Using Dynamic Information for the Iterative Recovery of Collaborations and Roles

Tamar Richner and Stéphane Ducasse
Software Composition Group, Institut für Informatik (IAM)
Universität Bern, Neubrückstrasse 10, 3012 Berne, Switzerland
{richner,ducasse}@iam.unibe.ch
http://www.iam.unibe.ch/∼{richner,ducasse}

Abstract

Modeling object-oriented applications using collaborations and roles is now well accepted. Collaboration-based or role-based designs decompose an application into tasks performed by a subset of the applications' classes. Collaborations provide a larger unit of understanding and reuse than classes, and are an important aid in the maintenance and evolution of the software. This kind of design information is lost, however, at the implementation level, making it hard to maintain and evolve an existing software application. The extraction of collaborations from code is therefore an important issue in design recovery. In this paper we propose an iterative approach which uses dynamic information to support the recovery and understanding of collaborations. We describe a tool we have developed to support our approach and demonstrate its use on a case study.
Keywords: *collaboration-based design, design recovery, program understanding, object-oriented reverse engineering, dynamic analysis.*

1. Introduction

In contrast to procedural applications, where a specific functionality is often identified with a subsystem or module, the functionality in object-oriented systems comes from the cooperation of interacting objects and methods[23, 12]. In designing object-oriented applications, the importance of modeling how objects cooperate to achieve a specific task is well recognized [24, 15, 19, 3]. 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.

The recovery of collaborations from existing code is an important aid for understanding and maintaining object-oriented applications [23]. However, 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.

To get a better understanding of the dynamic interactions between instances, developers often turn to tools which display the run-time information as interaction diagrams. Designers of such tools are confronted with the challenge of dealing with a huge amount of trace information and presenting it in an understandable form to the developer. Several visualization techniques, such as information murals[9], program animation[22] and execution pattern views[14] have been proposed to reduce the amount of trace information presented and to facilitate its navigation.

In this paper we propose an approach to the recovery of collaborations which uses dynamic information, but does not rely heavily on visualization techniques. Whereas most visualization tools display an entire trace and give the user a feel for the overall behavior of an application, our approach focuses on understanding much smaller chunks of interactions and the roles that classes play in these.

We have developed a tool prototype, the *Collaboration Browser*, to demonstrate the validity of our approach. We illustrate through examples how the Collaboration Browser is used to query run-time information iteratively to answer concrete questions about collaborations and interactions in Smalltalk programs.

The paper is structured as follows: in the next section we briefly illustrate the concepts of collaboration-based design. In Section 3 we discuss the challenges of recovering such design artifacts and give an overview of our approach. In Section 4 we introduce our approach and present the tool we have developed to support the recovery process. In Section 5 we walk the reader through an example of program understanding using the tool and in Section 6 we discuss issues arising from our case studies. The implementation is presented briefly in Section 7. In Section 8 we review related work. In Section 9 we conclude with a discussion of the approach and directions for future work.

2. Collaboration-based Design

In this section we illustrate the concepts of collaboration-based design with a small example. Consider a class model which describes a bureaucracy [18].

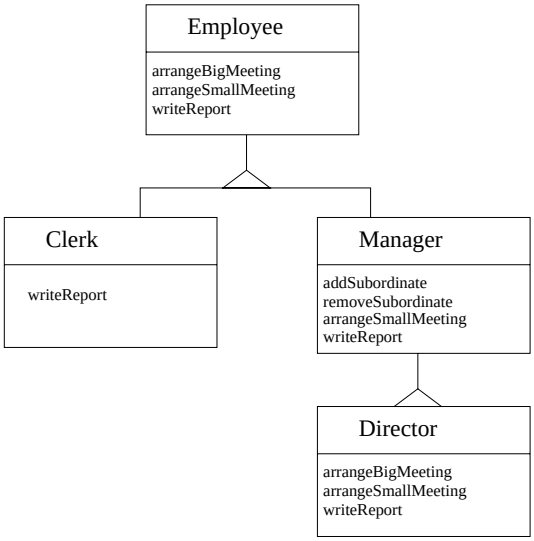

Figure 1. Class Diagram for Bureaucracy.

