Locate unused subsystems and unused code by comparing application scenarios and application structure.

2.7.2 Our Solution

The main points in our solution are as follows:

- The visualisation relies on a high-level model to get scalability.
- Collaboration models are visualized using collaboration-views that are able to display different levels of detail. The collaboration-views provide interactive facilities to inspect the collaborations and provide links to the source.
- Mechanisms for generating initial high-level models without detailed knowledge about the domain system are provided.

To validate the approach we build a framework called the *Collaboration Sniffer* (see Chapter 4.3). Our high-level model is built on top of *MOOSE* (4.1), an implementation of the FAMIX-Metamodel. For the views we implement extensions of *CodeCrawler* (see section 4.2) a tool for software visualisation. To extract collaboration based on static information we use *SOUL* (see section 6.1.2), a reflective logic language that is fully integrated in VisualWorks, to reason about Smalltalk-code.

2.8 Conclusion

In this chapter we introduced the terms *Reengineering*, *Reverse Engineering* and *Design Recovery*. Looking at different existing approaches in the domain of design recovery we focused on important aspects of how high-level models can be defined and on which information they are based.

In the next chapter we present our approach which integrates the facilities of getting an overview of a system based on collaboration taking concepts of Wim de Pauw [PAUW 93] in account as well as different concepts of the approaches relying on high-level models.

Chapter 3

Collaboration-Views

Our approach of software visualisation is built on three different concepts: we introduce *abstractions* using high-level models, we focus on the *behaviour of objects* by extracting collaboration and we use *views* for visualisation.

In this chapter these concepts are presented in detail. The high-level model consisting of entities and relations is defined. The two crucial requirements for the high-level model are: First, to take high-level information in account without loosing the details, and second, to provide mechanisms to generate an initial high-level model without detailed knowledge of the domain system.

We present visualisations of the collaboration models. Therefore we define two different views. The first view, the *browsing-view*, is a kind of a 'box and arrow sketch' enhanced with facilities to dive in lower levels. The boxes and arrows are linked interactively to provide more detailed information. The second view, the *collaboration-matrix*, relies on the inter-class call matrix of W. de Pauw and is enhanced by interactivity and facilities to represent high-level models.

3.1 Defining High-Level Models

The high-level model we define consists of entities representing one or more software artifacts (classes) and relations between those entities that represent different types of collaboration between the basic software artifacts. In the following sections we define the entity groups and the relations in more detail.

3.1.1 Defining Entity Groups

The existing approaches described in Section 2.4 [RICH 99][MURP 95] only take one level of abstraction into account. A high-level entity is a collection of software artifacts that are structural elements of a programming language like classes, methods and attributes. With the flat architecture focussing on the collaborations between the high-level entities the information about collaborations between the software artifacts are lost.

We overcome this loss of detailed information by introducing a hierarchical model. We define two types of entities, a HLE (High-Level Entity) and a LE (Leaf-Entity). A HLE can have one or more HLE's or LE's. In contrast, every LE relates to exactly one software artifact in the system (see Figure 3.1). While our current approach only considers classes there is no reason to limit it to this kind of software artifact.

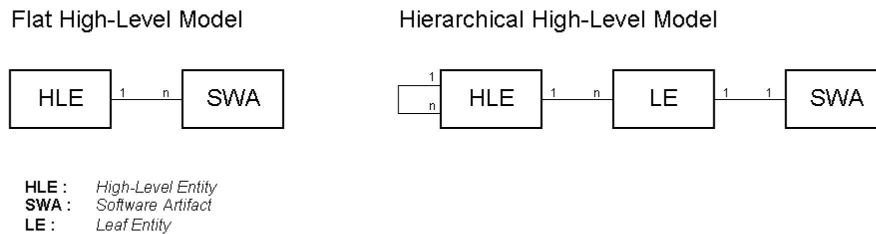

Figure 3.1: High-Level Models

A high-level model is built by creating root HLE's used as named containers. Root HLE's represent the components and subsystems of the software system. To add software artifacts (classes) to a component it has to be wrapped by LE's which can be added to the HLE. The process of building an initial high-level model manually by creating HLE's as groups and adding classes wrapped by LE's to the HLE's does not scale up and requires knowledge about the system. An engineer has to know already which software artifacts belong to a group and has to iterate over all groups. For reasons of scalability and simplicity, facilities to generate initial high-level models without any detailed knowledge about the domain system are essential. We introduce several automatic grouping mechanisms to help overcome the problem of building an initial model:

- Object-oriented languages typically have structural entities like packages in Smalltalk VisualWorks or Java. The simplest way to get an initial high-level model is to

map those existing structures to *HLE's*. We introduce automatic grouping for packages of VisualWorks Smalltalk. The *package grouping mechanism* generates *HLE's* for the packages consisting of its classes as *LE's*.

- As a second automatic grouping mechanism we introduce the *inheritance tree grouping*. This grouping mechanism generates *HLE's* for every hierarchy tree with all the classes of the hierarchy tree as its *LE's*.

- Addionally we add an interface to use the grouping mechanism of *Classifier* [TALE 03], a tool for group generation and visualisation.

Combining the hierarchical-model approach with the possibility to arbitrary add grouping mechanisms, we get a scalable and flexible model with which we can represent even large systems.

3.1.2 Defining the Relations

In designing object oriented applications, the importance of modeling how objects cooperate to achieve a specific task is well recognized [WILD 93]. We therefore define our relations between the group-entities as collaboration between classes. We take collaborations which are extracted from static information as well as information which is collected at runtime into account.

Collaboration based on static information

For the collaboration extracted from static information we define two categories of collaboration: collaboration based on instance variables and collaboration based on temporary variables or arguments.

Instance variables represent the state of an object. We define collaboration based on instance variables as follows: if class *A* has an instance variable *a* of Type *B*, we say that 'A uses B'. The useage of an instance variable expresses that an object uses another object not just as an information provider at a particular point in time. Therefore we value the use of instance variable as a strong indication of collaboration between objects.

Temporary variables and arguments store values and references only during an execution of a particular part of code. Even if local variables and arguments can change

the state of an object, compared to instance variables the local variables are not part of the state of an object. We define collaboration based on local variables as follows: If class *A* has a method *m1* which has a local variable of Type *B* we say that 'A uses B locally'. The use of a local variable expresses that an object uses another object as an information provider just locally to a certain method and constitutes a collaboration between objects. In general, collaboration based on local variable is less strong than collaboration based on instance variables but more frequent.

Collaboration based on dynamic information

Collaboration based on dynamic information is defined as message-sends at runtime from one object to another. While collaboration based on static information can be obsolete since there does eventally not exist any runtime-scenario which it is involved in, collaboration based on dynamic information is by definition part of a real scenario.

Note that semantically two different classes are receivers of a message-send, the class of the instance on which the method is called on and the class where the method is defined in. If for instance a message *x* is sent to an object of Type *A* which inherits from class *B* and the method *x* is defined in *B*, both of those classes are receivers of the method. To capture this difference we define two different types of collaboration at runtime.

Runtime Collaboration at Method-Level represents the collaboration between the class of the object sending the message and the class where the method is defined in.

Runtime Collaboration at Instance-Level represents the collaboration between the class of the object sending the message and the class of the instance receiving the message.

3.2 Visualisation

CodeCrawler 4.2 is used to display the resulting high-level model graphically. Two different approaches visualize the high-level model. The *browsing-view* allows the user to dive inside the nodes (group entities) and the *collaboration-matrix* gives an overview of the system, highlights "bad" collaborations and aids in the process of component detection.

3.2.1 Browsing-View

Typically architectures are visualized using boxes for instances, connected by arrows representing relationships. Engineers commonly use "box-and-arrow" sketches to communicate software design and reason about high-level models. Such sketches are also used by tools to visualize architectures [RICH 02] [MURP 95].

In our browsing-view (see Figure 3.2), boxes represent the entities and arrows represent the collaborations. For every collaboration type we use a different color, so that they can be compared instantly.

Even if sketches with boxes and arrows are very intuitive and accurately represent high-level architectures, the information they provide is limited.

- Why an arrow between two high-level entities does exist is not shown.
- Only the inter-collaboration of a high-level entity is shown, but not the inside-collaboration.

To overcome these limits, we enhance the view with additional facilities and user-interaction:

- To make details of a collaboration-arrow visible we add a tip-view which pops up when the mouse is moved over and shows which entities are concerned by the collaboration.
- To get more details, the selected collaboration can be inspected. Following the principle of information on demand, superfluousness of information is avoided and permits us to add more useful information to the view without raising the complexity for the user.
- To make the *degree* of inside-collaboration of a high-level entity visible, we colorize the entities according to the amount of inside-collaboration. A box with a white background indicates that there is no collaboration inside of the entity. The darker the background of a box the higher the collaboration inside. To visualize the inner collaboration of an entity, we add the facility to dive inside an entity. This opens a browsing-view with all the child-entities of the selected entity and shows all the collaborations on that level.

We define the basic and general characteristics which can be extracted by browsing the graph-view as follows:

Strong Outside-Collaboration. This is the most general and intuitive characteristic of sketches with boxes and arrows. If a box has plenty of arrows as connectors to other boxes the item which is represented by the box has strong outside-collaboration.

Strong Inside-Collaboration. If a box in our view is dark colored this is equivalent to the fact that there is a lot of collaboration between the child-entities of the concerned item.

Core Component. We define the term core component in our context for an entity which has strong outside-collaboration as well as strong inside-collaboration.

The ability to browse through the hierarchy of collaborations by diving inside entities and the possibility to inspect the collaboration edges by moving over them improves the process of understanding a system.

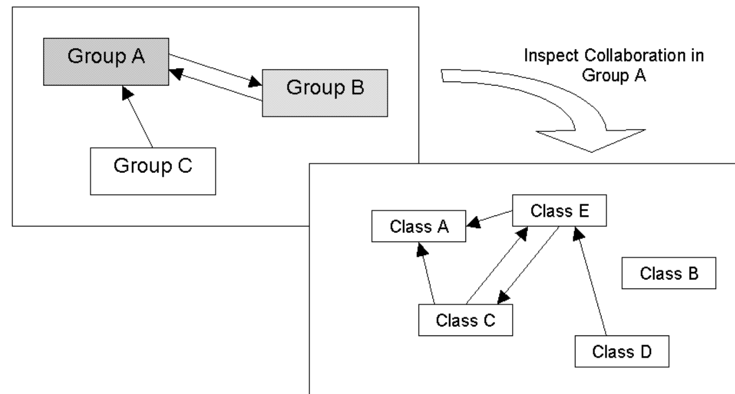

Figure 3.2: Browsing-View

3.2.2 Collaboration-Matrix

While the browsing-view is useful for browsing through collaborations on different levels inspecting their properties. The second view we introduce, the collaboration-matrix, provides an overview of collaborations on different levels of granularity and provides facilities to detect patterns of collaborations and “strange” entities with respect to collaborations.

