Recovering Behavioral Design Views: a Query-Based Approach

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Tamar Richner-Hanna

von Gränichen, AG

Leiter der Arbeit: Prof. Dr. O. Nierstrasz, Dr. S. Ducasse
Institut für Informatik und angewandte Mathematik

Recovering Behavioral Design Views: a Query-Based Approach

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Tamar Richner-Hanna

von Gränichen, AG

Leiter der Arbeit: Prof. Dr. O. Nierstrasz, Dr. S. Ducasse
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Der Dekan:

Bern, 17. Mai 2002 Prof. Dr. P. Bochsler

Abstract

The reality of software development is such that engineers must often perform maintenance tasks with missing or out-of-date documentation and without the support of the original developers. To understand the software *as it is now*, engineers use reverse engineering tools to recover information from the code itself. Most such tools analyze only static information about the system and so provide engineers with structural, rather than behavioral models. It is, however, critical to understand the behavioral aspect of the software system in order to carry out certain maintenance tasks.

To better understand program *behavior* engineers turn to tools which use dynamic information collected during program execution. Such tools typically display all the dynamic information at very fine granularity, making it difficult to extract manageable models of behavior. They then rely on visualization and navigation techniques to help the engineer locate information relevant to the change task.

In this dissertation we propose an approach to recovering behavioral models from object-oriented software which is based on *perspectives*. Our approach enables an engineer to declaratively define perspectives through which the dynamic information can be viewed. It supports an iterative recovery process in which successive views of the software system help the engineer to answer questions related to the maintenance task to be performed. We claim that such an approach can overcome the difficulties of recovering succinct and focused views of object-oriented software from dynamic information.

A perspective is a model of the kind of information that an engineer is interested in. Our approach supports the construction of principally two kinds of such models: component-connector models and collaboration models. We first identify a meta-model for describing object-oriented software and its execution, then develop a simple declarative way to express perspectives in terms of this meta-model: component-connector perspectives express a range of static groupings and dynamic relations; collaboration perspectives abstract from execution sequences to class collaborations. Using case studies we demonstrate the validity of our approach by showing how perspectives are used in an iterative process to recover both high-level and low-level succinct behavioral views.

Acknowledgments

First, I would like to thank the members of my Ph.D. committee. I thank my advisor, Oscar Nierstrasz, for his support over the years, and for his help in improving the presentation of my work. Many thanks to Stéphane Ducasse, who co-supervised this research. He has followed my work through most of its stages and I thank him for his continuous encouragement, for our many discussions and for the fruitful collaboration on several papers. I thank Kai Koskimies for writing the referee report and for coming from Finland to be the external examiner. My thanks to Professor Torsten Braun for chairing the committee.

I thank my colleagues at the Software Composition Group: thanks to Serge Demeyer for his guidance during the earlier stages of this work and to Roel Wuyts for many constructive comments towards the end. Thanks to Matthias Rieger for help with widgets, and to Sander Tichelaar and Michele Lanza for discussions about 'their' case studies. To my office mates, Franz Achermann and Juan Carlos Cruz, thanks for being there to listen, to laugh with – and of course for enlightening discussions on software architecture, composition and coordination. Thanks to Gabriela Arévalo for cheering me on at the tiring end phase, and for cookies. Thanks also to Alexandre Bergel, Isabelle Huber, Peng Liang, Nathanael Schärli and Therese Schmid. I thank you all for discussions, for reading drafts, and for the fun we had.

Several people I met at conferences or as visitors to our group have commented on my work and pointed me to interesting questions. In particular, I thank Gail Murphy, Patrick Steyaert, Bjorn Freeman-Benson, Stan Jarzabek, Spencer Rugaber, Tarja Systä and Andrew Black for mostly brief, but helpful discussions.

I am grateful to the Swiss National Science Foundation for awarding me a Marie Heim-Vögtlin grant at the beginning of my doctoral work. My thanks to Professor Dieter Hogrefe who initially sponsored my grant application and to Stefan Leue for several discussions and for much encouragement while I was writing the grant proposal.

I am especially grateful for the logistic support for finishing this work. Lee-Kaja Jost, and more recently, Maloti Bordoloi-Sharma were at home for the kids when I was working. Most of all I thank my mother-in-law, Verena, who always readily took over in the absence of our babysitters.

Finally, I thank my friends and family for their support. In particular, I thank Heinz for his unfailing encouragement and for his patience in enduring through this enterprise. Last but not least, I thank my children – Naomi, Jonas and Joel – for just being who they are.

Tamar Richner, May 2002

Contents

List of Figures

List of Tables

1

Introduction

> "A program that is used in a real world environment necessarily must change or become progressively less useful in that environment"[LB85]

Software must constantly change and evolve. In order to perform changes on existing software, engineers must be able to understand the software. The reality of software development is such that development teams must often perform maintenance tasks with missing or out-of-date documentation and without the support of the original developers.

In this context it is important to have access to tools which aid in program understanding. There is a large spectrum of approaches and tools which can help in this [WTMS95][HYR96][KC98], ranging from debuggers [LHS97] to metric approaches [FP96][LD02][DD99], from program visualization [KM96][PLVW98][JR97] and animation techniques [WMFB+98] to design conformance checkers [MN97][Wuy98][Ciu99]. These tools help engineers to *reverse engineer* a conceptual model from the software itself.

In considering models of software, we distinguish here in particular between structural and behavioral models. Structural models describe how the software artifacts are organized, be it through inheritance relationships, packages or modules, whereas behavioral models describe how these artifacts collaborate at runtime to carry out a certain functionality. Most program understanding and reverse engineering tools make use only of static information to extract models of the software. However, for object-oriented software, where there is a class-object dichotomy and polymorphism makes it hard to determine what actually happens at run time, models extracted using static information only are limited in conveying information about program behavior.

To better understand program *behavior* engineers turn to tools which use dynamic information collected during program execution. Most of these tools display all the runtime message exchanges and give information at a very low level of granularity. These approaches thus recover verbose models and rely on visualization and navigation techniques for helping the engineer to locate information relevant to the change task.

In our research we explored the feasibility of using dynamic information in a different way than these tools. Whereas most approaches retain the time element present in the execution trace and present it either as a vertical time line in a sequence diagram display

[JR97][KM96] or as simulated or real time through animation techniques [WMFB+98], we decided to ignore the time element altogether. We then investigated how the dynamic information *without the time dimension* helps in forming concepts about the high-level and low-level design of the software.

Our motivation was the following: although we want access to low-level run-time information, reverse engineering is not debugging – we also want help in forming general concepts about the software's behavior. When performing changes on software we want compact high-level views which give us an idea of the relations of parts of the software to each other and guide us in deciding where to focus. We need low-level views for understanding the run-time interactions of objects in *specific* parts of the software, rather than everywhere. We want to know which parts of the software are involved in carrying out a certain functionality and what their role is. We therefore want an approach which can extract both high-level and low-level views, but which does not present us with *all* the low-level information. We want an approach which helps us to focus on the part of the software which is of interest for the maintenance task we must undertake, without overwhelming us with details.

1.1 Approach and Contributions

In this section we describe our approach and the hypothesis it rests on, then outline the contributions made in the dissertation.

1.1.1 Research Hypothesis: the Nature of Design Recovery

Our work on reverse engineering has been guided by the following two observations about the nature of design recovery [MN97]. These two guidelines can be seen as a research hypothesis which forms a basis for our approach.

The design recovery activity is task-specific. The kinds of questions we ask in design recovery depend on the software maintenance task with which we are confronted. If we want to replace the user-interface of a software system, for example, we will need to discover the dependencies of the system on the present user-interface to evaluate the work involved. If instead, we want to add functionality to the same system, we will want to understand how related functionality is now carried out, and which parts of the software have a role in carrying out this functionality.

The task we must undertake will dictate the questions we ask about the software. We therefore do not need to extract a view which answers all possible questions about an application, but rather the ability to tailor the recovered view to the question at hand.

Design recovery is an iterative process. Whether or not we are using a tool, human intervention is always required to interpret the result of design recovery. The design

> recovery cycle is then as follows: analyze the subject system with the goal of understanding a specific thing about it – extract a view – interpret the view and revise the question – go back to the system to obtain an answer – revise the hypothesis and question, and so on, until we have sufficient information for answering the task-specific question.

Thus, ideally, the recovery approach must support an iterative process which is steered by the engineer confronted with a maintenance task.

1.1.2 Approach

In this dissertation we describe an approach to recovering behavioral views of object-oriented software. The essence of our approach for recovering views can be stated as follows:

> We enable a developer to construct a model of the kind of information that he or she is interested in, then create a view of the software by showing the dynamic information *through* this model.

We call the model constructed by the developer a *perspective*: it is the declarative lens through which we view the dynamic information. Our approach supports the construction of basically two kinds of perspectives which we considered important in understanding the behavior of object-oriented programs.

The first kind of perspective enables an engineer to recover a *concept view* of the software. Such a view presents the software as a set of components and connectors. The semantics of the components and the connectors are defined by the engineer: components are created by grouping together static elements of the software, such as classes or methods; connectors are defined by expressing a dynamic relationship between these static elements, such as invocation or creation relationships. Concept views thus accommodate a range of different views.

The second kind of perspective enables an engineer to recover a *collaboration view* of the software. Such a view presents the dynamic information as a collection of class collaborations. The goal here is to understand how instances collaborate at runtime to carry out a certain functionality by abstracting from similar execution sequences to a collaboration. These abstractions are created by applying pattern matching to the execution trace – the role of the engineer here is to specify what he or she considers to be similarity in execution sequences by modulating the pattern matching criteria. This gives semantics to the notion of collaboration.

A key part of our approach is the iterative querying it supports. Not every view created using perspectives will answer all the questions a developer might have. But each view will relate some information which helps the engineer to decide on the next view to generate, and the next query to launch.

The approach we propose is *lightweight* – a developer can quickly obtain initial views with a minimum of investment. This means that the approach does not rely on sophisticated analysis techniques for extracting an information base from the source code and

that formulating perspectives to obtain views is simple. The approach also produces views which are *succinct* – compact views which focus on the elements of interest and do not rely on visualization techniques for navigating the information.

1.1.3 Contributions

The overall contribution of this dissertation is the development of a methodology for reverse engineering behavioral design views of object-oriented programs using dynamic information. Our research contains the following contributions:

- we analyze the problems of design recovery for object-oriented software systems and survey current solutions and approaches. We look in particular at techniques of dealing with dynamic information.

- we identify a lightweight model for representing object-oriented programs. This model consists of a meta-model for representing static information in terms of the basic static entities and relations of object-oriented software – classes, methods, inheritance definitions, attributes, accesses and invocations – and a meta-model for representing program execution as a sequence of message send events.

- we introduce the notion of *perspective* as a declarative lens through which we will view dynamic information. A perspective is a model of the kind of information that a developer is interested in. We demonstrate that perspectives support the construction of two kinds of models which are particularly useful in understanding object-oriented programs. The first kind of model is a *component-connector* model. A component is a grouping of static elements of the software, a connector is a relation between the components. These models are useful in obtaining high-level views of the software, which we call *concept views*. The second kind of model is that of a *collaboration* – an abstraction of the interaction of instances to carry out a certain functionality. Collaborations are useful in obtaining low-level views of the software, which we call *collaboration views*.

- for concept view recovery, we develop a way to express perspectives using a logic programming language. We describe how component abstractions are specified by grouping together static entities, such as classes, into a component, and how the semantics of a connector is defined as a relation in terms of message send events. We identify and encode connector types which express several interesting dynamic relations, such as method invocation and object creation. We also demonstrate the encoding of design pattern abstractions using dynamic information.

- using case studies, we demonstrate the use of perspectives for the iterative recovery of concept views and identify several useful perspectives.

- for collaboration view recovery, we introduce pattern matching as a technique to identify similar execution sequences in the trace as instances of one collaboration

abstraction, and show how modulating the pattern matching criteria enables us to change the semantics of a collaboration to reflect the model we are interested in. We describe a simple query model for investigating collaboration patterns.

- using case studies, we demonstrate the recovery of collaborations. We show how the query model is used to characterize a particular collaboration and to decompose a program trace into collaborations.

- finally, we discuss how the two techniques of concept view recovery and collaboration view recovery could be integrated into one tool.

1.2 Structure of Dissertation

The dissertation is structured as follows:

Chapter 2 surveys the problems of design recovery and the existing approaches. We begin by presenting general approaches to design recovery and then focus in more detail on the recovery of design models from object-oriented software. The recovery of dynamic models, in particular, is discussed at length. At the end of this chapter, we distill a set of requirements for our own work.

Chapter 3 introduces the elements of our approach. We discuss the iterative approach in design recovery, present the meta-model for representing object-oriented programs and their execution, and introduce concept view recovery and collaboration view recovery.

Chapters 4 and 5 describe concept view recovery, which is used to recover views of a software system as *components* and *connectors*. In Chapter 4 we present a declarative framework on which concept view recovery is based, and the tool, Gaudi, which implements the framework. In Chapter 5 we then present the case studies conducted to validate the application and conclude the chapter with a discussion and evaluation.

In Chapters 6 we present collaboration view recovery, which is used to recover *collaborations* and *roles*. We describe the use of pattern matching in the recovery of collaborations, and introduce a query model for querying about collaborations. We describe the Collaboration Browser, a tool which supports the extraction and querying of collaboration, and present the case studies conducted to demonstrate the tool and validate the approach.

The dissertation concludes with Chapter 7, where we summarize the contributions made and discuss open problems and future work.

2

Design Recovery for Object-Oriented Systems: a Survey

The way we describe the process of software development has changed. Whereas ten years ago the waterfall model was still used to describe the sequential steps in the development process, it is now recognized that an iterative development cycle, as described by the spiral model [Boe88], better reflects the reality of the development process. There is a conscious effort in many organizations to encourage iterative development cycles which result in early prototypes, thus allowing for elucidation of requirements and detection of problems earlier in the software development process [Bec00].

With iterative development, the issue of software evolution is not restricted to the aging of software, or to legacy systems. The iterative cycle acknowledges that software evolves as we develop it, as we take into account more precise or newer requirements and adapt it to changing environments. Software evolves constantly – and this means that as we develop software we need to understand its current structure and behavior and to communicate this understanding to others.

Round trip engineering evokes the possibility of designing an application and automatically deriving its implementation, manually adapting the implementation and automatically updating the design documents to reflect the current state of the code [SZ99]. But until the advent of this "seamless integration of design diagrams and source code, of modeling and implementation"[DDT99], we must deal with the issue of 'manually' deriving models which represent the current structure and behavior of our software.

In order to obtain models which describe the software *as it is now*, we must recover design information from the software itself, *i.e.*, reverse engineer the software. So reverse engineering is not only important for legacy systems, it is an important part of the normal development cycle and is essential in many software maintenance activities.

In this chapter we survey existing approaches to design recovery, both general ones and ones specific to object-oriented software. From this survey we distill a set of requirements for our own work on the recovery of behavioral models. The chapter is organized as follows: In Section 2.1 we present an overview of reverse engineering and design recovery. The chapter continues with Section 2.2 where we look more closely at design

recovery for object-oriented systems. Finally, in Section 2.3 we introduces the requirements that have guided our work on the recovery of behavioral models.

2.1 Reverse Engineering and Design Recovery: an Overview

We first present definitions related to reverse engineering, then follow in Section 2.1.2 and Section 2.1.3 with a discussion of what design is and how it is documented. Finally in Section 2.1.4 we review general approaches to design recovery.

2.1.1 Definitions

To put our work in context, and to be clear in the use of certain terms, we present below accepted definitions for the important concepts which appear in the dissertation. Figure 2.1 illustrates these schematically.

> "**Forward engineering** is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of the system. "[CC90]

> "**Reengineering** ... is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."[CC90]

> "**Reverse engineering** is the process of analyzing a subject system to identify the system's components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction."[CC90]

> " **Design Recovery** recreates design abstractions from a combination of code, existing documentation (if available), personal experience, and a general knowledge about problem and application domains."[Big89]

The context for the work presented in this dissertation is reverse engineering and design recovery, and we will use these two terms interchangeably. As seen from Figure 2.1, reverse engineering is an essential part of reengineering. Although our work does not touch upon issues related to the alteration of the software system, it is clear that reverse engineering and design recovery are essential for many software maintenance tasks.

A software maintenance task is simply a task which involves making changes to the software. The definition of software maintenance offered by the ANSI/IEEE Standard dates from 1983, and makes a distinction between change tasks before or after product delivery, reflecting the waterfall model philosophy of its time:

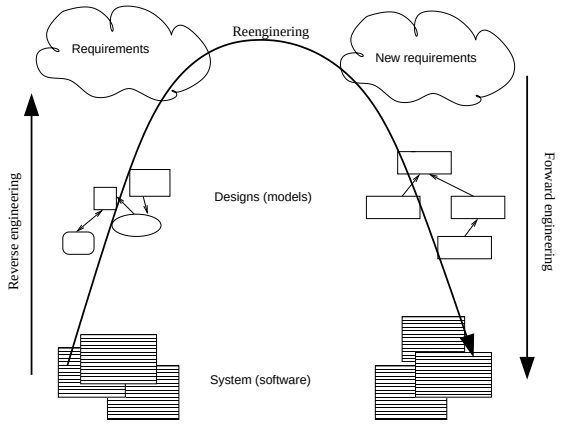

Figure 2.1: Reengineering concepts

> "**Software maintenance** is the modification of a software product [after delivery] to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment."[ANS83]

For our purposes, however, we ignore the bracketed words in the definition, *i.e.*, software maintenance is almost any software development task in which we make modifications to the software for some reason. Several industry surveys suggest that the amount of effort organizations spend on software maintenance as a percentage of their total programming effort is between 65% and 75% [Som92].

2.1.2 What is Design?

Recovering design information from an existing application is crucial to engineers confronted with a variety of software maintenance tasks. But what do we recover when we do design recovery? What is design? The words *design*, *architecture*, *architectural design* and *software structure* are used by some people interchangeably, whereas others will argue vehemently about the critical differences which distinguish these concepts from each other.

Design is more easily understood as a software development activity, the *design process*, than as a product:

> "Design is really two activities: architectural design and detailed design. Architectural design involves making strategic decisions about how system functionality is factored among independent components, how components relate and how control transfers from one component to another. It often includes a specification of how users give and receive information, and how the system communicates with other systems.
>
> Detailed design consists of tactical decisions, such as the choice of algorithms and data structures to meet performance and space objectives."[GR95]

What is embedded in the design process is the decision making process involved:

> "Design is a creative process which requires experience and some flair on the part of the designer. [...It] involves a number of different stages:
>
> (1) Study and understand the problem
>
> (2) Identify gross features of at least one possible solution. It is often useful to identify a number of solutions and to evaluate each of these. The choice of solution depends on the designer's experience, the availability of reusable components, the simplicity of the derived solutions [...]
>
> (3) Describe each abstraction used in the solution [...]"[Som92]

It is exactly this decision making process – identifying several solutions and choosing one among them – which gives us a key to understanding the resulting software system. But this decision making process does not appear in the software, and is rarely to be found in design documents. Indeed, one could argue that a good design document should document not only the chosen solutions, but also the rejected options, so as to make it easier to backtrack in the case of changing requirements or environment.

Although software design is a process, the word is also used to refer to the product of this process. The real products of the design process, that is the *design* or *architecture*, are all the important decisions taken about how to organize and structure the system. Many of these often remain undocumented – typically all we have is the final product: the implemented system. In the next section we will look more closely at what we want to recover when we do design recovery.

Design and architecture. Software architecture has been defined as the decomposition of the software into the main design elements (the components) of the system, the interactions among these components (the connectors), and the rules or conventions governing their assembly [SG96]. What is often presented as the architecture of an application is the strategy used to solve the foremost design problem.

We will use the words architecture and design interchangeably, favoring the word design. Architecture is usually taken to be high-level design or implementation-independent formalism to describe the decomposition of the system in to components and connectors. For the purposes of this work, there is no gain in making a distinction between architecture, design or software structure. The goal of recovery is to arrive at a model for understanding the software – so the formality of the notation or its implementation-independence is not important. The term *software architecture* is a more useful notion in the context of *architectural style*, that is, a template for a certain structure which can be recognized and reused.

2.1.3 Design Views

Any complex software system must deal with design problems along several axes. Decisions about the logical relationships between the domain concepts, for example, are

separate from decisions about concurrency and synchronization issues. Although design issues often cross-cut each other and design decisions along separate axes must be consistent with each other, we benefit from modeling these aspects separately. First, because we are usually interested in one particular aspect, and second, because there is no one notation which can be used to express all the aspects of a software design.

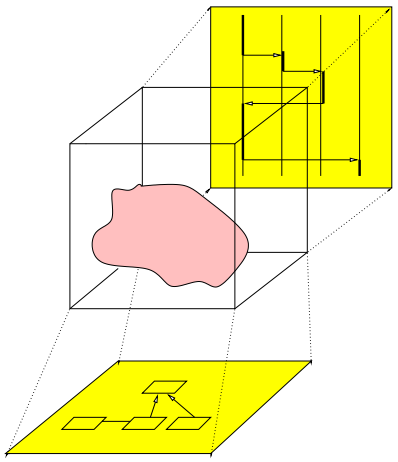

Figure 2.2: Design views as projections

We therefore describe a software design through models which express a certain view of the software. Kruchten [Kru95], for example, proposes four views: logical view, process view, physical view and development view, which are elaborated through the use of scenarios, representing the important variations of the systems functionality. A variety of strategies and techniques are used to map views on each other. The UML [BRJ99] provides for different kinds of models to express the design: use case models capture user requirements, static models describe the main elements and their relationships and dynamic models describe the system's behavior at runtime. Aspect-oriented programming [KLM+97] also recognizes the need to reconcile different aspects of a system, and proposes an approach for combining these at the programming language level. Figure 2.2 illustrates the idea of design views: each view can be seen as the projection of the software system through a window; each view models one aspect of the software system.

Luckily, we rarely need to understand everything about a software system. We need to understand just enough to carry out a specific maintenance task, with the assurance that the changes we effect in the system do not have a detrimental effect on the rest of the software. That is, in *design recovery*, we want to recover a model of the system which captures the particular aspect or view of the software which is of interest to us. So design recovery is task-specific, it is driven by a particular question of interest.

We cite here a few examples of researchers and practitioners who advocate the use of multiple views in software development and propose a repertoire of views: UML views [BRJ99], Kruchten's 4+1 views [Kru95], Siemens' architectural description [SNH95] and De Hondt [Hon98]. The IEEE *Draft Recommended Practice for Architectural description* [IEE99] aims to support any number of views of a system, rather than a fixed set. Though

there is no consensus on a minimal or complete set of views required in software development, it is possible to list a representative set of views which addresses most of the concerns, as shown in Table 2.1. Each view is described, together with references to the names of the corresponding view in other publications.

View	*Description*
Domain Model	A domain model view describes the system in terms of the main domain concepts and their relationships. This is the *logical* view in [Kru95], *conceptual architecture* in [SNH95].
Conceptual Architecture	A conceptual architecture view describes a model of the organization of the system which is shared among developers. In many projects a conceptual architecture is agreed upon, such as a view of networking software as a layered system, or a typical architecture for a compiler. This view sometimes mirrors the *development* view described in [Kru95]
Feature	A feature view [Hon98] groups together code elements which cooperate to provide a certain functionality, describing as well interaction between features. Such a view is related to UML *use cases* [BRJ99] and to the use of *scenarios* in [Kru95].
Development	A development view [Kru95] focuses on the organization of the software into subsystems or modules. Represented as *package* or *subsystem* view in UML [BRJ99], *module* or *code architecture* in [SNH95]. A development view can be further subdivided into views which represent different aspects, such as an ownership view, traceability view and customization view [Hon98].
Process	A process view [Kru95] is a model of the concurrent tasks in a system and their communication. It is used as well for looking at nonfunctional requirements such as performance and fault-tolerance. Corresponds to the *execution architecture* in [SNH95]
Physical	A physical view [Kru95] describes the distribution of the software on machines and processors. In UML this is called the *deployment* view. Soni et al. [SNH95] include this aspect under *execution architecture*.

Table 2.1: Views addressing main development concerns

Each of these views can be seen as a set of components (*e.g.*, processes in the process view, domain concepts in the domain model view) and the connectors between them (*e.g.*, process communication abstractions in the process view). The kind of task facing a developer determines the kind of view of the system that he or she is interested in.

2.1.4 General Approaches to Design Recovery

Several cognitive models have been developed to describe the strategies used by programmers to understand programs [vMV95]. These can be roughly categorized into *bottom-up* strategies and *top-down* strategies. In bottom-up strategies programmers chunk low-level code artifacts into higher-level abstractions, whereas in top-down strategies programmers hypothesize an initial model reflecting their current understanding, then analyze the code to confirm or reject the hypothesis and use the information to revise their model. Studies show that programmers typically switch between these two strategies in trying to understand the software, and that the kinds of tools available to the programmers influences the kind of strategy used [SWM97]. Reverse engineering techniques and tools all seek to represent the software at a higher level than that of the information which is directly

extracted from the code, but vary in the degree to which they support these two basic model building strategies. Tools which support only bottom-up strategies do not allow an engineer to articulate any expectations about the structure of the system: higher-level abstractions are discovered automatically, usually by clustering techniques. Top-down techniques are more expectation-driven: the engineer describes some of the expectations about the structure or about the kinds of views expected, and uses the tool to confirm or reject those expectations, and to revise his or her model. Many tools support both strategies: they offer some automatic clustering or concept recognition to suggest higher-level abstractions, while also allowing an engineer to create such abstractions manually.

Visualization tools are also used to recover design models, but do not usually allow for the explicit construction of higher-level abstractions or the description of the expected models. Instead they display the low-level software elements (*e.g.*, classes, calls between objects) and rely on visual cues to aid an engineer in recognizing patterns and clusters and to help to reject or confirm *mental* models.

Metrics are useful in indicating problem areas in the code and can help an engineer to focus the design recovery activity [FP96][Mar98][DD99], but they do not aim at recovering a model from the software. Visualization tools can also display metrics associated with software structures [DDL99] thereby coming closer to recovering a model of the software.

Below we survey some of the approaches and tools for design recovery. Though we have categorized these approaches as bottom-up, top-down and visualization approaches, these should be understood as loose categories, since many tools contain elements of each.

Bottom-up approaches. The Rigi environment [WTMS95] allows users to build high-level abstractions of code by repeatedly clustering lower-level elements into abstractions through an interactive editor which manipulates program representations. The clustering can be done automatically, based on cohesion and coupling criteria, or it can be specified by the user.

In [HYR96] *recognizers* are used to detect C/Unix idioms associated with specific architectural styles. The recognizers uncover relationships such as spawning, client-server and implicit invocation and thus make it possible to generate views which show these relationships between large program abstractions such as files or tasks. A similar approach is presented in [FTAM96], who also use recognizers to identify language-specific clichés.

Dali [KC98] is a workbench which integrates several extraction tools and allows for the combination of the views obtained from these different sources. Shimba [SKM01] is a reverse engineering environment which integrates tools for static and for dynamic analysis of software and provides mechanisms for reducing the information in a static view as a function of a dynamic view, and *vice versa*.

Top-down approaches. The Reflexion Model approach proposed by [MN97] allows an engineer to define a high-level model of the software system and a mapping of the source-code onto this model. A *reflexion model* is then computed which shows how close the high-level model comes to describing the source code. This allows the engineer to iteratively refine the high-level model until it better reflects the implementation.

The Reflexion Model approach is similar to a category of tools to check design conformance. Sefika [Sef96] uses Prolog clauses to describe a high-level model and check to see if the code complies with this model. This approach has also been used to check for the presence of design patterns [KP96], for conformance to design guidelines [Wuy98][Ciu99], and to codify software architectures [MWD99]. The Reflexion Model approach, however, not only reports whether or not the code matches the high-level model, but also reports the extent of deviation of the source code from the model.

Knowledge-based approaches to design recovery [DBSB91] allow an engineer to pose high-level queries relating domain concepts to particular code features, but these approaches operate on a source model which includes not only information extracted from the source code, but also information elicited from programmers and other experts. Hy+ [CMR92] is another query-based approach which uses a visual grammar to reason about representations of the code.

Visualization tools. Most reverse engineering tools rely on some kind of visual notation for displaying the high-level models recovered from the code. We distinguish, however, between these kinds of visual displays and those of visualization tools. Visualization tools do not display high-level models – they display the basic elements extracted from the code (either statically or dynamically) and use visual techniques [War00] such as layout algorithms, scaling and colour schemes to suggest patterns and groupings of these basic elements. There is a large range of approaches for the visualization of software [BE96]. Here we mention a few tools that can be applied specifically to object-oriented systems.

CodeCrawler [DDL99], for example, displays object-oriented metrics as node attributes in a range of graph layouts. Visualization techniques have been used especially for displaying dynamic information, to handle the great volume of information generated through program execution. The *information mural* technique used in the ISVis tool [JR97] allows the user to visualize a complete trace on one screen and to get a feeling for the patterns in the trace. Jinsight [PLVW98] displays interaction diagrams and makes extensive use of color to highlight similar patterns and to suggest relationships of elements in the program trace. Animation is another technique used to deal with dynamic information [WMFB+98].

2.2 Design Recovery for Object-Oriented Software

Whereas in the previous section we discussed design recovery strategies in general, we will now survey the recovery of design models for object-oriented software in particular. Though the main challenges in design recovery remain basically the same for object-oriented software as they are for procedural systems, some of the approaches taken are particular to object-oriented systems.

We start by reviewing some common models and notations used to describe object-oriented software. The two main kinds of design models used to describe object-oriented software are static models describing components and their structural relationships and dynamic models which describe the behavioral aspect of an application. The recovery of

these two kinds of models is discussed in Section 2.2.2 and Section 2.2.3. Finally, Section 2.2.4 presents the particular case of detecting design patterns in code. Design patterns have both a structural and a behavioral aspect and so their recovery does not fall under the previous categories.

2.2.1 Describing Object-Oriented Software Design

Software design is expressed through models which describe a view of the design. When it comes to models, there are two separate issues: content and notation – semantics and syntax. The first is the issue of what kind of information a model contains (what view it models), while the second is the issue of what kind of notation we use for expressing this information. The Unified Modeling Language (UML) [BRJ99] has now become an industrial standard for describing object-oriented design; it proposes a modeling language (a notation) as well as a set of model types (information content) to describe different aspects of a software system. The same information content can, however, be expressed using a different notation (*e.g.*, an ADL, another visual formalism), and there are model types (views) which UML cannot express very well (*e.g.*, process model).

Commonly used models. Below we list models which are commonly used to describe and document the design of object-oriented software, in terms of their information content. The issue of the notation used to express these is discussed later on.

- **Application domain model.** This is a description of the main elements of the problem domain and how they relate to each other. It is the result of a domain analysis and is usually a first step in object-oriented design: identifying domain concepts and making the relationship between them explicit.

- **Use-cases.** In object-oriented design these document the behavior of the system from the users' point of view, with each use case representing a coherent unit of functionality which the system supports. They are used to elucidate requirements and are valuable in project planning [Fow97].

- **Static models.** Static models describe classes and their relationships.

- **Dynamic models.** Dynamic models describe how typical objects interact at runtime to assure a certain functionality.

- **Models of a single object.** These describe the behavior of an object as a function of its internal state.

- **Process models.** Process models describe how execution threads are assigned and how processes communicate.

- **Packages and subsystems models.** These models describe how the software is packaged.

- **Deployment models.** Deployment models describe how the software is physically distributed on machines.

- **Design patterns.** Design patterns [GHJV95] have also been used to document software systems [BJ94][OQC97]. The strength of design patterns for program understanding is that they tap into familiar metaphors - and so their names give us a handle on understanding them before we delve into the details of their structure and collaborations. Such metaphors can succinctly describe components, the relationships between them, and at the same time provide us with familiar scenarios for their behavior.

- **Role-based design.** Role-based design is a way to decompose an object-oriented application into a set of collaborations between classes, where each collaboration encapsulates an aspect of the application. A class can participate in several collaborations, playing a distinct role in each. The participants are described by the role they play in the collaboration and the collaboration itself is described by how these roles interact [Ree96][RG98] .

These models are used to describe the views listed in Table 2.1.3. The application domain model corresponds to a domain model view of the software, and sometimes also to the conceptual architecture. The package and subsystem model corresponds to the development view, the process model to the process view, the deployment model to the physical view and use-cases to the feature view. Static models describing classes and their relationships may be used to represent a domain model, a conceptual architecture or a development view, whereas dynamic models may be used to represent feature views.

Notations. Below we briefly consider notations which can be used to express the models listed above.

- **UML.** The Unified Modeling Language (UML) can be used to describe most of the models listed above: domain models, use-cases, static models of classes and their relationships and dynamic models of object interaction. The UML's repertoire of models includes also activity diagrams, state diagrams, package diagrams and deployment diagrams, which will not be discussed here. The UML has limited support for expressing a process view of the design. The structure and collaborations of design patterns can be expressed using UML, though the intent, tradeoffs and other important documentation elements of a pattern remain elusive to any kind of shorthand notation. UML can be adapted to express role-based designs.

  Recently, UML is being studied for its suitability for the expression of general architectural views [Hil99], in the context of work on an IEEE *Draft Recommended Practice for Architectural description* [IEE99].

- **Architecture Description Languages.** In the last few years there has been a surge of work on architectural description languages (ADLs) [SG96][MT97][Ach02]. ADLs can be used to describe the high-level design or architecture of a system

in terms of components, connectors and their configuration independent of the programming-language which will be used for the implementation. As such, they are not specific for describing object-oriented systems. There is yet no consensus on what aspects of an architecture ADLs should model, and there is great disparity in the goals and uses of different ADLs. Some ADLs concentrate on modeling a process view of the architecture [All97], while others are used to describe particular architectural styles [MORT96][MDEK95]. Most ADLs are based on a formal semantic theory, but few of these languages allow a refinement from the architectural description to the implementation.

Since ADLs have so far been used only in a research setting and are in general not traceable to the implementation we will not consider their application in the context of reverse engineering. We will look here only at the recovery of design models which are currently used in object-oriented software development.

- **Other notations.** Many reverse engineering tools use their own visual notation to display models which correspond roughly to those listed above.

2.2.2 Recovering Static Models

Static models of object-oriented software are usually described by UML class diagrams. The notation of class diagram include the notions of class, with its operations and attributes, and two kinds of relationships, generalization and association, between these classes. Class diagrams, however, can be interpreted at different levels with respect to implementation classes. Such levels of interpretation, though not part of UML, are essential for avoiding confusion about the use of such diagrams. Fowler [Fow97] calls these levels of interpretation *perspectives*:

Conceptual. The conceptual perspective treats the UML class diagram as representing important domain concepts. The classes in such a diagram will be related to implementation classes, but there may not be a direct mapping. From this perspective, the attributes describe a basic property of the domain concept, the operations describe the principal functionality of the domain concept, and the associations describe conceptual relationships between these domain concepts.

Specification. The specification perspective focuses on the interfaces of the software. In this case a UML class represents a type, rather than an implementation class, and this type may have many implementations. From this perspective a class attribute means that this type must propose a way to query and to set the value of the attribute. An operation is interpreted as a public method, and an association between two classes represents a responsibility for navigating and for updating the relationship, though how this is in fact implemented is not specified.

Implementation. In the implementation perspective UML classes are taken to represent implementation classes. Attributes and operations then represent implementation

attributes and methods. The presence of an association between two classes means that there is some way to navigate from one to the other, either through a pointer, or attribute (this is also difficult to interpret).

It is important to be clear about how we are using UML class diagrams in order to make the right kind of interpretation. Fowler discusses perspectives in the context of forward engineering. For reverse engineering, the problem is more acute: we must decide on how to interpret code structures in order to map them to UML constructs of the particular perspective we are interested in. In addition to the source code we may have the running system as well, some documentation, users and developers which can provide cues on how to recover a perspective of the software.

In general, we are likely to be interested in the higher level perspectives, rather than in the implementation perspective. Consider the problem of recovering a conceptual perspective of the system. How can we distinguish between the important domain concepts and the less important implementation level information? How can we recover a domain concept which has been implemented by several classes in the code?

Even if we interpret a UML class diagram in the implementation perspective, the elements of the model do not have a one-to-one mapping to code [DDT99]. Though UML classes can be mapped to implementation classes, with UML operations mapping to the methods, and attributes mapping to instance variables, the case for the relationships of association and generalization is not straightforward. UML is a modeling language: it does not tell you how to implement the relationships in the model. As a consequence, there can be several implementations for the same modeling concept.

Problems. We summarize below the problems faced in recovering class diagrams from code:

- **Hard to map from programming language to UML.** The concepts of a programming language are not the same as those in the UML, or they can have different implementations.

  Different semantics in UML and programming languages. In a programming language we have subclassing based inheritance, whereas the UML generalization relationship is a subtyping relationship. In a similar vein, UML aggregation and composition relationships may be implemented identically in the code, so we cannot distinguish between these two kinds of relationships.

  Language-dependent interpretation. C++, for example, distinguishes between private and public methods whereas in Smalltalk all methods are public. So just by looking at the methods we cannot deduce the public interface of a class.

  Dynamically typed languages. For dynamically typed languages it is hard to derive associations from the code.

- **Too much information.** Even if we make explicit the mapping from our programming language to UML in order to arrive at a class diagram, this will be an implementation perspective of the code and contain too much information. In general,

we will need some heuristics to decide what kind of information to ignore in order to arrive at a model which is not overloaded with irrelevant information.

- **Granularity is too fine.** Several classes may represent one important domain concept. How can we group implementation level information to identify these concepts?

Existing Solutions. Tools like Sniff+ [Tak96] and CIA++ [Gra92] have been developed to support object-oriented application understanding based on static information and can display classes and relationships between them. Their use for design recovery remains limited, however, because of the difficulty of filtering relevant from irrelevant information in the great mass of data extracted. Similarly, commercial tools which can extract UML class diagrams from code [Rat98] produce diagrams which give an implementation perspective of the software, crowded with details which may not be of interest to the user. Heuristics are required to prune such diagrams in order to arrive at representations which present interesting information succinctly.

Rose/Architect [EK99] is one step in this direction. It is a tool prototype which uses simple structural reduction rules to remove extraneous information from UML class diagrams. Furthermore, it allows for the logical grouping of classes into 'planes', representing classes in a certain view, and for the projection of the class diagram through this plane, or view. Racz and Koskimies [RK99] describe a tool which compresses UML class diagrams by exploiting abstraction relationships such as inheritance, aggregation and implementation.

Finally, approaches such as the Rigi environement [WTMS95] and the Reflexion Model [MN97] operate on any set of relations, and so can be used on information extracted from the code using parsing tools. These tools produce high-level views of the software which do not have specific object-oriented design semantics.

2.2.3 Recovering Dynamic Models

Dynamic models are usually expressed using UML interaction diagrams. Interaction diagrams provide a notation for describing how objects interact to perform a task, and are typically used to model how the software realizes a particular use case, or a scenario which is part of the use case.

There are two kinds of interaction diagrams: collaboration diagrams and sequence diagrams. Sequence diagrams show the activity of each participating object as a vertical line, and the message exchanges between objects as horizontal lines. Collaboration diagrams depict objects as rectangles and represent message exchanges between objects as directed links. The sequence and the nesting of the messages is described by numbering the links with a nested numbering scheme. Interaction diagrams are intended to describe typical execution scenarios. They also provide special notations for expressing conditionality, iteration and multiple threads of control, though using these features often results in overly complex diagrams which are hard to decipher [Fow97].

At a design level an interaction diagram models an important scenario without necessarily representing all the details of message exchange at runtime. The interaction diagram is an abstraction of the collaboration of objects playing specific roles at runtime.

Using static information. In dynamically-typed languages, identifying collaborating classes is difficult from static information only. But even in statically typed languages, where we can identify collaborating classes through the attribute and method argument types, we do not have the control flow information necessary to understand how the classes collaborate in a particular scenario. Dynamic binding and polymorphism make it difficult to obtain precise control flow information using static tools which work for procedural languages, though some slicing tools have recently been extended to handle features of statically typed object-oriented languages [LH96][TAFM97].

Using dynamic information. To obtain interaction diagrams from the execution of a program we must be able to instrument the code in order to trace all the message sends which occur. It is possible then to generate a sequence diagram for the whole trace, but such a sequence diagram is huge. As an example, an Interaction Diagram [BFJR98] display was obtained by instrumenting the methods of 28 classes of the HotDraw [BJ94] graphical editor framework then running a simple scenario on the sample editor, (create a rectangle and a bezier figure, group them together and then ungroup them). This resulted in a trace in which 224 objects (vertical lines) appear and 2,6048 method invocations (horizontal lines) occur. Whereas the intention of UML sequence diagrams as a modeling notation is to succinctly describe a collaboration, at the right level of detail and granularity for communicating the behavior of objects, sequence diagrams generated by tracing message sends are nothing but succinct and include much uninteresting information. Collaboration diagrams are compacter since they show no time line, but they become crowded and unreadable when used to represent all message exchanges in a program execution.