This is a hierarchy of Director, Managers and Clerks which operates as described by the Bureaucracy pattern [18]. In effect, four of the GOF design patterns [7] 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 can not handle forwards the request up the hierarchy, as in Chain of Responsibility: the Manager 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 2 summarizes this information in a class/collaboration matrix [21]. It illustrates that an instance of a class participates in several collaborations, playing a distinct role in each.

Here we have described the collaborations and roles in terms of the design patterns they instantiate. Roles describes the responsibilities of objects in a collaboration, but how a role is actually modeled or specified is often left open [19]. Some design techniques model roles using interfaces [15], or as part of a behavioral contract between participants

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

Figure 2. Class-collaboration matrix for Bureaucracy. Each row represents a collaboration and each cell describes the role the class plays in the collaboration.

[8]. 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.

3. Challenges to Recovering Collaborations

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, objects 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 as two basic elements: *participants* and *roles*. The role of each participant is the part of the participant which enforces the interaction protocol.

In order to reverse engineer collaborations from code we must first recover interactions of instances from the code. Which class instances 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.

3.1. Recovering Interactions

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

Recording information about message exchanges between instances as the program executes provides us with control flow information required for deriving interactions and with information about the context in which methods of specific instances are invoked. Program tracing, however,

results in a great volume of information about the interactions of objects, where much of this information is duplicated many times over in an execution trace.

The main argument against the use of dynamic information is its incomplete coverage of the code. But this very property is also its advantage [1]. In the context of reverse engineering we do not always need complete information: a program trace provides information about the behavior of the system exercising a certain functionality, and so helps us to tie functionality to behavior.

Our Approach. To obtain control flow information we use *dynamic information* recorded from program execution. 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.

3.2. 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 the *important* interactions - these will be the collaborations we are interested in.

It has been observed that without guidance from a user the process of design recovery gives poor results [13]. Our own experience with reverse engineering tools corroborates this observation. We do not believe, therefore, in the automatic extraction of collaborations and roles, but rather in an iterative process steered by the engineer. Typically, an engineer 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 we are not interested in all collaborations, nor in all the details of a collaboration, choosing the right interactions to look at and the right level of detail is important.

Our Approach. In order to allow the engineer to focus on the details relevant for his or her investigation we support an *iterative recovery* process through *querying*.

3.3. Overview of Our Approach

The approach we propose for recovering collaborations and roles is:

Based on dynamic information. To obtain control flow information our approach uses dynamic information recorded from program execution. For each method invocation event we record the sender class, sender identity, receiver class, receiver identity, and name of invoked method.

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

Supports iterative recovery through querying. We enable the developer to identify the significant collaborations by specifying what kind of information he or see is interested in. 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.

4. Supporting the Recovery of Collaborations

The Collaboration Browser is a tool we have developed which supports such an iterative recovery process. In this section we first explain some of the underlying terminology and concepts, then introduce the Collaboration Browser.

4.1. Terminology and Concepts

Our starting point for the recovery of collaborations is the execution trace - a record of all method invocation events for the instrumented classes. A short sample of such a trace is given below.

```
1 EventDispatcher,5604,DrawingController,3548,handleEvent:
2     DrawingController,3548,DrawingController,3548,currentTool
3     DrawingController,3548,Tool,14970,handleEvent:
4         Tool,14970,ToolState,13668,nextStateForEvent:tool:
```

This trace sample consists of four method invocation events. For each event we record five items of information, which we illustrate with the values for event 1: the class of the sender (EventDispatcher), the identity of the sender (5604), the class of the receiver (DrawingController), the identity of the receiver (3538) and the method invoked on the receiver (handleEvent:).