The collaboration-matrix combines different levels of granularity in one single view. Collaboration at the level of classes and the level of groups of classes is displayed in the same view.

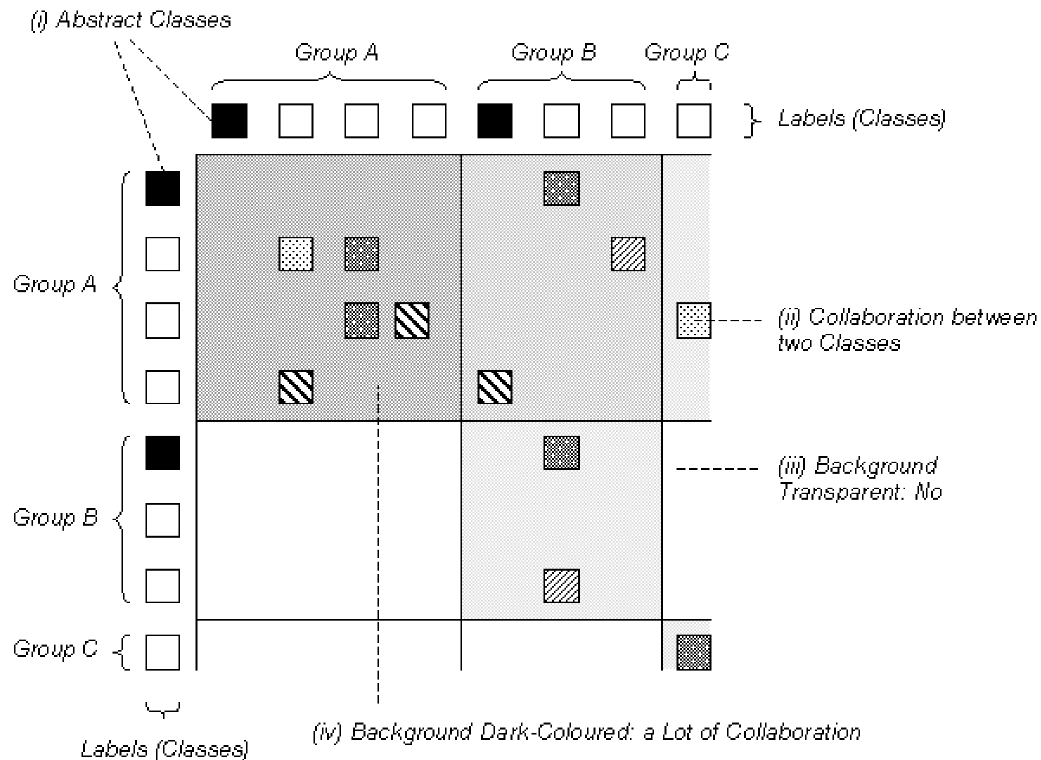

Figure 3.3: Collaboration Matrix

The collaboration-matrix is organized as follows: the classes are listed at the left side and the top side of the view (see “Labels” in Figure 3.3) in the same order. The class lists are classified according to the high-level entities they belong to. In the class-lists, the background for the classes is in general transparent, except for the abstract classes which have a black background color (see (i) in Figure 3.3).

To visualize collaborations between two classes, the point of intersection in the matrix is colorized (see (ii) in Figure 3.3). The *collaboration-points* colorized in the matrix give an overview of the basic collaboration. The color of the *collaboration-points* differ according to the type of collaboration.

The collaboration between the high-level entities is displayed by *collaboration-fields* in the matrix. A *collaboration-field* between two high-level entities is the area which covers all the possible *collaboration-points* between the classes of the involved high-level entity. The intensity of the collaboration between two high-level entities is the sum of all *collaboration-points* inside the *collaboration-field* and is expressed by the background color of the field in the matrix. If there is no collaboration marked inside a field the background color of a field remains transparent (see (iii) in Figure 3.3). The more collaboration inside a collaboration-field the darker the background color gets (see (iv) in Figure 3.3).

To link to the details of a collaboration, a *collaboration-point* can be inspected. This shows the details based on which the affected collaboration is extracted and completes the range of different levels of granularity which spans from the overview represented by the high-level entities to the source.

Comparing Application Scenarios with Application Structure

Besides detecting important classes, another application of the collaboration-matrix is to compare application scenarios with the application structure. Therefore we execute an application scenario and split up the system in two parts: those entities which are affected by the scenario executed and those which are not. The two parts provide two different kinds of information used for different aspects in reverse engineering:

Used Structural Entities. Combining the knowledge of a specific application scenario with the information about the structural entites in that scenario enhances the understanding of a system.

Unused Structural Entities. To have the relation between an application scenario and the unused structural entities enables us to find LE's (Classes) or even HLE's (Components) which are unused and can be removed.

There is one constraint which is not taken into account in this approach of finding *unused structural entities*. If an entity A is a subclass of an entity B, A can not be removed even if there is no runtime collaboration marked between the two entities.

3.2.3 Collaboration Patterns

We define patterns which are useful to extract important characteristics of the collaboration-matrix. There are different patterns for the component and the class level.

Collaboration Patterns of High-Level Entities

By focusing on the level of high-level entities of the collaboration-matrix we define six visual patterns (P1-P6) which are useful to extract important characteristics. The patterns to detect layered/non-layered architecture (P1/P2) and the patterns to detect cyclic/non-cyclic collaboration (P3/P4) are dependent on specific sort sequences of the high-level entities in the collaboration-matrix.

The following two patterns visualize wheter a system follows a clear layered architecture and shows possible failings of the layered structure. To detect these patterns the sort sequence of the high-level entities (layers) has to be according to the levels of the layers in the layered structure: The basic layer has to be the first high-level entity in the collaboration-matrix, the second layer the second entity and so on for all the layers.

(P1) Layered Structure. If there is only collaboration in fields on the diagonal and those fields directly above the diagonal the structure is clean layered (see Figure 3.4).

(P2) Non-Layered Structure. Collaborations outside the fields of the diagonal and directly above the diagonal indicate that the structure is not clean layered.

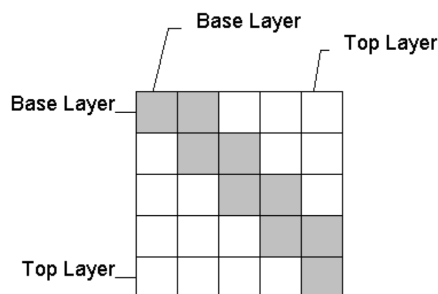

Figure 3.4: Clean Layered Structure

The following two patterns visualize wheter a there are cyclic collaborations between components. To detect these patterns the sort sequence of the high-level entities (components) has to be according to the collaboration-dependencies: The basic component which has no collaboration-dependency to any other component has to be the first high-level entity in the collaboration-matrix, the component which has only collaboration-dependencies to the first component has to be the second entity and so on for all the components.

(P3) Non-Cyclic Collaboration. If there is no collaboration in fields below the diagonal, then there is no cyclic collaboration between two high-level entities. It indicates a clean non-cyclic architecture.

(P4) Cyclic Collaboration. If there is collaboration in fields below the diagonal then there is a possible cyclic collaboration between two entities. It indicates “bad” dependencies between high-level entities.

The following two patterns are independent of the sort sequence of the high-level entities in collaboration-matrix. They visualize strong collaboration between two entities and inside an entity.

(P5) Dark-Colored Diagonal Field. The square-fields on the diagonal represent the collaboration inside a group. To have a dark-colored field on the diagonal means that there is strong collaboration of the entities inside the high-level entity.

(P6) Dark-Colored Non-Diagonal Field. A dark field outside the diagonal indicates strong collaboration between two different high-level entities.

Class Level Collaboration Patterns

Similarly to patterns defined on high-level entities we define patterns on the level of classes. These patterns represent characteristics of a class itself or in respect to the system.

(P7) Collaboration-Item on the Diagonal. Collaboration items on the diagonal indicate that a class collaborates with itself. In general, if a class collaborates with itself at runtime means that an object of this element is instantiated. There are certain cases where a class is instantiated without collaboration with itself.

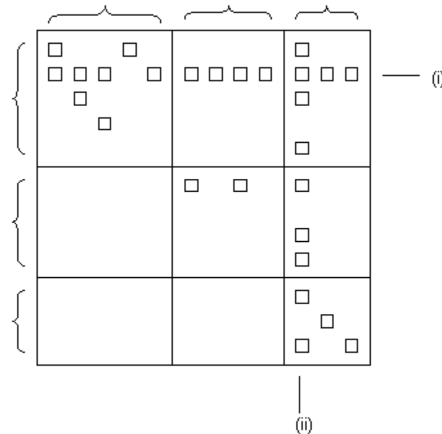

Figure 3.5: Collaboration Patterns

(P8) Collaboration-Items on a horizontal line. Multiple collaboration-items on a horizontal line indicate that the considered class is used by a lot of classes. The class collaborates with many classes (see (i) in Figure 3.5).

(P9) Collaboration-Items on a vertical line. Multiple collaboration-items on a vertical line indicate that the class uses a lot of other classes. The class is heavily dependent (see (ii) in Figure 3.5).

To summarize, the collaboration-matrix provides the following facilities:

- We get a high-level of abstraction by displaying collaborations between high-level entities.
- We can check the architecture for possible layered structure.
- The view visualises collaboration dependencies on the level of high-level entities.
- Different levels of collaboration are displayed in a single view.
- We get an intutitive link from the high-level entities to the source code.
- We get a scalable view.
- We get facilities to detect important classes and core components.

- We get facilities to detect strange classes.

