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 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. The extraction of collaborations 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 present the problems of extracting such information and describe a tool we have developed to validate our approach.

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[22, 12]. In designing object-oriented applications, the importance of modeling how objects cooperate to achieve a specific task is well recognized [23, 1, 8, 15, 18, 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 the code is an important aid for understanding and maintaining object-oriented applications [22]. 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. This difficulty is further aggravated in the case of dynamically typed languages like Smalltalk where no type definition is available at compile time and where methods are never statically bound.

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[21] 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 interactions and 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 first illustrate the concepts of collaboration-based design then discuss the obstacles to recovering collaborations from code. In Section 3 we introduce our approach and present the tool we have developed to support the recovery process. Section 4 walks the reader through an example of program understanding using the tool. Implementation issues are discussed next in Section 5. Section 6 reviews related work and Section 7 concludes with a discussion and evaluation of the approach and directions for future work.

2. Recovering Collaborations

In this section we illustrate the concepts of collaboration-based design, discuss the challenges of recovering such design artifacts and give an overview our approach.

2.1. Collaboration-based Design

Below we present a small example to illustrate the concepts of collaboration- based design. Consider a class model which describes a bureaucracy [17].

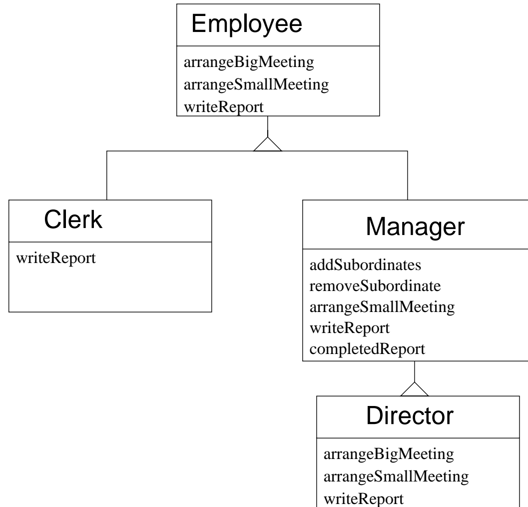

Figure 1. Class Diagram for Bureaucracy.

This is a hierarchy of Director, Managers and Clerks which operates as described by the Bureaucracy pattern [17]. In effect, four of the GOF design patterns 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 [20]. 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 [18]. Some design techniques model roles using interfaces [15], or as part of a behavioral contract between participants [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.

	Clerk	Manager	Director
Chain of Responsibility	Predecessor	Predescessor	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.

In contrast to the design idea of separating concerns as distinct collaborations, at run-time things look much more complex. Several design collaborations may be interleaved and instances of the same class often play different roles in one call sequence. This makes it difficult to disentangle concerns and to reconstruct collaborations as they were conceived at the design stage.

2.2. Problems with Recovering Collaborations

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 2, where each role describes the behavior of instances of this class in a specific context - in a collaboration. Below we discuss the challenges that must be tackled.

Static Information is not enough. As argued in the introduction, 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 start-

ing 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 study at applications whose design may not follow accepted design guidelines.

Dynamic Information is too much. Recording 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 of specific instances are invoked. Program tracing, however, results in a great volume of low-level information from which we must sift the details relevant for our investigation [14, 9, 21]. Here, the problems of focus, granularity and scalability must be addressed.

To be able to focus, it is necessary to break the program trace up into chunks representing collaborations. Also, we are not interested in all collaborations, nor in all the details of a collaboration, so choosing the right collaborations to look at and the right level of detail is important.

2.3. Overview of Our approach

The approach we propose for recovering collaborations and roles is:

Based on dynamic information. To obtain exact 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 addresses the problem of scalability since it does not confront a developer with the whole trace of execution information, but rather with *collaboration patterns*.

Supports iterative recovery through querying. To address the problem of focus and granularity we present a tool which lets the developer query the dynamic information in terms of classes and interactions of interest. The querying allows a developer to refine the investigation to focus on collaborations of interest.

3. Supporting Iterative Recovery

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.

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

3.1. Terminology and Concepts

As discussed in Section 2, how collaborations and roles are represented at the design stage and how a collaboration-based or role-based design is carried on to the implementation is left open most of the time. We reserve the term collaboration and role to talk about the high-level design concepts. Our starting point is, however, an existing execution trace from which we want to recover collaborations and roles approaching those used in design. In this context we talk about *collaboration instances* and *collaboration patterns*.