From the execution trace we aim to recover collaborations and roles approaching those used in design. We reserve the term *collaboration* and *role* to talk about the high-level design concepts introduced in Section 2. 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.

In the trace sample above there are four collaboration instances: the first one includes all four events and corresponds to the invocation of handleEvent: on DrawingController. The second one consists of event 2 only, the third of events 3 and 4, and the fourth of event 4.

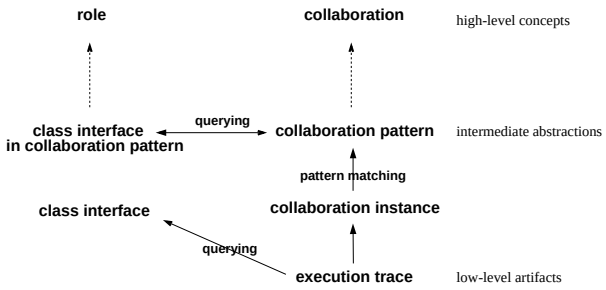

Figure 3. From an execution trace to collaborations and roles

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 3 above illustrates how pattern matching and querying support 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. Below we explain the pattern matching settings and the querying facility available to an engineer using this approach.

4.2. Pattern Matching Settings

In an execution trace there are many collaboration instances which are variations on the same prototype (design) collaboration. We use pattern matching to group collaboration instances into *collaboration patterns*. The settings for the pattern matching criteria specify what it means for two collaboration instances to be considered equivalent - they reflect what the engineer considers important about a collaboration.

The pattern matching settings can be modulated along three independent axes:

Information about an event. An event in the trace contains basically three items of information: the sender, the receiver and the invoked method. For each of these three items we can include or omit information in the matching scheme. For example, senders and receivers can be ignored, or matched on the identity of the object or the class. The invoked method can be ignored, or matched on method name or method category name.

Events to exclude. The matching scheme allow us to ignore certain events in the trace: we can ignore events in which an object sends a message to itself, events whose depth of invocation in the trace is above a given limit, or events whose depth of invocation in the collaboration pattern is above a given limit.

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 relationships between method invocations. In this scheme collaboration instances are treated as identical if they have the same method invocation events in their set.

4.3 The Query Model

Once pattern matching has been performed with the settings the engineer has specified, the dynamic information is presented to the engineer in terms of classes, methods and collaboration patterns. A developer focuses on the relevant collaboration patterns by querying this 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,Receive,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.

4.4. 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 4. 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