Chapter 4

Tool Support

To validate the applicability and the usability of our approach we implemented a framework called *Collaboration Sniffer*. The basic components used in this framework are MOOSE, a language independent environment for reengineering of object-oriented systems (see section 4.1) and CodeCrawler, a flexible and extensible software visualization tool (see section 4.2). Besides those existing components we implemented operators to extract collaboration information, a tool to generate high-level models and views as extensions to CodeCrawler.

In this chapter we introduce the reengineering environment basically consisting of MOOSE and CodeCrawler and describe the architecture of the *Collaboration Sniffer* framework.

4.1 MOOSE: using Software Models

MOOSE is a tool environment to reengineer and reverse engineer object-oriented systems. It consists of a repository to store models of software systems. A model itself consists of software artifacts such as classes, methods, attributes etc. and relations between those artifacts.

While for certain tasks like detection of code duplication [DUCA 99] the source code is essential, in some cases performing the reengineering task on an abstract level has major advantages:

- To apply the reengineering tasks on an abstract software model encapsulates the

language specific details.

- It enables us to concentrate on general object-oriented issues and at the same time preserves language independency as far this is provided by the abstract software model.

Using MOOSE we get the ability to reason about a software system on a level of abstraction suitable to our requirements.

The following characteristics of MOOSE are important for our approach:

Language independency. It supports different object-oriented languages which means that the applications using it are language independent.

Extensibility. The model is not restricted and can be extended with new entities and special-purpose relationships.

Scalability. The model can deal with huge legacy systems consisting of millions of lines of code.

MOOSE itself is an implementation of a meta model called FAMIX [DEME 01]. The main goal of FAMIX is to support reengineering activities in a language-independent way [TICH 01]. The entities of FAMIX represent source-code artifacts of object-oriented languages. A *language-independent core* part is defined, to provide reusability of activities between different languages without changes, while the *language extension* part defines certain language specific extensions.

4.2 CodeCrawler: a Software Visualization Tool

CodeCrawler is a language independent software visualisation tool written in Smalltalk. CodeCrawler supports reverse engineering through the combination of metrics and software visualisation. Besides its own coarse-grained, fine-grained and evolutionary software visualisations it provides facilities to define additional view specifications [LANZ 03b].

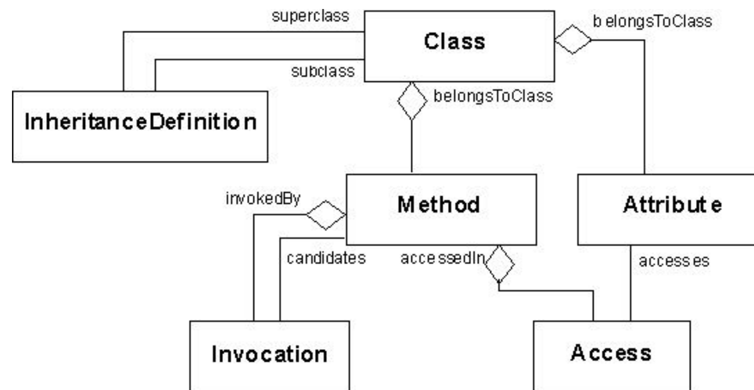

Figure 4.1: Famix Core

4.3 Collaboration Sniffer

The framework called *Collaboration Sniffer* and its tools are implemented in VisualWorks 7.0. In general the framework can be split into three parts:

Extraction and storage of collaboration. The extraction is implemented by *Type Annotators* and *Interaction Application* and the information is stored by *Information Repositories*. We look at those tools in Section 4.3.2.

Building of hierarchical groups. The grouping of software artifacts is implemented with the *Group Generator* described in Section 4.3.3.

Visual representation of the model. The high-level models are visualized by extensions of CodeCrawler 4.2.

4.3.1 Architecture of Collaboration Sniffer

Basically *Collaboration Sniffer* consists of several existing components integrated with new components or extensions of existing components. Figure 4.2 shows a sketch of the architecture by displaying the main data flow between the single components.

The basic component is MOOSE (see Section 4.1). While MOOSE itself extracts the source model, collaboration-dependencies are added to MOOSE by running *Type-Operators* on the source or by running the *Interaction Application* which collects

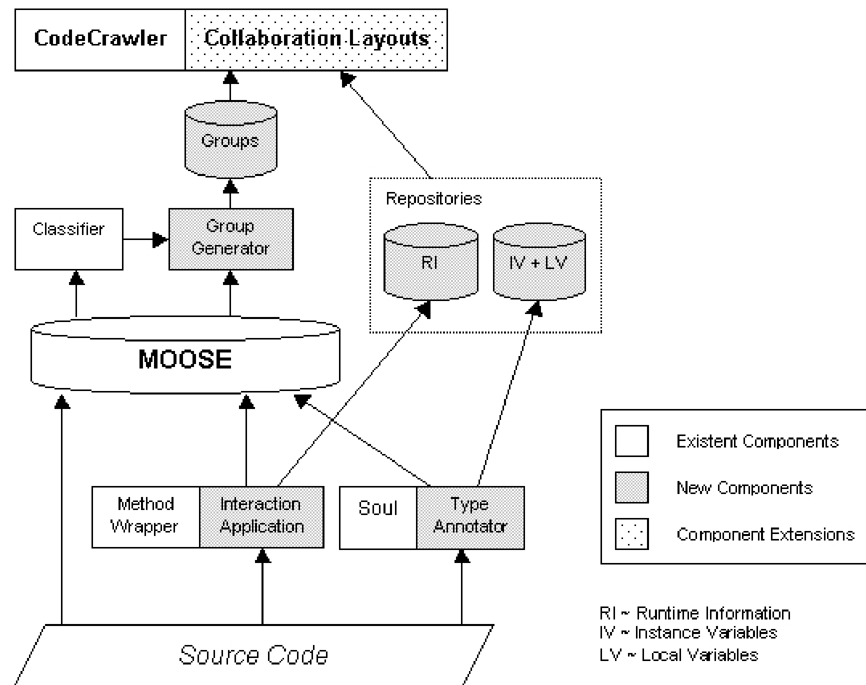

Figure 4.2: Architecture of *Collaboration Sniffer*

information at runtime of an application and adds this information as collaboration-dependencies to MOOSE. While the operators just add the pure dependency information to MOOSE, more detailed information about the extracted dependencies can be stored into the *Information Repositories*. This additional information is not used to visualize collaborations but is used to show why a certain collaboration is extracted. For the static collaboration it is the message sends or the assignments to a local variable or an instance variable while for the dynamic information it is the message sends from one class to another. For reasons of efficiency the collection of this information can be abandoned.

The software model provided by MOOSE is enhanced by collaboration-dependencies. It is then used by the *Group-Generator*, a Tool to manage and store the grouping of the software artifacts. It provides elementary operations to create and delete groups and to add and remove elements. Furthermore it provides operations to create automatically the groups based on the packages in Smalltalk. To generate the groups directly out of MOOSE there is the possibility to use *Classifier* 3.1.1, a tool to build groups by logic

queries, which is also based on MOOSE.

To visualize the high-level model two different views 3.2 as an extension to *CodeCrawler* 4.2 are used. These views are based on the *Group-Repository* to make the high-level model visible and uses the *Information Repositories* to inspect a single collaboration and to browse its detailed information.

4.3.2 Information Repositories

To reason about collaborations, different levels of detail must be taken in account. The first level of collaborations between classes is the simple ‘use’-collaboration between two entities. The second level of additional information is the type of collaboration. The third level of detail is the information about how the collaboration is extracted.

The first level of detail provides already enough information to show a useful overview of a system. At the second level of detail, adding information about the type of collaboration does not raise complexity but improves the value of the collaboration model. Therefore we do not differentiate between the first two levels of detail.

The *Interaction Application* and the *Type Annotator* add the first two level of information directly to the software model in MOOSE. By running the applications for extracting collaboration information the options can be set wheter the information of the third level of detail should be extracted and stored in separate repositories or not. The reason to store this information in separate repositories is because it is not used to build the collaboration model itself. It is just used for an interactive browsing by the user, to provide the facility to inspect a single collaboration and to inspect the ‘background’-information of a collaboration. To leave out the third level of detail does not affect the collaboration model but the possibility to browse and inspect the details of collaborations is lost.

4.3.3 Representation of the High-Level Model

The High-Level Model is represented by the *Group Repository*. This repository consists of named groups which consist of a collection of base entities, the MOOSE classes. The user interface for this repository is called *Group Generator* and provides facilities to build arbitrary groups out of the classes in MOOSE. Besides the arbitrary building of groups there are facilities to build groups automatically by taking package information or class-hierarchy information into account. An other very powerful

enhancement is the facility to import groups of the *Classifier*.

The *Group Generator* also calculates collaboration between groups. Other features integrated in the *Group Generator* are the *Attribute Viewer* which shows the extracted types of instance and local variables.

Chapter 5

Validation: Case Studies

To validate our claim that collaboration views provide useful information on different levels of granularity, we choose to apply our approach to systems of different size and complexity.

The systems we apply the layouts on are CodeCrawler and Jun. CodeCrawler is a small system but highly complex. The problem in the understanding of such a system therefore lies in its complexity, not its size.

We take the average-sized system Jun as a case study to make statements about scalability and to focus on aspects our approach provides on a completely unknown system.

For the experiments we extract the collaborations and generate the collaboration-matrix. We try to get an overview and look for the collaboration-patterns (defined in 3.2.3) at component and class level to find the important and extraordinary entities.

5.1 Collaboration inside CodeCrawler

CodeCrawler v.4.274 consists of 76 classes with almost 6000 lines of code. Although it is a relatively small system, its complexity can be classified as quite high (see Table 5.1). The reason for this fact is that CodeCrawler has passed frequent and fundamental refactorings.

The 152 classes are divided into 6 structural packages. By classifying the classes into these packages we get a higher level structure which is explicit in the source. To take this explicit structural information in account simplifies the process of building a high-

level model. Taking a one-to-one mapping from packages to higher level entities gives an appropriate model for reasoning about the system. Therefore the high-level model consists of 76 base entities (the class and its metaclass is always taken as one base entity) grouped in 6 higher-level entities, the packages.