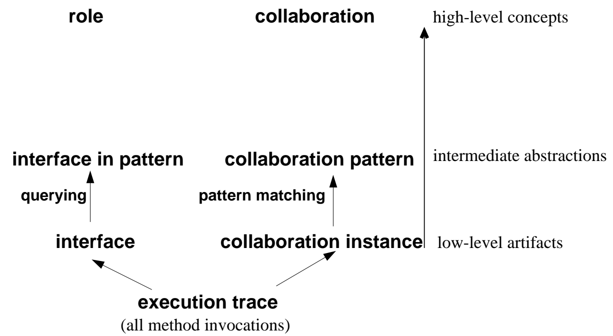

Figure 3. Relationships of terms.

Figure 3 above illustrates schematically the relationship of the terms we use.

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).

There are as many collaboration instances as method invocations in the trace. But 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*.

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

The pattern matching criteria is flexible and can be modulated along three independent axes:

Method invocation information. Each of the items: object identity (for sender and receiver), class identity (for sender and receiver), and name of invoked method can be either taken into account or ignored in the matching scheme. The matching scheme also allows the developer to make use of other static information related to these items.

Depth of invocation. The depth of the method invocation is specified so that method invocations at a greater depth are ignored in the matching criteria.

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

In Section 5 we discuss how this pattern matching is implemented.

3.2. 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 two collaboration pattern lists is explained further below, as is the function of the two button panels f and g.

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 4.

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. In the presentation below we have numbered the queries (Q1,Q2,Q3,Q4 and Q5), editing (E1, E2, E3, E4) and displaying functions (D1, D2, D3) in order to refer to them later on in Section 4.

Query about the interface of a class. These queries are of the form

Q1: *what methods of class A are invoked by class B?* Selecting sender and receiver classes, the user requests a list of the methods displayed in panel c.

Example: in Figure 4 a sender class, DrawingController and a receiver class, Tool, have been chosen. Panel c lists the methods of class Tool which are invoked by an instance of DrawingController.

Q2: *which classes are senders and receivers of the following methods?*

Conversely, selecting a set of methods in panel c, the developer can request the list of senders and receivers to be updated to the senders and receivers of the selected methods.

Query about a collaboration. By selecting sender and receiver classes, and methods of interest (panels a, b and c respectively) the following queries can be answered:

Q3: *what collaboration patterns result from this method invocation?* Panel d lists the collaboration patterns resulting from an invocation of one of the selected methods on an instance of one the selected receivers by an instance of one of the selected senders.

Q4: *what are the shortest collaboration patterns in which all the following methods are invoked?* Panel e lists the shortest collaboration patterns in which *all* the methods selected come into play, for the selected receiver and sender.

Example: in Figure 4 three methods have been selected in panel c: controller:, cursor and selected. Panel e lists the collaboration patterns in which all these three methods of class Tool come into play. The list shows five collaboration patterns, each with the name DrawingController-changedTool, but each with a different identity number (hidden in the screen shot). These are five different patterns which result from the invocation of changedTool on an instance of DrawingController.

Query about a role. These queries are of the form:

Q5: *given a collaboration pattern and a set of methods what are all the senders and receivers that participate in this collaboration pattern?* By selecting a set of methods (panel c) and a collaboration pattern in which this set of methods comes into play (panel e) the developer can request a list of receivers which play this role in the