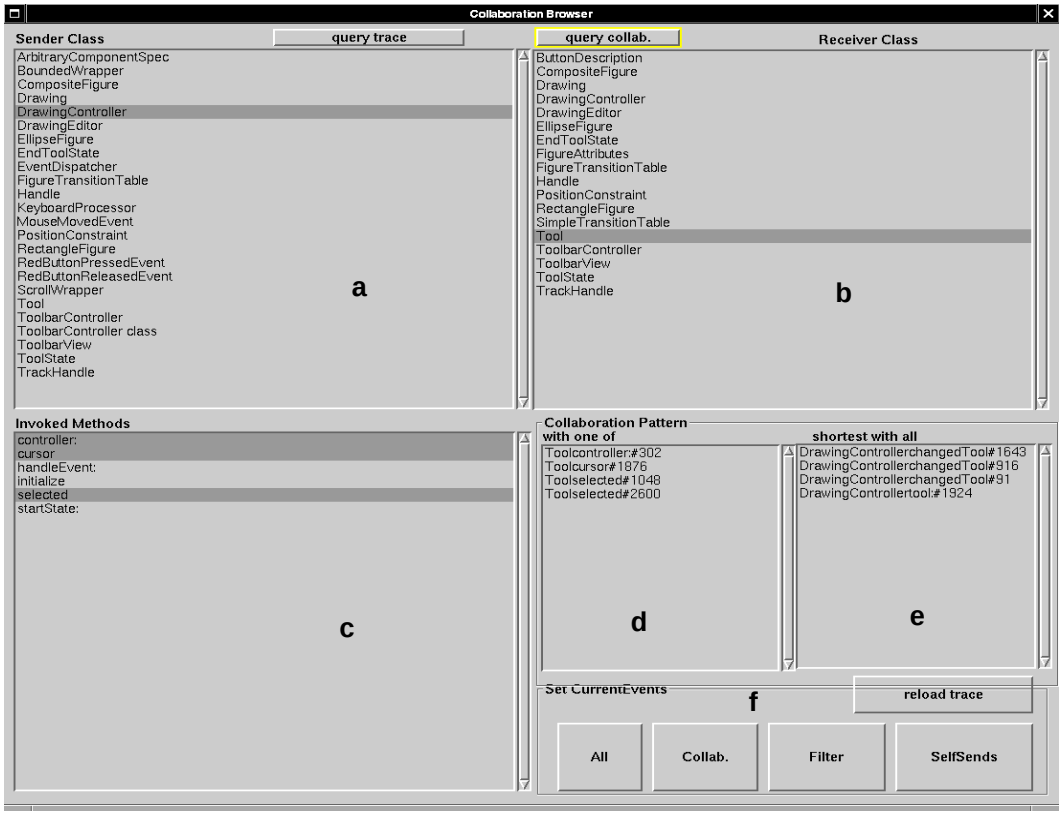

Figure 4. Collaboration Browser window. Panels a b and c list the sender classes, the receiver classes and the invoked methods respectively. Panels d and e both list collaboration patterns. Panel f provides functionality for filtering out information.

two collaboration pattern lists is explained further below, as is the function of the button panel f.

The Collaboration Browser supports the two key operations for the recovery of collaborations: pattern matching and querying. Querying is done through the browser window, whereas the pattern matching criteria are currently set by hand. In addition, the Collaboration Browser enables the developer to filter out dynamic information, and to display interaction diagrams.

In this section 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 5.

Querying 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 4, 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.

Querying 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 relationships of sender classes, receiver classes, invoked methods and collaboration patterns.

Example: in Figure 4 three methods of class Tool have been selected in panel c: controller:, cursor and selected. Panel d lists 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* (in terms of number of method invocation events) 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, we can ask which collaboration patterns include particular receivers and invoked methods. Second, selecting a collaboration pattern either from panel d or from panel e, we can ask about the senders, receivers and invoked methods in the pattern, again using the 'query collab.' button. If senders, receivers or invoked methods are also specified, the missing (unselected) elements will be returned as a response to the query. Three queries are here of particular interest:

collaboration pattern for given participants: Selecting a list of participant classes, we ask in which collaboration patterns instances of these classes occur together.

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.

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

Filtering out dynamic information. To focus the investigation on the events of interest the developer can filter out method invocation events for selected senders, receivers and methods, or focus on an instance of a selected collaboration pattern. This is done using the buttons in panel f.

Displaying an instance of a collaboration pattern. The interaction diagram window displays an instance of the selected collaboration pattern as a sequence diagram.

Example: In Figure 4 the collaboration pattern called DrawingcontrollerchangedTool#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 5.

5. Understanding Tools in HotDraw

In this section we demonstrate how our approach supports the understanding and recovery of collaborations by applying it on the HotDraw framework [2][4].