To validate the approach on CodeCrawler the following steps have to be done:

- The source code of CodeCrawler is loaded into MOOSE as a software-model.
- The Type-Annotator is run to extract static collaboration information.
- The collaborations based on dynamic information are extracted by applying the unit tests as test scenarios provided by CodeCrawler itself. Using these scenarios we get reproducible information and cover the main part of the system.
- The high-level model is built and an order set to the HLE's. The order we apply is to have the least possible collaboration in the rectangles below the diagonal (see the collaboration-patterns in Section 3.2.3).
- To get an overview of the system, we apply a collaboration-matrix consisting of all types of collaboration.

Metric	value
Number of Classes	152
Number of Methods	1175
Number of Attributes	110
Number of Typed Attributes	62
Number of Local Variables	744
Number of Typed Local Variables	368
Number of extracted collaboration at instance level	195
Number of extracted collaboration at method level	147
Lines of Code	5926

Table 5.1: CodeCrawler

Having this coarse overview, we apply every collaboration type separately, to collect collaboration patterns. We filter the extracted collaborations and look for positive false by browsing the source code manually. Finally, we summarize specific aspects and apply the patterns which supports the understanding of the system.

5.1.1 Coarse Overview

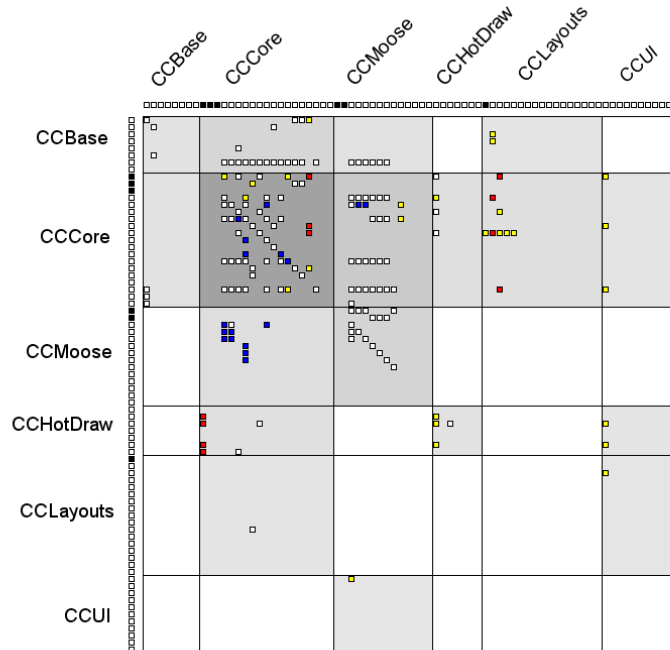

Figure 5.1: Collaboration-Matrix of collaboration in CodeCrawler version 4.274 by runtime, local variables and instance variables

The collaboration-matrix applied on the high-level model of CodeCrawler gives an overview of the complete system and allows us to make some general statements about the system (see Figure 5.1). A more detailed discussion about local collaborations is given in the following sections where the different collaboration types are analyzed separately. One of the most eye-catching things is that one group, the *CCCore* group is the one with the most inside- as well as outside-collaboration. Especially, *CCCore* is used by every other group of the model. The second high-level entity which plays an important role with respect to collaboration is the group *CCMoose*. Even if the outside-collaboration is restricted to the entities 'CCCore' and *CCBase* it is of a high intensity and additionally shows a strong inside-collaboration as well.

Looking at the collaboration types in detail we focus on two kinds of items. The first kind are those below the diagonal, expressing a sort of cyclic collaboration. The

second sort of items we are interested in are those lying on a horizontal or vertical line, which expresses that they are used a lot (horizontal row), or use a lot of other items (vertical column).

5.1.2 Collaboration by Local Variables

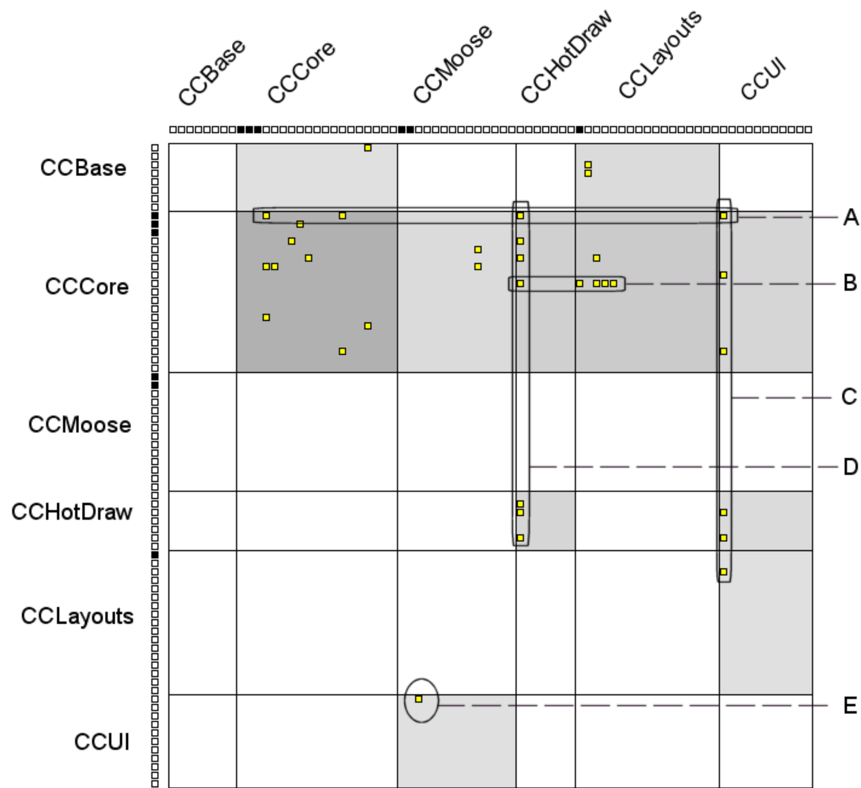

Figure 5.2: Collaboration-Matrix of collaboration in CodeCrawler version 4.274 by local variables

Looking at the collaboration by local variables we observe that just one single item lies below the diagonal and expresses a cyclic dependency in respect to our high-level model. The class which is represented by the entity is *CCFAMIXClassNodePlugin* (E in Figure 5.2) and the responsible method for this extracted collaboration is *spawn-*

ClassBlueprint. Looking at the method itself shows that the extracted collaboration is not a positive false. The class *CCFAMIXClassNodePlugin* of the *CCMoose* package really accesses the class *CodeCrawler* from the *CCUI* package using a local variable. While this collaboration dependency is not necessarily problematic, it should be brought to the attention of an engineer.

The other kind of items, lying on a horizontal or vertical line, do not show exceptional collaborations but help to reveal important classes. We look at four classes in more detail. Two of them because they seem to use a lot of classes, and the other two because they seem to be heavily used by other classes.

CCDrawing is a class of the package *CCHotDraw*. The collaboration layout indicates that the class uses seven classes (D in Figure 5.2) while four of them are false positives. The messages sent to the variable being the indicator for these collaborations are only very general messages like 'isEmpty' or 'width' and 'height'. Looking at the source code confirms that the classes shown do only match this general interface but are not the used classes.

The three classes which seems to really be used by *CCDrawing* are *CCEdgeFigureModel*, *CCNodeFigureModel* and *CCItemFigureModel*. The collaboration of those classes indicates a collaboration transferred to the higher level which is that the package *CCHotDraw* uses the package *CCCore*.

CodeCrawler is a class of the package *CCUI* and plays a central role, as can already seen from its name. It uses the six classes *CCEmbededSpringLayout*, *CCRectangleFigure*, *CCNamedFigure*, *CCItemPlugin*, *CCItemFigureModel* and *CCViewSpec* (C in Figure 5.2). There are no false positives among those classes and as they come from the different packages *CCCore*, *CCHotDraw* and *CCLayouts* it supports our assumption that this class plays a central role.

CCNodeFigureModel is a class of the package *CCCore* and is used by five other classes (B in Figure 5.2). Four of them are Layout classes from the package *CCLayouts* and the fifth is the class *CCDrawing*. This class is the main point of communication between the package *CCCore* and *CCDrawing*.

CCItemFigureModel is a class of the package *CCCore*. There are four classes which are using this class: *CCGraph* and *CCItem* from package *CCCore*,*CodeCrawler* from package *CCUI* and *CCDrawing* from package *CCHotDraw*. So this class seems to be important being used in most of the packages.

Looking at the collaboration based on local variables we focused on four classes which seems to play important roles using other or used by other classes and we extracted a collaboration which is not corresponding to the expected dependency model.

5.1.3 Collaboration by Instance Variables

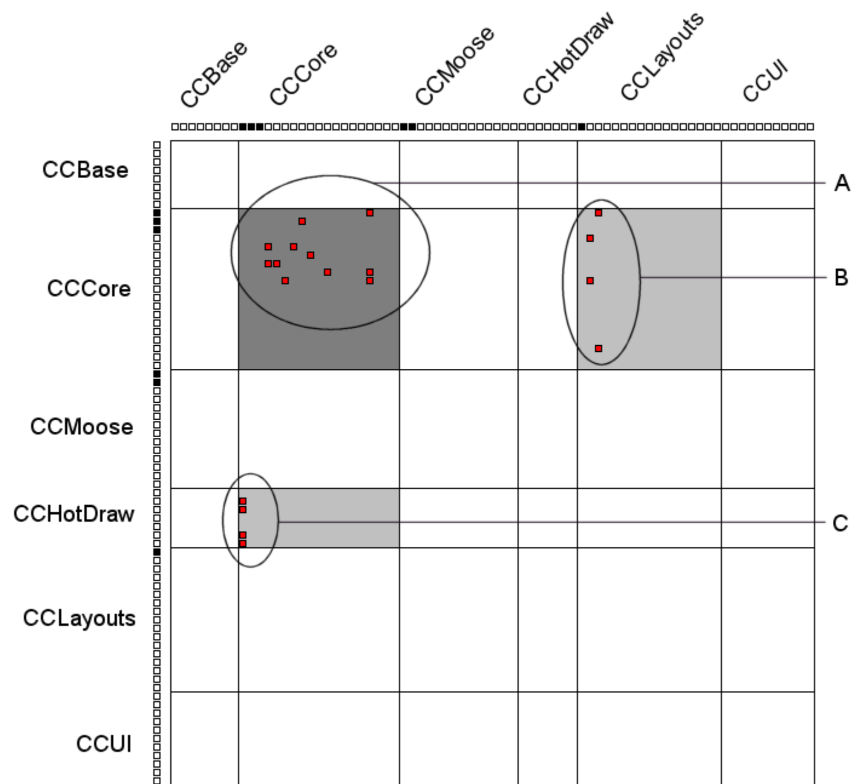

Figure 5.3: Collaboration-Matrix of collaboration in CodeCrawler version 4.274 by instance variables

The most surprising thing looking at collaboration by instance variables is that there are only very few packages affected by this type of collaboration. The collaborations which correspond to the dependencies in our high-level model are collaborations inside the package *CCCore* (A in Figure 5.3) and collaborations of classes in the package

CCLayouts using classes in the package *CCCore* (B in Figure 5.3). Not corresponding to the dependency of our high-level model is the collaboration of the class *CCItemFigureModel* using different figure classes of the package *CCHotDraw* (C in Figure 5.3). The explanation of this collaboration is found in the fact that the class *CCItemFigureModel* uses the interface of *Figure*, a class of HotDraw, which is not part of our high-level model and that all classes in CCHotDraw which inherits from this class are used by the class *CCItemFigureModel* even if there is no direct use defined. In this sense it is not a class use but a use of an equal interface which is not contradictory to our defined dependencies.