Problems. We summarize below the problems faced in recovering interaction diagrams from code:

- **Static information is not enough.** Detecting and deciphering interactions of objects in the source code is not easy: polymorphism makes it difficult to determine which method is actually executed at runtime, and inheritance means that each object in a running system exhibits behavior which is defined not only in its class, but also in each of its superclasses. This difficulty is further aggravated in the case of dynamically typed languages like Smalltalk where no type definition is available at compile time.

- **Dynamic information is too much.** Program tracing results in a great volume of low-level information from which we must sift the details relevant for our investigation. Here, the problems of focus, granularity and generalization must be addressed:

  Focus. In extracting an interaction diagram, we generally want to focus only on one small aspect of the program behavior. The problem here is that of locating the aspect that interests us inside the program trace.

Granularity. Trace level information gives us the interaction of objects. But we are often interested in the interaction between several logical groupings of objects, rather than in all the low-level method invocations.

Generalization. Even if we have located an interaction of interest in the program trace, and have represented it at a coarse granularity, we would like to abstract from all such instances of a collaboration to a more general representation of a collaboration with the same semantics.

Existing Solutions. When using dynamic information to recover behavioral models, the volume of information generated through program tracing is one of the main challenges. This challenge has been addressed by a variety of tools using strategies along three different axes:

- **Summarization through Metrics.** In this strategy information collected at runtime is summarized in some compact way, using a metric or some other statistical measure, for example, the frequency of calls or the number of objects. This measure is usually visually rendered [PHKV93][WMFB+98][SSC96].

- **Filtering and Clustering.** In this strategy the amount of information to be displayed or analyzed is reduced using filtering and clustering techniques [JR97][WMFB+98][LN95a][SKM01], thus addressing the problem of focus and granularity discussed above.

- **Visualization Techniques.** A variety of techniques permit the display of large volumes of information. The time element in a screen display can be removed through *animation* [PHKV93][WMFB+98][SSC96]. The display can be simplified by *elision* of some of the information which remains accessible through hyperlinks [KM96] or the use of other visual mechanisms [PLVW98]. The information can be compressed to be displayed as a *visual pattern*, as done in the Information Mural technique [JR97].

As can be seen from the list above, many tools integrate two or three of these strategies. Most of the tools, however, rely to some degree on visualization techniques. That is, they do not display high-level models, but rely on visual techniques and cues to suggest patterns and grouping of basic low-level elements.

Walker et al. [WMFB+98], for example, use program animation techniques to display the number of objects involved in the execution, and the interaction between them through user-defined high-level models. Their tool thus uses all three strategies: animation as a visualization technique, the summarization strategy for showing the number of live objects as a histogram, and reduction of the information space by allowing the user to cluster together code elements to create a high-level model.

De Pauw et al. [PHKV93] use both animation and summarization. They propose three kinds of layouts for displaying the communication behavior of an application as it executes: spring layout, histogram layout and matrix layout. These layout are used to

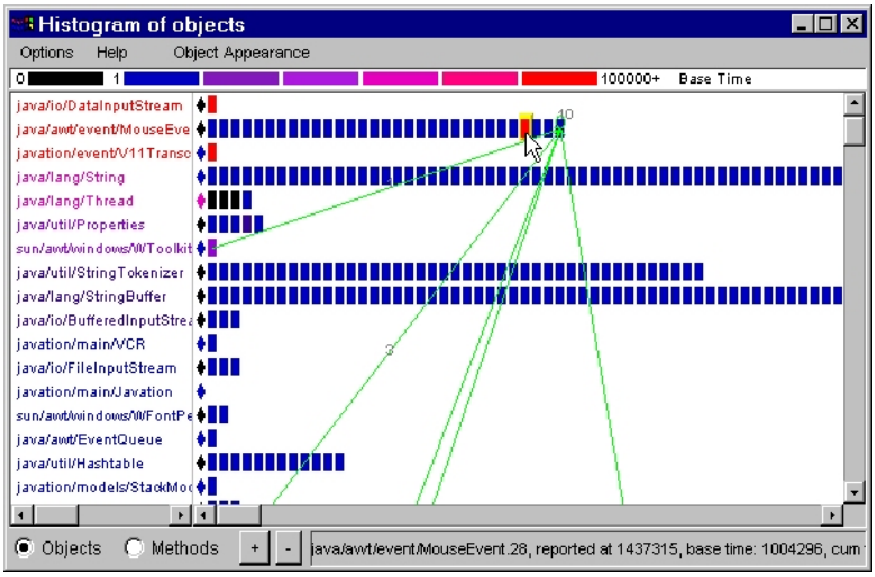

Figure 2.3: Histogram of objects graph of the Jinsight tool

produce seven kinds of graphs, a few of which are discussed below. The *inter-class call cluster* graph uses a spring layout. It shows class names as floating labels; as the program executes classes which communicate frequently gravitate towards each other, whereas those which communicate rarely repel each other. The histogram layout is used to display a *histogram of instances* showing the number of instances alive of each class, using a color scale to indicate the relative number of messages each instance has received so far. Figure 2.3[1] is an example of a histogram layout which uses color to indicate the level of activity of the objects and also displays the current communication between them. Matrix layouts are used for the *inter-class call matrix, allocation matrix, functions-instances matrix* graphs which show the relative frequency of class communications, the allocation relationship between classes and and the invocation of methods on instances, respectively.

The visualizations proposed by Sefika et al. [SSC96] are similar to the spring layout of [PHKV93]. [SSC96] also display static information in a similar layout for the purpose of comparing and contrasting information extracted statically with information obtained through program execution. The tools of [Sef96] can also display the interaction of architectural units (rather than classes, as in [PHKV93]), but to do this they depend on an instrumentation strategy which is particular to the domain of the application (the Choices operating system).

Several tools are based on a sequence diagram display. The Scene tool [KM96] uses hyperlinks to help a user navigate information which can not be displayed on one screen and also connects the sequence diagrams to the source code. SCED [KSTM98] displays message exchanges in a notation similar to sequence diagrams, and can also extract information on constraints, conditional branching and iteration. Shimba [SKM01] uses Rigi [MWT95] to create static abstractions so as to simplify the sequence diagram displayed

[1] from Jinsight pages http://www.research.ibm.com/jinsight/

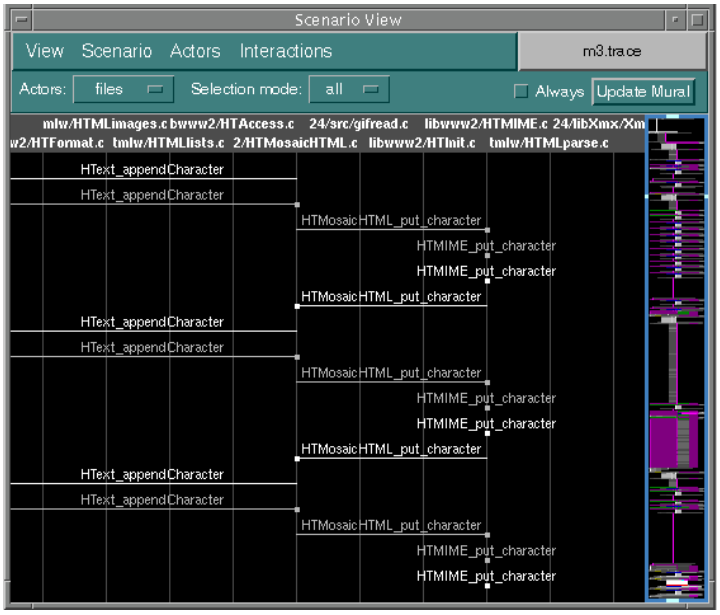

Figure 2.4: ISVis display

(using SCED) by reducing the number of vertical lines. It also offers pattern matching capabilities to detect similar sequences of message exchange.

ISVis [JR97][JSB97] displays sequence diagrams showing the communication between actors, where an actor is a clusterings of objects. As seen in Figure 2.4[2] an information mural display to the right of the sequence diagram shows the whole trace as a visual pattern and acts as a navigational aid through the more detailed sequence diagram. The ISVis tool uses filtering and clustering to reduce the amount of information to be displayed: the trace data can be edited to remove specific interactions, and actors in the trace scenario can be clustered together to produce a composite actor. This has the effect of reducing the number of horizontal lines (through filtering) and the number of vertical lines (through clustering) to be displayed. The tool also has features for replacing a low-level interaction (exchange of messages between actors) with a reference to a scenario – further reducing the number of horizontal lines to be displayed.

More recent work of De Pauw et al. on Ovation [PLVW98], whose features are now integrated into the Jinsight tool, experiments with sequence diagram based displays which allow an engineer to use a variety of elision mechanisms to control the amount and level of information to be displayed. The tool uses pattern matching to identify recurring interaction sequences, so that the visual manipulation can be applied to all occurrences of a pattern. As in ISVis [JR97], this work aims at reducing the conceptual overload for the user by identifying similar patterns in a trace.

Program Explorer [LN95a] offers simple sequence diagram and collaboration diagram like layouts intended for displaying a small part of a trace. In this tool static and dynamic information is reified as Prolog facts, allowing for the coupling of static and dynamic

[2] from ISVis User's Guide at http://www.cc.gatech.edu/morale/tools/isvis/isvis.html

information in checking for the presence of design patterns in code. The visualizations offered are, however, at the level of object interactions and require a good understanding of the application in order to be interpreted correctly.

Evaluation of Existing Approaches. The approaches discussed above all rely on visualization to convey information to the user. They can be divided into two categories: *discrete* approaches retain the individual run-time events and *summary* approaches display cumulative information. Summary approaches are good at indicating certain characteristics of an application, without giving exact information. De Pauw et al. [PHKV93], for example, present a catalogue of different kinds of displays together with a guidelines on how to interpret visual patterns in each display. "A program that exploits code inheritance extensively will show many blue squares over time" and "Bars that grow rapidly can reflect object creation in tight loops" are two examples of such guidelines. In contrast, discrete approaches are better at answering specific low-level questions, and seem to be very useful in debugging. Discrete approaches which incorporate pattern matching in order to generalize from a specific event sequence to a message exchange pattern [PLVW98][JR97] are better at revealing overall behavior of an application, but their display is still verbose, making it cumbersome to find the information of interest.

Reverse engineering approaches which operate on any set of binary relations can be used to build higher-level models using dynamic information instead of static information (*e.g.*, operating on object send relations), and so result in more succinct models. The Reflexion Model approach [MN97], for example, would allow a user to view the message send relation through a high-level model defined by the user. Such an approach is, however, not good at answering low-level questions about object interactions.

Role models [Ree96][RG98] and collaboration-based designs provide a succinct way to express the interaction of classes, and to decompose an application into significant collaborations. In this sense they span two kinds of levels: one level is the decomposition of an application into coarse units (the collaborations), the lower level is the one which describes the contents of the collaboration (how objects interact to ensure a certain functionality). Except for [Hon98], who reports work on the extraction of collaboration contracts [Luc97] using full parse tree information, there is currently no published work on the recovery of roles and collaborations. The recovery of design patterns is related to the problem of recovering roles and collaborations, because design patterns represent a template of a collaboration. As such, design pattern detection is also an aid in understanding program behavior.

2.2.4 A Case of Design Recovery: Detecting Design Patterns

Because the recovery of design patterns from code incorporates aspects of the recovery of both static and dynamic models, it is interesting here to look at the challenges of this problem. Whereas UML class diagrams and interaction diagrams only record the design without giving us insight into the rationale behind it, it has been argued that design patterns [GHJV95] [BMR+96] give us insight into the 'why' of the design. Because each design pattern presents a solution to a specific design problem, describing the design pat-

terns in a framework can give readers an understanding of the problem that the current design is intended to solve [BJ94].

Brown [Bro96] discusses the difficulties in detecting design patterns in code. In order for a pattern to be detectable its template solution (structure) must be *distinctive* and *unambiguous*. Distinctive means that the implementation must be due the use of this pattern and not some other pattern. Unambiguous means that there is a unique way (or at least only a few ways) to represent this pattern in the code.

Few patterns satisfy this criteria: consider for example the Strategy, State and Command patterns. Strategy and State have almost identical code structures, and a substructure of Command is similar to this structure, shown in Figure 2.5. In other words, the template solution for these patterns is not distinctive: by looking at the code we would not be able to tell which pattern has been applied without a deeper understanding of the domain and the intended functionality.

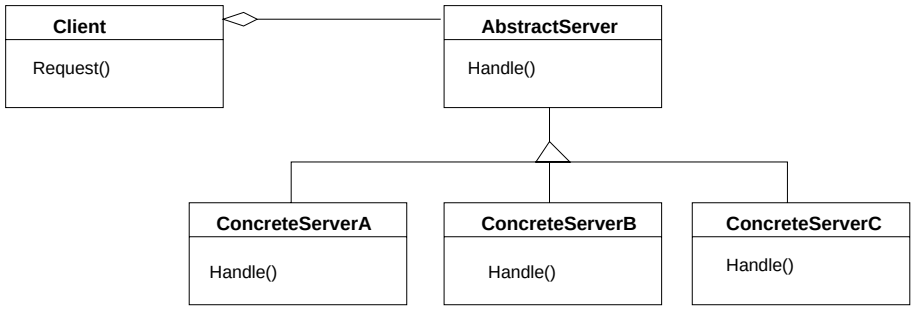

Figure 2.5: Structure occurring in Strategy, State and Command patterns

Many patterns can also be implemented in different ways and are implemented differently in different programming languages, making the problem of their detection difficult. Some simple patterns like Composite and Template can more easily be recovered. These are patterns whose structure is simple and whose intent is so general that we can rarely make a mistake in interpreting it. But detecting Composite and Template patterns in large code may not be of much help in obtaining an interesting design view of the code.

Detecting design patterns in the code, therefore, requires enough understanding of the code to be able to make some interpretation about the intent of specific pattern structures.

Problems. We summarize the problems as follows:

- **Ambiguity.** Several design patterns have a similar code structure.
- **Non-distinctiveness.** A design pattern may be implemented in different ways.
- **Crowded model.** Where design patterns overlap this may be hard to untangle.
- **Incomplete model.** Extractable design patterns represent just a small part of the code.

Existing Solutions. Work on recovering design patterns in code ranges from approaches of automatic detection [KP96] to integrating design pattern recovery in an interactive reverse engineering environment [KSRP99]. [KP96] use Prolog to look for structural design patterns (Adapter, Bridge, Composite, Decorator and Proxy) in C++ code. They analyze C++ header files to extract design information, making specific assumptions about how aggregations and associations have been implemented in the code, and represent this information as Prolog facts. Each pattern is then described by a Prolog rule which chains a list of facts specifying necessary (but not sufficient) properties for the presence of the pattern in the code. In the experiments conducted most of the pattern instances in the code were found, but the number of false positives was high (ranging from 14% to 40% for any significant number of patterns found). In SOUL [Wuy01], a logic-programming language integrated into Smalltalk, rules are used to describe design patterns, using full parse tree information. In [Sef96] design patterns are also represented as Prolog rules, and conditions are specified for violation of the pattern as well as those for compliance to the pattern. [Bro96] make use of dynamic information for the detecting Composite, Decorator and Template Method patterns in Smalltalk applications.

In the SPOOL project [KSRP99] pattern detection is integrated in an interactive environment for design recovery. Patterns can be reified as 'abstract design components' which are stored in a design repository. The code repository is then queried to search the source model for this structure. The authors make a case for the visualization of patterns: "we claim that only the *visualization* of the implemented patterns in the *context* of the application at hand will make documentation with patterns truly effective, elucidate the rationale behind the framework's design and make the applied patterns more tangible and understandable". For each pattern description the authors posit one unique reference class. When a reference class is identified in the code, a bounding box is drawn around it on the screen. Classes in the code which are reference classes for several design components will have several bounding boxes and so will stand out on the display. This visualization allows a user to interpret the structure in its context. The authors discuss the recovery of Template Method, Factory Method and Bridge patterns, conceding that human insight is necessary in reliably detecting design patterns in code.

2.3 Our Work: Scope and Requirements

In surveying existing solutions to the extraction of behavioral models we see that most approaches based on dynamic information result in verbose displays of sequence diagrams. These approaches combine the sequence diagram displays with techniques to reduce the amount of information presented: visualization, navigation, filtering and clustering, in order to help developers arrive at the information of interest [KM96][PLVW98] [JR97][SKM01]. Approaches which summarize statistics from dynamic information [PHKV93], or use static information only [WTMS95][MN97] produce more succinct models, but they are limited in conveying the behavioral aspect of the software.

In our work we explored an alternative approach for the recovery of behavioral models. We enable a developer to view the dynamic information, without the time dimension,

through a model he or she has specified – a perspective. We demonstrate that an approach based on perspectives can overcome some of the limitations of current solutions for extracting behavioral models. Our contribution is to add a new point in the spectrum of design recovery techniques by demonstrating a *lightweight* approach which extracts *compact* behavioral models.

In this section we distill a set of requirements against which we will later evaluate our work and contribution.

2.3.1 Scope of Our Work

We first clarify what we mean by *behavioral* models. We then make explicit what kinds of questions we do not aim to answer to delineate clearly the scope of our work.

Behavioral models. A behavioral model is a model which captures the dynamic aspects of a software system: how objects collaborate and cooperate in performing a task, and how responsibilities are distributed among these objects. This does not mean that the elements in a behavioral model are objects and the message sends between them. A behavioral model can abstract from the actual runtime interactions to illustrate relationships between components which are, however, anchored in the runtime events. Behavioral models should provide answers to the following kinds of questions:

- how do the main domain elements relate to each other?
- what parts of the software implement a specific feature?
- to which messages does an instance of a certain class respond?
- how many instances of a class are present at runtime?
- which objects are responsible for creating instances of a certain class?
- which class instances participate in the interaction resulting from the invocation of a particular method?
- what are variations on the way a method is executed?

Purpose. We do not aim to develop a debugging tool, nor a tool for coverage analysis, memory analysis or performance tuning. Our goal is to extract models which help a developer quickly gain the understanding necessary to perform maintenance tasks such as adding functionality, extracting a reusable components, detecting changes from a previous version or refactoring the code.

Other behavioral models. The UML [BRJ99] provides for four kinds of behavioral models: sequence diagrams, collaboration diagrams, state diagrams and activity diagrams. Sequence diagrams and collaboration diagrams have been discussed in Section 2.2.3. Our work considers mostly the behavioral information conveyed by such models. State diagrams describe the state machine of a single object responding to external stimuli. In our

work we do not consider the extraction of state diagrams [KSTM98]. Activity diagrams are a variant on state diagrams and are used to describe how activities are coordinated. They are particularly useful for modeling dependencies between several tasks an operation has to achieve. In our work we do not address the recovery or modeling of activity diagrams.

Language model. We restrict ourselves to single-threaded software applications, so we do not handle the notion of process in our model. The tools we have developed currently operate on Smalltalk programs – but we argue in the dissertation that the concepts and techniques we apply are general enough to be applicable to other class-based object-oriented languages.

2.3.2 Requirements for the Recovery of Behavioral Models

We have claimed that current approaches suffer from the following problems: they produce initial verbose views with too much low-level information which makes it hard for a developer to focus on the information of interest. In order to provide focus, they then rely on visualization and navigation techniques. By contrast, we aim to provide focus earlier, without showing all low-level information. Here we derive a set of requirements for our approach:

Lightweight. We would like a recovery approach which does not require a high investment for the extraction of information from the code. We also want to be able to quickly obtain information which guides a developer in the recovery process. We therefore want an approach based on a *lightweight information model* – an information model which is easy to populate for any object-oriented language, and one with *simple view specification* with which a developer can quickly obtain an initial useful view of this information.

Succinct views. We do not want to present the developer with lots of low-level information which requires visualization or navigation techniques to answer questions. Rather, we would like compact views which enable a developer to focus on interesting information at the appropriate level.

We want, of course, to extract views that really aid in program maintenance tasks. Demonstrating that our approach indeed aids developers in carrying out a variety of software maintenance tasks would require industrial strength tools and a set of controlled field trials. Instead, we will evaluate the usefulness of our approach in a different way. We aim to support a design recovery process which is consistent with the research hypothesis we stated in Chapter 1. We then evaluate our approach and case studies with respect to how they meet these criteria.

As discussed in Chapter 1, our hypothesis about the nature of design recovery process is that it is iterative and task-specific. This means that we want to put the developer in charge of the process. Also, since we cannot foresee all the maintenance tasks a developer might be faced with, we want to support the recovery of a large range of views.

Developer is in charge. We want the developer to be in charge of the recovery. First

the developer should be guiding the process, rather than the recovery tool determining the process – *developer guides the process*. Second, we want *extensible view specification* so that the developer can specify the kinds of views that he or she is interested in, rather than be restricted to a limited set of views dictated by the tool.

Supports a range of views. To answer the *behavioral questions* we have listed above we want to be able to generate *high-level views* where the developer can create high-level abstractions, while not loosing the possibility of obtaining *finer-grained views* which show what is happening at the level of object interactions.

2.4 Conclusions

In this chapter we analyzed the problems of design recovery in general and of design recovery for object-oriented systems in particular. We surveyed current approaches and techniques and looked in depth at approaches for using dynamic information to understand program behavior.

We then circumscribed the scope of our work on using dynamic information in design recovery. Finally, we derived a set of eight requirements against which we will evaluate our work:

1. Lightweight information model
2. Simple view specification
3. Succinct views
4. Developer guides the process
5. Extensible view specification
6. Behavioral views
7. High-level views
8. Low-level views

3

The Iterative Query-Based Approach

This dissertation presents an iterative, query-based approach for the design recovery of behavioral models from dynamic and static information. The goal of this chapter is to introduce the basic elements of the approach, and to describe briefly the two applications developed: concept view recovery and collaboration view recovery. The approach is based on an *iterative recovery process* which allows a user to *query* a *source model* representing information about a software system.

The iterative nature of the design recovery process is presented in Section 3.1, along with preliminary concepts and terminology. In Section 3.2 we describe how the source model is represented using a meta-model for static and for dynamic information. In Section 3.3 we then gives an overview of the two applications of our approach.

3.1 The Iterative Process of Design Recovery

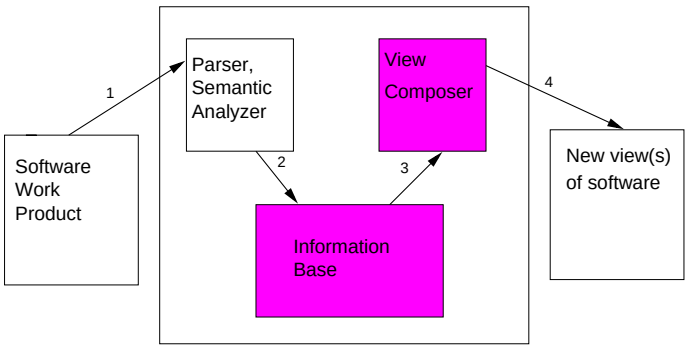

Figure 3.1: Basic tool architecture (after [CC90])

Chikofsky and Cross [CC90] observe that most tools for reverse engineering, restructuring and reengineering use the same basic architecture. Figure 3.1 shows this basic architecture with the arrows representing the flow of data. Though this basic architecture is indeed satisfied by most reverse engineering tools, what is not illustrated in such a schema is the actual design recovery process of a user interacting with a tool. Whereas

the steps illustrated by arrows 1 and 2 are typically only done once for a particular software product, the steps shown by arrows 3 and 4 may be repeated several times until an appropriate view is arrived at, or until the views generated so far answer the question of the user. Figure 3.2 adds this iterative cycle to the basic process shown in Figure 3.1.

The user. Typically, the user of the tool is a developer or maintainer of the code who must answer some questions about the code in order to carry out a maintenance task. The maintenance task might involve, for example, adding functionality to the software system, refactoring it to remove code duplication, or making changes to allow the system to run in a distributed environment. The developer interacts with the tool by querying, focusing on the parts of the code that must be better understood. As a function of the response to the query the developer launches a new query, obtaining a new response and so on. This enables the developer to steer the recovery process using the tool rather than let the tool dictate the kinds of views which will be extracted.

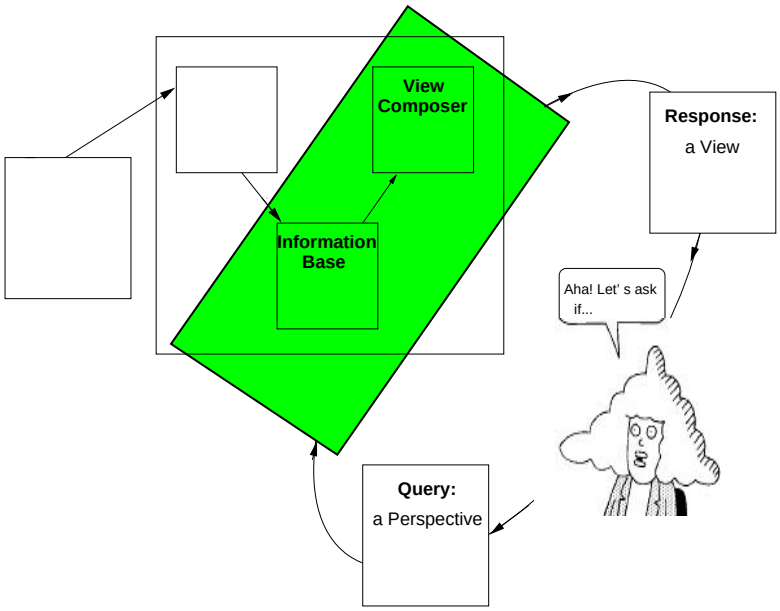

Figure 3.2: The iterative process of using a reverse engineering tool

The iterative process. It has been observed that without human guidance the process of design recovery gives poor results [MN97], and that automatic techniques often result in models whose abstractions are not recognized by the developers [WTMS95]. Our own experience with reverse engineering tools corroborates this observation. It is important that developers be able to steer the process by specifying abstractions which match their own *mental model* of the software.

We do not believe, therefore, in a purely automatic extraction of design artifacts, but rather a process guided by the developer or maintainer of the software. Typically, a developer approaching the code has a specific question in mind – asking something like “How is this task achieved?” rather than “how does everything work in this application?” – and this question steers the recovery process.

Since the maintenance questions a developer has determine the kinds of views that will be recovered there is no unique path in navigating through this iterative process using a tool. Guidelines, however, can be elaborated on the kind of queries which are useful for obtaining information as a function of the goal of the recovery. This kind of methodology can, for example, be described by a system of patterns [DDN02].

The iterative cycle we describe here is characteristic of the use of many design recovery tools. Note that a query can take many forms, depending on the tool. In CodeCrawler [DDL99], for example, a tool which combines metrics and visualization, a query corresponds to the user requesting a certain kind of graph for displaying specific metrics. The resulting display is the response to the query. The information obtained from the display is then used to decide about the next graph to request [DL01]. For the Reflexion Model tool [MN97] all queries take the form: "how close does my high-level model come to representing the source code?". Information about deviations from the model is used to alter the source-to-model mapping, and the query is then repeated.

3.1.1 Concepts and Terminology

Here we introduce some of the terms used throughout the dissertation and relate them to their role in the iterative process illustrated in Figure 3.2.

Source model. The source model is the base of information about the software system we are reverse engineering.

Perspective. A perspective is a specification of the aspects of the source model that we are interested in. A perspective is given as a declarative specification in terms of the software meta-model, or in terms of the source model itself. In the recovery process a perspective is the query of the developer.

View. A view of the source model consists of the elements of the source model which meet the declarative specification of a given perspective. In the recovery process a view is the response to a query.

A perspective is an *intension* – a definition of the characteristics or properties of the information we are interested in, whereas a view is its *extension* – the set of elements in the source model which satisfy these characteristics or properties.

The source model we use includes both static and dynamic information about the software system. The meta-model for our source model is presented in Section 3.2. The view composer shown in Figure 3.2 corresponds to a reverse engineering tool. The view composer creates a *view* of the source model by applying a *perspective* to the source model.

We use the analogy of photographing a scene to explain the notion of perspective and view and their relationship. The reverse engineering tool can be thought of as a camera, a perspective as a camera lens, a view as a photograph, and a source model as a scene in front of us. We mount a lens on the camera because we want a photograph of the scene

with a certain kind of effect. When we press the camera button we ‘apply’ the lens to the scene in front of us, and obtain a photograph.

Figure 3.3 illustrates this schematically. For example, perspective 1 is “inheritance relationship between classes” – view 1 shows this information for the classes in the source model. Perspective 2 is “calling relationship between inheritance hierarchies” – view 2 shows five bubbles, each corresponding to a grouping of classes belonging to the same inheritance hierarchy; the arrows between the bubbles correspond to the calling relationship between these inheritance hierarchies.

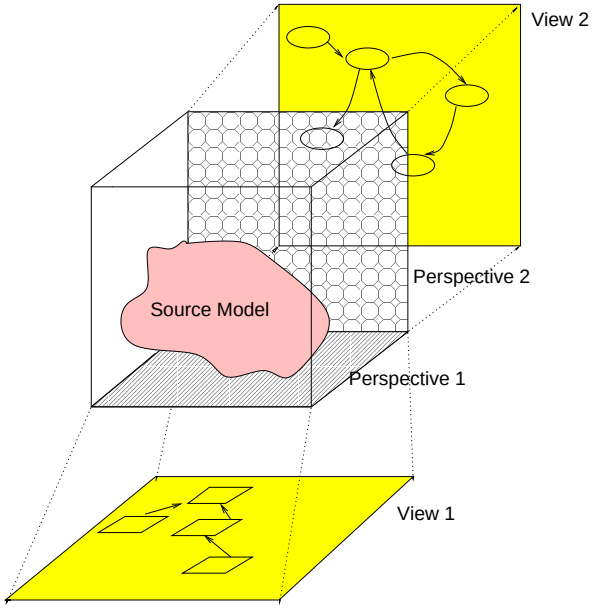

Figure 3.3: A *view* is the source model as seen through a *perspective*

The next sections of this chapter explain each of the elements illustrated in Figure 3.2. Section 3.2 presents the meta-model used to represent the source model (information base). Section 3.3 then makes more concrete the notions of perspective and view and introduces the two applications (view composers) we have developed.

3.2 The Source Model

The source model on which the approach operates is the base of information about the particular software system being studied. The source model includes both static and dynamic information about a software system. The static information models the static elements and relations in the object-oriented program, while the dynamic information models an actual execution of a program.

This section present the meta-models for representing this static and dynamic information. Since many design recovery techniques rely only on static information, we first argue for the use of dynamic information. Section 3.2.2 and Section 3.2.3 then present the modeling of static and dynamic information, respectively.

3.2.1 Why Dynamic Information?

In Section 2.2.3 we argued that static information is not sufficient for recovering behavioral models. While the volume of dynamic information generated through program execution is a challenge to design recovery, dynamic information presents several advantages for the recovery of behavioral models:

- **It acts like a program slice** with respect to control flow, since it limits the scope of our investigation to the particular scenario executed and so provides focus in the investigation.
- **It is always precise with respect to the executed scenario** since we know exactly which method has been invoked on which object.
- **It is easy to obtain compared to static control flow analysis** which proves difficult for large programs [WMH93].
- **It provides information not obtainable from any static analysis** such as the number of instances and the multiplicity of relationships between objects.

The main argument against the use of dynamic information is its incomplete coverage of the code. But this very property is also its advantage [Bal99]. In the context of reverse engineering and program understanding we do not always need complete information: we need information that helps us to form concepts about the software structure and helps us to formulate new hypotheses and questions. A program trace provides information about the behavior of the system in a scenario exercising a certain functionality, and so helps us to tie functionality to behavior.

3.2.2 Modeling Static Information

We model static information using FAMIX [DTD01][Tic01], a meta-model developed in the context of the FAMOOS[1] project. FAMIX provides for a language-independent extensible representation of object-oriented code and contains the required information for the reengineering tasks performed by a variety of tools. Since it is language independent it can be used to represent software systems in different implementation languages (C++, JAVA, Smalltalk, ADA). Its extensibility means that it can be extended to represent information that might be needed in future tools. The model can also be extended with language-specific features using language plug-ins.

The core FAMIX model is shown in Figure 3.4 and specifies the entities and relations that are extracted from source code. The model consists of the main object-oriented entities, namely Class, Method, Attribute and InheritanceDefinition. In addition there are two associations: Invocation and Access. An Invocation represents the definition of a method calling another method and an Access represents a method accessing an attribute. These

[1]The FAMOOS ESPRIT project investigated tools and techniques for transforming object-oriented legacy systems into frameworks.

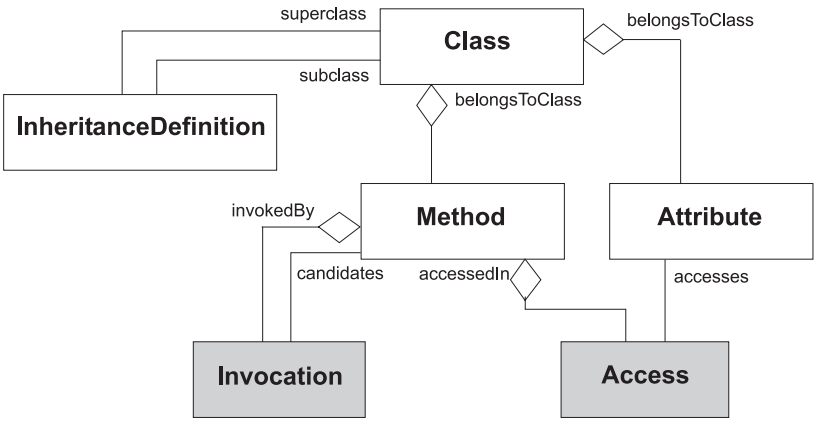

Figure 3.4: Core model of the static information

abstractions are needed for several reverse engineering and reengineering tasks. These associations are different than class associations in UML. The question as to why a new model has been defined instead of extending the UML model is discussed in [DDT99].

Table 3.1 shows the entities and associations of the core model and explains their attributes. Note that though the model is language-independent, how certain attributes are recognized in the source code is a language dependent issue. For example, in JAVA and C++ methods that have a hasClassScope attribute set to true are static methods - in Smalltalk these correspond to class methods, which are not exactly the same.

The two associations merit some further comments. The Access association represents the definition in source code of a behavioral entity (method or function) accessing a structural entity (an attribute, a local variable, an argument, a global variable). A separate access association is used to represent each access which occurs in the source code, even if the same structural entity is accessed several times in the body of the same method. The Invocation association represents the definition in source code of a behavioral entity (method or function) invoking another behavioral entity. As for the Access association, a separate Invocation association is used to represent each invocation in the source code. Note also that due to polymorphism, there exists at parse time a one-to-many relationship between the invocation and the actual entity invoked: a method, for instance, might be defined on a certain class, but at run-time be invoked on an instance of another class. This explains the presence of the base and the candidates attributes.

In the next section we describe the meta-model for representing dynamic information obtained from program execution.

3.2.3 Modeling Program Execution

The model for representing program execution is shown in Table 3.2. Program execution is modeled through an ordered list of entities which represent run-time events, of which there are basically two kinds:

Class	
name	the name of the class.
isAbstract	a predicate specifying whether the class is declared abstract.
belongsToPackage	the name of the package defining the scope of the class.
sourceAnchor	a reference to the location in source code.
Method	
name	the name of the method.
belongsToClass	a name referring to the class owning the method.
hasClassScope	a predicate telling whether the method has class scope or instance scope.
isAbstract	a predicate telling whether the method is declared abstract.
isConstructor	a predicate telling whether the method is a constructor.
sourceAnchor	a reference to the location in source code.
Attribute	
name	the name of the attribute.
belongsToClass	a name referring to the class owning the attribute.
accessControlQualifier	a string with a language dependent interpretation, that defines who is allowed to access the attribute.
hasClassScope	a predicate telling whether the attribute has class scope or instance scope.
InheritanceDefinition	
subclass	a name referring to the class that inherits.
superclass	a name referring to the class that is inherited from.
accessControlQualifier	a string with a language dependent interpretation, that defines how subclasses access their superclasses.
index	In languages with multiple inheritance, this is the position of the superclass in the list of superclasses.
Access	
accesses	a name referring to the variable being accessed.
accessedIn	a name referring to the method doing the access.
isAccessLValue	a predicate telling whether the value was accessed as a location value or a value on the left side of an assignment.
Invocation	
invokedBy	a name referring to the method doing the invocation.
invokes	a qualifier holding the signature of the method invoked.
base	the unique name of the class where the method is defined.
candidates	a multi-valued attribute holding a number of names of classes. Each name refers to a class that may be the actual one invoked at run-time.

Table 3.1: Attributes of the core FAMIX entities and associations. Attributes in bold font are mandatory, attributes in normal font are optional.

- **Send Event.** A send event represents the invocation of a method on an object by another object.
- **Return Event.** A return event represents the exit from an invocation.

InvocationTrace	
events	an ordered list of events. Each event is one of a Send, an Indirect-Send or a Return.
Send or **IndirectSend**	
time	global time.
sequenceNumber	integer giving the sequence number of the event.
senderClass	the name of the class of the sender object.
senderId	the identity of the sender object.
receiverClass	the name of the class of the receiver object.
receiverId	the identity of the receiver object.
invokedMethod	the name of the method invoked.
invokingMethod (for Indirect-Send)	the name of the method invoked by the sender object .
Return	
time	global time.
sequenceNumber	integer giving the sequence number of the event.

Table 3.2: The basic model of dynamic information

Direct and indirect sends. Since in collecting trace information not all methods are instrumented, there are two kinds of send events: direct sends and indirect sends. A direct send corresponds to the invocation of an instrumented method from within an instrumented method. An indirect send corresponds to the invocation of an instrumented method from within an *uninstrumented* method. (In the two cases of an uninstrumented method being invoked from within an instrumented or uninstrumented one no send event is recorded.)

Figure 3.5 illustrates the distinction between direct and indirect sends. The sequence diagram to the right shows a sequence of method invocations, with returns from invocations denoted as dashed arrows. The two traces to the left each show a trace corresponding to these events: Trace 1 corresponds to the case where all the methods were instrumented, and Trace 2 corresponds to the case where method m2 was not instrumented. The resulting indirect send indirectsend(a,m2,c,m3) is interpreted as follows: object a invoked method m2 on an unknown object and this resulted in the invocation of method m3 on object c.

Discussion. The model presented above treats an instance creation event as just another method invocation event. In Smalltalk classes are also run-time objects – each class being the sole instance of its metaclass. The creation of an instance of a class then corresponds to the invocation of a creation method on the class. However, the model can be extended to treat object creations as distinct from method invocations, when the instrumentation technique allows us to distinguish between the two kinds of events [Duc99].

Trace 1:
all methods instrumented

```
send(u,a,m1).
send(a,b,m2).
send(b,c,m3).
return.
return.
send(a,c,m4).
return.
return.
```

Trace 2:
method m2 not instrumented

```
send(u,a,m1).
indirectsend(a,m2,c,m3).
return.
send(a,c,m4).
return.
return.
```

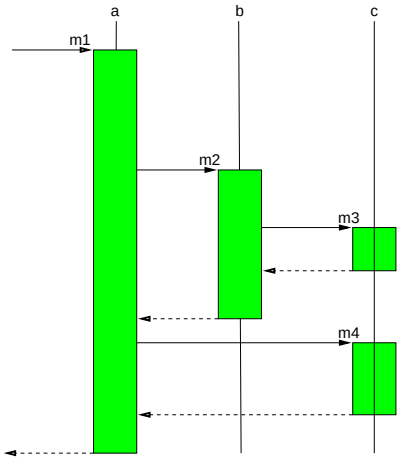

Figure 3.5: Direct and indirect sends

In our approach we are not concerned with global time since we do not seek to answer questions about performance and speed. The global time is required, however, by the interaction diagram tool we use to display a collaboration instance [BFJR98]. Some tools which display dynamic information also record events corresponding to a variable access as well as the value of the variable [LN95a]. Though this information is useful for debugging purposes, it was considered too fine-grained to help in building succinct behavioral models.

3.3 Using Perspectives to Recover Design Views

In this dissertation we present two applications of the query-based approach which produce views of a software system from perspectives: *Concept View Recovery* and *Collaboration View Recovery*. These two applications correspond to two different ways to decompose a software system.

In the first application, concept view recovery [RD99], we view the software system as a set of *components* and *connectors*. We name these views 'concept views' because components represent a range of concepts, and connectors represent a range of relations between these concepts. The perspectives we define, therefore, give semantics to the components (groupings or clustering of source model elements) and to the connectors (relations between the components). Since these components and relations can correspond to many different concepts, the views which result range from high-level views of the system, to views of fine granularity.