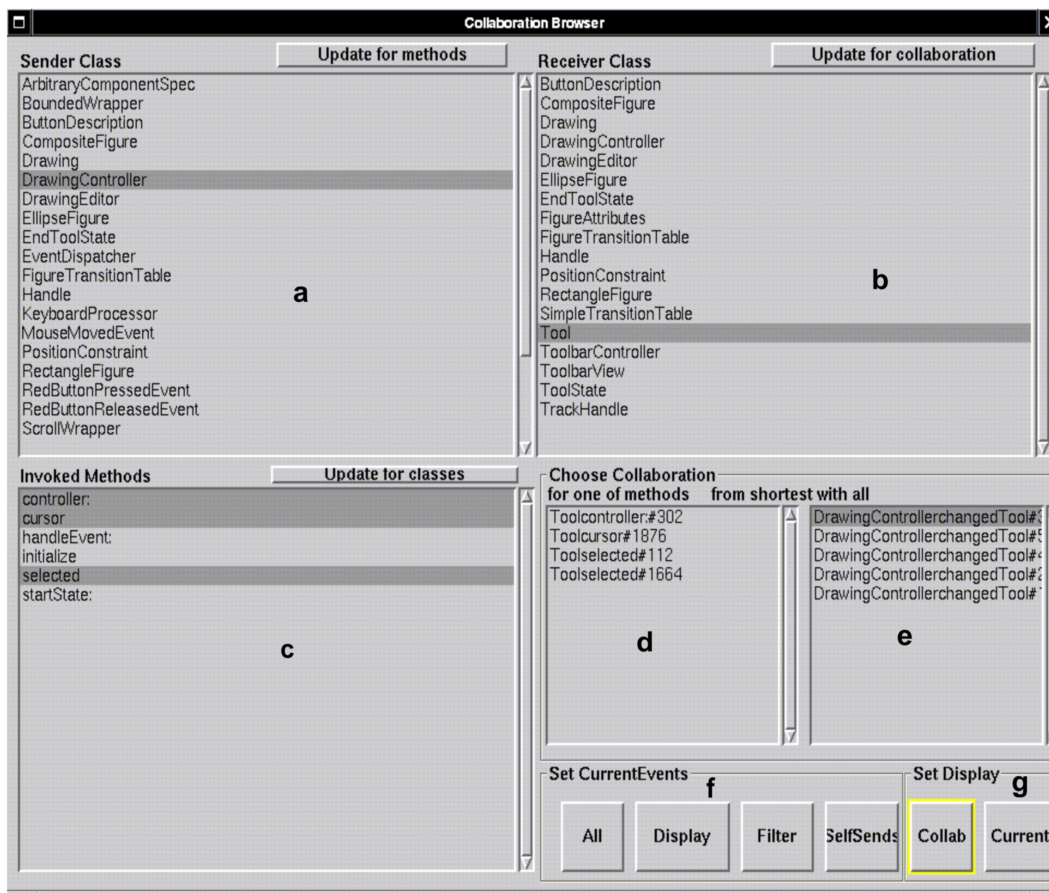

Figure 4. Collaboration Browser window. Panels a b and c list the sender classes, the receiver classes and the invoked methods respectively. Panels d and e both list collaboration patterns.

same collaboration pattern, and the corresponding senders.

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 can be set to display (using the buttons in panel g) either an instance of a selected collaboration pattern, or the whole of the current trace. The call tree of a collaboration pattern can be displayed (using buttons on the interaction diagram window, see Figure 5) 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 4 a collaboration pattern called DrawingcontrollerchangedTool#3426 has been selected. Figure 5 below shows an instance of this collaboration pattern displayed as an interaction diagram. It shows how four objects, instances of DrawingController, Drawing,Tool and ToolState, interact when the method changedTool is invoked on a DrawingController.

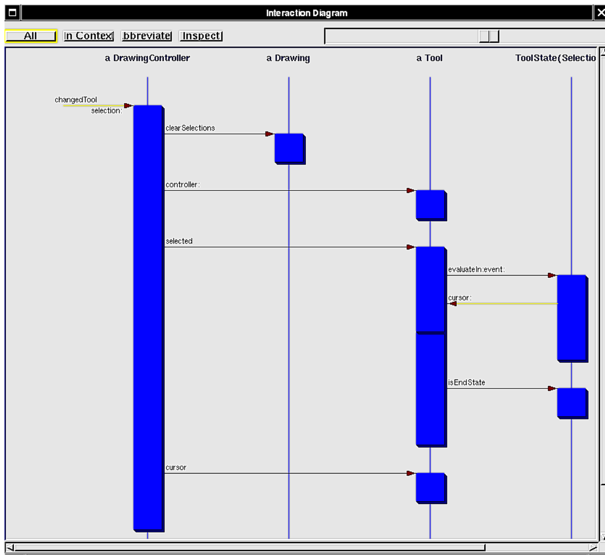

Figure 5. Interaction Diagram window. The interaction diagram corresponds to the collaboration pattern selected in Figure 4.

4. Investigating Collaborations 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].

4.1. HotDraw

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 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: 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:

- how are user events handled (e.g. selecting a tool and pressing a mouse button) ?
- with which classes does the class tool collaborate?

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. This generates 53,735 method invocation events.