To summarize, the instance variable collaboration-view shows very few collaborations. The package *CCLayouts* uses *CCCore* which itself uses the package *CCHotDraw* indirectly. As the use by instance variables indicates in general a stronger and longer live span than this of local variables, we denote these dependencies as strong.

5.1.4 Collaboration at Runtime (Method Level)

Exploring the collaboration at runtime on the method-level we focus on entities below the diagonal and on entities which seem to be important because they use a lot of other classes or are used by a lot of classes.

Two of the entities below the diagonal can be identified as the classes *CCEdgeFigureModel* and *CCNodeFigureModel*. Both of them are subclasses of *CCItemFigureModel* which uses some other classes of the package *CCCore* by instance variable (I in Figure 5.4). This collaboration between *CCItemFigureModel* and classes of package *CCCore* was already shown in Section 5.1.3. This example of collaboration at runtime can be seen as a validation of collaboration extracted using static information.

Another kind of collaboration which - at the first glance - contradicts to the defined dependency of the high-level model is found. For example the class *CCNode* of the package *CCCore* sends a method-call which is defined in the abstract class *CCNodePlugin* of the same package to a subclass of this abstract class, *CCFamixClassNodePlugin*, defined in the package *CCMoose* (H in Figure 5.4). As there are only method-calls to the abstract interface, this is not contradictionary to the defined dependencies. The same kind of collaboration is found between the packages *CCCore* and *CCLayouts* where the class *CCViewSpec* uses the interface of an abstract superclass of *CCQuadraticLayouts* (J in Figure 5.4).

The third occurrence of mutual collaboration is detected between the packages *CCBase*

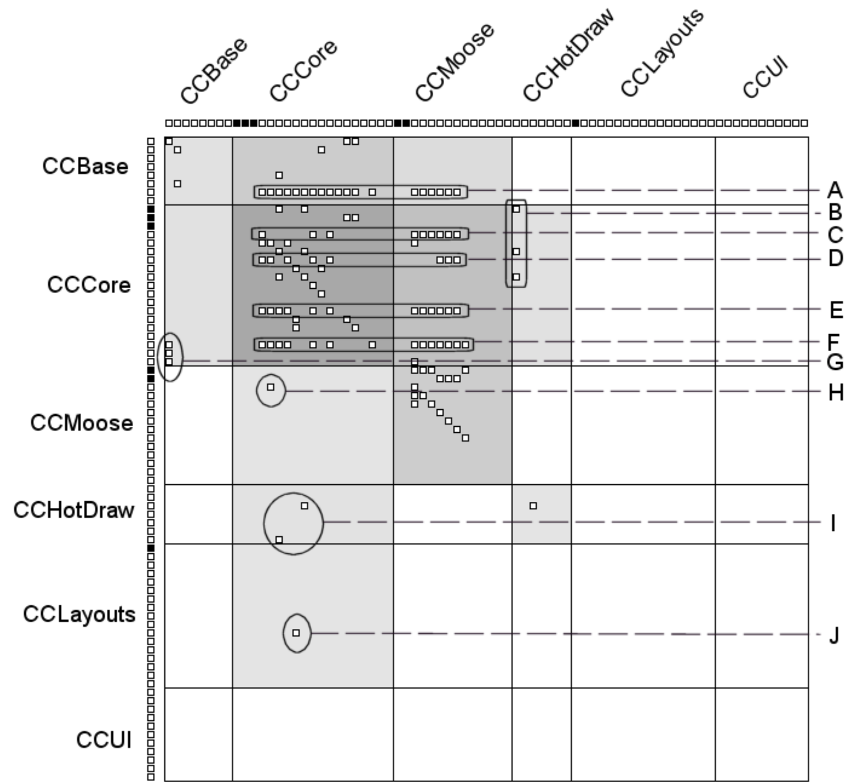

Figure 5.4: Collaboration-Matrix of collaboration in CodeCrawler version 4.274 at runtime (method level)

and *CCCore* (G in Figure 5.4). Looking at the code we can see that this dependency is not hard-coded as the method call is applied just on sub-classes of *CCRoot* which is part of *CCBase*. If there are no subclasses at all, the method-call would not be done, so it is an 'if-exists' dependency.

Additionally to the collaborations which do not correspond to the dependency model we can find some other 'important' classes, by looking for items lying on a vertical or a horizontal line.

CCRoot is a class of the package *CCBase*. The collaboration layout indicates that the class is used by 14 classes (A in Figure 5.4). Looking at the details of the

collaborations we detect that the reason of this is the message 'initialize'. That means that the 'initialize' method of *CCRoot* is not overwritten for those 14 entities and is used at runtime.

CCGraph is a class of the package *CCCore* and is used by eight other entities (C in Figure 5.4). Looking at the details shows us that this class really plays an important role in this scenario, because the messages are directly send to an instance of this class and not to subclasses of it as there are no subclasses.

CCEdge is a class of the package *CCCore*. It is used by *CCGraph* and *CCNode* sending several messages to it. Besides it is used by the subclasses of *CC-FAMIXEdgePlugin* sending the messages *fromNode:toNode:* and *plugin*.

CCItem is a class of the package *CCCore*. Besides several subclasses of *CCItemPlugin* it is used by the classes *CCGraph*, *CCEdge*, *CCNode* and *CCEdgeFigureModel*(E in Figure 5.4). They mainly send the message *plugin()* which indicates that it is used as the wrapper of the item-plugin.

CCItemPlugin is a class of the package *CCCore*. This class has the same collaboration pattern as the *CCItem* class which expresses the fact that it is used by the same classes (F in Figure 5.4). This two classes seem to be tight together very strongly.

CCDrawing is a class of the package *CCHotDraw*. It uses three other classes of the package *CCCore* which are *CCCore*, *CCEdgeFigureModel*, *CCNodeFigureModel* (B in Figure 5.4).

5.1.5 Collaboration at Runtime (Instance Level)

The collaboration-matrix showing the runtime collaboration at instance level can not tell anything about component dependencies in the source code. Figure 5.5 shows a strong cyclic dependency between the packages *CCCore* and *CCMoose*. These cyclic collaborations are based on subclass dependencies of classes in *CCMoose* which inherit from classes in *CCCore*. This impact of class inheritance on collaboration dependencies between instances is based on a complex pattern which is explained in detail in Section 6.2.

The collaboration-matrix showing collaboration at instance-level represents collaboration between instances. Looking for important entities it confirms what we extracted

of the other collaboration types. Important entities which are used (A in Figure 5.5) and using (B in Figure 5.5) as well a lot of other classes are *CCGraph*, *CCNode*, *CCEdgeFigureModel*, *CCEdge*.

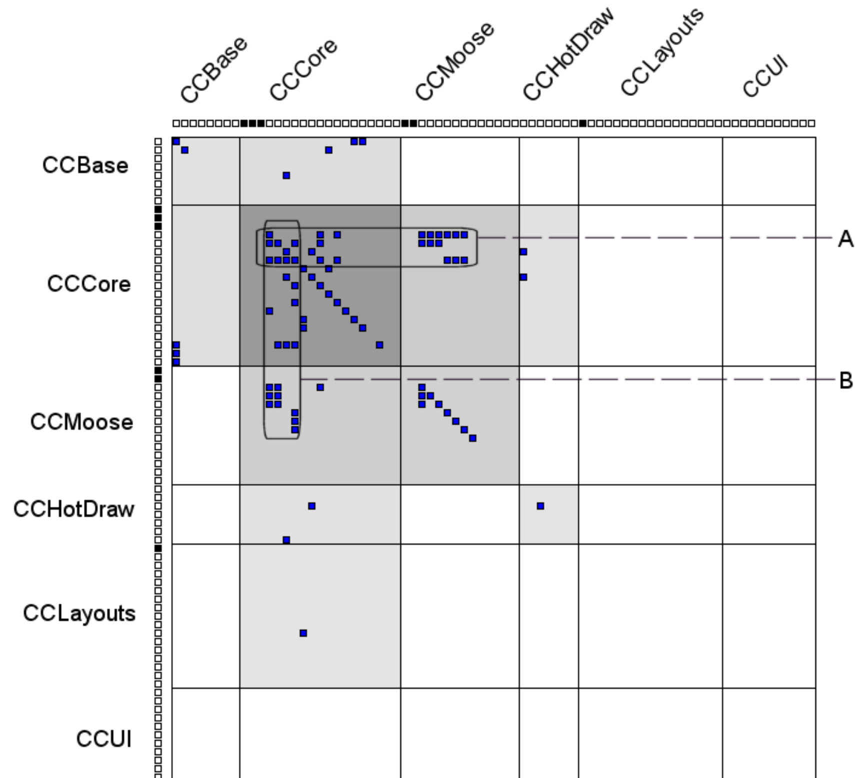

Figure 5.5: Collaboration-Matrix of collaboration in CodeCrawler version 4.274 at runtime (instance level)

5.1.6 Collaboration summarized

Applying collaboration-views on CodeCrawler we discover one core component the package 'CCCore' and a second package 'CCMoose' with a high inter-collaboration and of a high importance in respect of inside-collaboration as well. We notice a non-cyclic architecture with one single collaboration-entity really beeing in conflict with

the property of non-cyclic architecture. Additionally we notice a quite strong collaboration between the package 'CCCore' and 'CCLayouts'. The approach of first looking for the collaboration-patterns (defined in Section 3.2.3) turned out to be useful. Nevertheless, it is necessary to browse the source on the found patterns, because of false negatives and furthermore to evaluate the collaborations.