In the second application, collaboration view recovery [RD01], we view a software system as a collection of class collaborations. Whereas in the first application we often group together classes (usually corresponding to domain concepts), in the second application we group together sets of methods or partial interfaces of classes, which conceptually correspond to a collaboration to carry out a certain functionality. This kind of decomposition is close to a feature view of the software.

Why did we select these two kinds of decompositions as a basis for our perspectives? We began our work with concept view recovery. Varying the semantics of components and connectors allowed us to obtain a large repertoire of interesting perspectives and recover several useful behavioral views (*e.g.*, invocation relationship between groupings of classes, multiplicity of creation relationships between classes). We also used component-connector perspectives to see what was happening at the object level within a method invocation, but the resulting views were not satisfactory. First, it was hard to display all the method information on such a view, and second and more important, we could not generalize from one specific method invocation to the more abstract collaboration of which this invocation was just one instance. We therefore developed collaboration view recovery. Collaboration views span two kinds of levels: one level is the decomposition of the trace into coarse units (the collaborations), the lower level is the one which describes the contents of the collaboration (how objects interact to ensure a certain functionality).

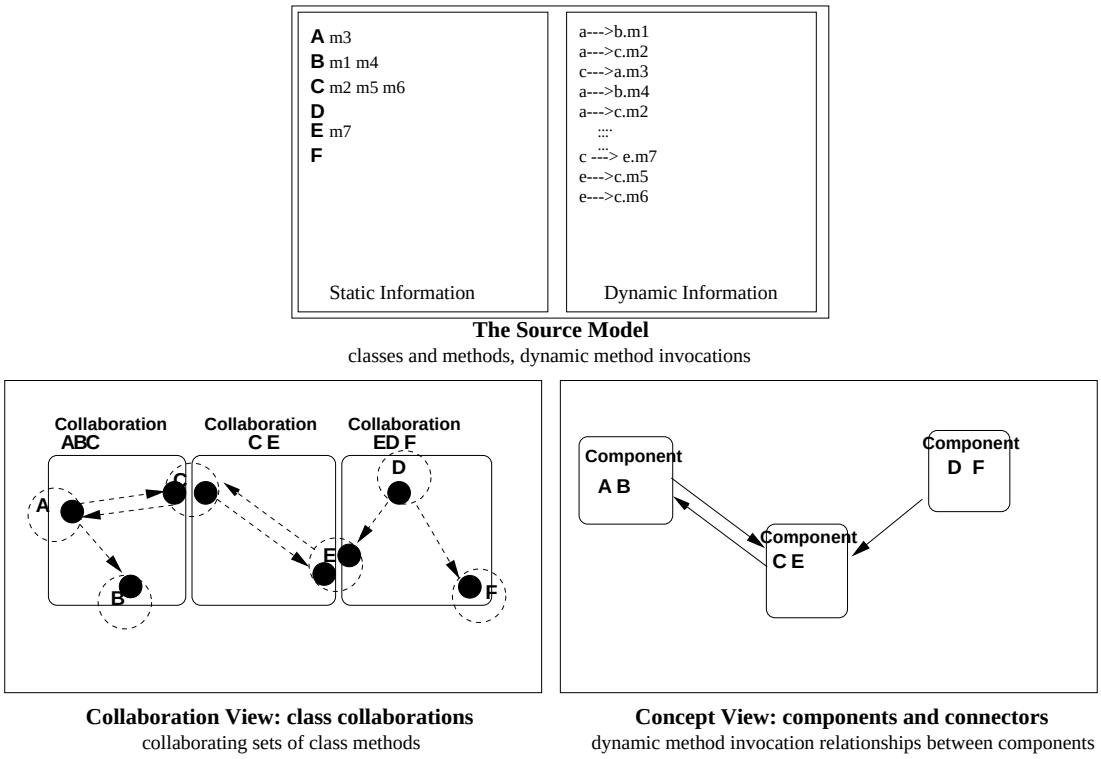

Figure 3.6: Concept views and collaboration views as a two different decompositions of a software system

Figure 3.6 illustrates the two kinds of decompositions schematically. It shows the static source model as information about classes in the system and their corresponding methods. Dynamic information is represented as a sequence of method invocations. The decomposition on the right hand side shows a concept view: classes are grouped into components and connectors represent method invocation relationships between these components. On the left is the collaboration view: it shows how partial interfaces (illustrated as a

```
                composedView(sendsCreate,component).

connectors:     sendsCreate(Class1,Class2).

components:     component(ComponentName,ListOfClasses).
```

Figure 3.7: A perspective for creation invocations between components

black circle inside the class construct) are grouped together to represent a logical method invocation sequence.

For both views we make use of the same source model of static and dynamic information. Static information is used to create components for the component-connector perspective, and to express similarity using pattern matching for the collaboration perspective. The semantics of the connectors is specified using the meta-model of dynamic information. Finally, dynamic information is shown through these perspectives to create a concept view and a collaboration view. Thus concept view recovery and collaboration view recovery are part of the same approach, and as such we will evaluate them together. In the next sections we give an overview of each of the applications, then briefly consider their integration into one tool.

3.3.1 Concept View Recovery

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. It enables the developer to specify perspectives which result in high-level views of the application as well as perspectives which result in views of finer granularity. The framework is supported by the Gaudi tool, described in Section 4.3. Here we provide only a short introduction to the application which will be presented in greater detail in Chapter 4 and Chapter 5.

Perspectives and views. A perspective is defined by specifying two elements: *components* and *connectors*. These are defined declaratively using a logic-programming language. A view is the result of applying a perspective to a source model.

Example: Figure 3.7 shows the specification of a perspective. The perspective composedView(sendsCreate,component) is a second-order logic predicate which takes two arguments. The first argument is a logic predicate which specifies a dynamic relation, and gives the semantics of the connectors between the components. The second argument is a logic predicate which clusters together static elements, giving the semantics of the components.

```
sendsCreate(Class1,Class2) :-
    sendsToMethod(Class1,MetaClass,Method),
    metaclassOf(MetaClass,Class2),
    methodCategory(MetaClass,Method,'instance creation').

sendsCreate(Class1,Class2) :-
    indirectsend(_, _,Class1, _,'new',Class2, _, _).
```

The rule sendsCreate above defines a creation relation between two classes which holds true if an instance of Class1 creates an instance of Class2. The rule component below defines a partitioning of classes in the HotDraw framework into several components. Classes which are not assigned to a component with this rule, constitute a component in themselves (here, for example, DrawingEditor, Drawing, DrawingController and CompositeFigure).

```
component('Figure',L]) :- allInCategory('HotDraw-Figures',L).
component('Handle',L) :- allInCategory('HotDraw-Handles',L).
component('Constraint',L) :- allInCategory('HotDraw-Constraints',L).
component('Toolbar',L) :- allInCategory('HotDraw-Toolbar',L).
component('Tool',L) :- allInCategory('HotDraw-Tools',L).
```

When the perspective described above is applied to the source model of the HotDraw framework, a view is obtained as shown in Figure 3.8. This view shows a graph where each component is seen as a node, and each connector as a directed edge. A directed edge from component Tool to component Figure means that the relation sendsCreate(Class1,Class2) holds for at least one pair of classes, (Class1,Class2), where Class1 is in component Tool and Class2 is in component Figure.

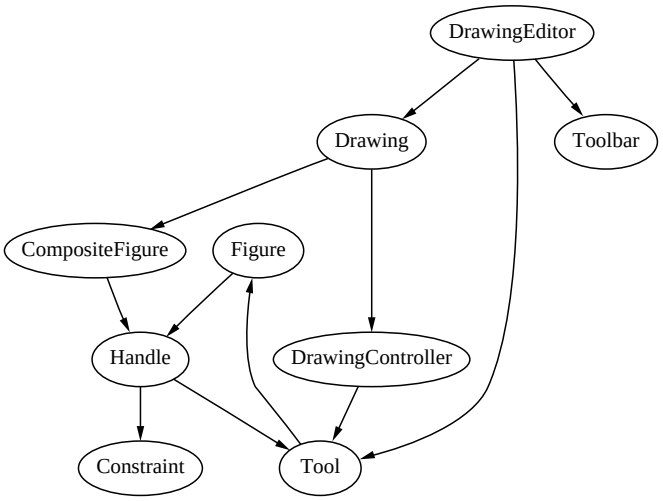

Figure 3.8: Creation invocations between components in the HotDraw framework

3.3.2 Collaboration View Recovery

While the extensible framework for concept view recovery proved flexible for answering a large range of questions, it was not satisfactory when we wanted to know which methods are used in which context and how classes collaborate in an interaction, and to generalize from specific object interactions to patterns of class collaborations. We therefore developed a new application of the approach which lets us recover a model which cross-cuts the system classes, and looks at groups of methods. Collaboration view recovery is supported by the Collaboration Browser tool, and is presented in detail in Chapter 6.

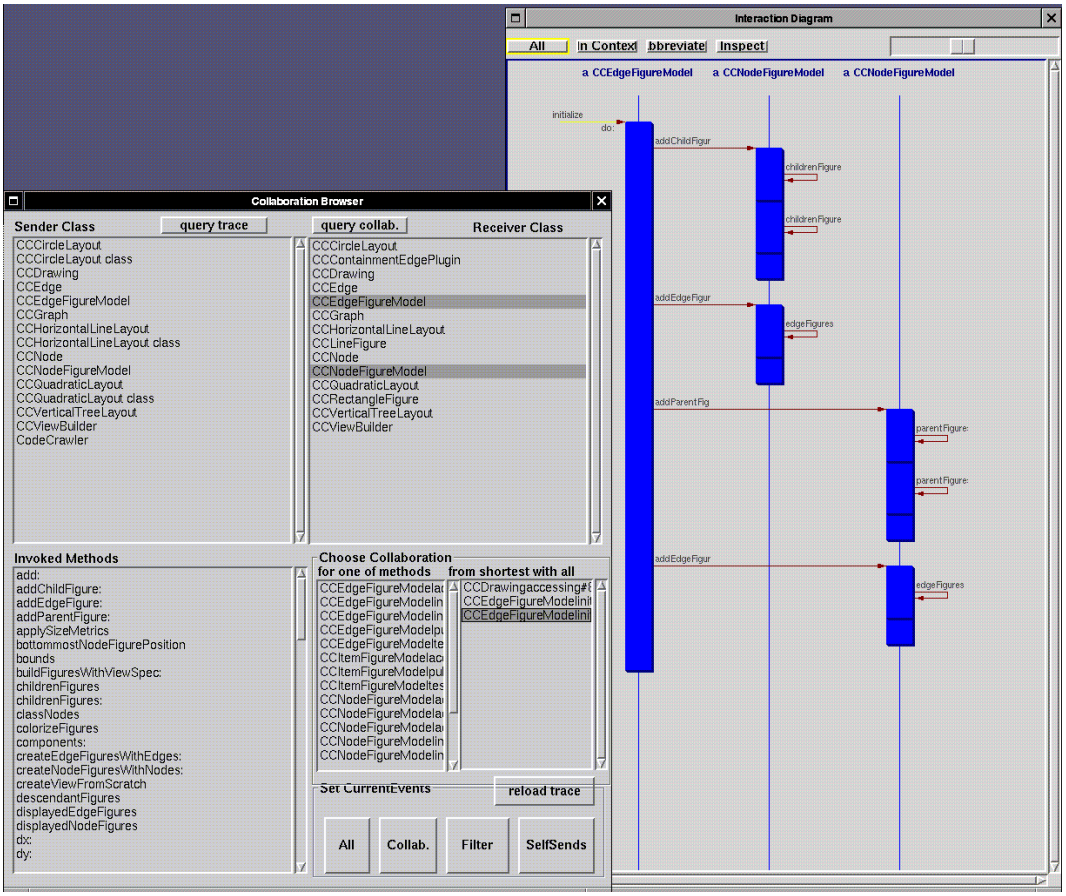

Figure 3.9: The Collaboration Browser

Perspectives and views. In collaboration view recovery perspectives have a more restrictive meaning than in concept view recovery. Whereas in concept view recovery we can recover views displaying different kinds of static components and dynamic relations, in collaboration view recovery our view – our window on the dynamic information – is always represented in terms of four basic elements: sender classes, receiver classes, invoked methods and collaboration patterns. In collaboration view recovery we group together similar sequences of object interactions in the trace into *collaboration patterns*. Perspectives determine what information we see about sender classes, receiver classes,

invoked methods and collaboration patterns. We specify perspectives through two operations: pattern matching and querying. Setting pattern matching options determines which execution sequences will be considered similar, so it gives the semantics of a collaboration. Querying allows us to find the relevant and important collaborations by choosing collaborations in which certain classes participate and by querying about the roles these classes play in the collaborations.

Example: a screen shot of the Collaboration Browser tool is shown in Figure 3.9. The Collaboration Browser window is divided into four sections, listing (clockwise from the top left corner) sender classes, receiver classes, collaboration patterns and invoked methods. Figure 3.9 shows this information for a scenario of CodeCrawler [DDL99]. Pattern matching has already been performed, so the execution trace is seen as a collection of *collaboration patterns* which can be queried. Two receiver classes have been selected: EdgeFigureModel and NodeFigureModel in the top right hand panel of the window. A query was then launched to discover which collaboration patterns these two classes engage in together. The response appears in the lower right hand panel of the window, and shows a list of three collaboration patterns. One of these collaboration patterns has been selected and an instance of the pattern displayed as an interaction diagram.

3.3.3 Combining Concept and Collaboration View Recovery

The two applications we present in the dissertation use basically the same approach. Both support an iterative process of design recovery in which perspectives are declaratively defined to extract views of the software. Both operate on the same meta-model of static and dynamic information. Both create abstractions by viewing the dynamic information stripped of its time dimension.

They are also complementary: concept view recovery extracts views showing binary relations between (mostly high-level) components, collaboration view recovery extracts finer-grained views of collaborations and the roles classes play in them.

We have implemented each of these as a separate tool and also present each of them separately in the dissertation. But because of they are similar and they complement each other we can easily see them being integrated into one tool which supports the two kinds of perspectives and view composer mechanisms (grouping as in concept view recovery, pattern matching as in collaboration view recovery). In particular, such a tool would support the recovery of collaborations in the context of a specific concept view. The question of integrating the two applications into one tool is discussed again in Section 7.3.

3.4 Conclusions

In this chapter we introduced the elements of our approach. We argued for the need for an iterative recovery process and discussed how a reverse engineering tool is embedded in this process. We introduced the notion of *perspective* as a query on the source model, and *view* as a response to the query. The meta-model used to represent the static and dynamic

information of the *source model* was presented and discussed. Finally, we introduced the two applications we have developed, concept view recovery and collaboration view recovery. For each of these we briefly described how perspectives are defined and how views are extracted.

The meta-model we use to represent static information represents a software application in terms of program entities and their relationships rather than at the detailed level of abstract syntax trees (AST). This information is thus relatively easy to extract from applications written in any class-based object-oriented language. The meta-model we use to represent program execution is a simple one, representing method invocation events in terms of sender class and identity, receiver class and identity and invoked method. This kind of information is also easily obtained from an instrumented program. We thus satisfy the requirement we set for ourselves in Section 2.3 of having a *lightweight information model*. In fact, as will be seen later from the case studies performed, a reduced meta-model of static information about classes, methods and inheritance definitions suffices to obtain a rich repertoire of perspectives.

Though we have not yet demonstrated the process of design recovery that our approach supports, we argued again for an iterative process which puts the developer in charge of the process and in charge of specifying the kinds of views that he or she is interested in.

In Section 3.3 we have given a preview of how perspectives declaratively specify views in each of the applications we present. We argued that concept view recovery and collaboration view recovery are part of the same approach, that they complement each other in design recovery, and that they can be thought of as being integrated into one design recovery tool.

In the next chapter we present the first of these two applications: concept view recovery.

4

Concept View Recovery: the Declarative Framework

In the next two chapters we present concept view recovery. The approach is presented in two parts: this chapter introduces a declarative framework which supports the definition of component-connector perspectives using a logic-programming language. It also describes a tool which implements the framework. In the following chapter we present two case studies to demonstrate how this framework is applied to real software applications.

We show how a logic programming language is used to describe the source model and to derive properties of this source model using layers of predicates. Perspectives are second-order predicates which combine two predicates: one to specify the clustering of static elements into a component and one to specify the semantics of the connectors. The semantics of the components and connectors is not fixed, but can be specified by the developer or maintainer of the software.

This chapter is structured as follows: in Section 4.1 we introduce the framework as a set of layers, and show how logic predicates in each layer are used to define static and dynamic relations of software artifacts. Section 4.2 presents the topmost layer of the framework – the perspective layer – whose predicates support the extraction of views. Section 4.3 describes Gaudi, a prototype tool developed to support the declarative framework. Finally, we discuss the framework in Section 4.4 and conclude the chapter with Section 4.5.

4.1 A Declarative Framework for Perspectives

We use a logic-programming language to define a declarative framework which supports the specification of perspectives. Figure 4.1 illustrates this framework schematically. The declarative framework is a set of rules, conceptually organized in layers, where each layer makes use of the layers below it [Wuy01].

The bottommost layer is called the *representation layer* since it consists of predicates which most directly represent the software – those found in the source model. The next highest layer is the *base layer*, consisting of predicates which describe relations derived

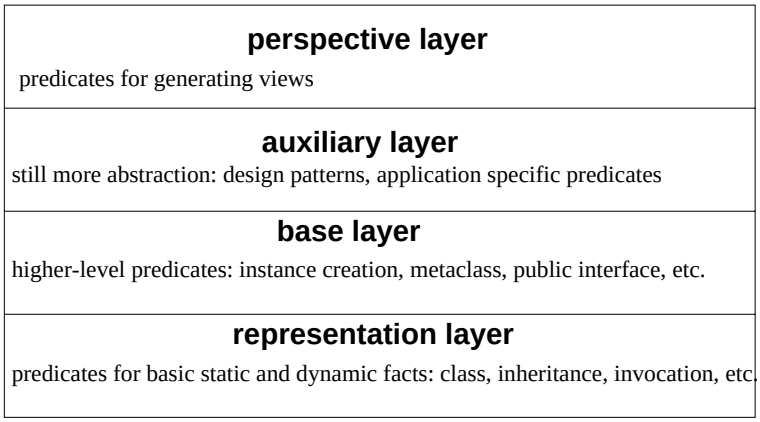

Figure 4.1: The declarative framework as layers of predicates

from the representation layer. Both the representation layer and the base layer are predefined in the framework. By contrast, the *auxiliary layer* consists of rules defined by the developer. These may be rules which are specific to the application being investigated, or rules which codify design abstractions not found in the base or representation layer. The auxiliary layer is not essential to the declarative framework – it may be empty. The *perspective layer* consists of rules which specify component-connector perspectives. Several types of perspectives are predefined – but this layer may be augmented by the developer.

Figure 4.2 illustrates how this declarative framework is used in a query-based approach. The source model is represented by static and dynamic facts about the software application to be analyzed. Using the predicates of the declarative framework a developer makes queries about this source model. More specifically, the developer requests to view the source model through a particular perspective. A perspective is specified by two predicates from the declarative framework, r and C. The predicate C specifies clustering to be applied to static elements in the source model to create *components*, whereas the predicate r specifies a relationship between static elements – the *connectors*. The response to such a query is a view of the software as seen through the perspective.

The notion of perspectives and their use in querying will be clarified in more detail in Section 4.2. In the next sections we first describe each of the layers of the declarative framework, starting from the lowest one.

4.1.1 The Representation Layer

The representation layer consists of predicates which represent the source model extracted from the software. Both the static model and the dynamic model described in the previous chapter (Section 3.2) are represented in terms of logic facts.

Static Relations. Static information is represented as facts in a logic-programming language [SS86]. Table 4.1 below shows the representation of the six FAMIX core entities and associations as logic facts.

Note that the parameter Candidates of predicate invocation gives the potential re-

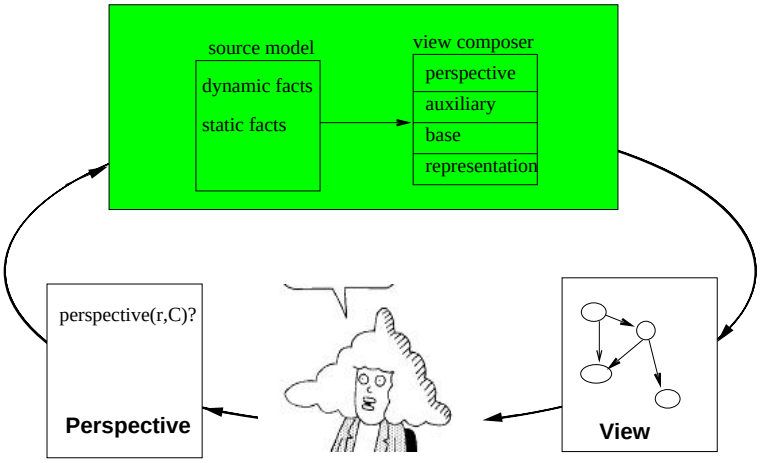

Figure 4.2: The framework as used in an iterative query cycle

ceivers of the invocation; an empty list means that no candidates were found within the classes of the application.The SourceAnchor parameter of the predicate class is language-dependent. In the case of Smalltalk it is taken to be the class category in which the class is defined.

As an example of how these predicates are used to represent static information we present below a sample which provides information about the class EllipseFigure of the HotDraw framework [BJ94]. Figure 4.3 illustrates this information in a class diagram.

```
(1) class('EllipseFigure','HotDraw-Figures').
(2) superclass('Figure','EllipseFigure').
(3) method('EllipseFigure','displayFilledOn:',false,'displaying').
(4) access('EllipseFigure','self','EllipseFigure','displayFilledOn:').
(5) invocation('EllipseFigure','displayFilledOn:','fillColor',['Figure']).
```

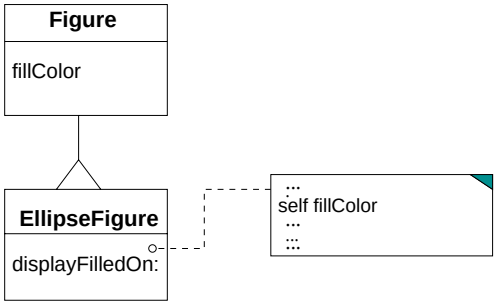

Figure 4.3: Class diagram for the static information above

Dynamic Execution. Dynamic information is represented as facts about method invocations in a program's execution (Table 4.2). Only two kinds of facts are represented: send and indirectsend. These are numbered according to sequence order (SN) and stack level (SL). Return events are not directly represented as facts: they can be deduced from the

stack level numbers of the send events. Each send or indirectsend fact corresponds to the invocation of an observed method on an instance of a class.

As an example, the send facts listed below record the method invocations that follow the invocation of EllipseFigure(instance#39).fillColor. This sequence corresponds to one execution of the invocation described in line 5 of the static information above. Its representation as a UML interaction diagram is shown in Figure 4.4.

```
send(49,7,'EllipseFigure',139,'EllipseFigure',139,'fillColor').
send(50,8,'EllipseFigure',139,'Drawing',685,'fillColor').
send(51,9,'Drawing',685,'FigureAttributes',4426,'fillColor').
send(52,9,'Drawing',685,'FigureAttributes',4426,'fillColor').
send(53,7,'EllipseFigure',139,'Drawing',685,'compositionBoundsFor:').
```

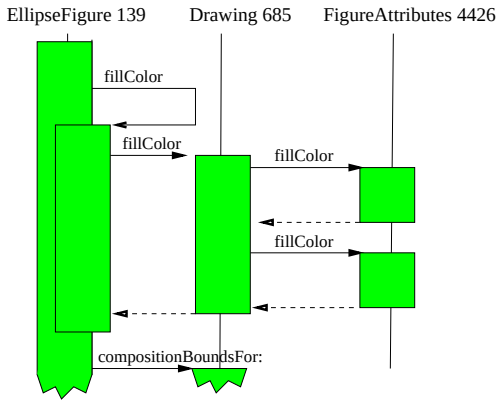

Figure 4.4: Sequence diagram for the sequence of method invocations above

class(ClassName, SourceAnchor) a class and its source artifact
superclass(SuperClass, SubClass) an inheritance relationship
attribute(Class, AttributeName, AttributeType) class defines an attribute of a certain type
method(Class, MethodName, IsClassMethod, Category) a class defines a method belonging to a Category
access(Class2, Attribute, Class1, Method) an attribute of Class1 is accessed by Method of Class2
invocation(Sender, Method, ReceivedMethod, Candidates) Method of Sender invokes ReceivedMethod on one of the Candidates

Table 4.1: Predicates of the representation layer: the static entities and associations

send(SequenceN, StackLevel, Class1, Instance1, Class2, Instance2, Method) an instance Instance1 of Class1 invokes Method on instance Instance2 of Class2. SequenceN is the sequence number of the event, and StackLevel is the stack level of the method call.
indirectsend(SequenceN, StackLevel, Class1, Instance1, Method1, Class2, Instance2, Method2) an instance Instance1 of Class1 sends the message Method1, which is unobserved. The next observed invocation is the execution of method Method2 on instance Instance2 of Class2.

Table 4.2: Predicates of the representation layer: dynamic relations

4.1.2 The Base Layer

The predicates of the representation layer are used to represent the source code and its execution – the *source model* on which a reverse engineering tool operates. This section presents the base layer: predicates which form a logic layer above the source model and allow for more sophisticated reasoning about the structure and behavior of the application. Some of these rules make use of static information only, some of dynamic information only, while some of the rules combine static and dynamic information. We present examples of each of these kinds of rules.

Of the base layer predicates, two kinds of rules are of special interest since they form the basis for the definition of perspectives. The first kind is a predicate which defines a relation between two entities, for example, the inheritance relationship between two classes, or the method invocation between two instances. This kind of predicate is used to give semantics to the *connectors* in the perspective. The second kind is a predicate which defines a relation between a component name and the set of entities in the component. An example of the second kind is the association of a class category name to the set of all classes in that category. This form of predicate is used to specify components in the perspective, and is called a *component clustering* rule.

Tables 4.3, 4.4, 4.5 and 4.6, summarize the predefined predicates of the base layer. Table 4.3 lists the predicates which use static information only. Table 4.4 lists predicates which are generally useful in querying, and Table 4.5 lists predefined component clustering predicates. Finally, Table 4.6 presents predicate for instance and class relations which rely on dynamic information.

The basic static predicates are used to formulate rules expressing structural concepts of object-oriented software, such as method overwriting, inheritance hierarchy and so on. In the same vein, the basic dynamic predicates are used to express behavioral relationships between instances and classes. We illustrate this below with some examples.

Rules using static information only. Rule 1 below specifies that a subclass Subclass overrides a method Method defined in a class Class. It makes use of rule 2, commonMethod, which says that Class1 and Class2 define a method with the same name Method, and rule 3, inHierarchy, which defines what it means for a class to be in the inheritance hierarchy of another class.

```
rule1 :   overrides(Class, Subclass, Method) :-
              commonMethod(Class, Subclass, Method),
              inHierarchy(Class, Subclass).

rule2 :   commonMethod(Class1, Class2, Method) :-
              method(Class1, Method, IsClassMethod, _),
              method(Class2, Method, IsClassMethod, _).

rule3 :   inHierarchy(Class, Class).
          inHierarchy(Class,Subclass) :-
              superclass(Superclass,Subclass),
              inHierarchy(Class,Superclass).
```

Rules using dynamic information only. Rule 4 below specifies that an instance of Class1 invokes, either directly or indirectly, a Method on an instance of a class, Class2.

```
rule4 :   sendsToMethod(Class1,Class2,Method) :-
              send(_,_,Class1,_,Class2,_,Method).

          sendsToMethod(Class1,Class2,Method) :-
              indirectsend(_,_,Class1,_,_,Class2,_,Method).
```

Rules combining static and dynamic information. Dynamic and static predicates are combined to express more complex relationships, such as a creation relationship between classes. An example is presented below.

Rule 5 below defines a *create* relationship between two classes – an instance of class Class1 creates an instance of Class2. Its use for obtaining a *creation* view of an application will be demonstrated in the case studies described in Chapter 5. The rule specifies that an instance of Class1 invokes a Method on an instance of a metaclass of Class2, where the Smalltalk category of Method is instance creation (Smalltalk creation methods, defined at the class level, appear as methods of a metaclass in the model. There is a Smalltalk convention to group instance creation methods into a category named instance creation). Rule 6 allows one to go up the inheritance hierarchy to find the Smalltalk category of the Method, in case it is not defined by MetaClass.

```
rule5 :   sendsCreate(Class1,Class2) :-
              sendsToMethod(Class1,MetaClass,Method),
              metaclassOf(MetaClass,Class2),
              methodCategory(MetaClass,Method,'instance creation').

rule6 :   methodCategory(Class,Method,Category) :-
              method(Class,Method,_,Category).

          methodCategory(Class,Method,Category) :-
              inHierarchy(Superclass,Class),
              method(Superclass,Method,_,Category).
```

Rules for component clustering. As will be explained in Section 4.2, the definition of perspectives allows us to declaratively define component clustering of the software elements. This is done through logic predicates. The following rule, for example, groups together all classes in the same inheritance hierarchy:

```
rule7 :   allInRootHierarchy(RootClass,ListOfClasses) :-
              setof(Class,inRootHierarchy(Class,RootClass),ListOfClasses).
```

inHierarchy(SuperClass,Class) Class is in the inheritance hierarchy of SuperClass
rootClass(RootClass) Class is a root class of an inheritance tree
inRootHierarchy(Class,RootClass) Class is in the inheritance hierarchy of RootClass, which is a root class
commonMethod(Class1,Class2,Method) Class1 and Class2 both define Method
overriddenMethod(Class1,Class2,Method) Class1 overrides a Method defined in Class 2
overriddenMethodRoot(Class,Method,Root) Root is the topmost class which defines Method overridden by Class
understands(Class,Method) Class understands Method
rootMethod(Class,Method,Root) Class understands Method, which is defined topmost at Root
classInterface(Class,ListOfMethods) all the methods that Class understands
methodCategory(Class,Method,Category) a Method understood by Class belongs to the method category Category
metaclass(Class) Class is a metaclass
metaclassOf(MetaClass,Class) Class is a metaclass of Class

Table 4.3: Predicates of the base layer: static relations.

numberOfInstances(Class,Number) number of instances of a Class in the scenario
singleton(Class) true if there is only one instance of Class in the scenario
publicInterface(Class,Methods) Methods are all the methods invoked on class Class by other instances

Table 4.4: Predicates of the base layer: rules based on dynamic information.

allInRootHierarchy(RootClass,ListOfClasses) associates to a component named RootClass all the classes in the inheritance hierarchy of RootClass
allInCategory(CategoryName,ListOfClasses) associates to a component named CategoryName all the classes in the class category CategoryName

Table 4.5: Predicates of the base layer: component clustering rules.

Instance Relations
sendsToInstance(Class1,Instance1,Class2,Instance2) true if Instance1 of Class1 invokes a method on Instance2 of Class2
createInstance(Class1,Instance1,Class2,Instance2) true if Instance1 of Class1 creates Instance2 of Class2
sendInstanceInStack(Class1/Inst1,Class2/Inst2,N,Start) true if Inst1 of Class1 invokes a method on Inst2 of Class2, within the call stack starting at send event numbered Start. N is the sequence number of the particular send event. This relation is used to show ordering of invocations within a call stack.
sendInstanceInStackMethod(Class1/Inst1,Class2/Inst2,Method,N,Start) true if Inst1 of Class1 invokes Method on Inst2 of Class2, within the call stack starting at send event numbered Start. N is the sequence number of the particular send event. This relation is used to show ordering of method invocation within a call stack.
Class Relations
sendsTo(Class1,Class2) true if an instance of Class1 invokes a method on an instance of Class2 (directly or indirectly)
sendsToMethod(Class1,Class2,Method) true if an instance of Class1 invokes Method on an instance of Class2 (directly or indirectly)
sendsToNotSameClass, sendsToNotSameInstance, sendsToMethodDirect, *etc.* variations of the method invocation relationship between classes
sendsCreate(Class1,Class2) true if an instance of Class1 creates on an instance of Class2 (directly or indirectly)
oneToOneSend(InstanceRelation,SenderClass,ReceiverClass) true if the Instance Relation is one-to-one for classes SendeClass and ReceiverClass
oneToManySend(InstanceRelation,SenderClass,ReceiverClass) true if the Instance Relation is one-to-many for classes SendeClass and ReceiverClass
manyToOneSend(InstanceRelation,SenderClass,ReceiverClass) true if the Instance Relation is many-to-one for classes SenderClass and ReceiverClass
manyToManySend(InstanceRelation,SenderClass,ReceiverClass) true if the Instance Relationis many to many for classes SendeClass and ReceiverClass

Table 4.6: Predicates of the base layer: dynamic relations of instances and classes.

4.1.3 The Auxiliary Layer

Whereas predicates in the representation and base layer are predefined in the framework, the auxiliary layer is a placeholder for predicates which are defined by a developer who uses the framework. Figure 4.5 illustrates this: shaded areas correspond to predefined predicates, white areas to places where a user of the tool can add predicates.

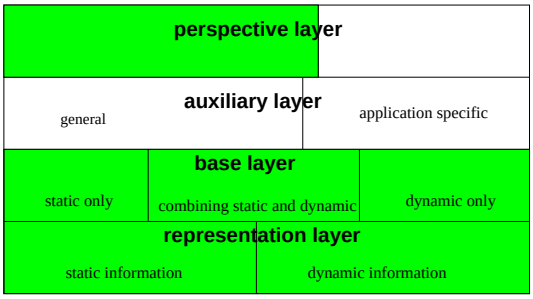

Figure 4.5: The declarative framework: shaded areas correspond to predefined predicates

The auxiliary layer consists of two kinds of predicate: those which represent general relations independent of the particular software application being investigated, and those relations which are application specific. General high-level predicates are design patterns, design conventions, idioms [Cop92] and best practice patterns [Bec97]. For example, [Wuy01, Men00] use a logic programming language to check conformance to such conventions and guidelines.

General predicates. Below we give an example of a general high-level predicate, adapted from [Wuy01]. It is a logic predicate accessingViolator which checks for the violation of the coding convention *accessor methods*. This convention states: *never access instance variables directly, but always through accessor methods.* The rule below states that this convention is violated if a method which is not an *accessor* accesses one of the class's instance variables directly. An accessor method is specified using the rule accessor(Class,Method) and identifies a method as an accessor method if it is in the Smalltalk method category accessing. The direct access of an instance variable is specified using the rule accesses(Class,Variable,Method) which checks if the method accesses any of its instance variables (also inherited ones). These rules make use of static relations of the representation and base layers.

```
accessingViolator(Class,Method,Variable) :-
    accesses(Class,Variable,Method) ,
    not(accessor(Class,Method)).

accessor(Class,Method):-
    method(Class,Method,_,'accessing').

accesses(Class,Variable,Method) :-
    access(Class,Variable,Class,Method).
accesses(Class,Variable,Method) :-
    access(Superclass,Variable,Class,Method),
    inHierarchy(Superclass,Class).
```

Design patterns also represent application independent rules and conventions. An example of a predicate which codes for the structure of the Composite design pattern is given in Section 4.4.1.

Application specific predicates. Application specific predicates are rules which use information specific to the software application being investigated. Below we present two examples of such rules. The first predicate, given below, defines an invocation relation between two classes, where either the sender class or the receiver class (or both) is the class RefactoringManager. This predicate is used in the case study presented in the next chapter to obtain a view of all classes which communicate with instances of RefactoringManager and the messages received or sent by instances of RefactoringManager.

```
sendsToFromRefactoringManager(C1,C2,M) :-
    sendsToMethod(C1,'RefactoringManager',M);
    sendsToMethod('RefactoringManager',C2,M).
```

The second example, presented below, is a set of component clustering predicates. These rules are used to cluster some of the classes of the Refactoring Browser into three components: Parser, Conditions, and Browser+Stuff. Classes which are not clustered with these rules represent components on their own.

```
inComponent('Browser+Stuff',Class) :- class(Class,'Refactory-Environments').
inComponent('Browser+Stuff',Class) :- class(Class,'Refactory-Browser').
inComponent('Browser+Stuff',Class) :- class(Class,'Refactory-Code Tools').

component('Parser',L) :- allinCategory('Refactory-Parser', L).
component('Conditions',L) :- allinCategory('Refactory-Conditions',L).
component('Browser+Stuff',L) :- setof(Class,inComponent('Browser+Stuff',Class),L).
```

4.2 The Perspective Layer

We have defined a perspective as 'a specification of the aspects of the source model that we are interested in' (Section 3.1.1) and explained that a perspective is given as a declarative specification in terms of the software meta-model, or in terms of the source model

itself. In this sense, any query of the source model is a perspective. Querying the source model, for example, with the predicate accessingViolator(Class,Method,Variable) specifies that we are interested in seeing all the triples (Class,Method,Variable) which violate the accessor methods coding convention.

Though the use of simple queries as perspectives also relates interesting information about the software system, we want to extract views which give us a more global understanding of the system. To this end, we introduce component-connector perspectives. The logic predicates for expressing component-connector perspectives form the topmost layer of the declarative framework. These predicates are used to generate concept views of a software application.

A concept view of the source model is a directed graph which displays *components* and a relationship between these components, the *connectors*. Below we define views, then go on to explain how perspectives are specified.

4.2.1 Views

A view $V = \{C, R\}$ is defined as a set of components C and a set of connectors R between these components. It corresponds to a directed graph: each node in the graph is a component C_i, and each directed edge between components C_i and C_j represents the presence of the relationship R between these components.

More precisely, let:

- $E = \{e_1, e_2, ..., e_n\}$ be a set of elements in the source model.

  For example, E can be the set of all classes, all methods, or all object instances in the source model, or a set of classes and methods.

- $r \subseteq E\text{x}E$ be a relation over two elements of E, or

  $r \subseteq E\text{x}E\text{x}L$ is a relation over two elements of E and an element of L, where L is a set of labels.

  So r is a binary relationship between elements in the source model or a relationship between two elements and a label, or value. For example, r can be the inheritance relationship between classes, the overriding relationship between methods or the method relationship between classes and methods. It can also be the message send relationship between two instances and a method, where the method name is the label associated to the relationship.

- $c \subseteq E\text{x}E$ is an equivalence relation over elements of E.

  That is, c induces a partition on E. For example, c can be the relationship 'in the same method category' between methods, or 'in the same inheritance hierarchy' between classes. These relations partition the elements of E into equivalence classes of all methods classified under the same method category, or all classes in the same inheritance hierarchy, respectively.

- we derive a set $E_r \subseteq E$ which includes all the elements in the domain and range of r:

  $E_r = \{x \in E | \exists y \in E, (x,y) \in r \vee (y,x) \in r\}$, or

  $E_r = \{x \in E | \exists y \in E, \exists l \in L, (x,y,l) \in r \vee (y,x,l) \in r\}$

  For example, if r is a message send relation between classes from the dynamic information, and E is the set of all classes, then E_r is the set of all classes which are related through this send relation to other classes. Classes for which no instances send or receive messages will not be included in E_r.

Then a view $V = \{C, R\}$ where:

- $C = E_r/c = \{C_1, C_2, ..., C_m\}$, a partition of E_r induced by c.
- $R \subseteq C \text{x} C$, where $R = r/c = \{(C_i, C_j) | \exists e_i \in C_i, \exists e_j \in C_j, (e_i, e_j) \in r\}$, or
  $R \subseteq C \text{x} C \text{x} L$, where $R = r/c = \{(C_i, C_j, l) | \exists e_i \in C_i, \exists e_j \in C_j, (e_i, e_j, l) \in r\}$

What does this mean? A view is a set of components and connectors: C is the set of components in the view, and R is the set of connectors. In order to specify a view, we need a relation r, and an equivalence relation c which will induce a partition on E_r, the elements in the range and domain of r. The view will then show a set of components. It will show a directed edge from component C_1 to component C_2 if there is at least one element e_1 in C_1 and one element e_2 in C_2 for which relation r holds. It will show a directed edge labeled l from component C_1 to component C_2 if there is at least one element e_1 in C_1 and one element e_2 in C_2 for which (e_1, e_2, l) is in relation r. Each component will be attached by an edge to at least one other component, since all the elements in E_r are either in the domain or range of r.

For example, if r is a message send relation between classes from the dynamic information, and c is an equivalence relation which partitions classes into separate inheritance hierarchies, then the view we obtain will show the message send relation between inheritance hierarchies. An inheritance hierarchy which does not 'send' or 'receive' a message will not appear in the view.

4.2.2 Perspectives

From the discussion above we see that a perspective must be expressed by specifying r and c as defined above. This declarative specification is then used to generate a view by using c to partition the set of elements into a set of components C and using r to derive the relation R between these components.

