Interactive Exploration of UML Sequence Diagrams

Richard Sharp
Ohio State University

Atanas Rountev
Ohio State University

Abstract

Sequence diagrams are commonly used to represent object interactions in software systems. Reverse-engineered sequence diagrams, which are constructed from existing code, are becoming widely available to more programmers through modern commercial and research UML tools. However, due to their large size and inefficient spatial design, such diagrams can easily become useless.

We discuss the visual limitations of UML sequence diagrams and present a set of techniques for overcoming these limitations. These techniques allow a programmer to explore interactively various aspects of large real-world sequence diagrams in order to gain insights about the behavior of the underlying software. We have implemented a prototype tool based on these techniques, and we have used it to enhance our comprehension of sequence diagrams that were constructed from code in the standard Java libraries. This paper discusses some insights from our experience, and their implications for the builders of visualization tools.

1 Introduction

The Unified Modeling Language (UML) has become a *de facto* standard for representing different aspects of software structure and behavior. Sequence diagrams are key UML artifacts for modeling the behavior of software [10]. The diagrams encode the flow of control during object interactions, for the purposes of software design, documentation, comprehension, and validation. Figure 1 shows a simple diagram representing part of the behavior of method `max` from class `java.math.BigDecimal` from the standard Java libraries.

In modern iterative development, which interleaves design, implementation, and testing within the same project iteration, it is often necessary to perform *reverse engineering* of the diagram from the existing code. A typical scenario is to perform design recovery through reverse engineering of class diagrams and sequence diagrams in the beginning of the current iteration, based on the last iteration's code. As pointed out in one popular book on modern software development [8], reverse-engineering of sequence diagrams is an important item on the "wish-list" for UML tools.

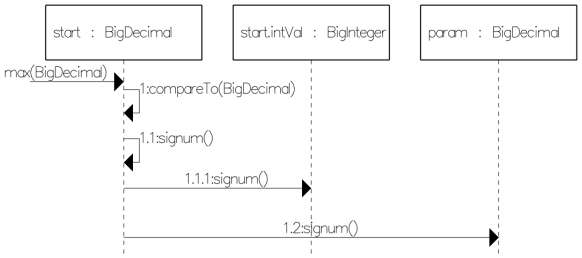


Figure 1. Diagram derived from method `max` in class `java.math.BigDecimal`.

Reverse-engineered sequence diagrams also provide essential insights for *software understanding and maintenance* of large-scale software systems. The comprehension of such systems is often challenging; e.g., the original system designers and developers may have moved on to other projects, and the design documentation may be incomplete, outdated, or non-existent. Reverse-engineered sequence diagrams can be beneficial for design recovery activities. Such diagrams are also important for *software testing*, because they provide guidelines and coverage criteria for writing high-quality test cases [1, 13].

The problem of building sequence diagrams or similar representations through program analysis has been investigated by various authors [2, 3, 6, 9, 11, 15, 16, 12, 14] and there exist commercial tools that provide such functionality (e.g., Together ControlCenter by Borland and EclipseUML by Omondo). The work presented in this paper was performed in the context of the RED tool for reverse engineering of sequence diagrams [12, 14].[1] The goal of this project is to provide high-quality tool support for reverse engineering of UML 2.0 sequence diagrams from Java code. One of the central questions for this and similar tools is the following: *How should the diagrams be presented to the tool user?* Experiments with RED indicated that, in many cases, displaying a reverse-engineered diagram is not enough: some of the diagrams produced from real-world

[1] `http://presto.cse.ohio-state.edu/red`

Java code are just too large and complicated. Existing work on reverse-engineering analyses offers little in the way of addressing this issue: the problem of effective visualization of complex reverse-engineered sequence diagrams is still a challenging one. This paper describes our initial efforts to solve this problem. The presented work is a contribution towards realizing effective visualization and exploration of sequence diagrams in modern UML tools.

The problem of effective visualization of reverse-engineered sequence diagrams has important implications for real-world software development. In the last decade, UML has become *lingua franca* for the software engineering community. Sequence diagrams are central artifacts of UML (together with class diagrams), and they are especially valuable for modern object-oriented software. Reverse-engineered sequence diagrams are becoming available to more programmers, through commercial and research software tools. However, without effective visualization of such diagrams, they could become essentially useless. The work presented in this paper is a step towards solving this problem. The specific contributions of our work are as follows:

- We identify and describe the visual deficiencies of reverse-engineered UML sequence diagrams.
- We propose a set of techniques for addressing these deficiencies. At the core of our approach is the observation that a programmer must be able to *interactively* explore different aspects of the diagram, in order to focus her attention on subsets of the expressed behavior.
- We describe our initial experience with the tool. These preliminary observations clearly show that the builders of visualization tools for reverse-engineered sequence diagrams face significant challenges. Future work that addresses these challenges is essential for building useful real-world visualization tools for these diagrams.

2 Visualization Challenges

Reverse-engineered sequence diagrams can quickly become large and cluttered, making them unreadable and essentially useless for system comprehension. In the worst case every new message appended to a diagram can add one more object thus increasing the physical size of the diagram by one "element" of width and height. This potential quadratic growth quickly makes a diagram unusable for a number of reasons, the least of which is the inability to print it on paper at any readable size.