5.2 Comparing Application Scenarios with Application Structure

We outline the approach of finding *unused structural entities* on the example of CodeCrawler and the appropriate test scenarios.

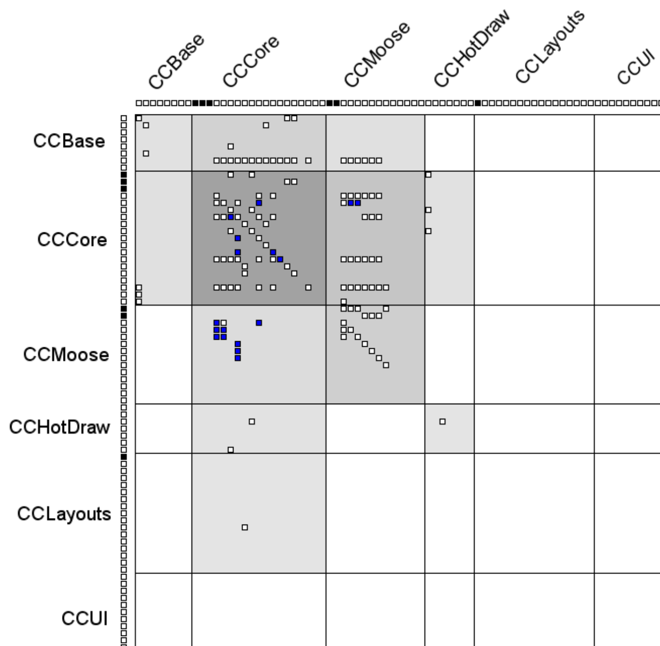

Figure 5.6: Collaboration-Matrix of collaboration in CodeCrawler version 4.274 at runtime

As we can see in Figure 5.6 the high-level entity *CCUI* is not affected by the executed

test-scenario. Thus removing the *CCUI* entity from the system should have no impact on the test scenario.

5.3 Collaboration in an Average-Sized System: Jun

A crucial property of reengineering tools is their ability to scale up. To show the strengths and limits of our approach we apply it on *Jun*, an average sized 3D Graphic Library. Jun for Smalltalk has been developed to bind OpenGL and QuickTime to VisualWorks, and for the implementation of fast 3D-graphics and multi-media applications.

We extract collaborations based on static information of version 501 of June which consists of more than 900 classes and 10'000 Methods in 110'000 lines of code.

We load all the classes in a software model in MOOSE and run the operator to extract collaboration information of the source code and annotate it to the software model. Since the classes of Jun are not grouped explicitly in packages we use the tool *Classifier* [TALE 03] to build an abstract model consisting of 16 groups.

Analyzing the collaboration-matrix of the high-level model of Jun we show how our approach scales and what we can gain of an average-sized system. In this analysis we are interested in an overview of the system and do not focus on low level details (Figure 5.7).

1. The inside-collaboration of the system is strong and shows that the supposed grouping conforms to the implicit grouping in the source.

2. The structure of the system in respect to collaborations is not cleanly layered. There is a lot of mutual collaboration dependencies between the particular groups.

3. The group *Jun-Geometry* is the core component in respect to collaborations with a lot of inside-collaboration as well as outside-collaboration.

5.3.1 Facilities and Limits

The application of our system on Jun shows that our approach provides useful information even on an average-sized system. The zoom-facilities allows the user to zoom out, to see the general collaboration structure and to zoom in to see the details - both

kinds of viewpoints present significant aspects. The limits lie in the extraction of collaboration information. The extraction of the static collaboration information take 24 hours (1Ghz processor pc) and the extraction of the dynamic collaboration information failed because it is not possible to install method wrappers on such a big number of methods (11'000).

Metric	value
Number of Classes	918
Number of Methods	10'877
Number of Attributes	1'211
Number of Local Variables	744
Lines of Code	114'387

Table 5.2: Jun version 501

5.4 Conclusion

To validate our approach we applied the collaboration-matrix on two systems of different size. We were able to identify core components and important collaborations between components, even if we did the validation on the average-sized system Jun without looking at the class level. For CodeCrawler we applied also collaboration-patterns at the class level and identified important classes and their roles.

What the validation of these two systems approved especially was the ability to provide an initial overview of an unknown system of any size in a short amount of time.

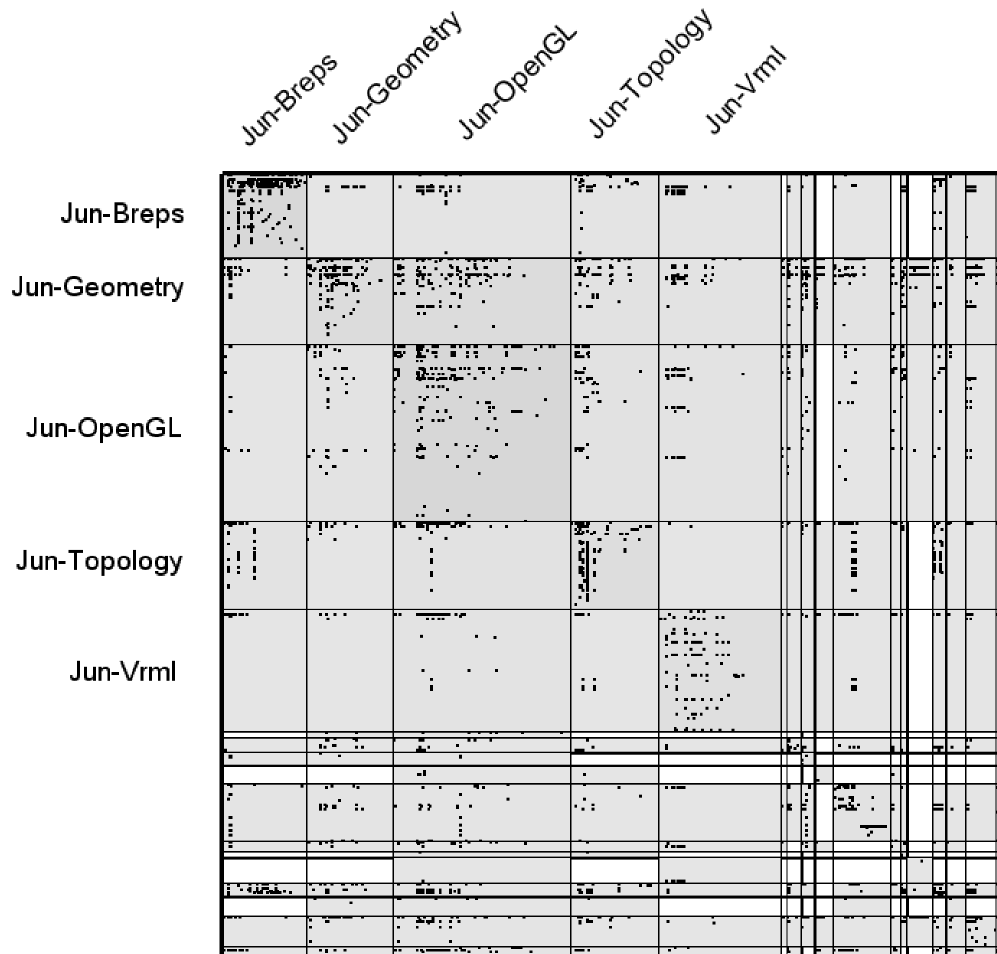

Figure 5.7: Collaboration of static information: Jun version 501

Chapter 6

Extracting Collaboration Information

The software model provides just the structure of the model. In order to make a high-level model from the software model, collaboration between source code entities has to be added, as explained in Section 3.1.2.

This chapter describes how collaboration on static and on dynamic information is extracted. This process is not language independent and we focus on how to extract collaboration information from Smalltalk code.

The fact that Smalltalk is a dynamically typed language makes it besides polymorphism [RICH 02] quite difficult to extract collaboration out of static information compared to C++ or Java which are statically typed. We present a heuristic approach to calculate types in a dynamic typed language. For the extraction of collaboration based on dynamic information, we describe an approach using method wrappers that record message sends between the objects at runtime.

6.1 Calculating types in an dynamically typed language

In Section 3.1.2 we defined collaboration extracted from static information as a kind of relation between the entities. To express this relation of collaboration we use type information for the considered variables. Since we use Smalltalk as the base language for our examples we somehow have to infer the types of the variables to be able to detect collaboration.

By using a trace graph for inferring types the results are sound. Parlsberg and Schwartzbach

describe an approach to infer types using a trace graph. Their algorithm constructs a set of conditional type constraints and computes the least solution by the least fixed-point derivation. The disadvantages of this approach are its worst-case time complexity which is exponential and its incapability to deal with meta-programming techniques (such as the Smalltalk-*blocks* and the *perform:*-method). [PALS 91]

Type inference using a trace graph calculates the *concrete types* of the variables. The definition of concrete types is as follows:

> *The concrete type of an object is its exact class.*

To reduce complexity and to overcome the incapability to deal with meta-programming techniques we apply heuristics to get type information about variables.

6.1.1 Heuristic Approach

We apply basically two different heuristics to get type information for a variable which are:

- Extract abstract type if possible
- Enhance the type information by evaluating expressions assigned to the variable

Abstract types are known as *interfaces, protocols or specification types*. They are sets of messages and the objects that understand all messages satisfy the abstract type [PALS 91]. The formal definition is as follows:

> *An abstract type T is a set of messages. An object x has abstract type T if x understands all messages in T.*

To extract the abstract type of a variable all messages sent to this variable are collected. We reduce in our approach the scope to look for message sends of the variable to the local environment of the variable like the class (for instance variables) or the method (for local variables). Most of the useful information for inferring types of a variable by heuristics like message sends or assignments is in the local environment. Therefore, not much of the available information is lost by restricting the scope to the local environment but the time complexity is reduced to less than exponential.