We specify perspectives as second-order logic predicates of the form perspective(relation,component) where relation and component are also logic predicates. Specifying relation corresponds to specifying r. In order to specify the equivalence relation c, we directly partition the elements of E using the predicate component. This is explained below.

1. relation(Element1,Element2) or relation(Element1,Element2,Value) is a relation between two elements of E, which may or may not associate a value to the relation, and corresponds to specifying the relation r for the view.

   For example, the predicate sendsTo(Class1,Class2) specifies the invocation relationship between classes, based on the dynamic information. The predicate sendsToMethod(Class1,Class2,MethodName) specifies the invocation relationship between classes and the name of the method invoked. The predicate superclass(Class1,Class2) specifies the inheritance relationship between classes, the predicate methodOf(Class,Methodname) specifies that a method is defined in a class.

2. component(ComponentName,ComponentElements) is a rule which clusters a set of elements ComponentElements into a component ComponentName. This rule must induce a partition on the elements of E.

   For example, the rule allInCategory(Category,ListOfClasses) partitions the classes in the source model into their respective Smalltalk class categories. The name of each component is the name of the class category. The predicate allInHierarchy(RootClass,ListofClasses) partitions the classes according to the inheritance hierarchy, where the name of each component is the name of the root class of the hierarchy.

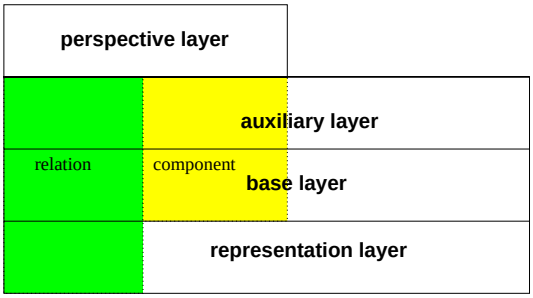

Figure 4.6: Perspective predicates are based on logic predicates from the lower layers

The perspective layer consists of predicates which specify different kinds of perspectives, discussed in the next section. These use relation and component predicates from the lower layers, as shown in Figure 4.6.

Using perspectives. We have described components as groupings of static elements, and connectors as relations derived from dynamic information. This is not inherent to perspectives themselves, but is due to our use of perspectives. We use them to view dynamic information in order to answer behavioral questions.

Connectors can represent static relationships, such as the inheritance relationship between classes or the access relationship between methods and attributes. As these relationships are extractable with many tools which perform only static analysis, we are not interested in them here. Components can also be expressed using dynamic information,

for example, to group together all classes whose instances send messages to instances of a specific class – in this case we use dynamic information to group together static entities. Finally, components could also be used to group together several method invocation events. It is not clear then what interesting relations can be defined between method invocations. Using dynamic information to create abstractions from sequences of method invocations is addressed by collaboration view recovery.

An example. In the next chapter we illustrate the use of perspectives through case studies. Throughout that chapter we present perspectives in a format illustrated in Perspective 1 below. The name of the perspective is presented at the top – here it is composedView. The relation predicate used in denoted by r, and the component predicate is denoted by C.

In the example below rules 8 and 9 define a relation r and a clustering C respectively. In this case E is the set of all classes and is defined implicitly by rule 8. Rule 8 defines a predicate sendsTo(Class1,Class2) which holds true if an instance of Class1 invokes any method on an instance of Class2. Rule 9 defines a clustering predicate, allInCategory(Category,ListOfClasses). This predicate is true when each class in ListOfClasses is in the Smalltalk class category Category. The predicate composedView combines rules 8 and 9 to specify a view in which components are Smalltalk class categories and the relation between them is the message send relationship between classes.

```
composedView(sendsTo,allInCategory).

r rule8 :  sendsTo(Class1,Class2).

C rule9 :  allInCategory(Category,ListOfClasses) :-
               setof(Class,class(Class,Category),ListOfClasses).
```

Perspective 1: Dynamic invocations between class categories

We then invoke composedView(sendsTo,allInCategory) as a query to generate a view of the source model. The view is generated automatically: the second argument, allInCategory, is used to partition the classes into components according to their class categories, the first argument, sendsTo, is used to derive the relation between these components. The generated view is displayed as a directed graph (using the dot tool [KN]). As an example, the application of this perspective to the source model of HotDraw results in the view displayed in Figure 4.7. Each node in the graph corresponds to a HotDraw class category and each directed edge $A \rightarrow B$ means that at least one instance of a class in category A invokes a method on an instance of a class in category B.

4.2.3 Specifying Perspectives

Conceptually, a perspective combines a relation predicate and a clustering rule to generate a view. Practically, there are some variations on the form of a perspective, induced by two factors:

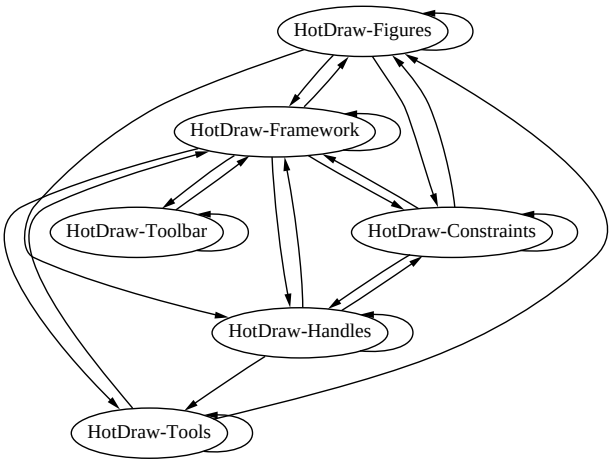

Figure 4.7: View 1: Invocations between class categories for the HotDraw sample editor

Presence or absence of component clustering: when a clustering predicate is missing then the equivalence relation c is the identity relationship I – that is, each element in E_r is itself a component. This is a *simple perspective*, in contrast to a *composite perspective*. Perspective 1 above is a *composite* perspective since it combines two rules to create a view.

The form of the relation: The relation r may be of the form $r \subseteq E\text{x}E$ or $r \subseteq E\text{x}E\text{x}L$. The second form has the effect of associating a label to each edge in the view. We often want a view of an application in which the directed edges of the graph are labeled with a value, for example, with the name of a method in the case of an invocation relationship.

Below we give some examples of different relation predicates derived from the meta-model for dynamic information. Note that these are just examples – the perspective framework is built to accommodate user-defined relations to tailor the views generated to the maintenance question guiding the investigation. More examples and their use in the recovery process will be provided in the presentation of the case studies in Chapter 5.

$r(e_1, e_2)$: sendsTo(C1,C2) defines the invocation relationship between classes.

$r(e_1, e_2, label)$: sendsToMethod(C1,C2,Method) defines the invocation relationship between classes, giving the name of the method invoked.

sendsToFrequency(C1,C2,Frequency) defines the invocation relationship, associating an integer to the relationship representing the frequency (from the dynamic information) with which instances of class C1 invoke methods on an instance of class C2.

$r(e_1, e_2, sequenceN)$: sendsToInstance(C1/I1,C2/I2,SequenceN) defined the invocation relationship between instance I1 of class C1 and instance I2 of Class C2. SequenceN gives the sequence number of the send event. This relation can be defined

to recover all events within a call stack, for example, and to produce a view which shows the control flow between instances within a method invocation event. Edges are labeled with integers to indicate the control flow.

$r(e_1, e_2, label, sequenceN)$: sendsToInstanceMethod(C1/I1,C2/I2,Method,SequenceN)
The same relationship as above, adding the method name as a labeling value. The resulting view is similar to a UML collaboration diagram: nodes correspond to instances and edges are labeled with a method name and are also ordered to show the sequence of method calls.

Table 4.7 lists the eight variations on perspectives predefined in the declarative framework, each as a function of the type of relation r given, and of the presence or absence of a clustering predicate C. The first column lists the simple perspectives, whereas the second column lists the composite perspectives. Both simple and composite perspectives can be of different types depending on whether or not the relation r includes a label and/or an ordering.

	C not specified	$C(Name, ListOfElements)$
$r(e_1, e_2)$	*view(r)*	*composedView(r,C)*
$r(e_1, e_2, label)$	*labelledView(r)*	*composedLabelledView(r,C)*
$r(e_1, e_2, sequenceN)$	*orderedView(r)*	*composedOrderedView(r,C)*
$r(e_1, e_2, label, sequenceN)$	*labelledOrderedView(r)*	*composedLabelledOrderedView(r,C)*

Table 4.7: Perspective types as a function of r and C

In contrast to the representation layer and the base layer, for which we listed the predefined predicates, the predicates of the perspective layer combine predicates from the lower layers, so they cannot be listed exhaustively. The framework is extensible: new perspectives can be obtained by combining new connector relationships and component clusterings; new perspective types can also be defined.

4.3 Tool Support: Gaudi

We have developed a prototype tool, Gaudi[1], to support the perspective framework. The Gaudi environment for query and view recovery has been implemented as an integration of tools for extraction, analysis and visualization. This is illustrated in Figure 4.8, which is explained below.

[1]named after the Spanish architect Antonio Gaudi (1852-1926).

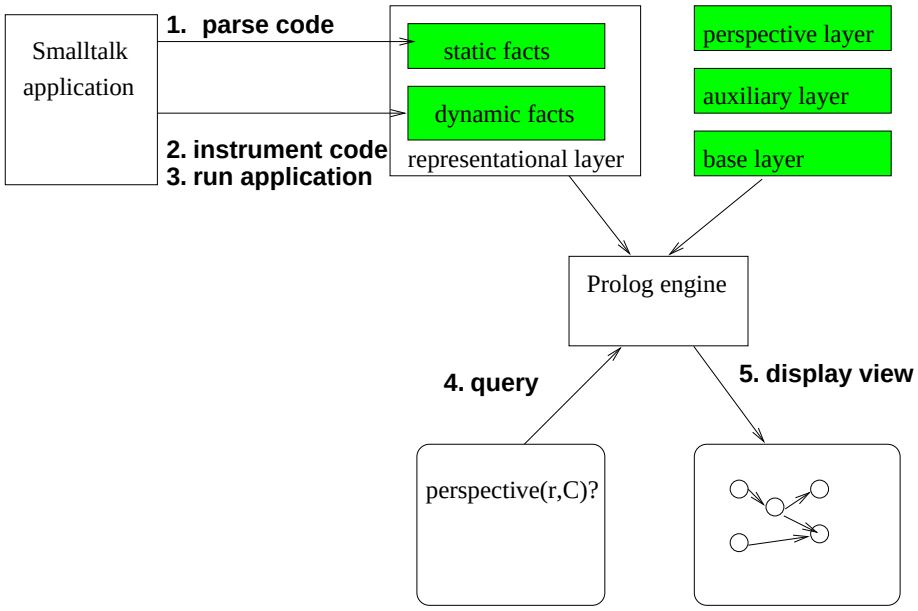

Figure 4.8: Implementation of Gaudi

To build up the source model on which the declarative framework operates, we extract static and dynamic information from the software application to be analyzed. The MOOSE tool [DLT00] is used to parse the code and represent it in the FAMIX model (**1** in Figure 4.8). This information is then written to a file as Prolog facts. The Smalltalk applications are instrumented using Method Wrappers [BFJR98] (**2**), which allow instrumentation at the level of individual methods. The application is then run (**3**) and the tracing information obtained from the execution is written to a file as Prolog facts.

The declarative framework is written in SICTUS Prolog [Pro95]. Queries are posed directly in the Prolog application (**4**). The view resulting from a perspective query is automatically written as a file in a format for the dot tool [KN], which is used to display the view (**5**).

Figure 4.9 shows a screen shot of Gaudi, in which the user has requested to see a creation relationship between components, for the HotDraw graphics editor framework. The window on the right is the emacs Prolog environment. The window on the left is the display of the dot tool.

4.3.1 Discussion of Gaudi

Here we discuss some of the choices we made in implementing Gaudi, the advantages they present and limitations they impose. We also consider possible extensions to the tool.

Prolog as a query language. Logic-programming languages are well suited for representing knowledge and for formulating declarative queries. The expressive power of a logic-programming language is also an advantage: multi-way queries can be succinctly

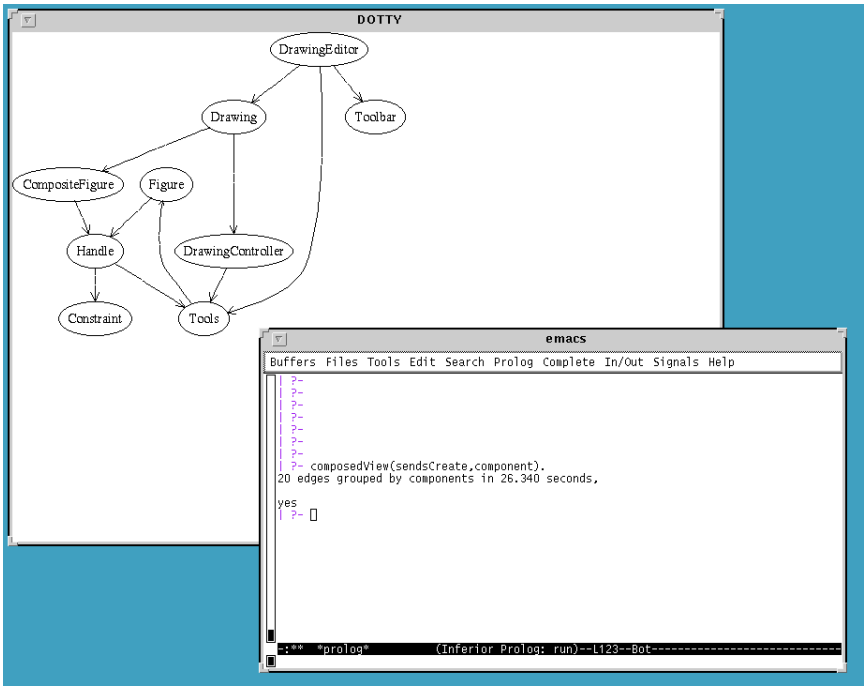

Figure 4.9: The Gaudi tool

expressed, and recursion is supported. We have chosen to use a logic programming language mainly for its simplicity and convenience as a query language for prototyping our tool. Could a database query language be used instead of a logic-programming language? Earlier database query languages provided little support for general recursive queries. They were also algorithmic in nature and required knowledge of the schema and its navigation. However, recent database query languages are more declarative in nature, and also support recursive queries to some degree. Further experiments are required to determine if database languages are suited to our approach.

In Gaudi, many useful queries are already defined as rules. However, an engineer also defines new Prolog rules in order to formulate queries tailored to the specific task or question. As our tool is a prototype meant to demonstrate the feasibility of the approach, we decided to use Prolog as the query language rather than defining a special query language. However, Prolog could be used as a back-end to a query language which is tailored more to the specific information model we are querying. This can be done by defining a domain specific language with Prolog's support for Definite Clause Grammars [SS86].

Display and Handling of Views. Our approach and tool relies on a simple display, rather than sophisticated visualization techniques. As implemented, the current prototype does not allow the manipulation of the source model through the visual display. The dot tool [KN] allows the editing of the resulting view, but the layout is done automatically, and so cannot be specified by the user.

Filtering. Currently elements in the source model are filtered out from a view if they do not enter in any relation r defined by the perspective. For example, for a perspective of

class invocations, a class will not appear in a view if none of its instances send or receive a message. In order to filter out some elements which do enter in the relation r we can group them together in a component. Information about the software can also be filtered out by instrumenting more selectively so as to capture the behavior of only the elements of interest.

4.4 Discussion of the Declarative Framework

Here we first discuss some work related to our declarative framework, then make an excursion into the issue of design pattern detection.

Wuyts [Wuy01] presents a declarative framework used to express programming conventions, design pattern structures and UML class diagrams. Though our declarative framework is similar to that of Wuyts in that it is a layered structure of logic predicates, the two are very different in scope and intent. First, our framework makes use of static information but relies most heavily on dynamic information for the extraction of behavioral views, whereas the SOUL Synchronization Framework uses only static information. Second, our framework is intended principally to support view recovery whereas the SOUL Synchronization Framework [Wuy01] is integrated in the development environment in a way which permits the synchronization of design and implementation. Design documentation over static information is expressed in the form of a logical predicate. Such documentation can be derived from the implementation, and implementation can be constrained by the design documentation.

Mens [Men00] chooses logic programming as a formalism for architectural conformance checking using static information only. Similar to the work on the declarative framework using SOUL [Wuy01], this approach uses a logic programming language to express the architecture of a software system and then to check the conformance of the software to the postulated architecture. In order to arrive at the description of the architecture in the first place, however, the code must be reverse engineered. This is a labor intensive work even with architectures that are well understood (mapping of architectural concepts to the code artifacts) but is very difficult for applications which are not well understood. This conformance checking approach does not address the reverse engineering step.

Prolog has also been used to query static program information in order to find structural design patterns [KP96] and to detect violations of programming conventions and rules [Sef96][Wuy98][Men00][Ciu99]. Our declarative framework also supports querying of the source model in terms of any logic relation which can be expressed over the representation and base layers. Though this kind of querying is useful when a user has a very specific question which can be precisely formulated, it is limited in conveying a more general understanding of an application, or in answering questions which cannot be easily formulated in a logic-programming language. In the next section we make a short excursion to illustrate the limits of such punctual querying for design extraction and program understanding.

4.4.1 Queries vs. Views: the Example of Design Pattern Detection

In this chapter we presented an approach to extracting design views of a software system using logic predicates. The source model can also be simply queried using the predicates described so far, for example to look for singleton classes in an application or to find out which classes create instances of another class. It is natural to ask about using such a querying approach for the detection of design patterns. In this section we demonstrate how this can be done and discuss the problems and limitations of this kind of querying. We argue that the iterative recovery of design views contributes more to our understanding of a software system than does punctual querying about the presence of a design pattern.

Using predicates in the representation and base layers, queries can be formulated to describe a design pattern structure and behavior. As an example, the structure of the Composite pattern in shown in Figure 4.10, and Rule 10 below shows one possible formulation for the Composite pattern.

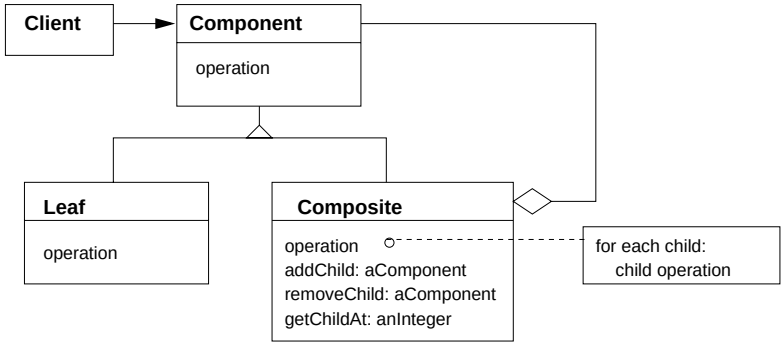

Figure 4.10: Structure of Composite Design Pattern

Rule 10 consists of two clauses: The first specifies that a Composite class must override Operation from a Component class, while the second models the delegation of Operation to the Composite's children using rule 11. Rule 10 makes use of static information only, whereas rule 11 makes use of dynamic information in the two predicates sendsToMethodSequence and sendToMethodInStack. Rule 11 specifies that there is an invocation of Operation on an instance of Composite which results in an instance of Composite invoking Operation on an instance of Leaf.

```
rule10 : compositePattern(Component,Composite,Operation) :-
             overriddenMethod(Composite,Component,Operation),
             compositeDelegation(Component,Composite,Operation).

rule11 : compositeDelegation(Component,Composite,Operation) :-
             sendsToMethodSequence(_,Composite,Operation,Start),
             sendToMethodInStack(Composite,Leaf,Operation,Start),
             inHierarchy(Leaf,Component),
             not(equal(Component,Leaf)),
             not(equal(Composite,Leaf)).
```

Rule 11 above looks for a delegation structure in the dynamic information where there is delegation to at least one instance of some Leaf subclass, rather than to several instances of Leaf, or to instances of different Leaf classes. An alternative formulation would be to look for delegation to several children or to check that operations corresponding to addChild, removeChild and so on are also present in the structure, either statically or from dynamic information.

Problems and limitations. As discussed in Section 2.2.4 on design pattern detection, formulating distinctive and non-ambiguous criteria for the presence of a design pattern is difficult. In particular, the formulation of the rule as given here would also detect an instance of the Proxy pattern, since in the Proxy pattern (Figure 4.11) a received request is also forwarded to an instance of a class of the same type.

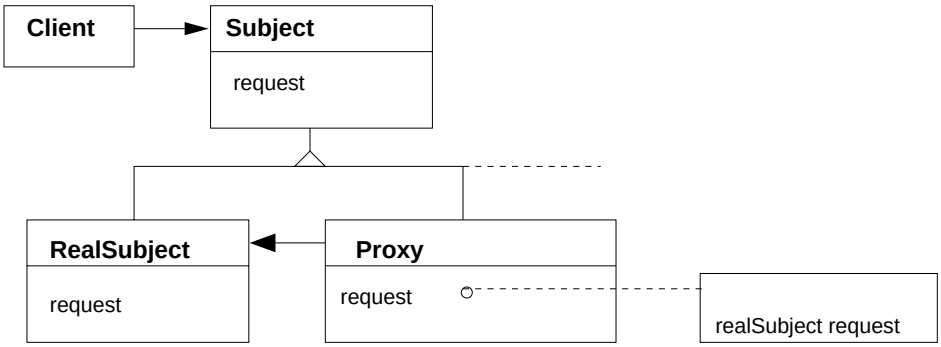

Figure 4.11: Structure of Proxy Design Pattern

It is, then, possible to detect the presence of design patterns using this approach, by encoding a heuristic as a logic rule. But that the inherent limitations of design pattern detection as discussed in Section 2.2.4 still remain. Whereas the detection of a design pattern rarely contributes to our global understanding of the structure and behavior of the software, it can help us to understand the role of a particular class or the interaction of a set of classes.

4.5 Conclusions

In this chapter we described a way to express component-connector perspectives for the recovery of concept views using a logic programming language.

To support component-connector perspectives, we developed a declarative framework of logic programming rules which consists of several layers. The lowest layer, the representational layer, contains predicates which express static and dynamic relations extracted directly from the code and its execution. The extracted information is thus represented as facts which constitute the source model. The base layer consists of predicates which express relations derived from those in the representational layer. Whereas the representational and base layer are predefined in the framework, the auxiliary layer is defined by

the engineer, and contains higher level relations and application-specific predicates. Finally, we presented the perspective layer: this layer defines predicates which are used to generate concept views.

We showed how component abstractions are defined by using a logic predicate to group together static entities of the software, and demonstrated how the semantics of a connector is given by defining a logic predicate in terms of message send events. Finally we presented the main types of perspectives. Perspectives are second-order predicates which take a component predicate and a connector predicate as arguments.

Perspectives specify concept views in a simple declarative way. The framework we present is extensible: new component clustering rules and connector relations can be defined by the developer and new perspectives can be obtained by new combinations of these. Though we have not yet seen the recovery process in action we have made a step towards showing that the view specification is *simple* and *extensible* – two requirements we have set for our approach in Section 2.3.

We also presented Gaudi, the tool we developed to implement the declarative framework, and discussed some of the design choices related to the tool. In the next chapter we describe two case studies which show how Gaudi supports reverse engineering and program understanding, and thus demonstrate the validity of our approach.

5

Concept View Recovery: Case Studies

To evaluate component-connector perspectives, as supported by the declarative framework described in the previous chapter, we applied the Gaudi tool on two case studies. Our claim is that the use of these perspectives in design recovery meets the requirements we have set for our approach, as outlined in Section 2.3. The goal of these case studies is is to lend support to this claim. The two case studies therefore 'walk' the reader through an iterative process of view recovery. We show how we obtain both high-level and low-level succinct behavioral views, and demonstrate the simplicity and extensibility of the view specification. At the end of the chapter we discuss and evaluate concept view recovery as a whole.

Each case study is presented as follows: we first briefly describe the software application to be investigated – information gleaned from existing documentation and from browsing the code. We then posit a question about the software. Next we walk the reader through the process we followed in seeking an answer to this question: extracting the source model, defining perspectives and analyzing the views which result from applying these perspectives.

This chapter is structured as follows: in Section 5.1 we present the first case study, in which we apply the approach to HotDraw in order to understand how the Tool metaphor works. The second case study is presented in Section 5.2, where we apply the approach to the Refactoring Browser to look for a facade. In Section 5.3 we summarize the lessons learned, sketch a methodology, discuss usability issues and present related work. In Section 5.4 we revisit the requirements and evaluate our approach. Conclusions are presented in Section 5.5.

5.1 Understanding Tools in HotDraw

In the first case study we investigate the HotDraw framework. HotDraw [BJ94] is a framework for semantic graphic editors. It allows for the creation of graphical editors which associate the picture with a data structure – that is, changing the picture also changes the data structure. The HotDraw framework consists of 114 Smalltalk classes and comes with several sample editors.

From the documentation we learn that HotDraw is based on the Model-View-Controller triad [KP88]; these roles are played by the classes Drawing, DrawingEditor and DrawingController respectively. Furthermore, it has a few other basic elements: *tools* are used to manipulate the drawing which consists of *figures* accessed through *handles*. *Constraints* are used to ensure that certain invariants are met, for example, that two figures connected with a line remain connected if one of the figures is moved.

Though HotDraw has been documented in several publications, including patterns for customizing the framework [Joh92], the overall view of the framework is never described. Moreover, several changes have been made to the framework over the years, in particular to the implementation of tools and of constraints, and most of the documentation is out of date with respect to the version 3.0 we were using.

Tools are used to manipulate the drawing: create new figures or manipulate the existing figures. On the drawing editor tools are represented by icons on the top panel (see Figure 5.1). Browsing version 2.5 and version 3.0 of the code shows us that the implementation of tools has changed considerably. In the earlier version tool responsibilities were handled by the classes Reader, Command and Tool, whereas in the current version (3.0) different tools are implemented through states. We seek to understand how tools are implemented in the current version.

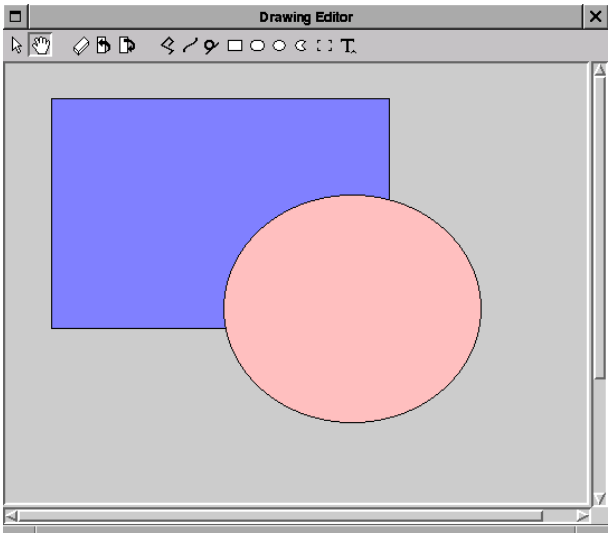

Figure 5.1: HotDraw sample editor

5.1.1 Extracting the Source Model

The logic database was first filled with static and dynamic information for the HotDraw framework. Static information was obtained by parsing, whereas dynamic information was obtained by instrumenting all the methods and running a scenario on one of the sample applications, DrawingEditor, to generate an execution trace. Section 4.3 discusses the process of creating the source model in more detail.

The scenario we ran on the DrawingEditor application to obtain an execution trace consisted of the creation of several kinds of figures (rectangle, rounded rectangle, bezier, text and image), deletion of a figure, grouping and ungrouping of two figures (rectangle and rounded rectangle), cutting and pasting one of the figures, moving a simple figure and a grouped figure, changing fill and line color of some of the figures and finally quitting the application. Running this scenario generated 59,277 method invocation events.

5.1.2 Understanding Tools

1. An initial view. The starting point is the first view generated, shown in Figure 5.2. This view was generated using Perspective 1, as given in Section 4.2.2. It shows all traced communication between instances, grouped by the Smalltalk category to which the class belongs.

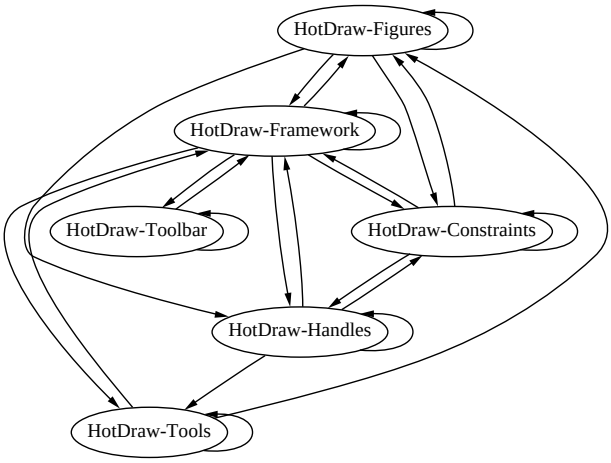

Figure 5.2: View 1: Invocations between class categories for the HotDraw sample editor

Analysis. This view gives some rough idea of the relationships of the main parts of HotDraw. In particular, we see that the HotDraw-Tools component communicates with the HotDraw-Framework, and the HotDraw-Handles components. We also see that the HotDraw-Toolbar component communicates only with the HotDraw-Framework component, that HotDraw-Constraints communicates with HotDraw-Figures and HotDraw-Handles as would be expected.

We make, however, two observations with respect to this initial view. First, since the HotDraw-Framework category contains several of the main HotDraw classes, in particular DrawingEditor, Drawing and DrawingController, we want to view these separately. Second, we do not necessarily want to see all the invocations in one view. In particular, we want to distinguish between invocations which create instances, and those in which instances just invoke methods on each other.

2. Clustering. We therefore first define a new component breakdown as shown below. This clustering creates five components: Figure, Handle, Constraint, Toolbar and Tool.

Classes which are not mapped into components are considered themselves components. This clustering is used to view invocations and creation relationships (see Figure 5.3 and Figure 5.4).

```
component('Figure',L) :- allInCategory('HotDraw-Figures',L).
component('Handle',L) :- allInCategory('HotDraw-Handles',L).
component('Constraint',L) :- allInCategory('HotDraw-Constraints',L).
component('Toolbar',L) :- allInCategory('HotDraw-Toolbar',L).
component('Tool',L) :- allInCategory('HotDraw-Tools',L).
```

3. Invocations between components. To obtain a view of the invocations between components as defined above, we specify Perspective 2. The rule sendsTo(Class1,Class2) evaluates to true if an instance of Class1 invokes a method on an instance of Class2. This perspective is used to obtain the view in Figure 5.3.

	composedView(sendsTo,component).
r	sendsTo(Class1,Class2).
C	component(ComponentName,ListOfClasses).

Perspective 2: Invocations between components

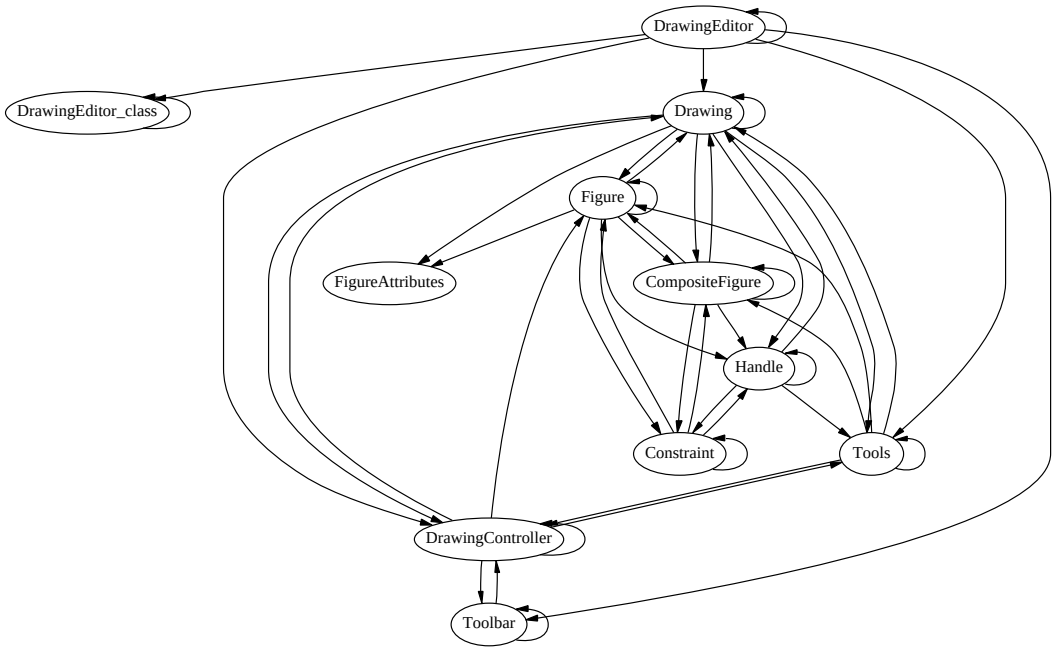

Figure 5.3: View 2: Invocations between components

Analysis. From Figure 5.3 we see that Drawing is central to the application – it communicates with Figure, Tool, Handle, DrawingEditor and DrawingController. FigureAttributes is invoked only by Drawing and Figure, and as the name suggests, probably stores attributes like line width and color which are relevant for all figures in the drawing. The Toolbar component is relatively independent of the other components and invokes only methods on DrawingController.

The Tools component receives invocations from the following components: DrawingController, Handle, Drawing and DrawingEditor. It sends invocations to the following: DrawingController, Drawing, Figure and CompositeFigure.

4. Creation Invocations. To distinguish between invocations which create instances and other kinds of invocations, we define a rule which specifies a *create* relationship between classes.

```
rule12 :  sendsCreate(Class1,Class2) :-
              sendsToMethod(Class1,MetaClass,Method),
              metaclassOf(MetaClass,Class2),
              methodCategory(MetaClass,Method,'instance creation').

          sendsCreate(Class1,Class2) :-
              indirectsend(_, _,Class1, _,'new',Class2, _, _).
```

The first part of rule 11 – the same as rule 5 in Section 4.1.2 – captures the events in which a method belonging to the instance creation Smalltalk method category is invoked on a metaclass. The second part of rule 11 captures the events in which Class1 sends an unobserved (not instrumented) method new, resulting in an invocation of some method on Class2. Perspective 3 below is then used to generate a create *creation* view in which these components appear, as shown in Figure 5.4.

	composedView(sendsCreate,component).
r	sendsCreate(Class1,Class2).
C	component(ComponentName,ListOfClasses).

Perspective 3: Creation invocations between components

Analysis. This view (Figure 5.4) tells us that DrawingEditor creates Drawing, which creates DrawingController, as expected from the Model-View-Controller pattern. In contrast to the other figures which are created by Tool, CompositeFigure is created by Drawing. As the components Handle, DrawingEditor and DrawingController all point to Tool, there is some ambiguity about the creation of this component. To find more about the creation responsibilities there we must split the Tool component into its constituent parts to get a closer look.

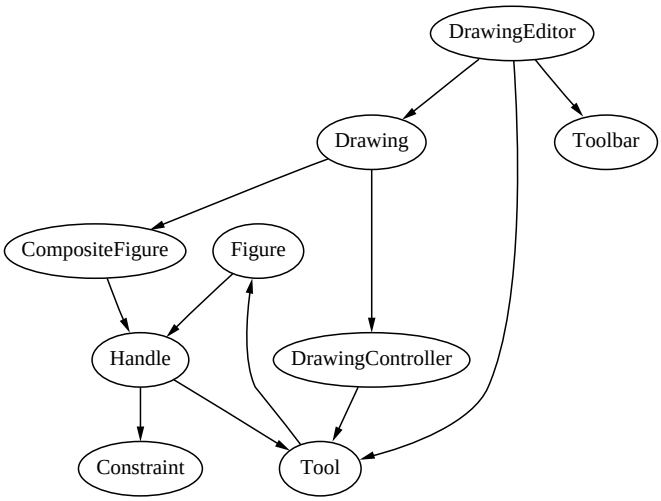

Figure 5.4: View 3: Creation invocations between components

5. Multiplicity of Creation Invocations. We want to get a better understanding of how tools are implemented. For this we generate a new view which gives us the creation relationships between the classes, rather than the components. Furthermore, we want to know the multiplicity of these relationships, that is, whether each instance of a figure creates only one or several instances of a tool. To do this we define Perspective 4, used to generate the view in Figure 5.5. In this view a filled edge $A \rightarrow B$ means that there is an instance of class A which creates only one instance of class B. A dotted edge $A \rightarrow B$ means that there is an instance of Class A which creates several instances of class B.

	view(oneToOneCreate), addView(oneToManyCreate).
$r1$	oneToOneCreate(Sender,Receiver).
$r2$	oneToManyCreate(Sender,Receiver).
C	components are classes

Perspective 4: Multiplicity of creation invocations between classes

Analysis. From this view (Figure 5.5) we see that instances of the class Tool are created by DrawingEditor, and that one DrawingEditor creates several instances. The class DrawingController also seems to create one instance of Tool; looking more closely at this, through querying about this creation event, we discover that when the application starts up and DrawingController is initialized, it invokes selectionTool on an instance of Tool_class. The other tools are created later when the DrawingEditor creates a tool for each icon in the toolbar.

In Figure 5.5 we had expected to see a creation relationship between Tool and ToolState. To understand this singularity we browsed the code and found that the Tool class (Tool_class in the source model) instantiates ToolState at load-time using class methods,

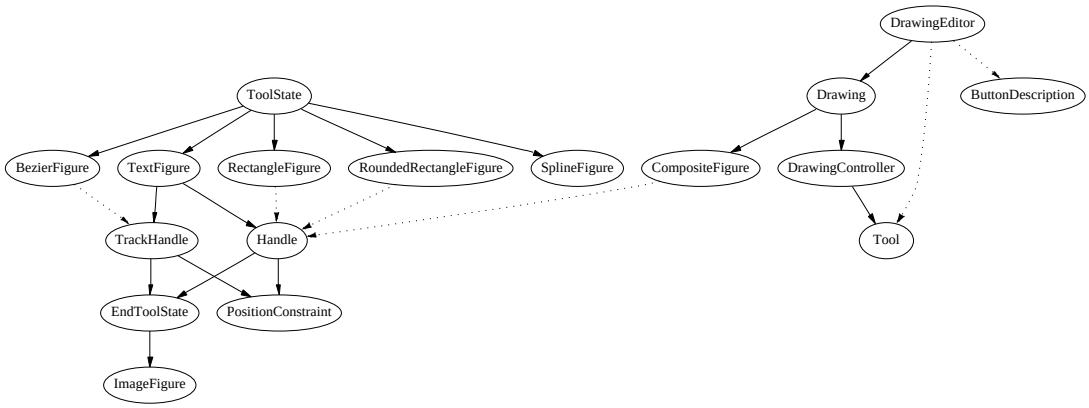

Figure 5.5: View 4: Multiplicity of creation invocations between classes. Filled edges shows a one-to-one creation relationship, dotted edges a one-to-many creation relationship.

and so instances of ToolState were created before the instrumented application was run. We also discovered by browsing that Tool class also creates an instance of EndToolState representing the last state of the tool state machine. Going back to the view in Figure 5.5 we wondered why instances of Handle and TrackHandle were creating more instances of EndToolState. Again, by browsing the code, we understood that a Handle contains a per default action that is to return to the end state of the tool state machine.

6. Tool, ToolState, SimpleTransitionTable: a Collaboration. We now have an inkling about the implementation of tools using state machines. To understand this in greater depth we define Perspective 5 to generate a view showing the communication between instances. By querying we detect a creation invocation and identify a trace sequence corresponding to a mini-scenario around the creation of an instance of RectangleFigure. We then display the invocations between instances (Figure 5.6) in a form similar to a UML collaboration diagram [BRJ99]: invocations are numbered sequentially in the order they occur. Rule sendScenarioMethodforHotDraw(Class1/Inst1,Class2/Inst2,Method,Number) defines an invocation relationship between instances. The predicate holds true if Inst1 of Class1 invokes method Method on Inst2 of Class2. Number represents the sequence number of the method invocation in the mini-scenario.

---

labelledOrderedView(sendScenarioMethodforHotDraw).

r sendScenarioMethodforHotDraw(Class1/Inst1,Class2/Inst2,Method,Number) :-
sendInstanceInStackMethod(Class1/Inst1,Class2/Inst2,Method,Number,Start).