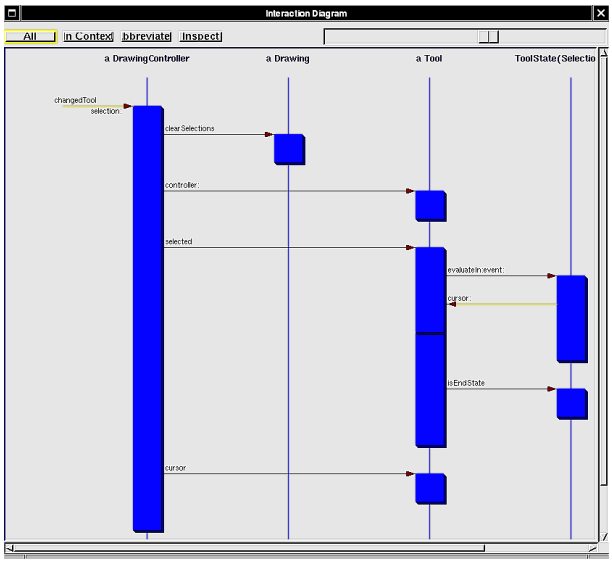

Figure 5. Interaction Diagram window. The interaction diagram corresponds to an instance of the collaboration pattern at the top of panel e in Figure 4.

HotDraw is a framework for semantic graphic editors which allows for the creation of graphical editors which associate the picture with a data structure. The HotDraw framework comes with several sample editors. From the documentation we learn that HotDraw is based on the Model-View-Controller triad: 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.

Formulating questions. From browsing the code we see that the documentation available describes an earlier version of HotDraw. 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. On the drawing editor tools are represented by icons on the top panel (see Figure 6). In a previous version tool responsibilities were handled by the classes Reader, Command and Tool, whereas in the current version different tools are implemented through states.

In order to understand how tools are implemented in this version of HotDraw we formulate several questions:

- with which classes does the class Tool collaborate?
- what role does the class Tool have in different collaborations?

- how are user events handled (e.g. selecting a tool and pressing a mouse button) ?

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.

Pattern Matching. We create collaboration patterns by running the pattern matching with the following options: (i) information about an event: sender: none, receiver: name of class defining method, method: method name, (ii) events to exclude: depth of invocation: 20, relative depth of invocation: 3, self-sends: to be ignored, (iii) structure of the collaboration instance: a set of events.

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

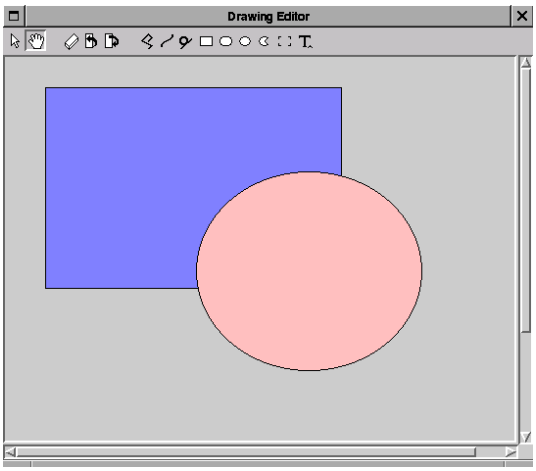

Figure 6. HotDraw sample editor

Looking at the collaborations of Tool**.** We look for collaboration patterns in which instances of Tool participate. As discussed in Section 4, each method invocation recorded in the trace is a collaboration instance, and each collaboration instance is mapped to a collaboration pattern. Thus many collaboration patterns correspond to a trivial interaction of just one method invocation. In general, then, to arrive at more interesting collaboration patterns, we identify patterns 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 query to obtain the shortest collaboration patterns in which both Tool and this class participate. All the collaboration patterns obtained in this way are summarized in Table 1. The name of each collaboration pattern corresponds to the event information specified in the matching scheme - in this case the name of the invoked method and the name of the class which implements the method. We have listed each collaboration pattern only once, though in some cases there are actually several collaboration patterns, variations on the execution of the method in question.

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 1. Collaborations involving Tool

Some of the collaboration patterns listed in the table are nested in each other. We query each collaboration pattern about its participants (senders, receivers and methods), and so deduce the nesting relationship between DrawingController handleEvent: and some of the other collaboration patterns.

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

Investigating Tool handleEvent:**.** From the nesting relationship illustrated above, we understand that when a tool handles an event it first asks the current state of the tool, ToolState, for the next state to go to (depending on the event). It then asks the next state to take over by invoking evaluateIn:Event:. It is this state object which does the rest of the work of handling the event.