For certain variables it is not possible to extract the abstract types using only the local environment. Either there are only pure accessors for the variable in the local environment or only assignments. In the case of pure accessors our approach fails and does not extract any type information. If expressions are assigned to a variable or boolean operators applied we use simple additional heuristics to extract type information such as evaluating the expression assigned to the variable. An example for such an additional heuristics is given by evaluating the following assignment:

x := Collection new.

If on right side of an assignment a class name occurs this class is taken as the type of the variable. Even if such heuristics can produce positive falses they improve the informal extraction of type information.

Aggregate Types

Besides the advantage of being able to reduce time-complexity significantly, a heuristic approach of type detection can be enhanced and optimized by adding additional detection patterns.

In particular for extracting collaborations between objects it is important to have additional type information for variables with aggregate types. We define aggregate types as follows: a variable `a` which has *Collection* or a subclass of *Collection* as its basic type is an aggregate type. A variable with an aggregate type has beside its basic type an element type, which represents the type of the elements inside the collection.

The reason why aggregate types are this important for extracting collaborations is illustrated by the following example: If class `A` has an instance variable `a` with Collection as its basic type, we say that - according to the definition of Section 3.1.2 - 'A uses Collection' which does not express the real collaboration relation in the view of high-level models. The real relation is only revealed by extracting the type of elements in the collection. This means that if the elements of the Collection are of the type B, the collaboration should be expressed as 'A uses B'.

We use a heuristic approach to extract not only the basic types of variables but also the element type of aggregate types . In the following section the implementation of our heuristic approach is presented.

6.1.2 Implementation of a Type Annotator

In this section we present some implementation details of our approach of extracting type information. We use a logic library called LiCoR which provides logic representation of the parse tree and a mechanism to traverse the parse tree. The power of this mechanism is its possibility to define patterns (the heuristics) and collect the information of the source-code that matches the defined patterns.

Besides looking at those mechanisms of parse tree traversal we present some results we got applying the approach on CodeCrawler 4.2, Moose 4.1, and the *Magnitude*-class hierarchy of VisualWorks.

Type annotation with LiCoR

LiCoR is a *Li*brary for *Co*de-*R*easoning. Besides basic predicates to reify code artifacts like classes, methods and attributes the library provides predicates to traverse the parse tree of methods.

LiCoR is based on SOUL (Smalltalk Open Unification Language), a logic programming language implemented in Smalltalk. SOUL is a logic interpreter implementing a set of basic logic predicates [WUYT 01].

Type extraction

The type extraction consists of two main parts: a logic parse tree and logic predicates which define patterns of the use of variables. By traversing the parse tree and by applying the logic predicates we extract the information to detect the type of variables. The power of this approach lays in its simplicity to enhance type detection by adding additional patterns in a declarative and simple way.

Logic Parsetree

Soul has a predicate which provides a parse tree for a method. The parse tree is directly extracted from Smalltalk code. The method-parsetree predicate looks as follows:

```
method(class,method,arguments,temporaries,statements)
```

The parse tree of a method consists of the class in which it is located, the name of the method and lists of its arguments, its temporaries and its statements.

We show the method parse tree for a simple example. The following Smalltalk method increments the argument and returns the result.

```
increment:  aNumber
      aNumber + 1
```

the following structure is the logic representation for this method:

```
arguments(<variable(aNumber)>)
temporaries(<>)
statements(<return(send(variable(aNumber),+,
      <literal([1])>))>)
```

While the list of the temporary variables is empty, the argument list contains one single element. The single statement in the statement list is nested with a return statement as the outer statement and a send statement as the inner statement.

With the this parse tree we get a logic representation of a method on which we can apply logic queries such as

```
send(receiver,message,arguments)
```

This query applied on the logic parse tree above returns true and the variables are instantiated as follows:

```
receiver -> variable(aNumber)
message -> +
arguments -> <literal([1])>
```

The power of having a logic representation of a method is the ability to define queries for detecting patterns in a very declarative way. To apply a set of patterns on a complex parsetree we use the LiCoR parsetree traversal predicates.

logic parsetree traversal

The parsetree is traversed verifying for every statement wheter it is interesting and collecting it as result if it matches the defined patterns. For the traversal of the parsetree two different kind of rules have to be defined. The 'found' rules which defined what has to be found in the parsetree and the 'process' rules which define how this found information should be processed.

'found' Rules

The meaning of the 'found'-rules is the following: if a statement matches one of the 'found'-rules in the parsetree the 'process'-rules have to be applied on the found statement. For the type detection the following rules are defined:

```
assign(?Var, ?Value)
send(?Receiver, ?Message, ?Arguments)
cascadeSend(?Receiver, ?Message, ?Arguments)
```

The assign rule matches on all assign statements in the Smalltalk code like

```
var:=Collection new.
```

Assign statements often - as in this example - directly reveal the type information of the variable. The send or cascadeSend statements do not supply type information directly. They are used to extract the interface of a variable. Examples of statements which are matched by the *send* and *cascadeSend* rule are the followings:

`var removeFirst.` (example for send)
`var x:width;y:height;normal.` (example for cascadeSend)

Only statements which match one of the described rule are of importance for the type extraction and are processed by the "process"-rules.

"process" Rules

The process-rules extract the type information of the statements. There are basically two different kinds of processings: first the processing of the *assign*-statements and

second the processings of the *send/cascade-send* statements. The approach for the assign statements is to directly extract type information of the right-side of the assign statements. We list the possible patterns by illustrating them with examples. If the right-side of the assignment consists of a class and a selector of the protocol 'create instance', the class is the type of the variable:

```
var:=Collection new.
```

If the right-side of the assignment consists of a send statement and the message contains a boolean selector (=, ==, , =, <, >, <=, >=), then the type of the variable is Boolean:

```
var:=(var1 size < var2 size).
```

Results of the Type Detection

In Table 6.1.2 the results are presented we got applying the approach on CodeCrawler 4.2, Moose 4.1, and the *Magnitude*-class hierarchy of VisualWorks. Since our heuristic to extract abstract types is limited to the local environment of a variable and also the additional heuristics can not extract information for all those variables there are always some variables where our approach can not provide any type information. The *Typed Instance Variables* in the table indicates the number of variables where type information is extracted and the line with the percentages indicate the number of *Typed Instance Variables* divided by the number of instance variables in the system. The *Aggregate Instance Variables* are those which are identified to be a subtype of *Collection* and the *Typed Aggregate Instance Variables* are those where even for there elements type information is found.

Application:	CodeCrawler	Moose	Magnitude
Instance Variables	121	238	31
Typed Instance Variables	70	119	16
Typed Instance Variables in %	57%	50%	51%
Aggregate Instance Variables	31	52	-
Typed Aggregate Instance Variables	19	5	-

Table 6.1: Results of Type Inference

6.2 Extracting runtime collaboration

We have defined runtime collaboration in Section 3.1.2 as the method-send of class A to class B which is expressed as ’A uses B’. We extract this information by installing method wrappers on all methods of the loaded source model. The method wrappers of a particular method record a message-send to the software model every time the method is called.

Instance-Level Trace versus Method-Level Trace

There are different aspects of information which can be extracted by tracing method calls between objects. At runtime, a method is called on an instance of a class which is not necessarily the same class where the method is defined in. Using inheritance a method can be invoked on an instance defined in one of its superclasses. So there are always two candidates for the receiver of a method call:

- The class of the instance on which the method is called.
- The class where the method is defined in.

With respect to collaboration both of them are important candidates and provide in fact different but meaningful information. While collaboration on the method-definition level expresses source-code dependency, collaboration on the instance-level expresses an abstract coupling.

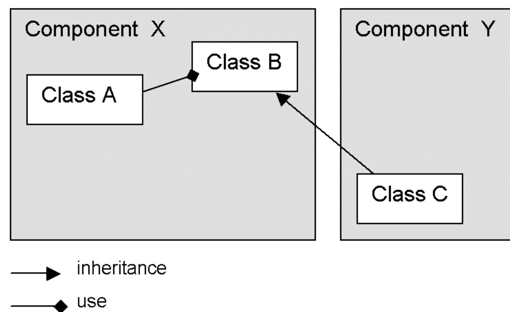

Figure 6.1: Collaboration and Dependency Scenario

The following example shows that collaboration of components on the instance-level even can even be inverse to dependencies in the source code (method-level). In Figure 6.1 a simple example-scenario is shown consisting of two components. *Component X* consists of two classes A and B, *Component Y* just consists of one class C. At source-level there are two relations defined between those classes(see Table 6.2).

Class Dependencies	Component Dependencies
A has a collection of B	X uses X
C is subclass of B	Y uses X

Table 6.2: Class Collaboration vs. Component Collaboration

At runtime the inheritance mechanism allows an instance of A to have instances of class C as elements of its collection. If A calls a method defined in B on one of the instances of C the following collaboration dependencies arise (see Table 6.2).

Method-Level:	A uses B	X uses X
Instance-Level:	A uses C	X uses Y

Table 6.3: Method-Level Collaboration vs. Instance-Level Collaboration

The collaboration at method-level correspond exactly to the one extracted from static information while the collaboration at instance-level leads to a cyclic collaboration between the two components (see Figure 6.2).

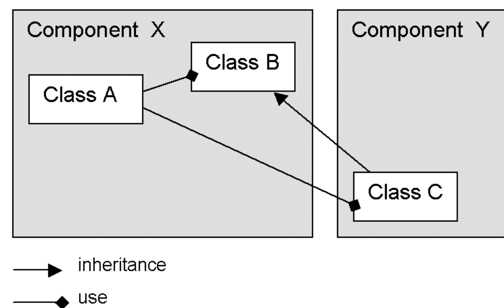

Figure 6.2: Collaboration and Dependency Scenario (cyclic dependency)

Level of Detail

The main problem when using execution traces is the large volume of low-level information it provides [RICH 02]. A message-send consists of the elements sender, receiver, message and timestamp. Storing this information for every single message send produces a huge amount of data.

To reduce the amount of information stored we only keep the existence of a collaboration between two classes (and not the number of messages sent between two classes). In addition we reduce the method send information to only store receiver and sender value (and not timestamp and message information). Using this approach we can limit the amount of stored traces to n^2 where n is the number of classes in the model.