C components are instances

Perspective 5: Instance invocations around the creation of a RectangleFigure

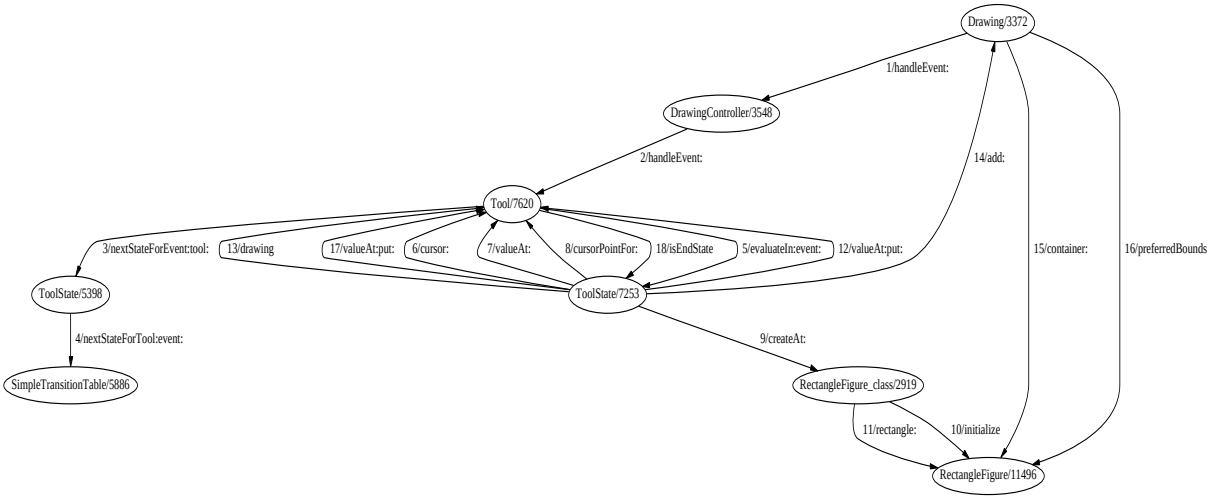

Figure 5.6: View 5: Instance invocations around creation of a RectangleFigure

Analysis. The collaboration of Figure 5.6 is interpreted as follows: when a user event occurs Drawing tells the DrawingController to handle the event. The DrawingController then tells the Tool to handle the event. The Tool consults with the current tool state ToolState/5398 (which in turn consults the SimpleTransitionTable) to get the next tool state for the event. It then tells the appropriate state ToolState/7253 to handle the event. ToolState/7253 creates an instance of RectangleFigure, and tells the Drawing to add this figure to itself. Drawing then repairs itself by asking RectangleFigure about its bounds.

7. Summary. We began with a coarse-grained view showing us the general pattern of communication between domain concepts in the HotDraw framework. This information was used to get a new component clustering and to focus in more detail on the communication in which the Tool elements engage. We learned about the creation of instances of Tool, and about how an instance of Tool handles the creation of a RectangleFigure. We see that we move from coarse grained views to finer grained views in which we finally look at the collaboration of instances. In this case study we looked at one collaboration in the program trace by querying to locate a creation event. In the next chapter we introduce collaboration view recovery: an approach which enables us generalize from one particular collaboration of instances to the notion of class collaboration.

5.2 Looking for a Facade in the Refactoring Browser

In this section we apply our approach to the Refactoring Browser [RBJ97]. Refactorings are behavior preserving transformations of source code [Opd92] [FBB+99]. The Refactoring Browser [RBJ97] is a Smalltalk browser which offers refactoring support: it lets a programmer make refactoring changes such as renaming a class, moving methods from one class to another, pushing a method down the inheritance hierarchy, *etc.* and automatically propagates these changes in the code so that it is in a consistent state again.

This case study is motivated by the work of a colleague on defining refactorings on the FAMIX meta-model [Tic01]. Sander wanted to use the Refactoring Browser as a back end to his language-independent refactoring system, to carry out the actual refactoring changes once a refactoring was deemed possible. So he wanted to know if there was a facade in the Refactoring Browser: that is, a class or component (grouping of classes) which presented an interface to which a description of a refactoring or of a series of elementary refactorings could be sent and which would then implement the elementary changes which together constitute a refactoring. Figure 5.7 illustrates this schematically.

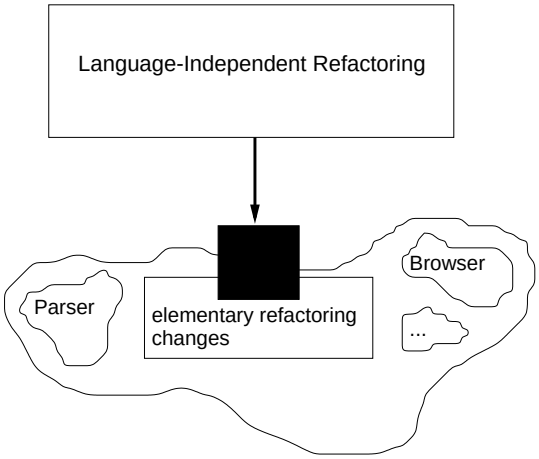

Figure 5.7: Looking for a facade in the Refactoring Browser

Though we hope to find a facade, if there is no such facade, then we want to know how the Refactoring Browser performs refactoring changes so that we can figure out how to write an application which can still make use of the Refactoring Browser.

5.2.1 The Refactoring Browser

In this case study we worked with version 3.0.1 of the Refactoring Browser. The Refactoring Browser consists of 151 Smalltalk classes arranged in 10 categories. Browsing through the code showed that the main hierarchies correspond quite well to the Smalltalk category arrangement.

- **Browser:** subclasses of the Smalltalk environment class ApplicationModel which handle the browser aspects.
- **Parser:** several hierarchies dealing with parsing of the code.
- **Condition:** a small hierarchy of conditions, to check if a refactoring can be carried out.
- **Refactorings:** contains classes named with the kinds of refactorings possible. All classes are subclasses of the root class, Refactoring.

- **Support:** contains classes named with more elementary kinds of refactorings. Most classes inherit from a root class RefactoryChange.
- **Navigator:** different kinds of navigator classes, all inheriting from Navigator, a subclass of BrowserApplicationModel.
- **Environments, OMT-diagram, Lint, Code-Tools**

We were interested only in the classes involved in the actual execution of a refactoring, and we therefore ignore the categories Browser, Environments, OMT-diagram, Lint and Code-Tools. Our initial understanding of how a refactoring is carried out was as follows: the Refactoring Browser first parses the code (using classes in the category Parser), then checks that the refactoring can actually be carried out (using classes in the category Condition), then carries out the refactoring. Since we were not interested in the parsing of the code or the checking of conditions, the categories which remain and which look the most promising for further investigation are Refactorings and Support.

5.2.2 Extracting the Source Model

To create the source model we first parsed the code to obtain a static model of the Refactoring Browser. To obtain the dynamic information, a first scenario was run, as described below. Later on in the case study the second scenario will be described. Though we have an initial idea about the roles of the different categories and classes, we first trace all classes in order to confirm our initial hypothesis about the application and to be able to focus on the relevant classes.

Scenario 1. We instrument all classes of the Refactoring Browser, and run a scenario which consists of the renaming of a class. This scenario generated 1,187 method invocation events.

We start by analyzing this first scenario. We assume here, of course, that there is some coherence to the system – that is, each refactoring is analyzed, handled and carried out in a similar way. This assumption is tested later in the case study when we run a new scenario with two different refactorings.

5.2.3 Looking for a Facade

1. Looking for a Facade pattern. Facade is a design pattern whose intent is to *"provide a unified interface to a set of interfaces in s subsystem. Facade defines a higher-level interfaces that makes the subsystems classes easier to use"* [GHJV95]. Since a Facade class provides a single interface to the services offered by other classes, its role at runtime is to forward requests to the appropriate objects. At runtime there is therefore usually only one instance of the Facade at any moment – it is either a singleton (a unique instance throughout the whole scenario), or it may be created and disposed of (in which case several instances will occur in the scenario). There can, however, be multiple instances of a Facade as well, each providing a facade to a different subsystem.

Rule 14 below tests for the presence of a singleton: a class for which there is a unique instance throughout the whole scenario. Rule 13 tests of the presence of a Facade. It says a class fulfills the role of a Facade *vis-a-vis* a Client class if:

1. it invokes methods on a set of classes, listed in SubSystemClasses.
2. the Client does not invoke any methods on any of the classes in SubSystemClasses

```
rule13 : facadePattern(Client,Facade,SubSystem) :-
             sendsToNoMeta(Client,Facade),
             setof(Class,(sendsToNoMeta(Facade,Class),not(equal(Class,Facade))),SubSystem),
             findall(Cl,(sendsToNoMeta(Client,Cl),member(Cl,SubSystem)),[ ]).

rule14 : singleton(Class) :- numberofInstances(Class,1),
             not(metaclass(Class)).
```

Although a declarative encoding for a Facade is quite straightforward, testing for the presence of a facade without knowing where to look (*i.e.*, which class could be a candidate for Client or Facade) is very inefficient. Furthermore, if a facade is not detected then we have learned nothing about how the application really works and can be modified to serve our purpose. We therefore choose to first generate a high-level view of the application, to get a better idea of where to look for a facade.

2. Obtaining a high-level view. We first generate a view which shows the communication going on in the execution of the first scenario. The objective here is to form a first hypothesis as to the location of such a facade and to build components out of the application elements which allow us to focus on some parts to the exclusion of others, by grouping together parts which we do not consider relevant for our investigation. To do this we first generate a view in which components correspond to Smalltalk class categories, as in Perspective 1 in Section 5.1.

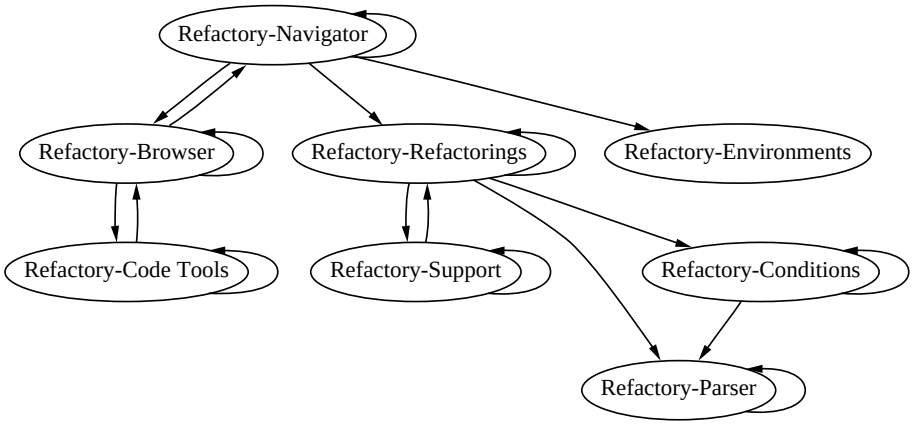

Figure 5.8: View 1: invocations between categories

Analysis. This view (Figure 5.8) shows a tree structure with three main branches. We know that we are interested more in what is happening in the middle branch, in which

Change and Refactoring classes participate. This view then confirms our suspicion that Browser, Code Tools and Environments categories can be ignored in the investigation. We want to open up the categories Navigator, Refactorings and Support, to see the communication between these.

3. Refining the high-level model. In order to get a better idea of what is going on we then define a new component breakdown where the Browser, Code Tools and Environment categories are grouped together; Parser and Condition are left as category groupings, whereas the Refactorings and Support categories are opened up to display the individual classes. This component breakdown is then used to visualize invocations and create invocations, as shown in Figure 5.9.

```
inComponent('Browser+Stuff',Class) :- class(Class,'Refactory-Environments').
inComponent('Browser+Stuff',Class) :- class(Class,'Refactory-Browser').
inComponent('Browser+Stuff',Class) :- class(Class,'Refactory-Code Tools').

component('Parser',L) :- allinCategory('Refactory-Parser', L).
component('Conditions',L) :- allinCategory('Refactory-Conditions',L).
component('Browser+Stuff',L) :- setof(Class,inComponent('Browser+Stuff',Class),L).
```

---

composedView(sendsToNoMeta,component),addComposedView(sendsCreate,component).

$r1$ sendsToNoMeta(Class1,Class2).
$r2$ sendsCreate(Class1,Class2).

C component(ComponentName,ListOfClasses).

Perspective 2: Invocations and creation invocations between components

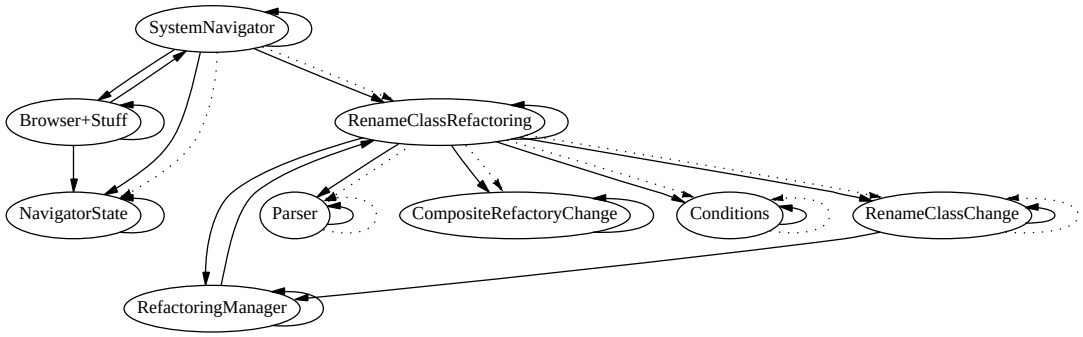

Figure 5.9: View 2: invocations and creations between components: dashed lines are create relationships.

Analysis. On the basis of its name, RefactoringManager shows some promise as our facade. The names of the classes RenameClassRefactoring and RenameClassChanges suggest that they handle changes particular for the rename class refactoring. The class

CompositeRefactoryChange suggests more generic functionality, but it looks like a dead-end, since it invokes no methods on other classes. We hypothesize that the job of RenameClassRefactoring is to invoke Parser to parse the code and to figure out if the refactoring can be carried out (using Conditions). It then requests RenameClassChange to carry out the actual refactoring. The roles of RefactoringManager and CompositeRefactoryChange is still unclear.

4. Querying about a Facade pattern. Because being a singleton is one indication for a facade, we query about the presence of singletons at this point, using Rule 14, and discover that two classes are singletons: RefactoringManager and RenameClassRefactoring. The name of RenameClassRefactoring does not suggest a class which would handle a range of different refactorings. We query using Rule 13 about whether these two classes play the role of a facade:

```
?— facadePattern(Client,'RefactoringManager',SubSystem).

Client = 'RenameClassChange',
SubSystem = ['RenameClassRefactoring']

?— facadePattern(Client,'RenameClassRefactoring',SubSystem).

Client = 'SystemNavigator',
SubSystem = ['CompositeRefactoryChange','Condition','ConjunctiveCondition',
'ParseTreeRewriter','RefactoringManager','RenameClassChange']
```

Analysis. The result of the first query suggests that RefactoringManager is a facade to the subsystem composed of RenameClassRefactoring, where the client class is RenameClassChange. Because we know from browsing the code that Change classes carry out elementary changes, while Refactoring classes are responsible for a set of elementary changes, it is unlikely that this is the facade we are looking for. We reformulate Rule 13 to check if a delegation occurs (An invocation of a method on the Facade results in a method invocation to an instance of a class in the subsystem).

rule15 :

```
facadePatternDelegation(Client,Facade,SubSystem) :-
    facadePattern(Client,Facade,SubSystem),
    sendsToMethodSequence(Client,Facade,_,Start),
    member(Class,SubSystem),
    sendToMethodInStack(Facade,Class,_,Start).
```

```
?— facadePatternDelegation('RenameClassChange','RefactoringManager',
        ['RenameClassRefactoring']).

no
```

Analysis. This second query confirms our suspicion that RefactoringManager is not the facade we are looking for. We learned from the previous query also that RenameClass-Refactoring is a Facade for the client SystemNavigator. This structure is also seen from Figure 5.9. Although we have not found a single class which plays the role of a facade, we might find a class, or set of classes which could be used as a facade. We next execute a second scenario in order to get a better understanding of how different refactorings are carried out.

5. Comparing with a second scenario. We run a second scenario in order to compare our view to the view given by another execution. This scenario differs from our first scenario in two ways: first, this time we trace classes in only two categories, eliminating some of the clutter we consider irrelevant for our investigation. Second, we run a scenario where we perform two different refactorings, so as to be able to compare how different refactorings are carried out.

Scenario 2. We instrument the classes in the two categories Refactory-Refactorings and Refactory-Support, as well as the class SystemNavigator, and run a scenario which consists of a refactoring to push a method down the inheritance hierarchy, then refactoring to rename a method. This scenario generates 2,492 method invocation events.

	view(sendsToNoMeta),addComposedView(sendsCreate).
$r1$	sendsToNoMeta(Class1,Class2).
$r2$	sendsCreate(Class1,Class2).
C	components are classes

Perspective 3: invocations and creations between classes

We then obtain a view which shows the invocation and the creation relations between classes, shown in Figure 5.10.

Analysis. The view of Figure 5.10 shows a structure parallel to the view of Figure 5.9. Parallel to the RenameClassRefactoring class, there are now two classes RenameMethodRefactoring and PushDownMethodRefactoring. SystemNavigator creates instances of these classes, which in turn create instances of CompositeRefactoryChange, which in turn creates instances of RemoveMethodChange and AddMethodChange. We hypothesize that CompositeRefactoryChange coordinates the elementary refactoring changes involved in one conceptual refactoring. RemoveMethodChange and AddMethodChange also create instances of each other, so this is a new puzzle. Further, as does RenameClass-Changes in Figure 5.9, they also invoke methods on RefactoringManager. The roles of CompositeRefactoryChange and RefactoringManager are still not clear. Though Figure 5.10 suggests that CompositeRefactoryChange could be a facade, information from our first scenario casts doubts on this.

6. Asking about RefactoringManager **and** CompositeRefactoryChange**.** We now use Scenario 2 to query about these two classes. We check to see if they are singletons:

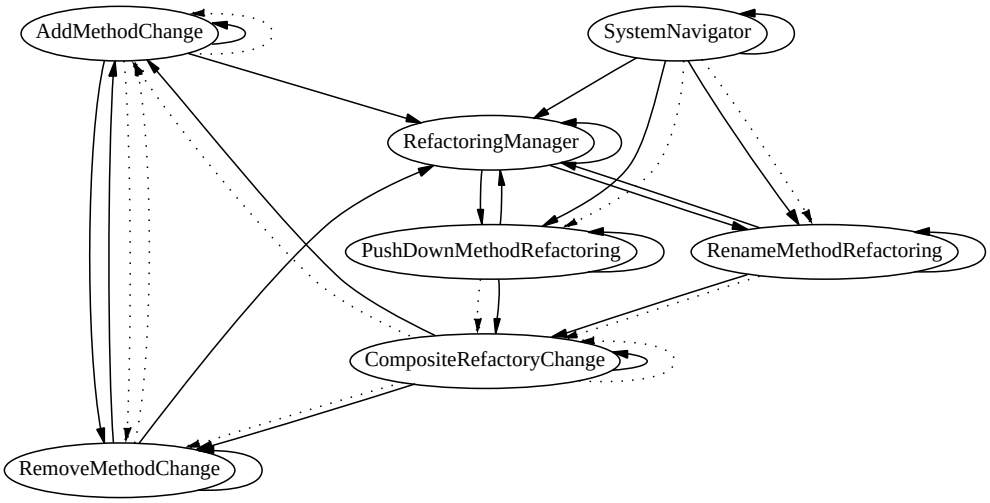

Figure 5.10: View 3: invocations and creations between classes. A filled edge corresponds to an invocation relationship, a dotted edge to a creation relationship.

RefactoringManager is a singleton, CompositeRefactoryChange is not. We query first about whether CompositeRefactoryChange is a facade, then ask about its public interface, and that of RefactoringManager.

```
?— facadePatternDelegation(Client,'CompositeRefactoryChange', SubSystem).

Client = 'PushDownMethodRefactoring',
SubSystem = ['AddMethodChange','RemoveMethodChange'] ? ;

Client = 'RenameMethodRefactoring',
SubSystem = ['AddMethodChange','RemoveMethodChange']


?— publicInterface('CompositeRefactoryChange',L).

L = ['addChange:','addChangeFirst:','compile:in:classified:','execute',
'executeWithMessage:','initialize','name:','removeMethod:from:']

?— publicInterface('RefactoringManager',L).

L = ['addRefactoring:','ignoreChangesWhile:','update:with:from:']
```

Analysis. We know that CompositeRefactoryChange is not a singleton. However, it can be a facade which is created and disposed of each time it is needed. We look at the interface it presents to its client classes to determine if it is receiving a description of a refactoring and executing it by carrying out elementary refactoring changes. Its public interface suggests that it keeps a list of changes (addChange: method), and then executes these changes (execute method). Browsing the code for CompositeRefactoryChange we

see that it presents an interface for different composite refactoring, and creates more elementary refactoring changes to carry out the composite refactorings. We would like to look at the run-time object structure to understand the role of different instances of CompositeRefactoryChange.

7. Role of RefactoringManager**.** Here we make a detour to find out what the role of RefactoringManager is. To see who sends what messages to RefactoringManager we create the view in Figure 5.11:

---

labelledView(sendsToFromRefactoringManager)

r sendsToFromRefactoringManager(C1,C2,M) :-
sendsToMethod(C1,'RefactoringManager',M);
sendsToMethod('RefactoringManager',C2,M).

C components are classes

Perspective 4: Senders and receivers of RefactoringManager

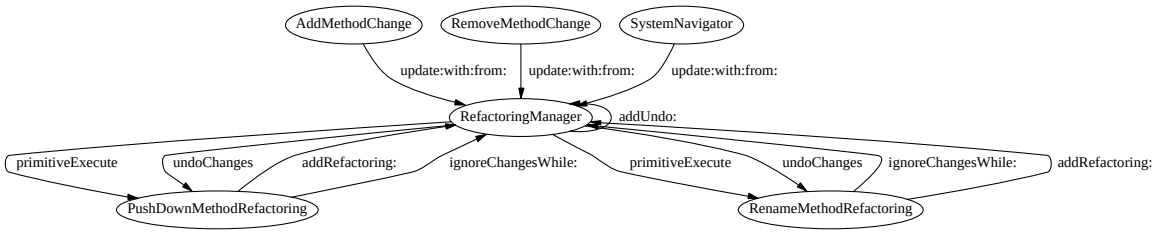

Figure 5.11: View 4: Senders and receivers of RefactoringManager

Analysis. RefactoringManager is a singleton which seems to play the role of a transaction manager with which elementary refactoring changes register so that they can be undone in case that the refactoring cannot succeed. So RefactoringManager is definitely not a facade to carry out refactoring changes. We now look at the object structure of a refactoring.

8. Looking at the object structure. To understand better what is happening at the object level, we query to locate an invocation of PushDownMethodRefactoring and ask about the invocations between objects within the call stack for this method.

?— direct(N, L, 'SystemNavigator',_, _, 'PushDownMethodRefactoring',_, 'execute').

L = 2,
N = 2135 ?

---

view(sendScenarioforRefactoringNoLabelNoMeta)

r sendScenarioforRefactoringNoLabelNoMeta(Class1/Obj1,Class2/Obj2):-
sendObjectInStackNoSelfNoMeta(Class1/Obj1,Class2/Obj2,_,2135).

C components are instances

Perspective 5: Object invocations in the execution stack of a Push Down Method refactoring

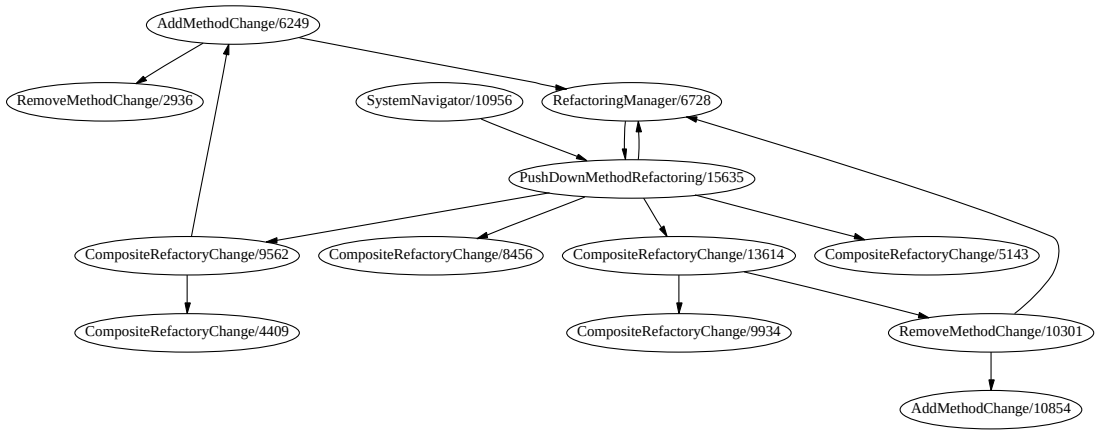

Figure 5.12: View 5: Object invocations in the call stack of Push Down Method refactoring

Analysis. Figure 5.12 shows the pattern of object communication in the call stack resulting from the invocation of execute on PushDownMethodRefactoring. Since we have already identified the role of the singleton RefactoringManager, and since in the resulting graph it acts as a central 'sink', we remove it from the graph through editing, and obtain Figure 5.13. This figure shows a layered object structure, and elucidates the structure of a refactoring. It shows that many objects are involved in carrying out a refactoring. In particular, there are several instances of CompositeRefactoryChange. We are puzzled by the different roles these seem to play: some instances invoke methods on instances of elementary refactoring like AddMethodChange and RemoveMethodChange, other instances invoke methods on instances of CompositeRefactoryChange, and several instances do not invoke any methods. Again, we browse the code to understand better what is going on. We understand that in this object structure there is in fact one object for each elementary and composite change that must be carried out, and a parallel object for undoing these changes. This suggests that we cannot really use the class CompositeRefactoryChange as a facade to which we send elementary refactorings – in order to use the RefactoringBrowser as a back end, we would have to create such an object structure to represent the refactoring.

9. Summary and resolution. Browsing the code shows that the class Refactoring, the root class of all the refactorings, relies on a method execute to carry out the refactor-

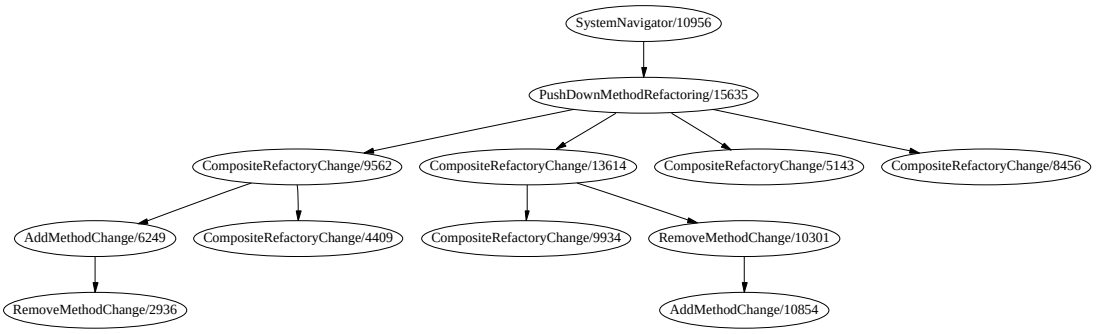

Figure 5.13: View 5: Object invocations in the call stack of Push Down Method refactoring. This is the same view as in Figure 5.12, with the node RefactoringManager/6728 removed, in order to illustrate the hierarchical structure of the refactoring

ing. Each refactoring is executed by first checking the preconditions, then making the necessary transformations, and finally by 'registering' the refactoring with the RefactoringManager.

From the analysis above we see that each kind of refactoring necessitates the creation of an object structure of composite and elementary changes. This poses a problem for using the Refactoring Browser as a backend for elementary changes only. Sander solved this problem as follows: he defined his own structure of refactoring classes parallel to the Refactoring hierarchy of the Refactoring Browser. These classes delegated the checking of preconditions to the language independent model of the code (instead of the parsed model the Refactoring Browser creates). Then the body of the execute method of each of his refactoring classes invoked the execute method of the elementary refactorings of the Refactoring Browser. These elementary refactorings then took over and created the appropriate Refactoring Browser objects to handle the actual changes. The transactional nature is still retained, since each elementary refactoring registers with the RefactoringManager [Tic01].

5.3 Evaluation and Discussion

5.3.1 Lessons Learned

Here we highlight some of the issues related to the use of the approach which came out of the case studies we performed.

The role of an initial question. The approach worked best when we started with a well-formulated question about the code. This allowed us to quickly formulate an appropriate perspective in order to confirm or reject the hypothesis we had about the code, and to move on to formulate a new hypothesis or question. We cannot yet evaluate how well the approach works when the maintainer starts with a very general objective, for example, "let's see if there is anything strange in this code" – that is, how good it is at revealing design anomalies or problems. As discussed in Chapter 4, we can query about violations

of design guidelines or rules, so these can also be formulated to be connectors which appear in a view, for example, to show all the methods of a class which violate the *accessor methods* convention.

Using the general querying facility. We used the logic programming predicates also outside the context of connector-component perspectives, to query the static or dynamic information. Examples are queries for the presence of a design pattern, such as the Facade Pattern, detecting singletons or asking about the public interface of a class.

Code browsing. In performing the case studies we occasionally resorted to browsing the code to get more insight into the function a particular method plays. So recovered views do not replace other ways of obtaining information about the software, but rather complement them. Views, however, allow us to quickly find out where in the code we might be interested in looking.

Partial information. The second case study in particular showed us that partial information is sufficient to effect changes in code – we do not need to understand everything in order to carry out a maintenance task. In looking for a facade it was possible to ignore many of the classes of the Refactoring Browser, either by clustering all the irrelevant classes into a component, or by not instrumenting them.

Using static relations to recover views. We performed several experiments using Gaudi with static information only. In these experiments we recovered views in which connectors were specified using the static invocation relationship between classes, using the invocation(Sender, Method, ReceivedMethod, Candidates) predicate of the representation level. As the number of classes which are candidate receivers can be quite large, the resulting views were very crowded and difficult to interpret. We also narrowed down the candidates for an invocation through dynamic typing: this did not simplify the views very much. This shows that even for typed languages the static invocation relationship gives too many candidates to provide behavioral information which can be usefully interpreted.

A view recovered from dynamic information from a particular execution scenario is much easier to interpret, because we can put the relations displayed in the view in the context of that particular scenario – of the specific functionality exercised.

The contribution of static information. At the beginning of our experiments we expected to get more mileage from combining static and dynamic information. In fact, in the case studies presented here we make use of only the basic static predicates Class, Superclass and Method. In practice, static information was used in (i) formulating predicates for the clustering of elements into components, (ii) formulating base level predicates which are purely static or combine static and dynamic information, (iii) formulating auxiliary level predicates for expressing design patterns or coding conventions.

5.3.2 Towards a Methodology

The case studies presented demonstrate the process of design recovery, but they do this in a stylized way, as is required to convey them on paper. That is, in practice there is

more backtracking: a recovered view may not answer the question as expected and the developer would then define a new perspective to get a better view. This occurs in mainly two situations:

- The view we obtain is too crowded to be meaningfully interpreted. In this case we define a new perspective which clusters some elements, thus presenting some of the information at coarser granularity.
- The view we obtain refutes our hypothesis. We expected a certain structure, but the view showed us that we must revise our hypothesis and formulate a new perspective.

Though we do not formulate a specific methodology, a certain pattern of usage arises from our experiments with Gaudi: for a first view, almost any kind of logical component clustering will do, as a way to get an overview and know where to focus. Second, we try to eliminate as much 'noise' as possible by defining a new component clustering, or by obtaining dynamic information for only a subset of the application's classes. Thirdly, we ask more pointed questions and define specific connector relations to get the answer to these. This phase of the investigation is the longest. From the case studies we see that method invocation and creation relations are central in understanding behavioral aspects of a software system. The method invocation relations can be qualified to represent the invocation of certain kinds of methods (based on method category name, for example). Finally, we often move from high-level overviews towards instance-level views in a quest to understand how a particular functionality is implemented. Table 5.1 summarizes the main kinds of perspectives used during an investigation.

Perspective		*Description*
Connector	*Component*	
Communication		
invocation	categories	Uses the existing software organization to create components and show a high-level view of communication.
invocation	components	Uses a user-defined component breakdown to show high-level communication.
Creation		
creation	components	Uses a user-defined component breakdown to show a high-level creation view.
Multiplicity of Communication		
invocation, creation	classes	Without showing instances gives an idea of the multiplicity of relations between classes.
Inside a Call Stack		
invocation	classes	Displays sequence of method invocations within a call stack.

Table 5.1: Main kinds of perspectives

Using several program traces. As seen from the second case study, it is often useful to compare the information obtained from the execution of two different scenarios on an application. We might start with a relatively simple scenario and once we gain more understanding of the application we are better prepared to analyze a more complicated

scenario. Furthermore, analyzing the first scenario suggests to us what we can safely ignore so that a second scenario can be run with more selective instrumentation. Instrumenting more selectively results in more focused views and in faster response time for the queries.

5.3.3 Efficiency

How efficient is the process of exploring a software application with Gaudi? In this section we discuss this issue and give some figures.

Extracting the Source Model. Whereas in exploring a software application we often ran several scenarios to collect dynamic information, static information about the software system was extracted just once. Extracting the static information from the software using the MOOSE tool [DLT00] is fast (less than a minute for our case studies).

Obtaining dynamic information was also quite quick. The instrumentation was done either through a browser that permits the instrumentation at the method level (select the classes, methods categories or methods to be instrumented), or by automatically instrumenting all classes in a specified Smalltalk category or application. The system must then be exercised with a scenario.

Querying and View Generation. The response times[1], varied from a fraction of a second for a view which shows the invocation relationship between instances (Perspective 5 in both case studies), to one and a half minute for a view which shows both creation and invocation relationships between classes (Perspective 3, Scenario 2, in the second case study). Case study 1 consisted of 59,277 method invocation facts and case study 2 consisted of 1,187 method invocations (1st scenario) and 2,492 method invocations (2nd scenario).

As a comparison, with the source model of case study 1, the time to detect an instance of the Composite pattern as codified in Rule 10 (Section 4.4.1) was just over 3 minutes.

The Iterative Process. The most time consuming aspect of using the tool is the actual process of analyzing the information displayed in a view and posing new queries. We quickly arrived at views which helped us to focus on a part of the software system. Getting further in the investigation sometimes required the formulation of new rules or the generation of a new scenario.

Discussion. Gaudi was intended as a tool with which we could easily explore a software system and experiment with the use of logic predicates for generating useful views from dynamic information. In this sense Gaudi had met our goal: it requires little preparation time (extraction of the source model), makes it easy to generate a few scenarios (easy to get dynamic information) and is easily extensible (adding logic predicates and queries is simple).

[1] On a SUN Ultra Enterprise 250 with 300 MHz and 1 Gigabyte main memory.

5.3.4 Generality of the Approach

Below we discuss issues which relate to the generality of the approach: can it be used on any object-oriented software system?

Language independence

We have applied this approach so far only on Smalltalk applications. We expect, however, that it is easily adaptable to other class-based object-oriented languages like C++ and JAVA. This is supported by the fact that:

- the model of static and dynamic information as presented in Section 3.2 is to a large degree language independent. Some exceptions are the interpretation of certain static facts. For example the interpretation of SourceAnchor in the relation class(ClassName,SourceAnchor) is language-specific. In Smalltalk this will correspond to a class category, in C++ to a file name and in JAVA to a package. Furthermore, whereas in Smalltalk no type information is available for attributes, or for identification of receiver candidates in an invocation, this kind of information can be obtained from statically typed languages like C++ and JAVA.

  Instrumentation techniques are language dependent. In Smalltalk the reflective capabilities make it relatively easy to obtain run-time information [Duc99]. Several instrumentation techniques exist for JAVA [SKM01] and C++ [LN95b].

- the Prolog rules used to query the information base and to generate views are also to a large degree independent of the implementation language of the application. Some of the rules for the derived relations hold for all object-oriented languages, for example overrides(Class,Subclass,Method), whereas others are more language specific. One example of a language specific relation is the sendsCreate(Class1,Class2) rule. Detecting the presence of a creation relationship is like recognizing a language-specific cliché [HYR96] in the base of dynamic information.

Scalability

Scalability can be addressed in three ways:

Controlling granularity. Specifying a component clustering rule which groups together many classes results in a manageable view even for large applications. As in the Reflexion Model approach [MN97] we declaratively map classes or other software entities into components to generate an initial model with coarse granularity. In our case studies we used the existing software containment relations such as class categories and inheritance hierarchies to group elements. In the same way such relationships can be exploited for larger software systems, grouping elements by applications or files and by exploiting naming conventions.

Focus. In order to focus on some of the elements of the source model to the exclusion of others, we cluster those elements we want to ignore into a large component. This was done in the Refactoring Browser case study. The addition of a domain predicate which gives a developer more control over the elements which appear in a view is considered in future work.

Selective instrumentation. Once we know what part of the system we want to focus on we collect dynamic information only from a small subset of the classes or methods. This was demonstrated in the second case study.

Scalability is also addressed by the iterative and open nature of the process itself. That is, we do not expect to answer all the questions we have with one view, and can use a variety of perspectives, each one to generate a view which focuses on one aspect or one part of the system. For very large scenarios the response time of Gaudi for some of the queries may be too long.

5.3.5 Related Work

The use of logic programming languages for querying static information was presented in Section 4.4. Here we discuss several approaches that are most related to ours.

Knowledge-based approaches, such as the LaSSIE system [DBSB91], allow an engineer to make queries at a higher level of abstraction, such as "What global variables are accessed by a function that flashes a display lamp at an attendant's console?". These kinds of approaches require that application specific information about the problem domain, the code structure and the features be used to populate the knowledge base, and, more importantly, that this knowledge base be kept up to date. Our approach can be seen as a lower-level knowledge-base system which incorporates knowledge general to the structure of all object-oriented systems. While it does not permit queries which require application specific information, it can populate the knowledge base automatically, without having to elicit information from experts. As seen in the approaches of [Wuy01, Men00], software engineering knowledge in the form of coding conventions and design patterns, as well as application specific information can also be added to such a framework to create a deductive knowledge base.

The work of Consens et al. on Hy+ [CM93] has many similarities with our work. Hy+ is a query and visualization system based on hygraphs – an extension of directed graphs in which *blobs* are incorporated. A blob represents a set, and is used to cluster together elements; blobs can also be nested. A *visual* declarative query language, called GraphLog, is used to define blobs (clustering) and edges to display. Queries are evaluated by translating the visual pattern expressed in GraphLog into programs for the back-end processor; Prolog, as well as other logical query languages have been used as back-ends. The formulation of queries in GraphLog seems quite complicated. Hy+ has been used for reasoning about dependency relationships between modules [CMR92] and for debugging concurrent programs through post-mortem animation of event traces.

Our use of perspectives is similar in spirit to Reflexion Models [MN97]. In the Reflexion Model approach an engineer posits a model of the system, similar to our component-connector model, and defines a mapping of elements of the code to the components. Using static binary relations between the code elements a reflexion model is then created: it shows the divergences (relations where none were expected), convergences (relations where they were expected) and absences (no relations where some were expected) of the source code from the engineer's model. The model can be successively refined by changing the mapping of the code to the model, refining the model itself or including more or other binary relations. Our approach differs from the Reflexion Model approach in several ways. Our model (perspective) is used to express what *kind* of information we want to see, *e.g.* creation relations between components, so it is not used in checking the conformance of a posited model to the source model. Our perspectives are based on a meta-model of object-oriented programs and their execution – whereas Reflexion Models are purely syntactic. Our approach supports multiple views, each with a different semantics. In Reflexion Models, this could be handled by a typed source model [Mur96] which treats different kinds of binary relations as typed edges, or by using a different source model when a developer wants to view a different relation. The Reflexion Model approach operates on any binary relation, so it can be used on message send events as well, though no such case is reported [Mur96]. The Reflexion Model approach offers a particularly simple and appealing methodology in which a source to high-level model mapping is repeatedly refined. So conceptually there is one path, rather than several paths to navigate in the methodology.

Other tools and approaches for understanding dynamic behavior have been presented in Section 2.2.3. Tools which summarize information [PHKV93] offer a fixed set of views. Tools which display sequence diagrams [PLVW98][JR97][KM96][SKM01] rely on visualization techniques and navigational aids to help a user find the desired information. Walker et al. [WMFB+98] display the interaction between objects through user-defined high-level model using program animation techniques. Their tool focuses on displaying the number of objects involved as the execution progresses. Our tool is based on a simple graph layout tool and so does not have the sophisticated visualization capabilities offered by these tools.

Our approach allows a user to declaratively define the kind of information of interest, and distills the dynamic information into a more succinct form which can be displayed compactly. Understanding message invocation sequences between instances does, however, seems important for getting a grasp on how an application works. Concept view recovery allows a user to display this kind of view as well, for a chosen call stack. In the next chapter we describe collaboration view recovery, in which we show how to generalize from specific object interactions to class collaborations.