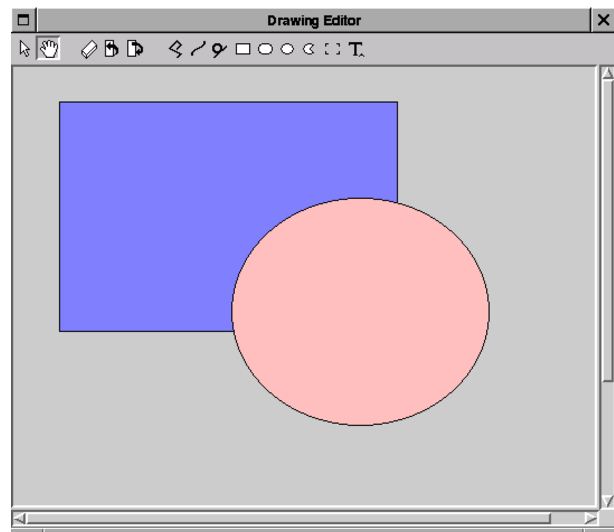

Figure 6. HotDraw sample editor

4.2. Using the Collaboration Browser for Understanding Tools in HotDraw.

In this section 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 3.2.

Querying about interfaces. We start by querying about the interface that class Tool presents to other classes in Hot-

Draw. The results of these Q1 queries are given in the Table 1 below.

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 1. Tool **interface matrix.** It shows the methods of class Tool which are invoked by other HotDraw classes.

From Table 1 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 Q3 queries. In the case of figureAtEvent: we see that the collaboration patterns for this method occur in two different contexts, as illustrated in Figure 7 and Figure 8 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.

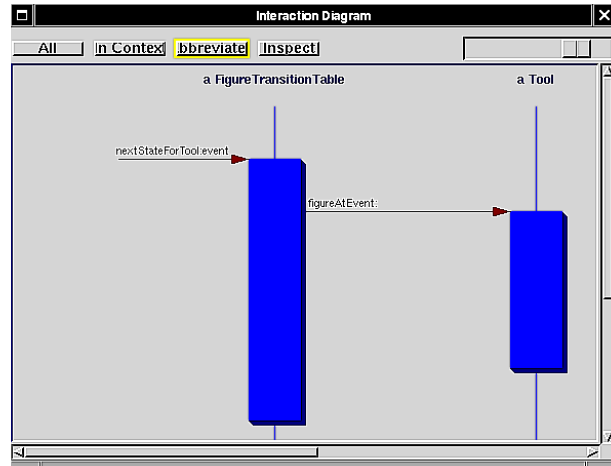

Figure 7. First context. The context in which a FigureTarnsitionTable invokes figureAtEvent: on a Tool

We repeat a new sequence of Q3 queries and D3 displays to compare the different contexts for the initialize method. When DrawingController is created it initializes a Tool, 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 (Q3 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 the class Tool**.** From browsing the code and using the Collaboration Browser we understand that tools are implemented using state diagrams. We first look more systematically at the collaboration patterns in which Tool methods are invoked, so as to choose the collaborations which are likely to be the key to understanding how tools handle user events.

As discussed in Section 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 interesting collaboration patterns, we identify patterns in which a subset of the methods of a class are involved.

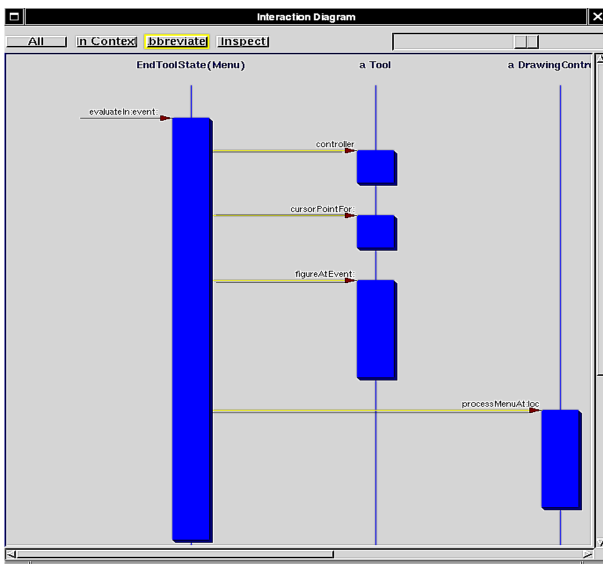

Figure 8. Second context. The context in which an EndToolState invokes figureAtEvent: on a Tool

Tool Interface	Collaboration Pattern Name
controller: cursor **handleEvent:** initialize selected startState	Tool handleEvent:
controller: **cursor** handleEvent: initialize **selected** startState	DrawingController changedTool
controller: cursor handleEvent: **initialize** selected **startState**	DrawingController initialize

Table 2. Collaborations involving the DrawingController-Tool interface. For each group of shaded method names, it gives the collaboration pattern in which all these methods are invoked.