We would like to understand which classes participate in the collaboration Tool handleEvent: and what role they play, and also to understand the different variations of this collaboration. We therefore query to obtain collaboration patterns resulting from the invocation of handleEvent: on Tool. The result of this query are four collaboration patterns. The differences between these is illustrated in Table 2: since the pattern matching criteria matched only to 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 are due to different methods executed. The table lists the four variations, each one in a separate row. For

	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 2. Different collaboration patterns for Tool handleEvent:**.**

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 3. 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.

each variation, the name of the class which implements the executed method is listed under the column of the method.

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 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 3, where the variable collaborations have been denoted by bold faced method names.

6. Discussion and Evaluation

The case study presented shows how querying with the Collaboration Browser is used to investigate interactions in HotDraw, and to recover some important collaborations. We continued in the vein of the investigation described above to discover the role of Tool in other collaborations as well. Each collaboration recovered represents an important task in which Tool interacts with other classes. By characterizing the collaboration and their variations we gain a better understanding of how the functionality of the software is carried out through the interaction of instances.

Experience with the Collaboration Browser. We also used the Collaboration Browser to decompose an execution trace into a class/collaboration matrix and to understand the function of a class by partitioning its interface into several roles and identifying the collaborations in which it plays these roles [16]. The case studies demonstrated 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. It is also a challenge to find the right 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.

The iterative process. 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.

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.

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.

Finally, we have argued in the introduction that although dynamic information is valid only for the particular scenario executed, it provides focus in the investigation: it acts as a program slice with respect to control flow and is always precise with respect to the executed scenario.

7. Implementation

The Collaboration Browser is implemented in Smalltalk and currently handles single-threaded Smalltalk applications. We instrument the application to be investigated using Method Wrappers[5]. This allows selective instrumentation at the method level. The visualization of collaboration instances as sequence diagrams is based on the Interaction Diagram tool[5]. Pattern matching is implemented using hashing.

8. Related Work

Our work on recovering collaborations in intended as a part of a query-based approach for iterative understanding of object-oriented applications. The recovery of collaborations provides us with low-level views of a software application, and as such is most useful when integrated in an approach which can also provide us with high-level views showing the interaction of components or subsystems [17][16].

Most of the work on understanding interactions in object-oriented applications has focused on visualization, where the challenge is to develop techniques for visualizing the large amount of information generated by program tracing [10, 11, 22, 20]. For a more thorough survey of these and other reverse engineering approaches we refer the reader to [16]. Here we compare our work with two visualization approaches which use pattern matching [14][9] to identify design abstractions.

The work of DePauw et al. [14], now integrated in Jinsight [1], experiments with a range of displays which allow an engineer to visually recognize patterns in the interactions of classes and objects. ISVis [9] is a visualization tool which displays interaction diagrams using a mural technique and also offers pattern matching capabilities. Our work is similar to these two approaches, both of which identify recurring patterns in a trace as an aid to recognizing important design concepts. In contrast to these, however, our work is not oriented primarily towards program visualization. We use only a simple sequence diagram visualization to display the collaboration pattern chosen. Our main focus is on querying the dynamic information to help in the recovery of collaborations and the understanding of the roles different classes play in these. We see our work as complementary to the visualizations proposed in [9] and [14]: whereas 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, our approach focuses on the roles of classes in much smaller chunks of interaction.

We know of only one other approach which explicitly tries to reverse engineer collaborations[6]. The approach uses static information to arrive at a description of participant-roles in a collaboration and relies heavily on the input of a user who must select the initial participants and their roles in the collaboration and determine appropriate acquaintances to include in the collaboration.