Although some work on program analysis tools [CCdCL98] and query-based debuggers [LHS97] is related to ours, these kinds of tools have a different goal than reverse engineering tools and are not well adapted to creating design models and revealing overall structures in the software.

5.4 Revisiting the Requirements

Before we close this chapter on concept view recovery, we evaluate our work in the light of the requirements we have outlined in Section 2.3. Note here that we consider the collaboration view recovery application presented in the next chapter as an integral part of our approach. This means that we will later evaluate our approach as a whole against this set of requirements.

1. **Lightweight information model.** As discussed at the conclusions of Chapter 3, the meta-models on which our approach operates are relatively simple to extract. Our experiences with concept view recovery suggest that in many cases we do not make use of the full static meta-model, and static information about classes, methods and inheritance relationships suffice for a rich repertoire of queries and perspectives.

2. **Simple view specification.** Obtaining an initial view of the software is simple, using predefined perspectives, as, for example, the calling relation between class categories. Though the notation of a logic programming language may not appear simple to all, the declarative nature of the queries is intuitively simple. We also discussed the possibility of implementing a domain specific language on top of the logic programming language to facilitate queries.

3. **Succinct views.** By succinct we mean that the views recovered are manageable in size. The extracted views shown in the case studies are manageable in size: they result in graphs which can still be understood and interpreted by the engineer to extract important information. This is not due to the logic programming approach, but rather to the facility it provides in managing granularity and focus.

4. **Developer guides the process.** The two case studies demonstrated the process of design recovery in which a developer, guided by a question, refines perspectives and queries to obtain information of interest. Both case studies show that through an iterative process of querying an engineer can arrive at an understanding required to answer the question posited at the beginning of the investigation. A view works as a catalyst for generating questions about the studied system and helps the engineer to focus the investigation of the code. It confronts the engineer with a new model to compare to his or her initial mental model and assumptions about the system.

5. **Extensible view specification.** Each case study had a clear objective at the beginning. However, the iterative nature of the process means that the nature of the question changes throughout the investigation. The two case studies thus demonstrate that perspectives can be tailored to a range of questions, enabling a developer to determine what aspects of the software he or she is interested in.

6. **Behavioral views.** Here we look again at the kinds of questions we want to be able to answer using behavioral views:

- how do the main domain elements relate to each other?

  By using an existing containment relationship, such as the Smalltalk class categories or the inheritance hierarchy, we can easily obtain an initial view of the relationships of these components to each other. The component breakdown can then be refined to better reflect our understanding of the domain elements.

- what parts of the software implement a specific feature?

  In executing a scenario to collect dynamic information we tie our information base to the feature or functionality we are interested in. Elements of the software which have no role in implementing this functionality will not appear in the dynamic information. This provides some initial focus. However, we will in general need more focus to discover which parts of the software are responsible for implementing a specific feature. The iterative process of extracting views helps us to arrive at this information.

- to which messages does an instance of a certain class respond?

  By querying the dynamic information, we can ask about the methods that have been invoked on a class. We can also ask about the interface that a component (a grouping of classes) presents in the system.

- how many instances of a class are present at runtime?

  This information can be obtained through a predefined query.

- which objects are responsible for creating instances of a certain class?

  The case studies showed that 'creation' views are useful in understanding responsibilities of classes. Creation views can also be obtained which show the multiplicity of the creation relationships.

- which class instances participate in the interaction resulting from the invocation of a particular method?

  In focusing our investigation we often want to see what is happening within a method execution: how many objects there are and how they interact. Such views can be obtained by defining a perspective which shows the method invocations inside a call stack of a method execution. In the next chapter we show how collaboration view recovery supports generalizing from one particular interaction to a class collaboration abstraction.

- what are variations on the way a method is executed?

  In order to obtain this information, we would have to look at several occurrences of the execution of the same method – this is possible, but cumbersome. In the next chapter we show how collaboration view recovery is used to extract and understand variations on the execution of a method.

7. High-level views. The clustering of entities enabled us to obtain several views of different granularities going from the interaction of sets of classes grouped together to the interaction between instances.

8. Low-level views. In trying to understand how a functionality is implemented we also created perspectives which extract the interactions of instances within a method invocation stack.

5.5 Conclusions

In this chapter we presented two case studies conducted to evaluate concept view recovery. In particular, we showed how a developer guides the process of design recovery and how perspectives are used to specify several different views at varying levels of granularity and so to answer questions at different abstraction levels.

We discussed some lessons learned through the case studies and identified several commonly used perspectives. We also argued that the approach is general enough to be applied to other class-based object-oriented languages and that it provides mechanisms for handling scalability problems.

In Section 5.4 we evaluated how the concept view recovery meets our requirements. We saw that it supports the recovery of succinct views which can answer a range of behavioral questions. We also noted that some of the low-level behavioral questions we pose, in particular about the interaction of objects within a method execution stack, are not very elegantly answered in concept view recovery. In the next chapter we present collaboration view recovery, where we propose an approach for understanding and recovering class collaborations. Collaborations can capture low-level information about object interactions in a succinct way, giving us insight into the roles that different classes play in an interaction.

6

Collaboration View Recovery

In this chapter we present collaboration view recovery. Like the concept view recovery, this application is based on the use of perspectives in an iterative process. But whereas component-connector perspectives use groupings and relations over all elements and relations in the source model, collaboration view recovery supports only perspectives for creating collaboration abstractions and querying the roles that classes play in collaborations.

The case studies presented in the previous chapter showed that in focusing our investigation of a software system we come to a point where we want to understand what is happening at the level of objects interacting to carry out a certain functionality. Looking at an instance level view shows us one instance of the interaction of objects, but does not let us generalize to a design abstraction. In particular, we want to know which classes play what role: how classes collaborate. Design artifacts which are good at answering such questions come from collaboration-based or role-based design methodologies. Collaboration-based or role-based designs decompose an application into tasks performed by a subset of the applications classes. They provide a larger unit of understanding and reuse than classes and can be an important aid in the maintenance and evolution of the software.

Collaboration perspectives enable an engineer to recover a collaboration view, where the software system is seen as a collection of class collaborations. These abstractions are created by applying pattern matching to the execution trace – the role of the engineer here is to specify what he or she considers to be similarity in execution sequences by modulating the pattern matching criteria. Collaboration view recovery also supports querying about which classes collaborate with each other, which methods are invoked by which classes, and which classes play similar roles.

The chapter is structured as follows: in Section 6.1 we present a short overview of collaboration-based design as it is used in forward engineering. In Section 6.2 we present the challenges of reverse engineering collaboration-based designs and introduce the approach we propose. In Section 6.3 we present our approach in more detail and in Section 6.4 we describe the Collaboration Browser, the tool we have developed to support the recovery of roles and collaborations. In Section 6.5 we present the case studies which validate the approach. Section 6.6 presents a discussion of the approach. In Section 6.7

we evaluate collaboration view recovery in terms of our requirements and conclude the chapter with Section 6.8.

6.1 Collaboration-based Design

The need for modeling collaborations in object-oriented applications is now well recognized. Design methodologies which recognize the distinct responsibilities of classes cooperating to achieve different tasks were first promoted by [WBW89, BC89], and are now integrated into most object-oriented modeling methods and notations [BRJ99]. Design patterns, for example, capture collaborations to solve a specific design problem [GHJV95].

Collaboration-based or role-based design decomposes an object-oriented application into a set of collaborations between classes playing certain roles. Each collaboration encapsulates an aspect of the application and describes how participants interact to achieve a specific task. In this section we first present a small example to illustrate the concepts of collaboration- based design. We then briefly survey approaches to specifying and implementing collaboration-based designs in forward engineering.

6.1.1 An Example

Consider a class model which describes a bureaucracy [Rie98], as shown in Figure 6.1. This is a hierarchy of Director, Managers and Clerks which operates as described by the Bureaucracy pattern [Rie98].

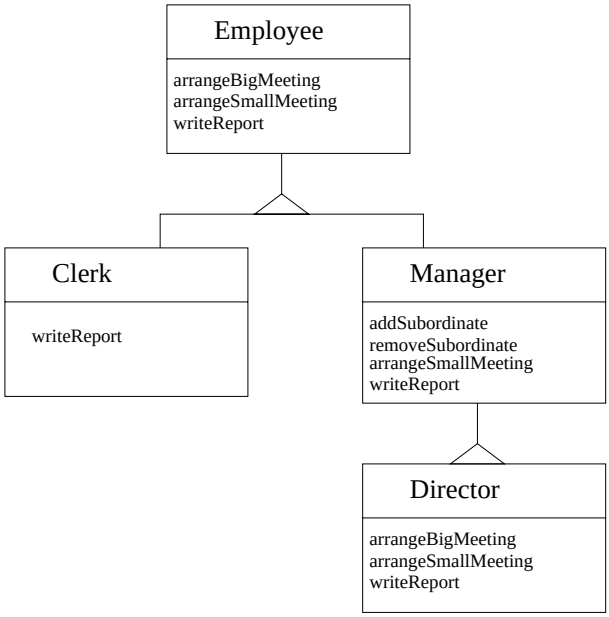

Figure 6.1: Class diagram for Bureaucracy

In effect, four of the GOF design patterns [GHJV95] govern the interaction of the objects. A Manager or Director who receives a request delegates work to its subordinates, as in the Composite pattern: the Clerk plays the role of Component and the Manager the role of Composite. A Manager or Clerk receiving a request it cannot handle forwards the request up the hierarchy, as in Chain of Responsibility: the Managers and Clerks play the Predecessor role and the Director the Successor role. Clerks or Managers who want to interact with each other first address their superior to coordinate them, as in the Mediator pattern: at the same hierarchy level the objects are Colleagues, whereas the superior acts as Mediator. Finally, when a subordinate changes state, such as completing some work, or being absent, it reports this change of state to its superior: thus the superior acts as Observer of its subordinate Subjects, as in the Observer pattern. Figure 6.2 summarizes this information in a class-collaboration matrix [VN96]. It illustrates that an instance of a class participates in several collaborations, playing a distinct role in each.

	Clerk	**Manager**	**Director**
Chain of Responsibility	Predecessor	Predecessor	Successor
Observer	Subject	Observer	
Composite	Component	Composite	
Mediator	Colleague	Mediator	

Figure 6.2: Class-collaboration matrix for Bureaucracy. Each row represents a collaboration and each cell describes the role the class plays in the collaboration.

Here we have described the collaborations and roles in terms of the design patterns they instantiate. Roles describe the responsibilities of objects in a collaboration, but how a role is actually modeled or specified is often left open [RG98]. Some design techniques model roles using interfaces [Ree96], or as part of a behavioral contract between participants [HHG90]. Collaborations are usually modeled using UML interaction diagrams. These show how participants interact to achieve a task: they are usually succinct and show only one instance of each kind of participant. In the next section we survey briefly a few of the notations and techniques for specifying and implementing collaborations and roles.

6.1.2 Collaborations and Roles in Forward Engineering

Apart from the standard UML collaboration diagram notation [BRJ99], several researchers have proposed ways to specify such behavioral compositions using contracts

[HHG90] and Reuse Contracts [Luc97] as an aid in reuse and evolution. There are also several proposals on how to move from a collaboration-based design to the implementation – that is how to map the roles in a collaboration onto the application's classes. Some of these propose mappings from roles to classes at the design level [Ree96][RG98], or strategies for implementing role-based designs [SB98][VN96][Ken99], while others propose features or extensions which retain the concept of role and collaboration at the code level [SB98][ML98] [DR97].

This active research area attests to the general recognition that role-based design artifacts can greatly aid the understanding and maintenance of object-oriented applications. In this section we briefly survey how role-based and collaboration-based designs have been specified in these different approaches.

UML. In the UML [BRJ99], interaction diagrams model how groups of objects collaborate and are typically used to describe a single use case, or a specific scenario in a use case. These formalisms model the flow of control between class (role) methods in carrying out a specific task.

OORAM. OORAM [Ree96] is a design method in which separate aspects of an application are modeled using *role models* and these are combined to arrive at the class model for the application. A role model [And97] is a set of roles, where each role is defined by a set of role paths. Each role path gives a name of a receiver and a list of messages which that role may send to the receiver. This is the output interface of a role – the messages it can send. The input interface is not explicitly specified: it is the union of the output interfaces directed towards that role by other roles.

Contracts. Contracts [HHG90] are design formalisms used to describe constraints on message exchange between participants. Collaboration contracts, called Reuse Contracts in [Luc97], have two aspects: the static structure refers to the participants and the acquaintance relations between them, the interaction structure describes the messages sent between the participants. The interaction structure states the invocation relationship between the methods of the participants, but it does not specify any ordering on these invocations.

Role modeling. Riehle presents an approach to framework design based on role-modeling [RG98, Rie00]. The approach shows how to start with role-models and compose them to arrive at a class-based design. This work remains at the design level and does not prescribe or suggest any type specification mechanism for describing a role and how it is embedded in a specification of a collaboration.

Roles and collaborations at the implementation level. Interfaces, as present in JAVA, can be used to specify roles at the implementation level. Extensions have also been proposed to enrich Smalltalk with roles [GSR96]. Roles have been implemented using C++ templates [VN96], Aspect-Oriented Programming [Ken99] and Role Object pattern [BRSW00]. These implementation techniques capture the notion of role, but not the interaction of roles in a collaboration. Below we discuss a few contributions to supporting collaborations at the implementation level:

Aspectual Components [LOML01], the more recent version of Plug-and-Play Components [ML98], describe template collaborations between participants. These are then instantiated, with classes playing the participants and methods of those classes matching the roles in the collaborations. Aspectual components thus add data members and function members to existing classes, and modify function members of existing classes to augment their behavior, in the context of a collaboration. In this formalism, a collaboration is a collection of interfaces; the semantics of the collaboration protocol is expressed using the base programming language (JAVA in this case).

Executable Connectors [DR97] are very similar to the collaboration notion of Aspectual Components, but they are used to add behavior to a collection of objects playing roles, rather to a set of classes. The semantics of the collaboration is specified in terms of rules which express how participant interfaces collaborate.

Smaragdakis and Batory [SB98] introduce Mixin Layers, a form of mixins that can contain several smaller mixins, as a way to represent collaborations at the implementation level and to build an application from the composition of these collaborations.

6.2 Reverse Engineering Collaborations and Roles

Since standard object-oriented languages do not provide language constructs to capture collaborations, design information about collaborations is lost in the implementation. In a collaboration, participants interact according to a protocol describing the set of allowed behaviors. At the implementation level the description of this behavior protocol is distributed throughout the code. At the implementation level a collaboration thus consists of two basic elements: *participants* and *roles*. The *role* of each participant is the part of the participant which enforces the interaction protocol. A collaboration can thus be seen as a collection of roles.

6.2.1 Describing Collaborations and Roles

In order to extract collaborations from existing code we must first make clear what kind of description we want to extract: who the *participants* are and how each participant's *role* is to be described.

Such a description should be *easy to understand* and be *tied to the code*. First, since we want a description which helps in program understanding, we are not interested in a formal notation, nor in a formal description of the interaction protocol. Visualizing a collaboration as an interaction of objects also helps in understanding collaborative behavior. Second, the description should be tied to the static structure of the code, that is, to classes and methods, rather than remain at the object level.

We therefore propose a simple description of a collaboration as a collection of classes (the participants) and their roles, where the role of each class is given by set of methods of the class which control the behavior of the class in the collaboration. We do not try to recover a description of the protocol which governs the interaction of the methods.

An example of such a collaboration description is given in Table 6.1 below. The participants in this collaboration are the classes DrawingEditor, Ellipse, Rectangle, and DrawingVisitor and their roles are the set of methods listed in the respective columns. Note that the Ellipse and Rectangle classes have equivalent roles in this collaboration. The collaboration corresponds to a Visitor pattern [GHJV95], in which DrawingEditor plays the role of Client, DrawingVisitor plays the role of Visitor, and Ellipse and Rectangle classes play the role of Visited Elements.

DrawingEditor	Ellipse	Rectangle	DrawingVisitor
refresh	acceptVisitor: lineColor lineStyle leftTop rightBottom	acceptVisitor: lineColor lineStyle leftTop rightBottom	visitEllipse: visitRectangle:

Table 6.1: Collaboration description for a collaboration DrawingEditor refresh

To recover collaborations and roles from existing code we need to discover the important tasks in which instances collaborate, and break up the behavior of a class into roles, as shown schematically in Figure 6.2, where each role describes the behavior of instances of this class in a specific context. Below we discuss the challenges that must be tackled.

6.2.2 Challenges to the Recovery of Collaborations

There are several challenges to the recovery of collaborations from code: first, we must recover interactions of classes from the code. Which classes interact with each other? Which methods are invoked in an interaction? Second, since object-oriented code is full of interactions the challenge is to find the *significant* interactions – the design collaborations are those which capture important behavioral concepts.

Recovering interactions. As argued in Section 2.2.3, static information does not provide us with the information necessary for identifying collaborations. To identify collaborations we need control flow information; this is difficult to obtain purely from static analysis, due to polymorphism, inheritance and dynamic binding.

The notion of role is also hard to recover from static information. Even when languages explicitly support an interface construct, such as present in JAVA, there is no semantics in terms of collaboration attached to this construct. That is, to recover collaborations and roles we need to understand the rules governing the run-time behavior of the instances. The interface construct is, however, a good starting point for discovering important collaborations.

The inheritance relationship is also not very reliable in deriving information about roles because a subclass does not necessarily play the same roles that its superclass plays. This depends on how inheritance has been used, and in reverse engineering we often examine applications whose design may not follow accepted design guidelines.

Recording dynamic information about message exchanges between instances as the program executes provides us with control flow information required for deriving collaborations and with information about the context in which methods are invoked. Program tracing, however, results in a great volume of information about the interactions of objects. Much of the information about interactions is duplicated many times over in an execution trace.

We use dynamic information to recover interactions. To reduce the volume of information, while still maintaining the information *content*, we use pattern matching to group similar sequences of method invocations in a pattern. This allows us to abstract from a particular execution sequence to a pattern of execution which occurs repeatedly in the trace.

Finding the important collaborations. Once we have obtained information about interactions – which instances interact with each other and the methods invoked in these interactions – the challenge remains to identify *important* interactions.

Every method invocation in the trace is mapped to a collaboration pattern. How do we find the important collaborations? The important collaborations are those which are important to understand in order to carry out the maintenance task. We use the iterative query-based approach introduced in Chapter 3. This allows the developer to create the relevant abstractions and views, and to steer the process of design recovery.

6.2.3 Overview of Our Approach

Below we summarize our approach for recovering collaborations and roles. The process is illustrated in Figure 6.3.

Based on a source model of static and dynamic information. Dynamic information is used to obtain exact control flow information and so to extract interactions. Static information is used to group similar interactions into collaboration patterns.

Uses pattern matching. We use pattern matching to find similar execution sequences in the execution trace. This condenses the amount of information from information about interactions of instances to information about patterns of interactions.

Supports iterative recovery through perspectives. As with concept view recovery, we let the developer specify what kind of information he or see is interested in by defining a perspective. This is done through two operations: specifying the desired pattern matching criteria and querying the dynamic information in terms of classes and interactions of interest.

6.3 Extracting Collaboration Views

In this section we first introduce some of the terminology and concepts, then explain how collaborations are extracted using pattern matching and querying.

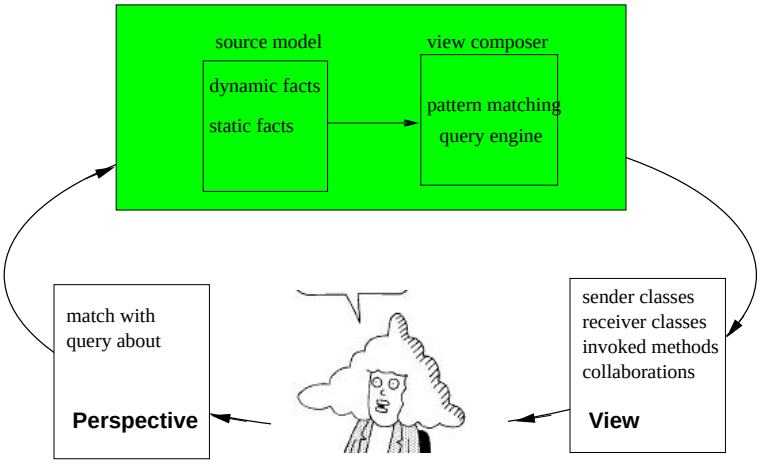

Figure 6.3: Pattern matching and querying in the iterative query cycle

6.3.1 Terminology and Concepts

We reserve the term *collaboration* and *role* to talk about the high-level design concepts. Our starting point for the recovery of collaborations is the execution trace. Each method invocation recorded in the execution trace gives rise to a sequence of method invocations, an interaction which we call a *collaboration instance*. We then identify *collaboration patterns* by comparing similar collaboration instances.

Collaboration instance. A collaboration instance is the sequence of message sends between objects, ordered as a call tree, which results from a method invocation (all message sends up to the return).

Collaboration pattern. A collaboration pattern is an equivalence class of several collaboration instances.

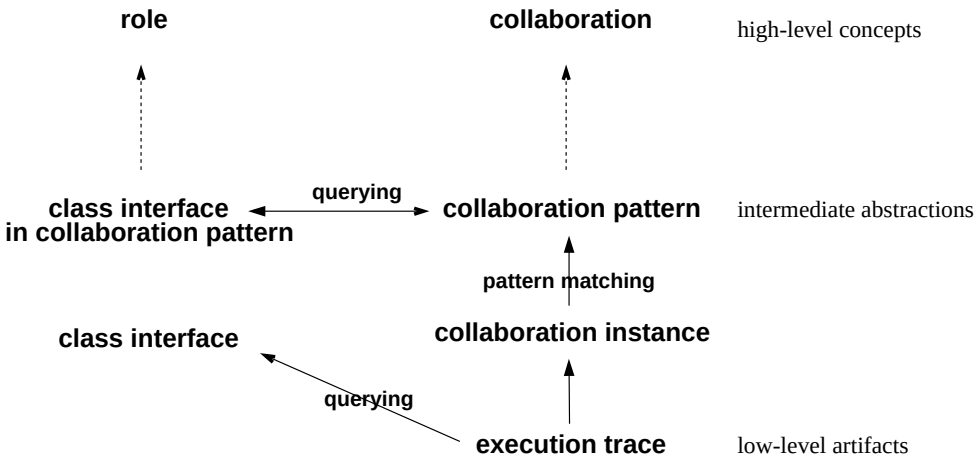

Figure 6.4: From an execution trace to collaborations and roles

A collaboration pattern is an approximation to the higher-level design concept of collaboration. The corresponding approximation to the high-level notion of role is the set of (public) methods that a class presents in the context of a collaboration pattern. We can obtain this information by querying about a collaboration pattern.

Figure 6.4 above illustrates how pattern matching and querying supports the recovery of collaborations. Pattern matching allows us to create the abstractions of *collaboration patterns*. These are indications for collaborations. The execution trace can be queried to obtain the interface of a class in the whole execution trace or in the context of a collaboration pattern. The interface of a class in a collaboration pattern is an indication for the role of the class in the collaboration.

Perspectives and views. A perspective is a specification of the aspects of the source model that we are interested in. In the context of collaboration view recovery, a perspective is a specification of (i) what we consider similarity in execution sequences – this determines how collaboration instances are grouped into collaboration patterns and (ii)) what we want to know about collaboration patterns. These two specifications correspond to the two operations, 'match with' and 'query about', illustrated in Figure 6.3. Pattern matching and querying are the two key operations in recovering collaborations. Pattern matching condenses the information about all method invocations present in an execution trace into a much smaller set of collaboration patterns. Setting the pattern matching criteria dictates what the developer considers to be important about a collaboration, be it the classes involved, the methods invoked or the nesting structure of the method invocations. The effect of choosing different pattern matching criteria will be discussed in Section 6.3.2. Querying about collaborations begins once we have a base of collaboration patterns. It helps us to find the important collaboration patterns by selecting them in terms of the senders, receivers and invoked methods involved.

In collaboration view recovery our 'window' on the source model is restricted to sender classes, receiver classes, invoked methods and collaboration patterns. This then corresponds to the view. Specifying similarity between execution sequences determines what collaboration patterns we will see and querying lets us put sender classes, receiver classes and invoked methods in relation to each other in the context of a specific collaboration pattern.

6.3.2 Pattern Matching

The settings for the pattern matching criteria specify which collaboration instances will be matched as instances of the same collaboration pattern. They reflect what the engineer considers important about a collaboration.

When the pattern matching is performed these settings are used to transform each collaboration instance to its pattern representation. Identical patterns are then grouped together as one collaboration pattern. We retain, however, some of the original collaboration instance information about senders, receivers and invoked methods in order to allow querying about these.

The execution trace can be seen as an ordered, labeled tree where each node corre-

sponds to a method invocation event. Each subtree in this large tree is a collaboration instance. So grouping collaboration instances into one collaboration pattern means that we match subtrees of the execution trace. But we are not interested in exact matches – this is much too restrictive to allow us to generalize from a particular execution sequence to a design abstraction. To support *approximate* matching of collaboration instances we propose a set of pattern matching criteria which are selected and set by the engineer. Below we present a small example to illustrate the ideas behind the pattern matching criteria we propose. These are later discussed in greater detail in Section 6.6.3 and Section 6.4.2.

An example. When are two collaboration instances similar enough to be considered the same? We illustrate the ideas behind the pattern matching criteria using an example. Consider the design of a simple program which simulates the forwarding of packets in a local area network (Figure 6.5). Each node is either a workstation or a printer. Printers can forward packets or print packets. Workstations can forward packets or originate packets. Packets can be asked, using isAddressedTo:, if they are addressed to a particular node.

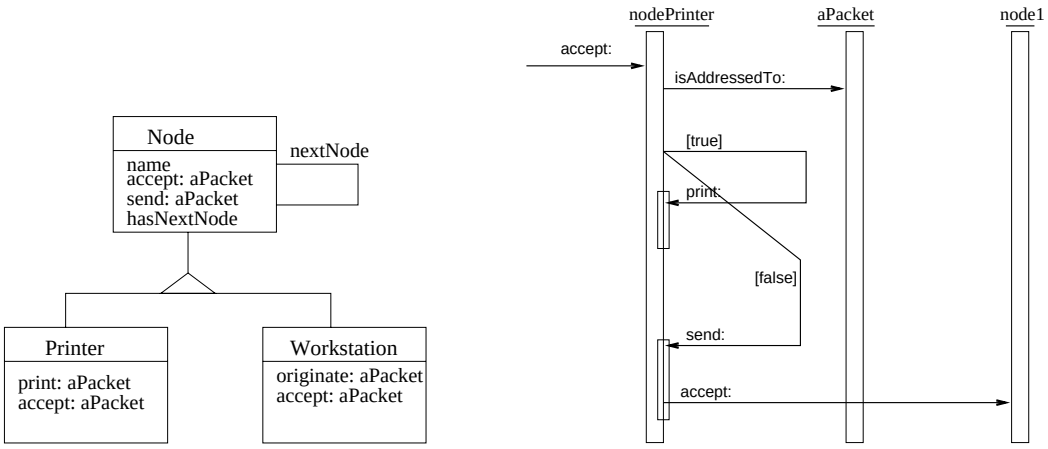

Figure 6.5: LAN design

Consider now what happens at runtime as such a program executes. A packet may be relayed several times before it reaches its destination. But what is happening is always the same: the node asks the packet if it is addressed to that node, if it is then the execution ends, otherwise the packet is forwarded to the next node in the LAN. One example of an execution is given in Figure 6.6.

If we take into account the identity of the nodes, then we will only recognize collaboration instances as similar if the very same nodes participate in them. If we consider instead the class of the nodes, then we match events with the same class rather than the same object. If we consider rather the superclass of the class, in this case Node, then we will match as similar events where both printers and workstations behave the same. So we can control matching by deciding what *information about an event* we are interested in.

The accept: method in this example is implemented recursively. So if we look at the call tree resulting from an invocation of accept:, we will obtain a shallower or deeper

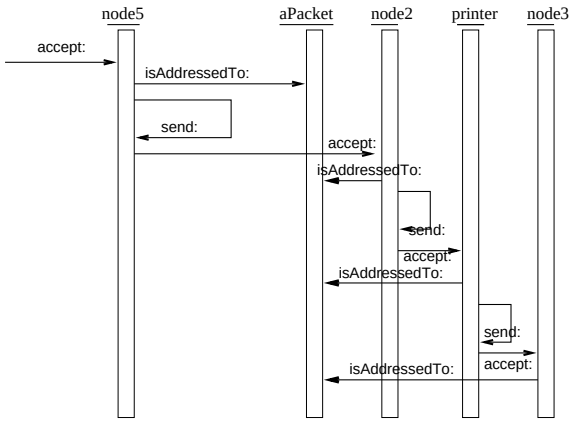

Figure 6.6: LAN execution

tree depending on the number of times the packet has been relayed in the network. If we treat collaboration instances as call trees of events then collaboration instances will match only if they have exactly the same number of recursive calls. If we flatten the tree, however, and match collaboration instances with the same *set* of events, then we ignore the extra information about recursion or iteration of the same event. The *structure of the collaboration instance* is thus also important as a matching criterion.

We can also choose to *exclude some events* we are not interested in. Self-sends, for example, often add no special semantics to the collaboration instance. In many cases we also want to obtain a high-level view of what is happening in the collaboration instance, so we prefer to look at only the first few levels of invocations. In our LAN example, for example, if we treat the structure of the collaboration instance as a tree, but look only at the two top levels of invocation, then each accept: invocation will be seen as consisting of a tree of three events where the topmost event is the accept: invocation and its children are the two events isAddressedTo: and send:.

This simple example illustrates the difficulties of abstracting from collaboration instances to collaborations. In most cases we try to abstract as much as possible – the extreme is to match collaboration instances based on the information for only the top level method invocation event. This would hide, however, interesting and important variations of a method invocation. Our experiences and experiments with different pattern matching schemes are described in Section 6.6.3.

Pattern matching criteria. To summarize, we propose the following matching criteria which can be modulated along three independent axes:

Information about an event. An event in the execution trace is a method invocation, as given in the model described in Section 3.2.3. An event contains basically three items of information: the sender, the receiver and the invoked method. Each of these three items can be taken into account or ignored in the matching scheme. Table 6.2 shows what information about each of these three items can be used in the matching scheme.

sender	none	object identity	sender class	
receiver	none	object identity	receiver class	name of class defining method
invoked method	none	method name	method category name	

Table 6.2: Pattern matching options for the method invocation events

Events to exclude. The matching scheme allow us to ignore certain events, on the basis of three criteria:

Depth of invocation. Method invocations at a greater depth (with respect to the whole execution scenario) are ignored.

Relative depth of invocation. The relative depth of invocation is the depth of invocation with respect to the first invoked method in the collaboration pattern. Using this criterion we specify that method invocation at a greater depth in the collaboration pattern should be ignored.

Self sends. Method invocations in which an object invokes a method on itself are ignored.

Structure of the collaboration instance. A collaboration instance is a tree of events. However, similar collaboration instances may differ in their tree structure and still have the same 'meaning'. Therefore, in the matching scheme it is also possible to treat collaboration instances as sets of events, thus ignoring all ordering and nesting relations between method invocations. In this way collaboration instances are treated as identical if they have the same method invocation events in their set.

6.3.3 The Query Model

A developer focuses on the relevant collaboration patterns by querying the dynamic information. The query model supports multi-way queries about the two basic relations which are of interest to us in recovering collaborations: method invocations in the executed scenario, and method invocations in the context of a collaboration pattern.

send(Sender,Receiver,Method) : this relation holds when there is an instance of the class Sender which invokes Method on an instance of the class Receiver, in the context of the whole execution trace.

sendInCollab(Sender,Receiver,Method,Collab) : this relation holds when there is at least one collaboration instance in the collaboration pattern Collab in which an instance of Sender invokes Method on an instance of Receiver.

For each of these relations, multi-way queries are supported. That is, in querying about a relation we specify the value of one or more arguments in the relation – the response to

the query provides the values of the missing arguments. This will be illustrated in greater detail when we introduce the Collaboration Browser tool. Table 6.3 lists the queries about the **send** relation, while Table 6.4 lists the queries about the **sendInCollab** relation.

Note that the **send** relation corresponds to the predicate sendsToMethod in concept view recovery, and the **sendInCollab** relation corresponds roughly to the sendInstanceInStackMethod predicate in concept view recovery. We draw the parallel here to emphasize that we consider concept view recovery and collaboration view recovery to be part of the same approach, though we have implemented two separate tools for each kind of application.

Participant classes. The participants in a collaboration instance are all the receiver classes in the collaboration instance. We do not include the sender of the first invoked method of the collaboration instance as a participant, so that each sender class in the collaboration instance is also a receiver class.

	Sender classes	Receiver classes	Invoked Methods	Query
Q1	senders	receivers	?	Given a list of senders and receivers, returns the union of all the invoked methods for each sender-receiver pair
Q2	senders	?	?	Given a list of senders, returns the union of all receivers for all sender-receiver pairs, and the corresponding invoked methods
Q3	?	receivers	?	Given a list of receivers, returns a union of all senders for all sender-receiver pairs, and the corresponding invoked methods
Q4	senders	?	methods	Given a list of senders and a list of methods, returns all the receivers which receive any of these methods from one or more of the senders
Q5	?	receivers	methods	Given a list of receivers and a list of methods, returns all the senders which send any of these methods to one or more of the receivers
Q6	?	?	methods	Given a list of invoked methods returns senders and receivers which send or receive these respectively

Table 6.3: Queries about method invocations in a trace

6.4 Tool support: The Collaboration Browser

To support the recovery of collaborations, we have implemented a tool prototype, the Collaboration Browser. The Collaboration Browser presents the dynamic information to the user through four basic elements of information: sender classes, receiver classes, invoked methods and collaboration patterns. Each of these four elements is displayed on the screen in a separate panel as seen in Figure 6.7. Panels a, b and c list the sender classes, the receiver classes and the invoked methods respectively. Panels d and e both list collaboration patterns. The distinction between these two collaboration pattern lists is explained further below, as is the function of the button panel f.

	Sender classes	Receiver classes	Invoked Methods	Collaboration Pattern	Query
A. Query to obtain collaboration patterns in which ALL selected elements participate					
Q7	participant classes			?	Returns the collaboration patterns for which there is a collaboration instance in which all the participant classes participate.
Q8	participant classes		methods	?	Returns the collaboration patterns for which there is a collaboration instance in which all the participant classes receive the invoked methods.
Q9			methods	?	Returns the collaboration patterns in which all the methods are invoked.
B. Query to obtain collaboration patterns in which any of selected elements participate					
Q10	participant classes			?	Returns the collaboration patterns in which any of the participant classes participate.
Q11	participant classes		methods	?	Returns the collaboration patterns in which one of the methods is invoked on one of the participants.
Q12			methods	?	Returns the collaboration patterns in which one of the methods is invoked.
C. Given a collaboration pattern query about senders, receivers and invoked methods					
Q13	?	?	?	collaboration	Returns the senders, receivers and invoked methods which participate in at least one collaboration instance in the pattern.
Q14	?	?	methods	collaboration	Returns the senders and receivers which correspondingly send and receive ALL the invoked methods
Q15	?	receivers	?	collaboration	Returns the union of all the senders for the given receivers, and the union of all the methods they invoke on the receivers.
Q16	senders	?	?	collaboration	Returns the union of all the receivers for the given senders, and the union of all the methods they receive from the senders.
Q17	senders	receivers	?	collaboration	Returns the invoked methods for all the senders and receivers.

Table 6.4: Queries about method invocations in a collaboration pattern

In Section 6.4.1 we explain the functionality of the Collaboration Browser, giving some small examples. The screen shots which provide the examples are from an analysis of the HotDraw application, which will be presented in greater detail in Section 6.5.1. In Section 6.4.2 we briefly discuss implementation issues.

6.4.1 Functionality of the Collaboration Browser

The Collaboration Browser supports 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. The pattern matching criteria are currently set by hand.

In the presentation below we have numbered the queries (Q1 to Q17 from Table 6.3 and Table 6.4), editing (E1, E2, E3, E4) and displaying functions (D1, D2, D3) in order to refer to them later on in Section 6.5.1.

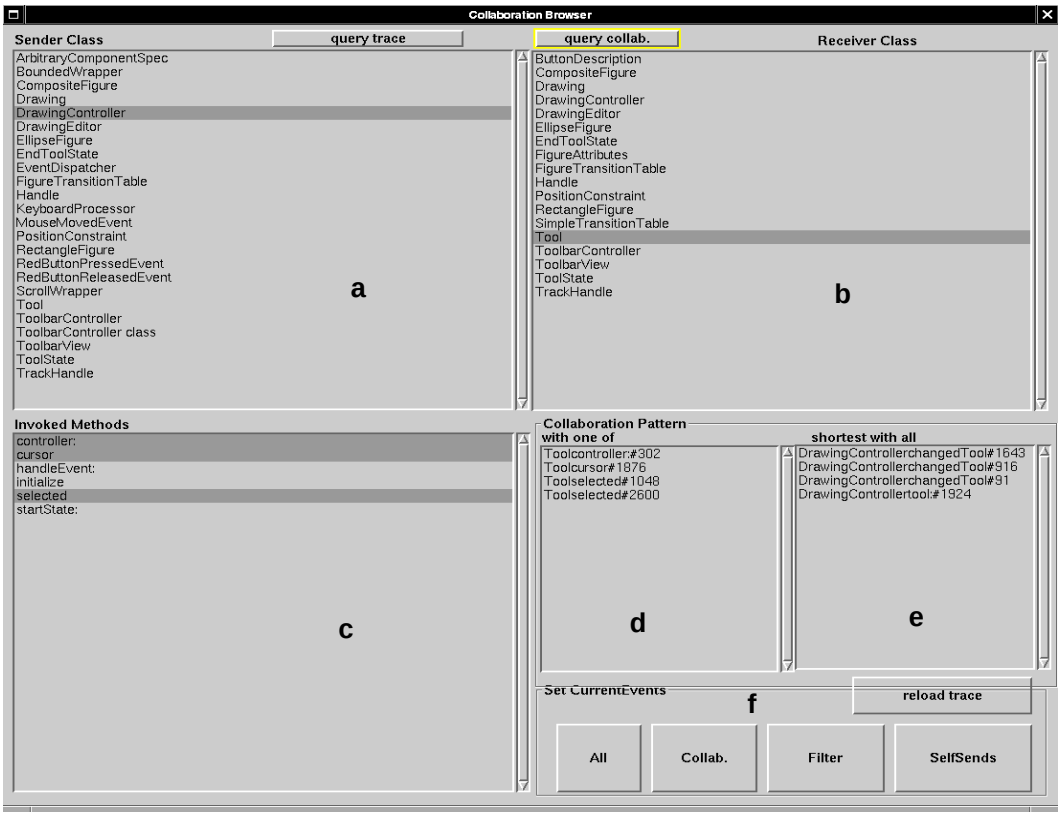

Figure 6.7: Collaboration Browser window: panels a, b and c list the sender classes, the receiver classes and the invoked methods respectively. Panels d and e both list collaboration patterns.

Querying the dynamic information

Q1-Q6: Query about senders, receivers and methods in the context of the whole scenario. The "query trace" button (at the top of the Collaboration Browser window) is used to query the relationships of sender classes, receiver classes and invoked methods in the context of the complete execution scenario.

Example: in Figure 6.7, a sender class, DrawingController and a receiver class, Tool, have been chosen. Querying the trace with these selected sender and receiver classes resulted in panel c being updated to list the methods of class Tool which are invoked by an instance of DrawingController.

Q7-Q17: Query about senders, receivers and methods in the context of a collaboration pattern. The "query collab." button (at the top of the Collaboration Browser window) is used to query the relationship of sender classes, receiver classes, invoked methods and collaboration patterns.

Example: in Figure 6.7 three methods of class Tool have been selected in panel c: controller:, cursor and selected. Panel d list the collaboration patterns resulting from the invocation of each one of the methods selected: Toolcontroller:, Toolcursor and two Toolselected collaboration patterns. In contrast, panel e lists the (shortest) collaboration patterns in which ALL these three methods of class Tool come into play. The list shows four collaboration patterns, three with the name DrawingControllerchangedTool, but each with a different identity number, and one named DrawingControllertool. The first three collaboration patterns result from the invocation of changedTool on an instance of DrawingController, the last one from the invocation of tool on an instance of DrawingController.

Note that the answer to the query about which collaboration patterns include particular participants and methods always provides the *shortest* collaboration pattern which meets the criteria. It is clear that there are many longer collaboration patterns which contain these shortest patterns and there is no interest in exhaustively listing all of them. Panel d lists the shortest collaboration patterns in which *one* of the selected participants occurs, whereas panel e lists the shortest collaboration patterns in which *all* the selected participants occur.