Using the Collaboration Browser we group, for example, methods in the interface Tool presents to DrawingController and look at the shortest collaboration patterns in which these groups of methods occur. We do this by interactively selecting several methods and posing queries of the form Q4: “what are the shortest collaboration patterns in which an instance of DrawingController invokes the following methods on an instance of class Tool?” These queries and the corresponding response are shown schematically in Table 2 below. The collaboration pattern is named after the method of a class which is the root of call tree.

Remark: How this is done using the Collaboration Browser is shown in Figure 4. This screen window shows the sender and receiver classes and the methods that have been selected. Panel e lists the shortest collaborations in which these methods are involved. Notice that there are several collaboration patterns with the same name (but different sequence number, hidden on screenshot): these are separate collaboration patterns which result from the invocation of the method changedTool on an instance of DrawingController. The difference between these collaboration patterns are due to variations in the participants of the collaborations; these differences may or may not be important for the understanding of an application - this depends on the questions which must be answered. The accommodations of variations on a pattern is adjusted to some degree using different pattern matching modes (presented in Section 3.1).

Investigating a particular collaboration. We choose to concentrate on the collaboration patterns Tool handleEvent:, to learn about how tools handle user events. Though there are many collaboration patterns resulting from the invocation of this method on an instance of Tool, their abbreviated form is similar (D3 display). Looking more closely at an instance of one of these patterns (D1 display) we see that DrawingController invokes handleEvent on Tool. Tool then invokes nextStateForEvent:tool: on an instance of ToolState and this object then consults SimpleTransitionTable to obtain the next state nextStateForTool:event:. It then asks the next state to take over by invoking evaluateIn:event:. It is this state object which handles the rest by, in this case, creating a rectangle and adding it to the drawing. By comparing a few of these collaboration patterns (D1 displays) we see that the many variations are due to variations in the call trees resulting from the invocation of evaluateIn:event: on a ToolState object. The variations in the call trees which result from this invocation are due to the different kinds of user event (tool chosen and mouse button activity).

Characterizing a collaboration. By loading an instance of this pattern as the current base of dynamic information (E3 function) and querying about interfaces (Q1 queries) we can arrive at a characterization of the handleEvent collaboration which describes the predictable participants, as seen in Table 3. We have left a wild card cell (rightmost cell) in this table to represent the participants of the evalu-

Tool	ToolState	EndToolState	TransitionTable	*
handleEvent: cursorPointFor: drawing valueAt: valueAtPut:	evaluateIn:event: isEndState nextStateForEvent:tool:	evaluateIn:event: isEndState	nextStateForTool:event:	*depend on event* evaluateIn: event:

Table 3. Class-Collaboration description for collaboration Tool handleEvent: Gives the role of each class in the collaboration. The wildcard cell represents variations on the collaboration.

ateIn:event: collaboration. In Table 3 we represent the role of the classes in the collaboration simply by the interface they present in the collaboration.

Recovering other collaborations. We continue in the vein of the investigation described above to discover other collaboration patterns in which instances of Tool participate. In this way, we extracted eight main collaborations in which Tool participates from this HotDraw scenario. Using the Collaboration Browser we identified the participants in these collaborations and the methods which are invoked for those classes. Each collaboration recovered represents an important task in which Tool interacts with other classes, and is represented as a row in the form of Table 3. As in Table 3 some rows have an empty wild card cell representing a variation on the collaboration which is not critical for understanding the its basic structure. These variations can be specified in greater detail, i.e. which are the candidate participants. In this way we arrive at a class-collaboration matrix as discussed in Section 2.1: and shown in Figure 2.

4.3. Discussion and Evaluation

We have shown how the Collaboration Browser is used to investigate interactions in HotDraw, and to recover some important collaborations.

The iterative process. Extrapolating from this example we sketch the process in which we use the Collaboration Browser. Below we describe the steps we take in recovering collaborations. The iteration in the process comes at step 5, where we then repeat steps 1-5 with more focus. We also iterate by formulating and launching new queries as we learn more about the application.