[1] http://www.alphaworks.ibm.com/tech/jinsight

9. Conclusions

In this paper we have presented an approach to the recovery of collaborations which is general enough to be applied to software applications implemented in any class-based object-oriented language. The approach begins with an execution trace and condenses this information by representing program behavior in terms of collaboration patterns. It presents this information to developers in terms of sender classes, receiver classes, invoked methods and collaboration patterns and allows developers to query each of these items in terms of the others. In this way it lets a developer focus on the aspect of the application of interest without wading through a lot of trace information.

We have shown through an example how the Collaboration Browser is used to discover important collaborations in an application and to understand the roles that classes play in these collaborations. Our initial experience with the Collaboration Browser on three case studies showed that the approach is promising, but it also demonstrated the limits of automatic recovery of design artifacts. To be successful the use of the tool must be embedded in an iterative recovery process steered by a particular question or hypothesis.

Our approach demonstrates the feasibility and utility of using dynamic information to extract collaboration abstractions without reliance on visualization techniques. There are tradeoffs to be made between our approach for extracting compact representations of collaborations and approaches which use visualization techniques to display interaction patterns over the low-level interactions in the whole trace. We therefore consider our approach as complementary to other reverse engineering techniques: no single tool can satisfy all the requirements for design recovery, rather guidance is needed as to which tools are best for which maintenance tasks.

Acknowledgments. Thanks to Matthias Rieger for his help and for his comments on the manuscript. We also thank Roel Wuyts for his helpful comments.

References

[1] T. Ball. The concept of dynamic analysis. In *Proceedings of ESEC/FSE'99*, number 1687 in LNCS, pages 216–234, 1999.

[2] K. Beck and R. Johnson. Patterns generate architectures. In *Proceedings ECOOP'94*, LNCS 821, pages 139–149. Springer-Verlag, July 1994.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[4] J. Brant. Hotdraw. Master's thesis, University of Illinois at Urbana-Chanpaign, 1995.

[5] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP'98*, LNCS 1445, pages 396–417. Springer-Verlag, 1998.

[6] K. DeHondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Vrije Universiteit Brussel, 1998.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[8] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings OOPSLA/ECOOP'90*, volume 25, pages 169–180, Oct. 1990.

[9] D. Jerding and S. Rugaber. Using Visualization for Architectural Localization and Extraction. In *Proceedings WCRE*, pages 56 – 65. IEEE, 1997.

[10] K. Koskimies and H. Mössenböck. Automatic synthesis of state machines from trace diagrams. *Software Practice and Experience*, 24(7):643–658, July 1994.

[11] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA'95*, pages 342–357. ACM Press, 1995.

[12] S. Lauesen. Real life object-oriented systems. *IEEE Software*, pages 76–83, March 1998.

[13] G. C. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 8:29–36, 1997.

[14] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[15] T. Reenskaug. *Working with Objects: The OORAM Software Engineering Method*. Manning, 1996.

[16] T. Richner. *Recovering Behavioral Design Views: a Query-Based Approach*. PhD thesis, University of Berne, May 2002.

[17] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 13–22. IEEE, Sept. 1999.

[18] D. Riehle. Bureaucracy. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 163–185. Addison Wesley, 1998.

[19] D. Riehle and T. Gross. Role model based framework design and integration. In *Proceedings OOPSLA '98 ACM SIGPLAN Notices*, pages 117–133, Oct. 1998.

[20] T. Systä, K. Koskimies, and H. Müller. Shimba – an environment for reverse engineering java software systems. *Software – Practice and Experience*, 1(1), January 2001.

[21] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings OOPSLA'96*, pages 359–369. ACM Press, 1996.

[22] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proc. OOPSLA'98*, pages 271–283, 1998.

[23] N. Wilde, P. Matthews, and R. Hutt. Maintaining object-oriented software. *IEEE Software (Special Issue on "Making O-O Work")*, 10(1):75–80, Jan. 1993.

[24] R. Wirfs-Brock and B. Wilkerson. Object-oriented design: A responsibility-driven approach. In *Proceedings OOPSLA '89*, pages 71–76, Oct. 1989. ACM SIGPLAN Notices, volume 24, number 10.