First, using queries Q7-Q12 (parts A and B of Table 6.4), we can ask which collaboration patterns include particular receivers and invoked methods. Second, using queries Q13-Q17 (part C of Table 6.4), we select a collaboration pattern either from panel d or from panel e and ask about the senders, receivers and invoked methods in the pattern. If senders, receivers or invoked methods are also specified, the missing (unselected) elements will be returned as a response to the query. Several queries are here of particular interest:

Q7: collaboration pattern for given participants. Given a list of participants, we ask in which collaboration patterns they occur together.

Q15: role of a class. Selecting a collaboration pattern and a receiver class we ask about the role (a set of methods) this class plays in the collaboration pattern.

Q14: role equivalence. Selecting a collaboration pattern and a role (a set of methods), we ask which classes play this role in the collaboration pattern.

Editing the dynamic information

To focus the investigation on the events of interest the developer can filter out method invocation events which are not relevant by specifying sender classes, receiver classes and methods to be filtered out. This reduces the amount of dynamic information to be analyzed and presented. Another option for focusing on events of interest is to load an instance of one collaboration pattern as the current base of information. This allows the developer to focus on analyzing one collaboration pattern.

The browser queries operate on a current execution trace. When the tool starts up, the current trace corresponds to original execution trace obtained through instrumenting and executing the application. In the course of the iterative process this trace can be edited by the user (using the buttons in panel f) to:

E1: remove method invocation events from the trace for selected senders, receivers and methods,

E2: remove method invocation events which are self-sends,

E3: set the current trace to an instance of a selected collaboration pattern, or

E4: reset the current trace to the original execution trace.

Displaying an instance of a collaboration pattern

The interaction diagram window displays an instance of the selected collaboration pattern as a sequence diagram. It can be displayed (using buttons on the interaction diagram window, see Figure 6.8) as:

D1: All: the whole call tree

D2: Abbreviated: an abbreviated call tree (calls up to depth of 2)

D3: Context: the context of the call tree (an abbreviated view from one level up the call tree).

Example: In Figure 6.7 the collaboration pattern called Drawingcontrollerchanged-Tool#1643 is listed in at the top of panel e. When this collaboration pattern is selected, an instance of the pattern is displayed as an interaction diagram, shown in Figure 6.8. This interaction diagram shows how four objects, instances of DrawingController, Drawing,Tool and ToolState, interact when the method changedTool is invoked on a DrawingController.

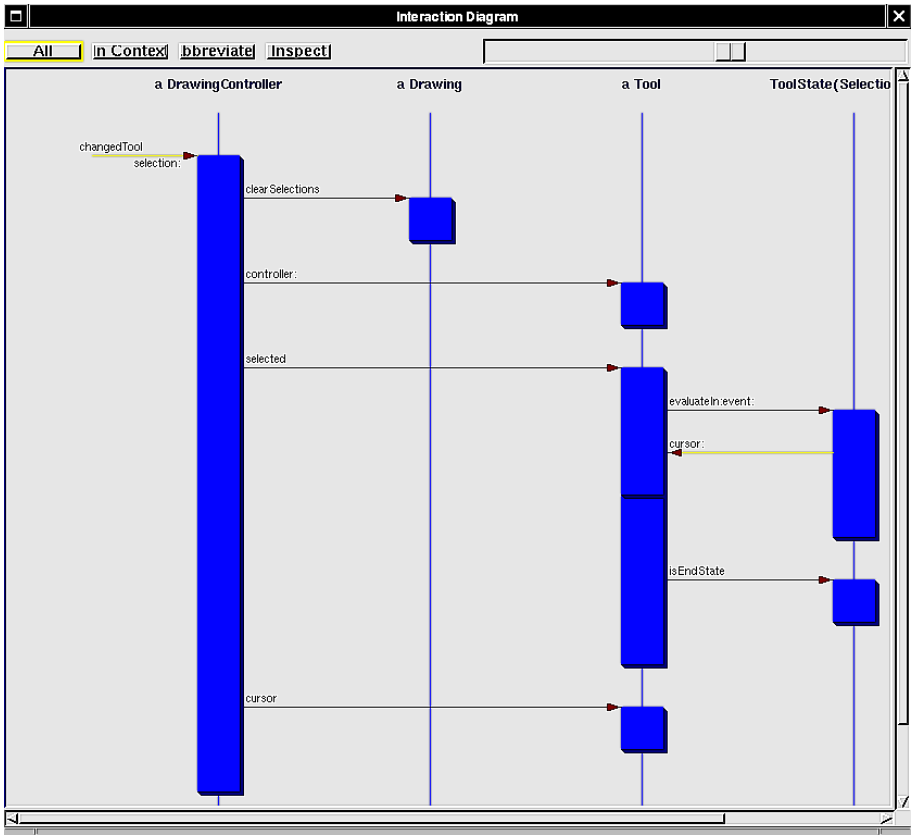

Figure 6.8: The interaction diagram corresponds to an instance of the topmost collaboration pattern in panel e of Figure 6.7.

6.4.2 Implementation

The Collaboration Browser is implemented in Smalltalk. As for the Gaudi tool, we collect dynamic information by first instrumenting the application to be investigated using Method Wrappers [BFJR98], then exercising a scenario on the application. Static information, when required for pattern matching, is obtained directly from Smalltalk. The visualization of sequence diagrams is based on the Interaction Diagram tool [BFJR98].

Pattern Matching. The pattern matching criteria specify what event information is used to label each event node in the call tree of the execution trace, and which event nodes should be ignored. We implement the pattern matching using hashing, by traversing the call tree in a bottom-up fashion and assigning a hash value to each event node in the tree. Hash conflicts are resolved using linear probing.

In assigning hash values we distinguish between the case where the collaboration pattern is treated as a *tree* of event labels, and the case where it is treated as a *set* of event labels:

Tree structure. The tree matching is implemented by computing a hash value for each node in the call tree such that the hash value of a parent node is a function of the

hash values of its children. This matches trees with unordered siblings.

Set structure. The set matching is implemented by building a set of event labels for each event node, then assigning a hash value to this set as a function of the labels in the set.

6.5 Validation of the Approach: Case Studies

In this section we present three case studies to demonstrate the use of the Collaboration Browser and to show how our approach supports the understanding and recovery of collaborations. In the first case study, we walk the reader through the process of design recovery. For the two case studies which follow we describe the process briefly and summarize the results obtained. In the first case study we aim to identify variations on a method invocation and to characterize the resulting collaboration patterns. In the second case study our aim is to decompose the execution trace into a set of collaborations and to understand which classes take part in these collaborations. Finally, in the third case study, we look at the interface of a class and partition it into the roles the class plays in different collaborations.

Note that the first step after the extraction of the source model is the pattern matching. The queries we describe then operate on the execution trace and the set of collaboration patterns created through pattern matching.

The particular pattern matching settings described in the case studies were selected after experimenting with different settings. Some of these experiments are discussed in Section 6.6.3. For all the three case studies, tree matching was considered too restrictive to be useful, so collaboration instances are treated as sets of events. Experiments with different relative levels of invocations and event labeling options led to selecting a high-level perspective of a collaboration instance (looking at only the first few invocation levels), and a ‘polymorphic’ perspective of a method invocation event: we match on the name of the invoked method and the name of the class (in the inheritance hierarchy of the object’s class) which defines this method.

In the first case study we refer to queries and functions of the Collaboration Browser as they are numbered (Q for queries, E for edit, D for display) in Section 6.4.

6.5.1 Investigating Collaborations of Tools in HotDraw

We have already used HotDraw as a case study in Chapter 5, to obtain several high-level views and some views of finer granularity which show invocations between instances around the creation of a rectangle figure. Although a concept view can show information about the invocation of instances, they do not help us to abstract from these to recover collaborations and roles. The Collaboration Browser provides a better user interface for the kinds of queries we want to pose in recovering collaborations.

Background information. HotDraw was presented in Section 5.1. We are interested in particular in the implementation of tools. Tools are used to manipulate the drawing:

create new figures or manipulate the existing figures.

- with which classes does the class Tool collaborate?
- what role does the class Tool have in different collaborations?
- what roles do other classes play which participate with Tool in handling a user event?

Collecting Dynamic Information. We instrument all methods in the HotDraw classes, then run a short scenario on the sample HotDraw editor in which we make use of different tools from the editor's upper panel: create a rectangle and color it, create an ellipse and color it, move the rectangle from back of the ellipse to the front, move the ellipse from back to front, group the two figures, move the two figures, ungroup the figures, move the ellipse.

Pattern matching. We set the following pattern matching options: (i) as information about an event, we choose the method name and the name of the class defining the method, (ii) for events to exclude: the depth of invocation is set to 20, the relative depth of invocation to 3, and we ignore self-sends, (iii) for structure of the collaboration instance: a set of events (rather than a tree).

The scenario executed generated 53,735 method invocation events. The pattern matching resulted in 183 collaboration patterns.

Querying about interfaces. We start by querying about the interface that class Tool presents to other classes in HotDraw. A Q3 query returns all the senders and methods for the receiver Tool. We then proceed with Q1 queries to obtain the methods invoked for each sender class. The results of these queries are given in the Table 6.5 below.

From Table 6.5 we notice that there is overlap in the table cells. That is, some methods of Tool are invoked by instances of two different classes. For example, both EndToolState and ToolState invoke cursorPointFor:, drawing and valueAt:, both FigureTransitionTable and EndToolState invoke figureAtEvent: and both DrawingController and DrawingEditor invoke initialize.

Understanding the context of method invocations. Using the Collaboration Browser we look at the collaboration patterns which result from the invocation of these methods using Q8 queries. In the case of figureAtEvent: we see that the collaboration patterns for this method occur in two different contexts, as illustrated in Figure 6.9 showing two D3 displays. In the first context the method nextStateForTool:event: is invoked on an instance of FigureTransitionTable, which in turn invokes the figureAtEvent: method on an instance of Tool. In the second context the method evaluateIn:event: is invoked on an instance of EndToolState, which in turn invokes three methods on an instance of Tool: controller, cursorPointFor: and figureAtEvent:, and then the method processMenuAt:local:for: on an instance of Drawing.

We repeat a new sequence of Q8 queries and D3 displays to compare the different contexts for the initialize method. When DrawingController is created it initializes a Tool,

Senders	Tool method
Drawing	passInputDown
DrawingController	controller: cursor handleEvent: initialize selected startState
DrawingEditor	initialize passInputDown: startState:
FigureTransitionTable	figureAtEvent:
EndToolState	controller cursorPointFor: figureAtEvent: drawing sensor valueAt:
ToolState	cursor: cursorPointFor: drawing valueAt: valueAt:put:

Table 6.5: Tool interface matrix. It shows the methods of class Tool which are invoked by other HotDraw classes.

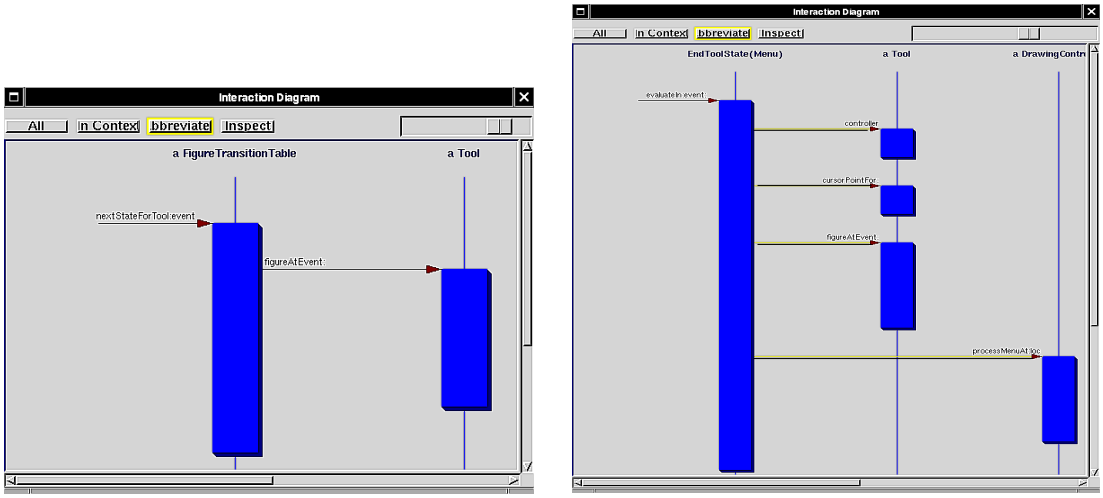

Figure 6.9: FigureAtEvent is invoked on Tool in two different contexts

sets its start state and sets the controller for that tool. DrawingEditor, on the other hand, invokes the initialize method on Tool when it builds the button description for the tool and associates it to an icon. This is a collaboration pattern resulting from the invocation of buildButtonDescriptionForTool:andIcon: on DrawingEditor.

Since ToolState is a subclass of EndToolState, the overlap in the interface Tool presents to these is expected. Using similar queries and displays (Q8 and D3) we discover that the collaboration patterns in which these classes invoke methods on Tool result from an invocation of evaluateIn:Event: on an instance of ToolState or EndToolState – but this method invocation gives rise to several collaboration patterns, depending on the kind of tool in question.

Looking at the collaborations of Tool**.** We look more systematically at the collaboration patterns in which instances of Tool participate. As discussed in Section 6.3, each method invocation recorded in the trace is a collaboration instance. Thus many collaboration patterns are not of great interest because they correspond to a trivial interaction of just one method invocation. In general, then, to arrive at more significant collaboration patterns, we identify patterns in which in which several classes participate or in which a subset of the methods of a class are involved.

For each class in the execution trace (listed in the receivers panel) we use Q7 queries to obtain the shortest collaboration patterns in which both Tool and this class participate. The collaboration patterns obtained in this way are summarized in Table 6.6. The name of each collaboration pattern corresponds to the name of the invoked method which generates the interaction and the name of the class which implements the method. We have also listed each collaboration pattern only once, though in many cases there are actually several collaboration patterns.

Collaboration Pattern Name
DrawingController changedTool
DrawingController tool:
DrawingController handleEvent:
Drawing handleForMouseEvent:
Tool handleEvent:
Tool figureAtEvent:
Tool startState:
Tool selected
ToolState nextStateForEvent:tool:
EndToolState evaluateIn:Event:
FigureTransitionTable nextStateForTool:event:
ToolbarController redButtonReleasedEvent:

Table 6.6: Collaborations involving Tool

Some of the collaboration patterns listed in the table are contained in each other. In particular, we deduce by looking at the collaboration patterns, the nesting relationship between collaboration patterns of DrawingController handleEvent::

```
DrawingController handleEvent:
    Tool handleEvent:
        ToolState nextStateForEvent:tool:
            FigureTransitionTable OR SimpleTransitionTable nextStateForTool:event:
        Tool changedToState:event:
            EndToolState OR ToolState evaluateIn:Event:
            EndToolState OR ToolState isEndState
```

We can thus reduce the number of non-overlapping collaborations in which Tool takes part. Which collaboration patterns we want to eliminate from the table depends on the question which is driving our investigation – on whether we are interested in understanding the elementary collaborations first, or prefer to have an overview of some of the larger collaborations.

Investigating a particular collaboration. We choose to concentrate on the collaboration patterns Tool handleEvent:, to learn about how tools handle user events. There are four collaboration patterns resulting from the invocation of this method. The differences between these is illustrated in Table 6.7: since the pattern matching settings specify a relative depth of 3, only differences in the method invocations up to a relative depth of 3 are seen in the collaboration pattern. The differences of the collaboration patterns are due to different methods executed. The table lists the four variations, each one in a separate row. For each variation, the name of the class which implements the executed method is listed under the column of the method.

	handleEvent:	nextStateForTool:event:	evaluateIn:event:	nextStateForEvent:tool:	isEndState
1	Tool	SimpleTransitionTable	EndToolState	ToolState	ToolState
2	Tool	SimpleTransitionTable	EndToolState	ToolState	EndToolState
3	Tool			ToolState	
4	Tool	FigureTransitionTable	EndToolState:	ToolState	ToolState

Table 6.7: Different collaboration patterns for Tool handleEvent:.

Looking more closely at an instance of each one of these patterns using the interaction diagram display we see that there are principally three variations, since collaboration pattern 1 and 2 are similar. In contrast, collaboration pattern 4 in which a FigureTransitionTable participates, differs considerably from the three others. For each one of these collaboration patterns we query about the participants of the collaboration pattern and their role. From these queries we learn that when the nextStateForTool:event: is invoked on an instance of FigureTransitionTable, rather than on a SimpleTransitionTable, then it in turn requests Tool to provide the figure associated with an event by invoking figureAtEvent:.

Characterizing a collaboration. By querying about each of these four collaboration patterns we extract the role that the participant classes play in each collaboration. This

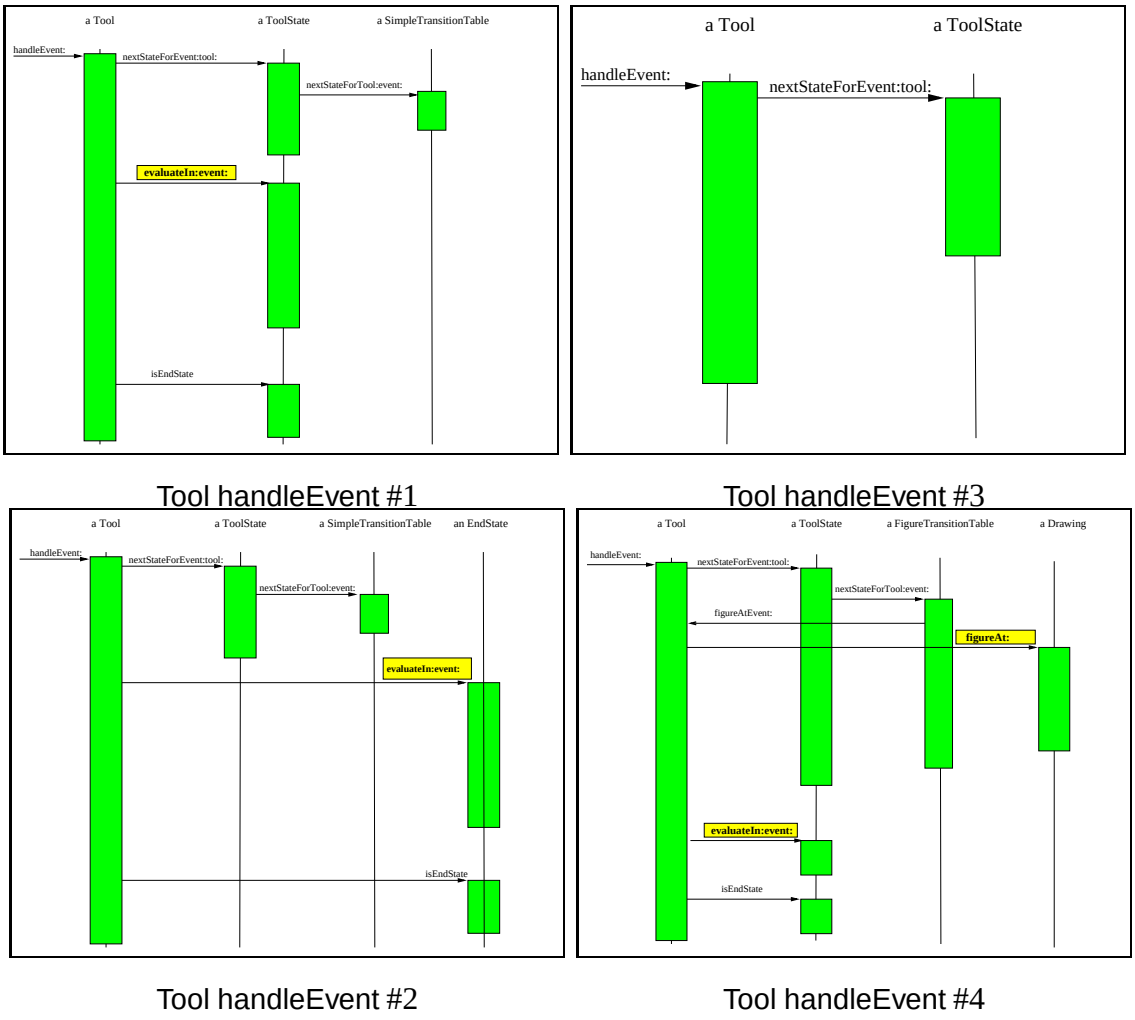

Figure 6.10: Collaboration patterns resulting from invocation of Tool handleEvent. The method invocations in boxes and bold font show invocations which have not been expanded in the interaction diagram.

information is not straightforward to present, since we see that there are two collaborations EndToolState evaluateIn:event: and Drawing figureAt:, whose participants are not predictable – they depend on the user event, and on the figures which are in the drawing. We therefore choose to characterize the predictable elements of the collaboration patterns and to leave the variable elements open. This can be seen in Table 6.8, where the variable collaborations have been denoted by bold faced method names.

Recovering other collaborations. We can now revise the list of collaborations of Table 6.6 : we remove DrawingController handleEvent:, Tool handleEvent:, ToolState nextStateForEvent:tool: and FigureTransitionTable nextStateForTool:Event:, since we understand how they fit together. EndToolState evaluateIn:Event and Tool figureAtEvent: are shaded in the table - these are the ones which are still to be investigated. We thus obtain Table 6.9.

Class name	handleEvent #1+#2	handle Event #3	handleEvent #4
Tool	handleEvent	handleEvent	handleEvent: figureAtEvent:
ToolState	nextStateForEvent:tool **evaluateIn:event:** isEndState	nextStateForEvent:tool:	nextStateForEvent:tool **evaluateIn:event:** isEndState
EndToolState	**evaluateIn:event:** isEndState		**evaluateIn:event:** isEndState
SimpleTransitionTable	nextStateForTool:event:		
FigureTransitionTable			nextStateForTool:event:
Drawing			**figureAt:**

Table 6.8: Class-Collaboration description for collaboration Tool handleEvent:. Each column corresponds to a collaboration, each row to a class. The table cells give the role of the class in the collaboration. The methods in bold represent unexpanded collaborations which result in variations on the collaboration patterns.

Collaboration Pattern Name
DrawingController changedTool
DrawingController tool:
Drawing handlerForMouseEvent:
Drawing controller
EndToolState evaluateIn:Event
Tool figureAtEvent:
Tool startState:
Tool selected
ToolbarController redButtonReleasedEvent:

Table 6.9: Collaborations involving Tool

We continue in the vein of the investigation described above to discover the role of Tool in these collaborations and the other participants of these collaborations. Each collaboration recovered represents an important task in which Tool interacts with other classes.

By characterizing the collaboration patterns and their variations we also learn more about interactions in the application.

6.5.2 Collaborations in CodeCrawler

In this section we present some results from using the Collaboration Browser to understand collaborations in CodeCrawler [Lan99][DDL99]. CodeCrawler is a tool which supports reverse engineering through the visualization of code metrics. It combines a range of code metrics with a variety of layout and display options to produce graphs which help an engineer to understand a software system and to detect problems and anomalies.

Background information. CodeCrawler is implemented in Smalltalk and consists of 86 classes. In our investigation we are interested in how instances of classes interact in creating a CodeCrawler display. We therefore look basically at four inheritance hierarchies: Item, ItemFigureModel, Figure and AbstractLayout. We know that a graph, a concept implemented in class Graph, is composed of nodes and edges and that it represents the Model behind the display. The actual view is represented by classes of the ItemFigureModel, Figure and AbstractLayout hierarchies. Figure 6.11 below illustrates these hierarchies with the main classes which appear later in the scenario.

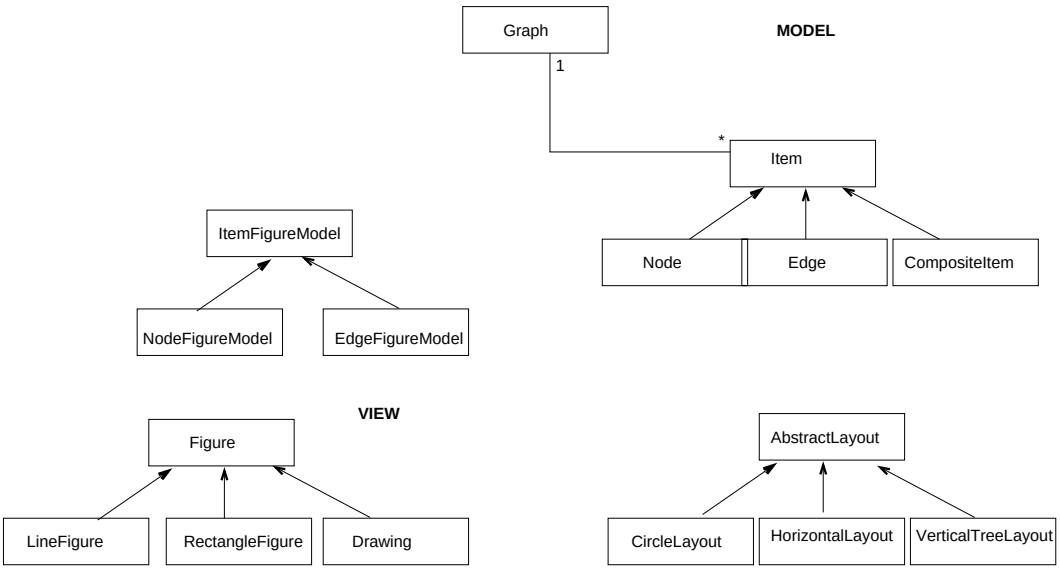

Figure 6.11: The known elements of CodeCrawler

We want to know how the main elements illustrated in Figure 6.11 collaborate to create a display in CodeCrawler. In particular we are interested in the different roles played by the classes in ItemFigureModel hierarchy.

Collecting Dynamic Information. We instrument all the methods of the classes in these four inheritance hierarchies, for a total of 41 classes. We then execute the following scenario: load a software application to be displayed, run Class BluePrint operator to

collect metrics, open CodeCrawler, choose inheritance classification display, display it, enlarge display, choose circular display, vertical tree display, horizontal display.

Pattern matching. We set the following pattern matching options: (i) for information about an event, we choose the method name and the name of the class defining the method, (ii) for events to exclude: the depth of invocation is set to 20, the relative depth of invocation to 4, self-sends are not ignored, (iii) for structure of the collaboration instance: a set of events (rather than a tree).

This scenario generated 7,132 method invocation events. The pattern matching resulted in 116 collaboration patterns.

Looking at the interface of ItemFigureModel **classes.** Since we are interested in the roles played by the classes NodeFigureModel and EdgeFigureModel we first look at the interface that these classes present to other classes. The results of the queries are summarized in Table 6.10 below.

Senders	NodeFigureModel	EdgeFigureModel
CircleLayout HorizontalLineLayout QuadraticLayout VerticalTreeLayout	translateTo: width height childrenFigures descendantFigures isRootFigure level level:	
Drawing	bounds figure isEdgeFigure isNodeFigure zoomByFactor:	figure isEdgeFigure isNodeFigure
EdgeFigureModel	addChildFigure: addEdgeFigure: addParentFigure: figure	
ViewBuilder	applySizeMetrics figureClass: fillColor: initialize initializeFigure isAttributeFigure isClassFigure isFunctionFigure isMethodFigure item item:	figureClass: initialize initializeFigure item: node1Figure: node2Figure:

Table 6.10: Interface matrix for NodeFigureModel and EdgeFigureModel.

From Table 6.10 we learn that the layout classes invoke methods on NodeFigure-

Model, but not on EdgeFigureModel. Furthermore, whereas EdgeFigureModel invokes methods on NodeFigureModel, the reverse does not hold. This asymmetry of nodes and edges suggests that nodes are more 'central' than edges in CodeCrawler. Layout is likely done for nodes, and the edges follow.

Recovering Collaborations. As in the first case study, we used queries to find the collaboration patterns in which these two classes participate. For each of the collaboration patterns we then recorded all the participants in the collaboration and the part of their public interface which is invoked in the collaboration. In Table 6.11 below we list the main collaborations and the participants in these collaborations. We have chosen to omit here the roles of the participants as this results in a very large table, and the details are not of interest here. The interesting thing is that we can 'decompose' the trace information into collaborations and we can see the patterns of collaboration: which classes interact with other classes, and in which context.

How did we decide which collaboration patterns to look at? In extracting collaborations we look at collaboration patterns which do not contain each other, and which are not trivial, *i.e*, they have two or more participants. The collaboration patterns shown in the table meet this criteria. This is why no ViewBuilder collaboration patterns are seen in Table 6.11: each ViewBuilder collaboration pattern also includes one or more of the patterns given in the table.

Through querying we realized that most of the trace generated consists of one large collaboration pattern (5,073 invocation events out of total of 7,132), from the execution of the method createViewFromScratch on an instance of ViewBuilder. This method, whose code is given below, builds up a CodeCrawler view from scratch. First the model aspects are handled: building the graph from nodes and edges then the view aspects of sorting the nodes and coloring them and finally the layout of the graph is done.

```
createViewFromScratch
    self initialize.
    self initializeAttributeNodePluginMetrics.
    self initializeClassNodePluginMetrics.
    self initializeMethodNodePluginMetrics.
    self initializeFunctionNodePluginMetrics.
    self inferNodesToBeDisplayed.
    self inferEdgesToBeDisplayed.
    self sortNodeFigures.
    self colorizeFigures.
    self layoutFigures
```

We have therefore sorted the collaborations in an approximate ordering corresponding to the execution of this method. Collaboration patterns which handle model aspects occur first, followed by collaborations which create the figure to be displayed and finally the collaboration which does the layout and displays the drawing. We have also sorted the participants in such a way that participants which belong to the model aspect appear first, followed by those that belong to the view concept. The table then shows a diagonal structure: as we proceed in creating the view, we use less and less the model classes

and more and more the view classes. Table 6.11 indeed confirms the central role that NodeFigureModel plays in the creation of a CodeCrawler display.

	Collaboration	Graph	Node	Edge	NodeFigureModel	EdgeFigureModel	RectangleFigure	LineFigure	Layout	Drawing
1	Graph classNodes	X	X							
2	Graph inheritanceEdges	X		X						
3	NodeFigureModel applySizeMetrics		X		X					
4	NodeFigureModel isClassFigure		X		X					
5	NodeFigureModel initializeFigure:				X		X			
6	EdgeFigureModel initialize				X	X				
7	EdgeFigureModel initializeFigure:			X	X	X		X		
8	AbstractLayout initialize:				X				X	
9	AbstractLayout LayoutAt:				X				X	
10	Drawing displayedEdgeFigure				X	X				X

Other collaboration patterns with the same participants as collaboration patterns:
4. isAttributeFigure, isFunctionFigure, isMethodFigure.
8. CCVerticalTreeLayout levelWidthOfSubtree:atLevel:, CCTreeLayout nodeFiguresOnLevel:outOf:, CCTreeLayout maxLevelOfNodeFigures: and CCTreeLayout initializeLevelOfChildrenOfNode:.
10. Drawing displayedNodeFigure.

Table 6.11: Class-Collaboration matrix for collaborations in CodeCrawler

6.5.3 Collaborations in the Refactoring Browser

In this section we present some results from looking at the collaborations of the class CompositeRefactoryChange in the Refactoring Browser [RBJ97].

Background information. Recall that in the case study in Section 5.2 we looked for a facade in the Refactoring Browser. In that investigation we wondered about the role of CompositeRefactoryChange. It is difficult with concept views to convey what is happening in the application in terms of the sequence of calls and the responsibilities of classes

in different contexts. We now use the Collaboration Browser to look more closely at the collaborations in which CompositeRefactoryChange takes part.

Collecting dynamic information. As in the second scenario presented in Section 5.2, we instrument the classes in the two categories Refactory-Refactorings and Refactory-Support, as well as the class SystemNavigator, and run a refactoring scenario to push a method down the inheritance hierarchy, then to rename a method.

Pattern matching. We set the same pattern matching options as for the CodeCrawler case study. The executed scenario generated 1,724 method invocation events. The pattern matching resulted in 125 collaboration patterns.

Querying about the interface of CompositeRefactoryChange**.** We look at the interface that CompositeRefactoryChange presents to other classes in the scenario – this is given in Table 6.12. It shows that both Refactoring classes instances invoke the same methods on instances CompositeRefactoryChange. In the table we also give the methods that instances of CompositeRefactoryChange invoke on *other* instances of the same class.

Senders	CompositeRefactoryChange method
CompositeRefactoryChange	changes: postcopy
RenameMethodRefactoring PushDownMethodRefactoring	addChange: addChangeFirst: execute executeWithMessage: initialize name removeMethodFrom: compile:in: compile:in:application:classified:

Table 6.12: Interface matrix for CompositeRefactoryChange

Querying about the collaborations of CompositeRefactoryChange **with Refactoring classes.** We query about the collaboration patterns in which CompositeRefactoryChange and a Refactoring class both participate. These are listed in Table 6.13. For each collaboration pattern we give the role that CompositeRefactoryChange plays in the collaboration.

We use the Collaboration Browser to look at the collaboration patterns in Table 6.13. We see that the structure and functionality of Refactoring convertMethod:for:using:notifying and PushDownMethodRefactoring pushDown:using: are similar – they are both collaborations in which a CompositeRefactoryChange creates an AddMethodChange object and sets its attributes. We also see that the two collaboration pattens Refactoring performChange: and Refactoring performChange:withLabel have similar structure and functionality – in both cases the elementary changes are executed. In both cases the CompositeRefactoryChange object which executes the changes, on the invocation of execute or executeWithMessage: also creates a deep copy of itself. Though we do not understand from this all the details of what is happening, we suppose that this is

Collaboration	Role of CompositeRefactoryChange
Refactoring initialize	name: initialize
Refactoring convertMethod:for:using:notifying	compile:in:
PushDownMethodRefactoring pushDown:using	compile:in:application:classified:
Refactoring performChange:	addChange: addChangeFirst: postCopy changes: execute
Refactoring performChange:withLabel:	addChange: addChangeFirst: postCopy changes: executeWithMessage:

Table 6.13: Collaborations for CompositeRefactoryChange

for the purpose of undoing changes in a transactional way. We thus compress Table 6.13, to arrive at Table 6.14.

Collaboration	Role of CompositeRefactoryChange
create a composite change	name: initialize
create and set a change object	compile:in: OR compile:in:application:classified:
execute the changes	addChange: addChangeFirst: execute OR executeWithMessage: *in execute:* *create a deep* postCopy *copy of itself* changes:

Table 6.14: Collaborations for CompositeRefactoryChange: here we have grouped together similar collaborations from Table 6.13.

From this table we see that these collaboration patterns partition the interface of CompositeRefactoryChange into roles. There remains, however, the method removeMethodFrom: which is unaccounted for. We query to obtain collaborations in which this method is invoked and obtain two: ChangeMethodNameRefactoring performRefactoring and PushDownMethodRefactoring performRefactoring. Selecting each of these collaboration patterns and querying about their participants, we obtain a nesting relationship which indicates that these collaboration patterns include also the collaboration patterns Refactoring performChange: and Refactoring performChange:WithLabel:. We thus see where each of the methods of the public interface of CompositeRefactoryChange comes into play.

6.6 Evaluation and Discussion

What do these three case studies demonstrate? In the first case study we walked the reader through the process of using the Collaboration Browser. We also showed how queries are used to identify variations on a method invocations and to characterize the resulting collaboration patterns. In the second case study we showed how the execution trace can be decomposed into a set of collaborations, thus giving us a feeling for the patterns of collaboration: which classes collaborate with each other, and in which context. Finally, in the third example we partitioned the interface of a class into several roles and identified the collaborations in which it plays these roles.

The case studies demonstrate that the queries aid us in locating interesting collaborations and in understanding the role of a class in a collaboration. They also show that the task is not simple: we cannot automatically obtain enlightening information – rather we must work in interpreting the information obtained and in deciding on the best way to explore collaboration patterns.

In this section we evaluate the approach in the light of the case studies conducted and discuss some critical issues.

6.6.1 Lessons learned

The case studies demonstrated that we can quickly locate the information of interest, and find collaboration patterns which should be further investigated. What we also see from these case studies is that it is rare that we can actually partition the public interface of a class into separate sets of methods which correspond to the roles of a class in different collaborations. It is also a challenge to find the right settings for the pattern matching criteria for each case study so that we are not presented with too many variations on a method execution, while at the same time getting some information about important variations. Section 6.6.3 discusses our experiences and results with a range of matching schemes.

By characterizing a collaboration and its variations we gain a better understanding of how the functionality of the software is carried out through the interaction of instances. First, it helps us to understand the function of a class, by seeing the different roles it plays. Second, it tells us where in the code we should look to understand the protocol of interaction. Finally, through the iterative process of recovering information about a collaboration we learn much about the program behavior.

Although we were interested to know how far we could go in collaboration recovery without visualization, in working with the tool we realized it was important to integrate a sequence diagram visualization in order to display collaboration instances. This was certainly a help in the process of collaboration recovery.

Table 6.15 below summarizes the size of the case studies in terms of the number of classes instrumented (number of methods), the resulting number of method invocations in the execution trace, and the number of collaboration patterns which resulted with the pattern matching scheme.

Application	Classes	Methods	Invocations	Collaboration Patterns
HotDraw	39	548	53,735	183
CodeCrawler	41	426	7132	116
Refactoring Browser	68	657	1724	125

Table 6.15: Size of case studies.

6.6.2 Towards a Methodology

The process of extracting collaborations using the Collaboration Browser is an iterative one – the result of one query leads to another query, and so the user focuses on classes and collaborations of interest. Below we sketch the process, giving a rough ordering of different kinds of queries. The first step in the process is the creation of collaboration patterns through the setting of the pattern matching options. In the process of querying, though, we may want to alter these settings to create a new base of collaboration patterns.

1. *Creating collaboration patterns.* We start by setting the pattern matching criteria and launching the pattern matching to create the collaboration patterns which form a base for the querying.

2. *Querying about interfaces.* In querying we generally start by finding out which classes communicate with each other. For this we query to find the interface a class presents to other classes.

3. *Looking for a collaboration pattern.* We query about collaboration patterns in which certain classes participate, or ones in which certain methods come into play.

4. *Looking at all the participants for a collaboration pattern.* Once we have obtained several collaboration patterns which are of interest, we want to know which classes participate in a given collaboration pattern, and what role each class plays.

5. *Understanding a collaboration.* The interaction diagram displays aid us in understanding a collaboration. We can also load an instance of a collaboration pattern as the current base of dynamic information, and begin at step 2. again, this time working with a smaller base of dynamic information.

6.6.3 Pattern Matching

In the case studies presented we concentrated on the process of recovering collaborations by querying and on showing what kinds of questions can be answered using this approach. In this section we look at pattern matching criteria as they are practically used. We start by summarizing guidelines from our experience with case studies. We then look at each of the pattern matching options. Finally, we look at the effect of the size of the trace on the number of collaboration patterns obtained.

Some guidelines. Our experiments suggested that treating collaboration instances as trees is too restrictive for design recovery. We therefore chose to treat collaboration instances

as sets of events. Self-sends can safely be ignored without loss of 'meaning'. The two criteria most useful to modulate are the relative depth of invocation and the event labeling options. Matching on object identity is in most cases too restrictive unless we want to detect the use of a particular object. Labels which allow for polymorphism, such as the name of the class defining the method, enable us to abstract away from the actual class of the object or the method name to something with the same 'meaning'.

Collaboration Structure. As mentioned in Section 6.3.2, the problem of matching similar collaboration instances is a problem of approximate tree matching. Each collaboration instance is a subtree in the execution trace with nodes labeled with send event information: (SenderClass, SenderInstance, ReceiverClass, ReceiverInstance, InvokedMethod). If we are flexible on how we match nodes, *e.g.* only some parts of the label should match, then we can match trees whose nodes differ slightly. The structure of the tree itself will also determine which trees will be matched as similar. Several algorithms exist for approximate tree matching [ZSW94], which measure similarity of two trees in terms of the deletion, insertion and relabeling operations which convert one tree to another. We have chosen to arrive at a approximate matching, though, by doing away with some nodes altogether, in effect filtering them out, and by considering another way to view the tree structure.

If collaboration patterns are treated as trees, then two collaboration instances match if they have equivalent call trees of method invocation events. Tree equivalence can be defined recursively: two trees are equivalent if the label of their root node matches, and their subtrees have a one-to-one matching.

This kind of matching, however, does not take into account some variations of a collaboration. For example, similar collaborations in which the number of participants is different will not be matched, since their call trees have a different structure. The same is true for iterations of message exchanges which vary in the number of iterations.

We can *flatten* the collaboration tree and represent it as a set of node labels. Here we make the assumption that this kind of structure is not likely to be duplicated for collaborations which differ semantically to a great degree. This seems like a reasonable assumption. Consider each method invocation event as a node-labeling token in an alphabet. For general trees (general language) the probability of a token labeling any node is the same, whereas trees which represent method invocation events represent a highly-structured language where the distribution of the tokens in specific subtrees is not random.