1. *Querying about interfaces:* we start by querying about interfaces (Q1) to understand which classes communicate.

2. *Looking for a role:* we break up the interface of a class into groups of methods which could represent roles. This was done in the HotDraw example first as in Table 1: breaking up the interface by sender class using Q1 queries.

3. *Looking for a collaboration for a role:* we then continue to form groups of methods to discover which collaboration patterns they occur in (Q4).

4. *Understanding a collaboration:* the context display (D3) and the abbreviated display (D2) aid us in comparing collaboration patterns which are similar, whereas the full display (D1) directs us further in the inquiry by giving more details on a collaboration pattern.

5. *Deeper understanding of a collaboration:* once we have found a collaboration pattern that we want to understand in more depth, we load an instance of this pattern as the current base of dynamic information (E3). The process can then begin again at step 1., this time working with a smaller base of dynamic information.

Our HotDraw example did not demonstrate the filtering functions (E1, E2) nor the query about roles (Q5). In our investigation we found filtering out self-sends useful as it eliminates some 'noise' from the interaction of instances. The filtering of senders, receivers and methods is useful when we already know what we can ignore. More useful is the E3 function of loading an instance of a collaboration pattern as the current trace.

Experience with the Collaboration Browser. Identifying the eight major collaborations in which Tool takes part required about thirty interactive queries. We have also used the Collaboration Browser on our own MOOSE tool[7]. The number of queries required to narrow down the information of interest depends very much on the familiarity we already have with the application. The kind of scenario that we exercise on the system is also important for the kind of information base we have to analyze.

We also experimented with different pattern matching modes. The mode which treats a collaboration instance as a sets of events rather than as a tree of events results as expected in more matches, without any 'false' matches. For the method invocation information, using the method name and name of class defining the method as labels is in most

cases sufficient for identifying matches. Over-restricted matching results in too many collaboration patterns.

Characterizing roles. In our characterization of collaborations we represent the role of a class in a collaboration be 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.

5. Implementation

The Collaboration Browser is implemented in Smalltalk and currently handles single-thread 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. As discussed in Section 3.1, we treat the structure of a collaboration instance in two ways: as a tree of method invocations, or as a set of method invocations. In the call tree of events of a collaboration instance, each node corresponds to a method invocation and contains five items of information: sender class, sender instance, receiver class, receiver instance and invoked method selector. This information is used to compute 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. Each of the five items of information can be taken into account, or ignored, in computing the hash value. In the case that the collaboration instances are treated as sets of events each is assigned a hash value as a function of the members of the set.

6. Related Work

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.

There are several tools which display interaction diagrams from program executions [5, 10]. Program Explorer [11] offers class and object based displays of dynamic information. Sefika et al.[19] can display the dynamic interactions of architectural units such as subsystems, but their approach requires an instrumentation specific to the application. Walker et al.[21] 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.

The work of DePauw et al. [14] 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 most similar to the work described in [9] and [14], 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, our tool 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 and 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. The classification browser approach relies heavily 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.

Finally, this work is related to reverse engineering and design recovery techniques in general. Our work on recovering collaborations in intended as a part of an environment for iterative understanding of object-oriented applications. For a more extensive survey of related reverse engineering approaches, the reader is referred to [16].

7. Discussion and Conclusions

The approach we have presented in this paper 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 be 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 two 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. We plan to conduct a more extensive case study in order to make a more thorough evaluation of the approach and to refine the iterative process described in the paper.

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. Since these are at the design level 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.

Acknowledgments. Thanks to Matthias Rieger for his help and for his comments on the manuscript. We also thank Oscar Nierstrasz and Franz Achermann for their helpful comments.

References

[1] K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings OOPSLA '89*, volume 24 of *ACM SIGPLAN Notices*, pages 1–6, 1989.

[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] S. Ducasse, M. Lanza, and S. 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.

[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] C. Laffra and A. Malhotra. Hotwire – A visual debugger for C++. In *Proceedings of USENIX C++ Technical Conference*, pages 109–122, 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 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.

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

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

[19] M. Sefika, A. Sane, and R. H. Campbell. Monitoring complicance of a software system with its high-level design models. In *Proceedings ICSE-18*, pages 387–396, Mar. 1996.

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

[21] 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.

[22] 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.

[23] 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.