Whereas execution trace always provide unambiguous collaborations, extracting collaboration of static information polymorphism and inheritance make it impossible to always detect the unique entities taking part in a certain collaboration. The limit of the collaboration extraction of dynamic information is the incomplete coverage of the source code. Collaboration is extracted only for the source code belonging to a certain scenario which is executed. If there exists widespread tests for the application, they provide a good facility to have well defined and reproducible scenarios to extract collaboration.

Since the limit of space used for storing trace information is $(n^2) * (traceinformation)$ independent of how often and how long the system is running, there are no scaling problems even with very large systems.

Differences between extracted collaboration

In general collaborations extracted using dynamic information are instances of possible collaborations extracted using static information. There are however exceptions to this general rule:

Runtime collaboration without representation in static collaboration
If an object sends a message to another object the sender has to have a reference to the receiver object. Usually the reference is stored either in a local variable, or in an instance variable, but they do not have to be stored obligatory. In Smalltalk for instance an object could be instantiated and a message sent to it without storing the reference in any variable. There is no way to detect possible collaboration with our approaches based on static information if the reference is not stored in any variable.

No Runtime Collaboration because of Dead Code
If there are variables in the system which can never be instantiated because they belong to dead source code there can be a possible collaboration based on static information which - based on dynamic information - never exists.

No Runtime Collaboration because of Incomplete scenarios
A certain runtime collaboration only exist, if there is a scenario which executes the corresponding code. Runtime collaboration usually is not complete with respect to the source code.

Chapter 7

Future Work

In this chapter we present some additional ideas and where possible extensions of this work could lie. In the following section we show an approach of refactoring component architecture where we already did some experiments.

- Provide a mechanism for defining subsystems with a logic approach like this is done in *Software Reflexion Models* [RICH 02] or *Declaratively Codifying Software Architecture* [MENS 99].
- Apply the approach on huge systems using more levels of details e.g. subsystems, sub-subsystems, classes. This could raise scalability and would provide facilities to find finer-grained restructurings for systems.
- To move from a collaboration between classes used as primary relation to a more general *dependency*-relation. Collaboration is just one kind of dependency. To add other types of dependencies between classes like inheritance-dependency using the same approach of visualisation would enhance the value of the resulting models.

7.1 Refactoring Component Architecture

A possible application using collaboration-matrix is this of refactoring component architecture by moving items between components, providing a clean layered architecture with non-cyclic collaboration between the components is the goal of this application.

In MOOSE there are two packages with a strong mutual collaboration which are *CC-Core* and *CCMoose*. In this simple example We outline how the collaboration-matrix can be used to reduce mutual collaboration between high-level entities by moving low-level entities from one to another.

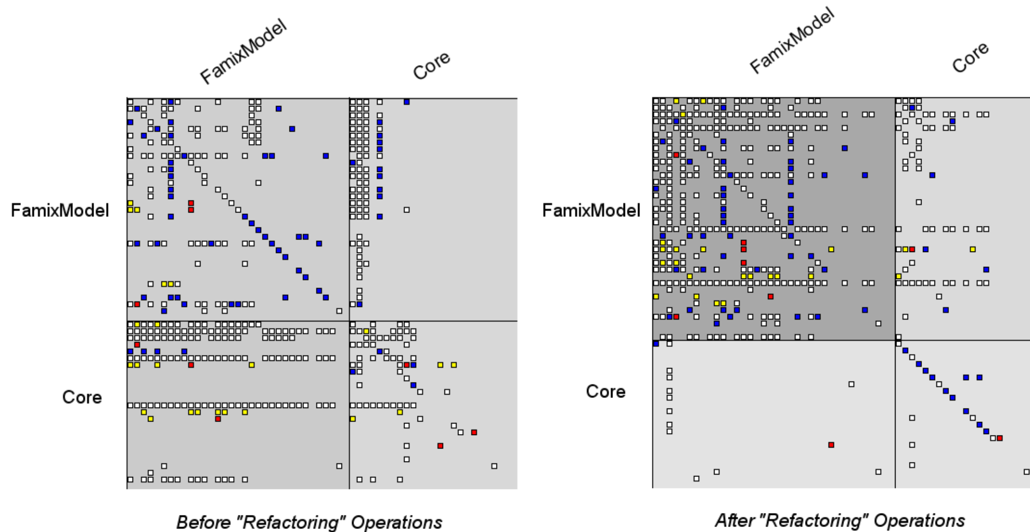

Figure 7.1: Visual Refactorings in Moose

The collaboration using the inital structure is shown in Figure 7.1 (Before Refactoring Operations). After the iterative process of manually interchange entities between the high-level entities the collaboration using the modified structure is shown in Figure 7.1(After Refactoring Operations).

This process of refactoring is done without using any internal knowledge about the system by just moving iterativly items with multiple collaboration in the down left square of the collaboration-matrix to the complementary high-level entity.

We do not state any completeness of this approach. The only goal is to show possible restructurings which reduce mutual collaboration between two components, based on simple visualized collaboration information.

Moved from *MooseCore* to *MooseFAMIXModel*:	Moved from *MooseFamixModel* to *MooseCore*:
MSEModel	FAMIXModelQueryFacade
MSEModelMVAttributeDescriptor	FAMIXModelSmalltalkQueryFacade
MSEModelAttributeDescriptor	MSEExpressionArgument
MSEModelInformation	MSEAccessArgument
MSEAbstractImporter	MSEAbstractArgument
MSEAbstractModelRoot	MSEInclude
MSEAbstractRoot	MSEFunction
MSEAbstractGroup	MSESourceFile
MSEMetric	
MSEIntegerIdGenerator	

Table 7.1: Classes moved between the Packages in Moose

Chapter 8

Conclusion

In this thesis we present an approach to build mental models of object-oriented systems in the context of reverse engineering. Complementary to other approaches which use information that is explicit in the source (like classes, class-hierarchies, methods and attributes), we use collaboration information between classes which we have to extract first. As there is no way of extracting complete collaboration information but only different types which all focus on different aspects, we combine the different types of collaboration to improve the overall completeness of the model.

We visualize the models we extract from the source which enables the engineer not only to browse through one level of granularity but as well to switch between different levels of details.

We show the value of our visualisations by applying them to two small but complex systems as well as an average one. The insights gained of the systems prove that this approach of providing facilities to browse between different levels and links to the source improves the work of a reverse engineer considerably.

8.1 Lessons Learned

- Applying our approach on case studies we notice that there is very little difference between the different types of collaboration at the level of subsystems. The dependencies between the subsystems are very similar for the different types of collaboration.

- Instance variables express important dependencies on the level of subsystems.
- We extract collaboration information from the source. As this information is not explicit in the source and has to be calculated this process can cause performance problems applied on large size systems.
- Our approach is appropriate to gain an overview of the system and extract roles on the level of subsystems. The facilities to extract roles of classes are limited.

Bibliography

[BIGG 89] T. Biggerstaff. *Design Recovery for Maintenance and Reuse*. IEEE Computer, pages 36–49, October 1989. (p 7)

[CHIK 90] E. J. Chikofsky and J. H. Cross, II. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, pages 13–17, January 1990. (pp 6, 7)

[DEME 01] S. Demeyer, S. Tichelaar, and S. Ducasse. *FAMIX 2.1 – The FAMOOS Information Exchange Model*. Research report, University of Bern, 2001. (p 27)

[DUCA 99] S. Ducasse, M. Rieger, and S. Demeyer. *A Language Independent Approach for Detecting Duplicated Code*. In H. Yang and L. White, editors, Proceedings ICSM '99 (International Conference on Software Maintenance), pages 109–118. IEEE, September 1999. (p 26)

[JERD 97] D. Jerding and S. Rugaber. *Using Visualization for Architectural Localization and Extraction*. In Proceedings WCRE, pages 56 – 65. IEEE, 1997. (p 1)

[LANZ 99] M. Lanza. Combining Metrics and Graphs for Object Oriented Reverse Engineering. Diploma thesis, University of Bern, October 1999. (pp 5, 7)

[LANZ 01] M. Lanza and S. Ducasse. *A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint*. In Proceedings of OOPSLA 2001, pages 300–311, 2001. (p 1)

[LANZ 03a] M. Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, may 2003. (pp 1, 2)

[LANZ 03b] M. Lanza. *CodeCrawler - Lessons Learned in Building a Software Visualization Tool*. In Proceedings of CSMR 2003, page to be published. IEEE Press, 2003. (p 27)

[LEHM 85] M. M. Lehman and L. Belady. Program Evolution - Processes of Software Change. London Academic Press, 1985. (p 5)

[LIU 96] C. Liu. Smalltalk, Objects, and Design. Manning Publications, 1996. (p 6)

[MENS 99] K. Mens, R. Wuyts, and T. D'Hondt. *Declaratively Codifying Software Architectures using Virtual Software Classifications*. In Proceedings of TOOLS-Europe 99, pages 33–45, June 1999. (pp 8, 9, 57)

[MURP 95] G. Murphy, D. Notkin, and K. Sullivan. *Software Reflexion Models: Bridging the gap between Source and High-Level Models*. In Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 18–28. ACM Press, 1995. (pp 8, 15, 18)

[PALS 91] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Inference*. In Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, pages 146–161, November 1991. (p 47)

[PAUW 93] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. *Visualizing the Behavior of Object-Oriented Systems*. In Proceedings OOPSLA '93, pages 326–337, October 1993. (pp 11, 13)

[RICH 99] T. Richner and S. Ducasse. *Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information*. In H. Yang and L. White, editors, Proceedings ICSM '99 (International Conference on Software Maintenance), pages 13–22. IEEE, September 1999. (pp 1, 15)

[RICH 02] T. Richner and S. Ducasse. *Using Dynamic Information for the Iterative Recovery of Collaborations and Roles*. In Proceedings of ICSM '2002 (International Conference on Software Maintenance), October 2002. (pp 2, 7, 8, 10, 18, 46, 55, 57)

[TALE 03] D. Talerico. Grouping in Object-Oriented Reverse Engineering. Master's thesis, University of Bern, 2003. (pp 16, 43)

[TICH 01] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001. (p 27)

[WILD 93] N. Wilde, P. Matthews, and R. Hutt. *Maintaining Object-Oriented Software*. IEEE Software (Special Issue on "Making O-O Work"), vol. 10, no. 1, pages 75–80, January 1993. (pp 5, 16)

[WUYT 01] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001. (p 49)