Practically speaking, flattening a tree structure means that a collaboration is represented as a set of node labels. Collaborations are considered equivalent if their label sets are identical. This matching scheme gets rid of nesting relation on the events and ignores iterations and recursions. The node labeling options control which event information enters into the label.

In our case studies we have treated collaboration patterns as sets of labels rather than as trees, since we considered tree matching too restrictive for the kind of information we were trying to extract. Table 6.16 below compares the number of collaboration patterns obtained for each kind of structure matching. The other matching parameters were set as follows: no limit on relative depth of invocation or on depth of invocation (it was set

greater than the deepest collaboration instance), ignoring self-sends, matching nodes on defining class and name of method.

Structure	Collab. Patterns	Collab. patterns > 1	Tool handleEvent
Set	295	235	25
Tree	428	368	33

Table 6.16: Number of collaboration patterns as a function of collaboration structure for the HotDraw case study

Table 6.16 lists the total number of collaboration patterns obtained, the number of collaboration patterns whose length (number of method invocations occurring in the pattern) is greater than 1, and also the number of collaboration patterns obtained for the collaborations resulting from the invocation of handleEvent: on Tool. Recall that in Section 6.5.1 we looked at four variations of this collaboration.

Relative depth. In the HotDraw case study we set the relative depth for matching at 3. Table 6.17 shows the effect or different values of relative depth on the total number of collaboration patterns found, on the total number of collaboration patterns which correspond to more than just one method invocation, and the number of collaborations found for Tool handleEvent. Here the total depth of invocation is set to 25, the event label to defining class and name of method. We ignore self-sends, and treat the collaboration structure as a set of labels.

relative depth	Collab. Patterns	Collab. patterns > 1	Tool handleEvent
1	127	68	1
3	183	121	4
7	287	225	21
max(23)	295	235	25

Table 6.17: Number of collaboration patterns as a function of relative depth for the HotDraw case study

Table 6.17 shows that there are 127 different node labels (defining class and name of method), as obtained with matching to a depth of invocation of 1. Also there are a maximum of 295 collaboration patterns when the relative invocation depth is set to the maximum depth of an invocation in the trace. When self-sends are taken into account as well, this figure rises to 572. In general, ignoring self-sends reduces the number of collaboration patterns by approximately a factor of 2.

In the HotDraw case study increasing the relative depth of invocation to 7 resulted in 21 different collaboration patterns for the invocation of handleEvent on Tool. The same 'dramatic' effect was not observed in the other two case studies, where there were much fewer variations on the execution of a method.

Event node labeling. The information we choose for labeling the event nodes depends on the aspects of a collaboration we are interested in. In design recovery we are usually not

interested in what is happening at the object level (exactly which instances send messages to each other), but rather in generalizing from these object interactions. Labeling an event node with the names of the sender and receiver classes and ignoring method names, for example, will match all collaboration instances which have the same sender-receiver pairs, regardless of the invoked method. Labeling an event node with the name of the selector category rather than the method name will match collaboration instances where methods with similar semantics (since they are in the same category), but different method names are invoked. For example, in the CodeCrawler case study, using defining class and method category as a labeling function, we match all the collaboration patterns listed in row 4 of Table 6.11 as one collaboration pattern NodeFigureModel testing, instead of the four different collaboration patterns obtained with matching on method name. In our case studies we used the method name together with the name of the superclass which defines this method as our node labeling.

Size of the trace. Our case studies showed that pattern matching compressed the amount of information we operate with considerably: in the HotDraw case study 50K method invocations resulted in 180 collaboration patterns. But what happens as the size of the trace increases? To get a feeling for this we ran a few more scenarios to obtain figures for larger traces. In each of these cases we treat the collaboration structure as a set, ignore self-sends, do not restrict the relative or absolute depth of invocations and label the nodes with defining class and method name.

Application	Classes	Methods	Invocations	Collab. Patterns	Deepest Collab.
HotDraw	39	548	35,435	177	19
HotDraw	39	548	53,735	295	23
HotDraw	39	548	140,585	393	23
CodeCrawler	41	426	7,132	116	11
CodeCrawler	77	669	19,369	128	8
CodeCrawler	77	669	86,992	186	13

Table 6.18: Number of collaboration patterns as a function of number of events

Table 6.18 lists the number of collaboration patterns as a function of the size of the trace. It also gives the number of classes and methods instrumented, and the depth of the deepest collaboration instance in the trace. This information is graphed in Figure 6.12. It is hard to project from these few data points to draw conclusions about the number of collaboration patterns found as the trace grows even larger. It is clear that this also depends on the *breadth* of the scenario – for a scenario which exercises the same basic functionality over and over we would expect the curve to level off as the size of the trace grows, but a scenario which exercises new parts of the system as time proceeds may result in a steady increase in the number of collaboration patterns found.

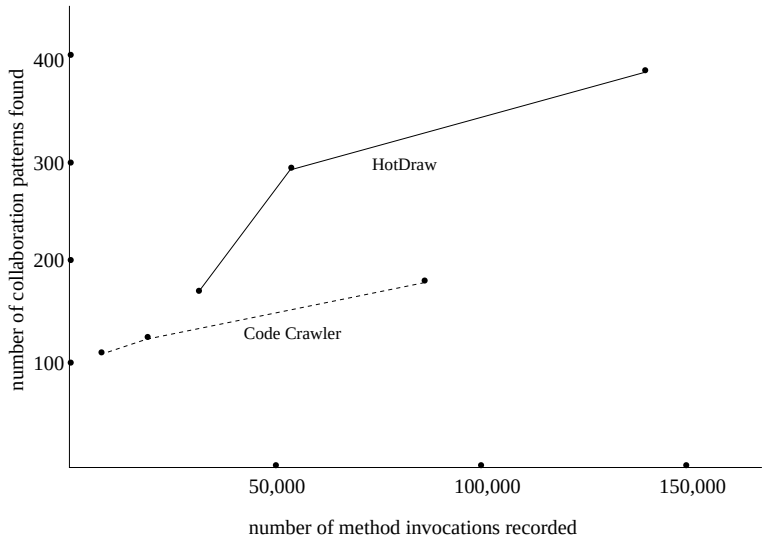

Figure 6.12: Number of collaboration patterns as a function of size of trace

6.6.4 Characterizing Collaborations and Roles

An important issue raised by this work is the characterization of collaborations. The notation currently used to model object-oriented collaborations are UML interaction diagrams. As discussed in Section 6.3, UML interaction diagrams are used at the design level, so it is hard to tie them to collaborations occurring in the code. It would be interesting to experiment with other ways of modeling collaborations which can express similarity of collaboration instances found in a trace.

In our case studies we have chosen to represent a collaboration as a set of participants and their corresponding role. This is also in synch with the pattern matching settings we use. Since we label each method invocation event with the name of the invoked method and the name of the class defining that method, we treat collaboration instances as equivalent when they have the same labels in their set. If, on the other hand, we include also the name of the sender class in an event label, then the collaboration pattern is better characterized as a matrix. For example, Table 6.19 shows the collaboration pattern Tool handleEvent: #4 (Table 6.8) as a matrix.

sender—receiver	Tool	ToolState	EndToolState	FigureTransitionTable	Drawing
Tool		nextStateForEvent:tool **evaluateIn:event:** isEndState	**evaluateIn:event:** isEndState		**figureAt:**
ToolState				nextStateForTool:event:	
EndToolState					
FigureTransitionTable	figureAtEvent:				
Drawing					

Table 6.19: Collaboration matrix description for collaboration Tool handleEvent: #4

Characterizing roles. The same issue of characterizing collaborations applies also to

the characterization of roles. In the Collaboration Browser roles are defined in terms of method names. Matching on roles is therefore textual: we look at the method name to see if it occurs in the collaboration pattern. But there are cases where two instances play a similar role in a collaboration though the method names differ – just as abstract classes generalize from their subclasses, an abstract characterization of a method would generalize from method names. We also experimented with matching methods by the method category they belong to, to arrive at a behavioral semantics of the method independent of its name.

6.6.5 Generality of the Approach

Language independence. The approach we present is general enough to be used for the extraction of collaborations from software applications implemented in any class-based object-oriented language. The approach relies most heavily on dynamic information, static information is used in pattern matching for assigning labels to nodes, *e.g.*, matching on method category name, matching on the name of the class defining the methods. The pattern matching scheme can easily be tailored to accommodate static groupings for other object-oriented languages.

Efficiency. Once pattern matching has been done, we are querying a relatively small database. Without putting much effort into query and algorithm optimization, the response times obtained in the case studies for both 'query trace' and 'query collab' queries were fast (less than 2 seconds).

Scalability. The collaboration recovery technique is scalable to large systems for the following reasons:

Compression of information through pattern matching. Data from our case studies shows that the number of collaboration patterns obtained as a percentage of the number of method invocations in the trace range from less than 0.5% (for the HotDraw case study) to 7.2% (for the Refactoring Browser case study). As discussed Section 6.6.3, if we execute a scenario to exercise only a specific functionality, we expect that the number of collaboration patterns found would form a decreasing percentage of the trace size as the trace size grows.

Information presented through classes and methods. The query model we support presents the information to the developer in terms of classes, methods and collaboration patterns. Even for large systems, this information can easily be searched and accessed. We can also consider grouping classes and methods as they are grouped in the Smalltalk browser: contained in applications or class categories, or in method protocols.

Meaningful collaborations are naturally small. By this we mean that in seeking to understand collaborations in a software system we look for interaction where only a subset of the classes participate.

Limitations. We treat a collaboration instance as the execution sequence of *all* the events which result from a method invocation, rather than looking at an arbitrary sequence of events within a method invocation. This has simplified the implementation of pattern matching as an operation on trees. But we could also consider a broader definition of collaboration instance, and as a result a broader notion of collaboration.

In our characterization of collaborations we represent the role of a class in a collaboration as a set of all the methods invoked on instances of that class in the collaboration. That is, in a single collaboration, we do not consider that different instances of the same class play different roles, or that a single instance could switch roles. A finer analysis of a particular collaboration pattern could yield a more refined partitioning of different roles.

The Collaboration Browser does not currently support queries which can relate one collaboration pattern to another (*e.g.*, which collaboration pattern is contained in another), but this might be a useful extension for a better navigation of collaboration patterns.

6.6.6 Related Work

Several visualization approaches to dynamic information have been discussed in Section 2.2.3. We compare this present work with two approaches which use pattern matching as a technique for reducing information overload [PLVW98][JR97][JSB97].

The work on execution patterns [PLVW98] reports experiments with a range of elision techniques which allow an engineer to visually recognize patterns in the interactions of objects. The dynamic information is presented as a modified form of a sequence diagram where the flow of control is depicted always from left to right (so there may be more than one time line for each object). Using this display as a basis, several chart metaphors are introduced to represent message send behavior more compactly. Pattern matching is also used to simplify the presentation of the data by hiding details not considered essential and so highlighting similar sequences in the execution trace.

In ISVis [JR97] [JSB97] pattern matching is used, as in our approach, to group together similar execution sequences. The patterns identified are used in several different visualizations. Execution patterns are plotted against all the classes of the application in a visual 'pattern matrix' which enables the developer to scan the patterns for ones with specific classes as participants. This is a form of visual querying similar to our querying about collaboration patterns with specific participants. Patterns are also plotted against the length of the trace in a mural view, so that the occurrence of a specific pattern in the trace can be localized.

The work we presented here is similar to the work described in [JSB97] and [PLVW98] in the goal of finding recurring patterns as a way of recognizing important design concepts. Our work is, however, not oriented towards program visualization: we use only a simple sequence diagram visualization to display an instance of the collaboration pattern chosen. Our main focus is on using the dynamic information to help in the recovery of roles and collaborations. We see our work as complementary to the visualizations proposed in [JSB97] and [PLVW98]: these tools display an entire trace and give the user a feel for the overall behavior of an application and the repeated occurrence of

patterns in order to identify different phases of execution, whereas our approach focuses on the roles of classes in much smaller chunks of interaction, without considering the time dimension of the trace.

Work on the detection of design patterns in code is also related to our work since design patterns often represent a collaboration. These have been discussed in Section 2.2.4. Finding collaborations may aid in detecting design patterns.

We know of only one other approach which explicitly aims to reverse engineer collaborations[Hon98]. In this approach, static information only is used to recover Collaboration Contracts [Luc97]. The approach is an incremental one, in which a *classification browser* is used first to classify a set of classes of the application as participants of interest and then to edit their interface, so as to arrive at a description of participant-roles in a collaboration. This classification of participant-roles is then enlarged by all the acquaintances of the participants already in the classification. Finally, by looking at the method bodies of all the participants-methods in the classification the calling relationships between methods can be inferred, albeit without an ordering relationship on the method calls. The classification browser approach relies even more heavily than our approach on the input of a user who must select the initial participants and their roles in the collaboration and in determining appropriate acquaintances to include in the collaboration.

6.7 Revisiting the Requirements

We now evaluate our work on collaboration view recovery in the light of the requirements we have set for ourselves in Section 2.3.

1. **Lightweight information model.** This has been discussed in Section 5.4. For recovering collaborations, we use static information only in pattern matching for labeling the method invocation events, *e.g.*, to generalize from a method name to a method category, or from a receiver class name to the name of the class in the inheritance hierarchy which defines the executed method. Collaboration view recovery thus requires static information about classes and the methods they define, and about inheritance relationships.

2. **Simple view specification.** The pattern matching criteria are simple to set and the query model proposes simple queries about basically two relations.

3. **Succinct views.** What is an extracted view in collaboration view recovery? The artifacts we usually aim to recover from this analysis are class-collaboration matrices, or some characterization of a collaboration pattern. We are usually not interested in carrying out this analysis for all the classes in the software system but for a particular subset of classes. The information which results can thus be summarized in a compact way.

4. **Developer guides the process.** The developer guides the recovery process by setting pattern matching options and by querying the dynamic information. The first case

study demonstrated the process of collaboration recovery in which a developer poses a question and launches queries in order to understand a collaboration and to characterize its variations.

5. **Extensible view specification.** In collaboration view recovery we are restricted to querying about two kinds of relations: method invocations in a trace, and method invocations in the context of a collaboration pattern. So collaboration view recovery does not provide for extensible view specification. This is provided for in concept view recovery. The proposed pattern matching criteria can, however, be extended by taking into account, for example, other kinds of static information in labeling method invocation events, or by considering other options for describing the structure of a collaboration instance.

6. **Behavioral views.** Here we look again at the kinds of questions we want to be able to answer using behavioral views:

   - how do the main domain elements relate to each other?

     In collaboration view recovery the main elements are classes, methods and collaboration patterns. We can query about their relation to each other within the two relations supported by the query model. Concept view recovery supports a wider range of relations and enables the developer to create component abstractions corresponding to domain concepts.

   - what parts of the software implement a specific feature?

     In collaboration view recovery we can ask which classes and methods implement the functionality of a particular method. Again, concept view recovery can provide answers to this question in a wider sense.

   - to which messages do instances of a certain class respond?

     This information can be obtained by querying the trace (‘query trace’ in the Collaboration Browser) about the senders and invoked methods for a particular receiver class.

   - how many instances of a class are present at runtime?

     Collaboration view recovery cannot provide us with an answer to this question. It can be obtained in concept view recovery.

   - which objects are responsible for creating instances of a certain class?

     Again, the answer to this question can be obtained in concept view recovery.

   - which class instances participate in the interaction resulting from the invocation of a particular method?

     Whereas in concept view recovery this question can be answered only for one particular method invocation, collaboration view recovery provides us with information about different collaboration patterns resulting from the invocation of a particular method.

- what are variations on the way a method is executed?

  This question can also be answered in collaboration view recovery. The first case study demonstrated how the querying facility of the Collaboration Browser helps us to understand the variations of a method execution.

7. High-level views. Though the views obtained in collaboration view recovery are low-level in the sense that they look at the method level, they can be considered high-level when they help to decompose a software system (for a particular scenario) into a set of collaborations, as was done for the CodeCrawler case study.

8. Low-level views. Low-level views in collaboration view recovery correspond to characterizations of a collaboration pattern.

6.8 Conclusions

In this chapter we presented collaboration view recovery. We argued that collaboration-based design artifacts are an important aid in understanding, maintaining and evolving object-oriented software, and presented the challenges to extracting these from code. We then introduced pattern matching as a technique to identify similar execution sequences in the trace as instances of one collaboration abstraction. We discussed how perspectives are specified using two operations: setting the pattern matching options and querying. We explained how the engineer gives semantics to the notion of collaboration by setting the pattern matching criteria and described a simple query model to support the recovery of important collaborations.

We presented the Collaboration Browser tool which supports our approach and demonstrated through case studies how it is used to characterize a particular collaboration, to decompose a program trace into collaborations, and to break down the interface of a class into several roles. We reviewed lessons learned from our use of the tool and sketched a methodology for the extraction of collaborations.

In Section 6.7 we discussed how collaboration view recovery meets our requirements. From this discussion we saw that collaboration view recovery is better than concept view recovery at answering some of the behavioral questions we pose. In the conclusions which follow we evaluate concept view recovery and collaboration view recovery together as one contribution.

7

Conclusions

In this chapter we summarize the contributions made in the dissertation, discuss the limitations of our approach and point to directions for future work.

7.1 Contributions

In this dissertation we presented a new approach for reverse engineering behavioral design views of object-oriented software systems using dynamic information. Our approach is based on perspectives – lenses through which the engineer views the dynamic information – and their use in an iterative recovery process.

We claimed that such an approach can overcome the difficulties of recovering succinct and focused views of object-oriented software from dynamic information because it enables an engineer to declaratively specify the kind of information that he or she is interested in, instead of presenting him or her with a lot of information at fine granularity.

To validate our claim we developed techniques to support the specification of perspectives and the extraction of design views. We developed methods for expressing basically two kinds of perspectives: the first kind is the component-connector perspective which supports the recovery of concept views of the software; the second is the collaboration perspective which supports the recovery of collaboration views. We demonstrated that these two kinds of perspectives enable the extraction of a range of views which answer many behavioral questions. The views obtained by each of the techniques are complementary: concept views are useful in obtaining high-level views of the software, whereas collaboration views are helpful in understanding how classes collaborate to carry out a certain functionality and the roles they play in different collaborations.

We first identified a model for representing object-oriented programs and their execution. This model consists of a meta-model for static information and a meta-model for dynamic information, and provides a cornerstone for the definition of perspectives. The meta-model for static information is used in perspectives for creating component abstractions and collaboration abstractions. Our case studies demonstrate that even a small core of this meta-model – only classes, methods and inheritance definition entities – suffices for obtaining a repertoire of interesting perspectives. The meta-model for dynamic infor-

mation represents program execution as a sequence of message send events between class instances. This meta-model is used in perspectives for creating connector abstractions. The two meta-models are conceptually simple, and they are *lightweight*: easy to populate for any object-oriented system.

For the recovery of concept views we introduced a framework to support the definition of perspectives using a logic programming language. We described how component abstractions are defined by grouping together static entities of the software into a component, and how the semantics of a connector is defined by defining a relation in terms of message send events. We identified and encoded several useful component and connector types and showed how an engineer can define new ones. Finally, using case studies, we demonstrated the use of perspectives for the iterative recovery of concept views.

Our work on concept view recovery showed that perspectives provide for *simple view specification*, and that the logic programming framework supports *extensible view specification*. The case studies described showed that the views obtained are *succinct*, and that they enable us to answer many *behavioral questions* at both a *high-level* of component relations and a *low-level* of object interactions. The case studies also demonstrated an iterative recovery process *guided by the developer*.

For the recovery of collaboration views we introduced pattern matching as a technique to identify similar execution sequences in the execution trace in order to abstract from the interaction of objects to class collaborations. Here the notion of perspective translates to focusing on interesting information through pattern matching and querying. We described how setting the pattern matching criteria determines the semantics of a collaboration, and showed through our case studies how querying is used to find and characterize important collaborations.

Our work on collaboration view recovery showed that perspectives provide for *simple view specification* through these two operations. Though we do not have a visual notation for a collaboration view, as we do for a concept view, the case studies described showed that the views obtained can be expressed *succinctly* as class-collaboration matrices which show the roles that classes play in different collaborations. These enable us to answer specific *behavioral questions* relating to the collaborations of instances at runtime. Characterizing a collaboration provides us with a focused *low-level* view of the software, whereas decomposing the execution trace into a set of collaborations provides us with a *high-level* view. The case studies also demonstrated an iterative recovery process *guided by the developer*.

In summary, we have provided evidence that an approach based on perspectives and their use in an iterative recovery process meets the requirements for the recovery of behavioral views which we outlined in Section 2.3.

Our overall contribution is the development of techniques to support the iterative recovery of behavioral views using perspectives and the demonstration of the feasibility and usefulness of such an approach. An approach based on perspectives can overcome some of the difficulties associated with current visualization approaches, in providing focused and succinct views earlier in the process of design recovery. Our work also makes some contributions in a larger research context:

Logic programming is a powerful paradigm for representing, describing and reasoning about design knowledge. It has been used as a meta-programming language for building software development support tools which use static information to check and enforce design constraints [Wuy01][Men00]. Our contribution here is to demonstrate that logic meta-programming can be used for reasoning about dynamic information as well, and can form a basis for design recovery tools. We use logic meta-programming as a vehicle for expressing perspectives – but our work shows that it can also be used to express invariants over dynamic information. Conformance checking using dynamic information is thus an area for future work.

De Hondt [Hon98] demonstrated the use of ad-hoc grouping of software entities, *software classification*, in organizing software in a flexible manner in a development environment and in recovering views from static information. Our work on perspectives also demonstrates the utility of such groupings, as in the mapping of software elements to components, *virtual classifications* as suggested in the Classification Browser approach [Hon98], in recovering software design.

Our work provides another example of the usefulness of partial design recovery [Mur96] based on incomplete information. Dynamic information is incomplete because it is based on a specific usage scenario, and so does not exercise all execution paths in the system. The approach is partial in that we focus on only a small part of the system, by instrumenting selectively and by defining perspectives to specify the kind of information we are interested in.

7.2 Discussion

In this section we address some of the usability considerations of our approach.

The recovery process. The advantages of an approach which hands over so much responsibility to the developer is that it lets the developer decide what kind of information he or she wants to see. But ‘deciding what I want to see’ might not be easy – so the burden of interpreting a view and formulating the next perspective might also overwhelm a developer. To overcome this problem it is important to elaborate guidelines on the use of such an approach; a first step was made in this dissertation in abstracting from the case studies to common patterns of usage of our prototype tools. In concept view recovery, for example, we first exploit the static organization of the software to obtain initial views, where simple invocation and creation connectors are central in conveying an overview. For collaboration view recovery we provided some guidelines for setting the pattern matching criteria and described a general querying process in which we start by looking at the invocation relationships between the classes we are interested in and then go on to identify and query collaboration patterns in which these classes participate. More work is required here to assess which perspectives are useful in which context, and to associate guidelines to different kinds of maintenance questions.

Visualization. In this research we wanted to avoid the use of visualization techniques. We were interested to see how far we could go in design recovery without relying on a

visualization form specific to object-oriented systems, such as sequence diagrams. But when trying to understand the low-level interaction of objects we decided to integrate a sequence diagram tool to view instances of collaboration patterns, and we sometimes missed not having an overview of the whole trace. So the absence of such a possibility might limit the usefulness of our approach for answering certain questions.

There is clearly a tradeoff to be made between extracting compact abstractions of behavior and seeing all the low-level interactions in the trace. Our approach can provide high-level views and still allow a developer access to fine-grained information about collaborations. But no single tool or approach will satisfy all the requirements for design recovery. Rather, guidance is needed as to which tools and approaches are best for which maintenance tasks.

In the spectrum of tools which deal with dynamic information our approach sits somewhere between the two extremes of *macroscopic* approaches which summarize program behavior through metrics and *microscopic* approaches [PLVW98] which retain and display each message send event. It provides an example of the feasibility and utility of using dynamic information to extract high-level abstractions and relationships without visualization techniques. As such, our approach is related to frequency spectrum analysis, coverage concept analysis [Bal99], dynamic differencing [RBDL97] and dynamic discovery of program invariants [ECGN99]. Design recovery approaches for object-oriented software might benefit from looking at these different kinds of dynamic analyses.

Scalability. We have argued that our approach can scale to handle large software systems by discussing the mechanisms it provides for dealing with big systems. Our basic assumption is that a developer will never need to understand everything about a system, and will not use this approach on a system when he or she has absolutely no idea where to begin an investigation. Some initial information about a software system can be obtained from looking at the static structure of the code, by browsing or using a tool. Case studies conducted for the Reflexion Model technique [Mur96] showed that developers were always able to come up with some initial hypothesis about the structure of the system they wanted to investigate.

Starting out with these assumptions about the use of the approach, the key in handling a large software system is to focus the investigation. This is done first, by instrumenting only those parts of the software that we want to understand, and second, by choosing the size of the view that will be presented. For concept view recovery this is done by creating component abstractions which cluster together many software elements. For collaboration view recovery this is done by choosing the size of collaboration pattern to investigate. Here we assume that in seeking to understand collaborations of classes in a software system we look for interactions where only a small subset of the classes participate. We have not, however, exercised our tool prototypes on a very large software system, so it remains to be shown that such an approach can indeed scale up as well in terms of efficiency and in terms of the design recovery process itself.

7.3 Future Work

We have already pointed to some general directions for future work in the two previous sections. Here we consider future work on the extension or use of our tools for concept view recovery and collaboration view recovery.

Our work demonstrated the feasibility of using an iterative approach based on perspectives for the recovery of compact and focused views from dynamic information. Much work remains to be done, however, in making these techniques usable in a broader context. We consider here some issues that should be addressed:

A vision for a tool. Though we have developed two separate tools for concept view recovery and collaboration view recovery, we see the two techniques as two complementary parts of the same approach, and as such it is clear that they could be integrated into one tool.

The usability of a tool is determined in a large part by the user interface it presents. Our goal, however, has been to demonstrate the kind of functionality that a design recovery tool should have rather than to investigate user interface issues. We envision a tool which integrates the techniques we propose with a more interactive interface. For example, elements of a view as obtained in concept view recovery (directed graph with nodes and edges) respond to context-sensitive clicking. Nodes and edges can both be queried about the elements they contain and collaboration queries, as in collaboration view recovery, can be launched on edges or on nodes.

Other usability issues such as the question of defining a domain specific query language and optimizing the time and space performance of queries must also be addressed. In order to better address scalability in concept view recovery, a domain specification element could be added to allow the developer more control over the elements which appear in a view.

Handling views. In concept view recovery a view is currently treated as purely visual. A view is, however, a directed graph and as such graph analysis algorithms could be applied to it to deduce properties. In order to analyze recovered views for the detection of architectural patterns, *e.g.*, layering, the semantics of such views must be taken into account. So far, we have left it up to the developer to interpret the extracted views in terms of the semantics he or she assigned to the component and connector abstractions. More work is therefore needed to evaluate how the graphs generated can be used for detecting architectural invariants.

Using the tools. The case studies presented in this dissertation demonstrate a certain pattern of usage. However, further experiments and case studies are needed to assess which views are most useful and what kind of extensibility is most desired. This would lead to enlarging the core of predefined perspectives.

One interesting use of our approach is that of tracking the evolution of a software architecture. Some initial experiments [Ric99] show how extracted views can be compared to locate design changes. We are also planning to use collaboration view recovery for the extraction of roles as an aid to refactoring classes into mixins.

Bibliography

[Ach02] Franz Achermann. *Forms, Agents and Channels - Defining Composition Abstraction with Style*. PhD thesis, University of Berne, January 2002.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[And97] Egil P. Andersen. *Conceptual Modeling of Objects: a Role Modelling Approach*. PhD thesis, University of Oslo, November 1997.

[ANS83] ANSI/IEEE Standard 729-1983, New York. *IEEE Standard Glossary of Software Engineering Terminology*, 1983.

[Bal99] Thomas Ball. The concept of dynamic analysis. In *Proceedings of ESEC/FSE'99*, number 1687 in LNCS, pages 216–234, 1999.

[BC89] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings OOPSLA '89*, volume 24 of *ACM SIGPLAN Notices*, pages 1–6, 1989.

[BE96] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, pages 33–43, 1996.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[BFJR98] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the Rescue. In *Proceedings ECOOP'98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[Big89] T.J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, pages 36–49, October 1989.

[BJ94] Kent Beck and Ralph Johnson. Patterns generate architectures. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP'94*, volume 821 of *LNCS*, pages 139–149, Bologna, Italy, July 1994. Springer-Verlag.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, 1996.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[Bro96] Kyle Brown. *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk*. Masters thesis, North Carolina State University, 1996.

[BRSW00] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. Role object. In Niel Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Language of Program Design 4*, pages 15–32. Addison Wesley, 2000.

[CC90] Elliot J. Chikofsky and James H. Cross, II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[CCdCL98] Gerardo Canfora, Aniello Cimitile, Ugo de Carlini, and Andrea De Lucia. An extensible system for source code analysis. *Transactions on Software Engineering*, 24(9):721–740, September 1998.

[Ciu99] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of TOOLS 30 (USA)*, pages 18–32, 1999.

[CM93] M. Consens and A. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2*, pages 511–516, 1993.

[CMR92] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, 1992.

[Cop92] James O. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison Wesley, 1992.

[DBSB91] P. Devanbu, R. Brachman, P. Selfridge, and B. Ballard. Lassie: A knowledge-based software information system. *CACM*, 34(5):34–49, May 1991.

[DD99] Serge Demeyer and Stéphane Ducasse. Metrics, do they really help? In Jacques Malenfant, editor, *Proceedings LMO'99 (Languages et Modèles à Objets)*, pages 69–82. HERMES Science Publications, Paris, 1999.

[DDL99] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In Francoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*. IEEE, October 1999.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. to appear, spring 2002.

[DDT99] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In Bernhard Rumpe, editor, *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, volume 1723 of *LNCS*, Kaiserslautern, Germany, October 1999. Springer-Verlag.

[DL01] Stéphane Ducasse and Michele Lanza. Towards a methodology for the understanding of object-oriented systems. *Technique et science informatiques*, 20(4):539–566, 2001.

[DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[DR97] Stéphane Ducasse and Tamar Richner. Executable connectors: Towards reusable design elements. In *Proceedings of ESEC/FSE'97, LNCS*, volume 1301, pages 483–500, 1997.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 - the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of ICSE'99*, May 1999.

[EK99] Alexander Egyed and Phillipe B. Kruchten. Rose/architect: a tool to visualize architecture. In *Proc. 32nd Annual Hawaii Conference on Systems Sciences*, 1999.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[Fow97] Martin Fowler. *UML Distilled*. Addison Wesley, 1997.

[FP96] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.

[FTAM96] Roberto Fiutem, Paolo Tonella, Giuliano Antoniol, and Ettore Merlo. A cliché-based environment to support architectural reverse engineering. In *Proceedings ICSM '96*. IEEE, November 1996.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[GR95] Adele Goldberg and Kenneth S. Rubin. *Succeeding With Objects: Decision Frameworks for Project Management*. Addison Wesley, Reading, Mass., 1995.

[Gra92] J.E. Grass. Object-oriented design archeology with CIA++. *Computing Systems*, 5(1):5–67, 1992.

[GSR96] Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings OOPSLA/ECOOP'90*, volume 25, pages 169–180, October 1990.

[Hil99] Rich Hilliard. Using the UML for architectural description. In Robert France Bernard Rumpe, editor, *Proceedings 2nd International UML Conference, UML '99*, volume 1723 of *LNCS*, pages 32 – 48. Springer-Verlag, October 1999.

[Hon98] Koen De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. Ph.D. thesis, Vrije Universiteit Brussel,Departement of Computer Science, Brussels - Belgium, December 1998.

[HYR96] D.R. Harris, A.S. Yeh, and H.B. Reubenstein. Extracting architectural features from source code. *Automated Software Engineering*, 3(1-2):109–139, 1996.

[IEE99] IEEE Architecture Working Group. *IEEE P1471/D5.0 Information Technology - Draft Recommended Practice for Architecural Description*, August 1999.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 63–76, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[JR97] D. Jerding and Spencer Rugaber. Using Visualization for Architectural Localization and Extraction. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 56 – 65. IEEE Computer Society, 1997.

[JSB97] Dean J. Jerding, John T. Stansko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of ICSE'97*, pages 360–370, 1997.

[KC98] Rick Kazman and S. Jeromy Carriere. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse*, Victoria, B.C., 1998.

[Ken99] Elizabeth Kendall. Role model design and implementations with Aspect-Oriented programming. In *Proceedings of OOPSLA'99*, ACM Sigplan Notices, pages 353–369, November 1999.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[KM96] Kai Koskimies and H. Mössenböck. Scene: Using scenario diagrams and active test for illustrating object-oriented programs. In *Proceedings of ICSE-18*, pages 366–375. IEEE, March 1996.

[KN] Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot*. AT & T Bell Laboratories, Murray Hill, NJ.

[KP88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.

[KP96] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedingsof WCRE '96*. IEEE, November 1996.

[Kru95] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995.

[KSRP99] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse engineering of design components. In *Proceedings of ICSE'99*, May 1999.

[KSTM98] Kai Koskimies, Tarja Systä, Jyrki Tuomi, and Tatu Männistoö. Automated support for modeling oo software. *IEEE Software*, 15(1):87–94, January/February 1998.

[Lan99] Michele Lanza. Combining metrics and graphs for object oriented reverse engineering. Diploma thesis, University of Bern, October 1999.

[LB85] M. M. Lehman and L. Belady. *Program Evolution - Processes of Software Change*. London Academic Press, 1985.

[LD02] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002*, pages 135–149, 2002.

[LH96] L. Larsen and M.J. Harrold. Slicing object-oriented software. In *Proceedings ICSE '96*, pages 495–505. IEEE, 1996.

[LHS97] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *Proceedings OOPSLA '97*, ACM SIGPLAN, pages 304–317, October 1997.

[LN95a] Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA'95*, pages 342–357. ACM Press, 1995.

[LN95b] D.B. Lange and Y. Nakamura. Program explorer: A program visualizer for C++. In *Proceedings of Usenix Conference on Object-Oriented Technologies*, pages 39–54, 1995.

[LOML01] Karl Lieberherr, Johan Ovlinger, Mira Mezini, and David Lorenz. Modular programming with aspectual collaborations. Technical Report NU-CCS-2001-04, College of Computer Science, Northeastern University, Boston, MA, March 2001.

[Luc97] Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Programming Technology Lab, Vrije Universiteit Brussel, Brussels, Belgium, 1997.

[Mar98] Radu Marinescu. Using object-oriented metrics for automatic design flaws in large scale systems. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, volume 1543 of *LNCS*, pages 252–253. Springer-Verlag, 1998.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeffrey Kramer. Specifying distributed software architectures. In *Proceedings ESEC '95*, volume 989 of *LNCS*, pages 137–153. Springer-Verlag, September 1995.

[Men00] Kim Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 2000.

[ML98] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98 ACM SIGPLAN Notices*, pages 97–116, October 1998.

[MN97] Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 8:29–36, 1997.

[MORT96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of the Fourth ACM SIGSOFT FSE Symposium (FSE4)*, pages 24–32, San Francisco, CA, October 1996.

[MT97] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of ESEC/FSE'97*, pages 60–76, Zürich, Switzerland, September 1997.

[Mur96] Gail C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.

[MWD99] Kim Mens, Roel Wuyts, and Theo D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999.

[MWT95] Hausi A. Müller, Kenny Wong, and Scott R. Tilley. Understanding software systems using reverse engineering technology. In V.S. Alagar and R. Missaoui, editors, *Object-Oriented Technology for Database and Software Systems*, pages 240–252. World Scientific, 1995.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[OQC97] Georg Odenthal and Klaus Quibeldey-Cirkel. Using patterns for design and documentation. In *Proceedings of ECOOP'97*, volume 1241 of *LNCS*, pages 511–529. Springer-Verlag, June 1997.

[PHKV93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 326–337, October 1993.

[PLVW98] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[Pro95] Programming Systems Group, Swedish Institute of Computer Science, Sweden. *SICStus Prolog User's Manual*, 1995.

[Rat98] Rational Software Corporation. *Rational Rose 98: Roundtrip Engineering with C++*, 1998.

[RBDL97] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of ESEC/FSE'97, LNCS 1301*, pages 432–449, 1997.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 13–22. IEEE, September 1999.

[RD01] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. Technical Report IAM-01-007, University of Bern, Institute of Computer Science and Applied Mathematics, December 2001.

[Ree96] Trygve Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.

[RG98] Dirk Riehle and Thomas Gross. Role model based framework design and integration. In *Proceedings OOPSLA '98 ACM SIGPLAN Notices*, pages 117–133, October 1998.

[Ric99] Tamar Richner. Using recovered views to track architectural evolution. In *ECOOP'99 Workshop Reader*, number 1743 in LNCS. Springer-Verlag, June 1999.

[Rie98] Dirk Riehle. Bureaucracy. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, pages 163–185. Addison Wesley, 1998.

[Rie00] Dirk Riehle. *Framework Design: a Role Modelling Approach*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2000.

[RK99] Ferenc Dósa Rácz and Kai Koskimies. Tool-supported compression of uml class diagrams. In Bernhard Rumpe, editor, *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, LNCS 1723, pages 172–187, Kaiserslautern, Germany, October 1999. Springer-Verlag.

[SB98] Yannis Smaragdakis and Don Batory. Implementing layered design with mixin layers. In Eric Jul, editor, *Proceedings ECOOP'98*, volume 1445 of *LNCS*, pages 550–570, Brussels, Belgium, July 1998.

[Sef96] Mohlalefi Sefika. *Design Conformance Management of Software Systems: an Architecture-Oriented Approach*. PhD thesis, University of Illinois, 1996.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[SKM01] Tarja Systä, Kai Koskimies, and Hausi Müller. Shimba – an environment for reverse engineering java software systems. *Software – Practice and Experience*, 1(1), January 2001.

[SNH95] Dilip Soni, Robert L. Nord, and Christine Hofmeister. Software architecture in industrial applications. In *Proceedings ICSE '95*, pages 196–207, Seattle, April 1995. ACM Press.

[Som92] Ian Sommerville. *Software Engineering*. Addison Wesley, 1992.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1986.

[SSC96] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring complicance of a software system with its high-level design models. In *Proceedings ICSE-18*, pages 387–396, March 1996.

[SWM97] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. How do program understanding tools affect how programmers understand programs? In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 12–21. IEEE Computer Society, 1997.

[SZ99] Wilhelm Schäfer and Albert Zündorf. Round trip engineering with Design Patterns, UML, Java and C++. ESEC/FSE 99 Tutorial Notes, September 1999.

[TAFM97] Paolo Tonella, Giuliano Antoniol, Roberto Fiutem, and Ettore Merlo. Flow insensitive c++ pointers and polymorphism analysis and its application to slicing. In *Proceedings ICSE '97*. IEEE, May 1997.

[Tak96] TakeFive Software GmbH. *SNiFF+*, 1996.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001.

[vMV95] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, 1995.

[VN96] Michael VanHilst and David Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings OOPSLA'96*, pages 359–369. ACM Press, 1996.

[War00] Colin Ware. *Information Visualization*. Morgan Kaufmann, 2000.

[WBW89] Rebecca Wirfs-Brock and Brian Wilkerson. Object-oriented design: A responsibility-driven approach. In *Proceedings OOPSLA '89*, pages 71–76, October 1989. ACM SIGPLAN Notices, volume 24, number 10.

[WMFB+98] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings OOPSLA '98*, ACM SIGPLAN, pages 271–283. ACM, October 1998.

[WMH93] Norman Wilde, Paul Matthews, and Ross Hutt. Maintaining object-oriented software. *IEEE Software (Special Issue on "Making O-O Work")*, 10(1):75–80, January 1993.

[WTMS95] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, January 1995.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure object-oriented systems. In *Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.

[Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[ZSW94] Kaizhong Zhang, Dennis Shasha, and Jason L. Wang. Approximate tree matching in the presence of variable length don't cares. *Journal of Algorithms*, 16(1):33–66, January 1994.

Curriculum Vitae

Name Tamar Richner-Hanna

Birth Date October 15, 1957

Nationality Swiss,Canadian

Education

1983 M.Sc. Computer Science, University of Toronto

1981 B.Sc. Computer Science, University of Toronto

Work Experience

1994-2002 Doctoral researcher[1], Institut für Informatik und angewandte Mathematik, Universität Bern

1987-1992 Research Collaborator, École Polytechnique Fédérale de Lausanne

1986-1987 System Engineer, Services Informatiques, Université de Gènéve

1984-1986 Member of Scientific Staff, Bell Northern Research, Toronto

1983-1984 Software Developer, Daedalian Systems, Toronto

1979-1983 Teaching Assistant, Department of Computer Science, University of Toronto

[1] Eight years!? No, with two long absences, a total of five years and four months.