To illustrate this issue, consider the diagram in Figure 2. This diagram was created using RED, based on method `getInstance` from class `java.text.Collator` from the standard Java libraries. The diagram contains 23 objects and 131 messages. This is not an unusually large

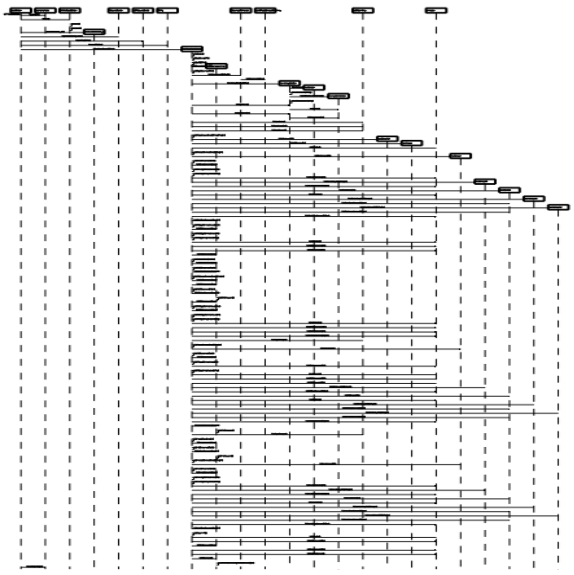


Figure 2. An unreadable sequence diagram.

diagram; in fact, the analysis inside RED restricted the size of the diagram by limiting the depth of the call stack to 5, and by considering only calls to a small subset of the library classes. Increasing the call stack depth or the set of classes would cause the number of messages and objects to grow even more. The diagram produced by RED has been resized to fit in Figure 2. Obviously the names of the objects and messages are a blur, but even at a standard 1280×1024 screen resolution the diagram is still unreadable.

Clearly, one major drawback of reverse-engineered sequence diagrams is their potentially large size which, in the worst case, makes them impossible to show to a human reader. However, there are other more subtle visual deficiencies that even a small diagram can exhibit.

When exploring a diagram, a programmer is often interested in a particular message—that is, the message name, the source object, and the destination object. However, zooming on a large diagram usually cannot relay this information. Consider for example Figure 3, where zooming on the middle of the image only shows the message names and not the source or destination objects. To get this information the user would have to translate right and left to find where the message intersects with the object lifelines, then translate up the lines to find the objects. This task quickly gets tedious and frustrating.

Even if the diagram is small enough to be read on one page or screen, the layout can mislead the reader. Specifically, the horizontal position of the source and destination objects for a given message is irrelevant, as is the length of the message line connecting them. However, the human visual system will perceive larger distances or longer lines to mean that the event described by that line is somehow

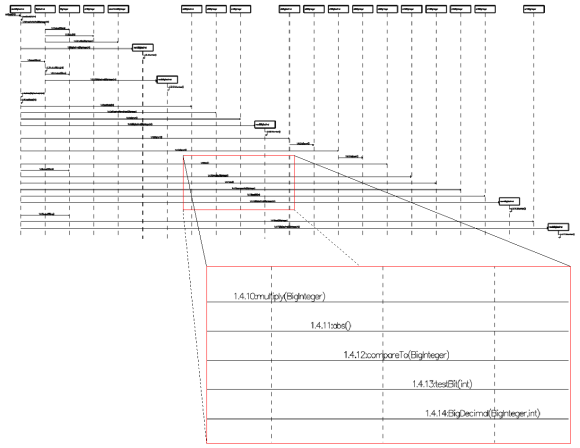


Figure 3. Zooming on a message offers little information.

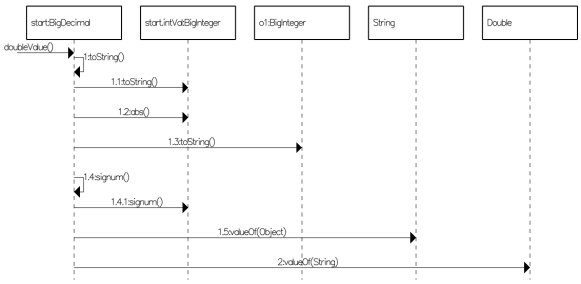


Figure 4. The length of the message implies greater importance or further "distance."

"larger" or "farther away" or perhaps more important. In fact, this instinct built into our perception system is often maliciously used by graphic designers to influence the way people interpret data, and is so common that it is given the term "lie factor" as described in the classic work by Tufte [17]. The last message of Figure 4 illustrates this issue.

A message represents the top of the run-time call stack at the particular moment of time when the corresponding method is called. A programmer often needs to reason about the state of the entire call stack, in order to answer calling-context questions such as, "How did the flow of control get here?". For example, for a message numbered `1.3.2.5`, the call stack consists of the messages labeled `1.3.2`, `1.3`, and `1`. The sequence of these messages is "buried" in the diagram, and the messages are spatially disjoint. As a result, a programmer has to trace backwards the call stack by performing multiple translations left and up. This could easily become frustrating and time-consuming in a large diagram.

Version 2.0 of UML defines *interaction fragments* as diagram to represent control-flow aspects of object interactions [10]. For example, an *opt* fragment (see Figure 7a) encloses an optional part of the diagram. A *loop* fragment represents repeated behavior, and an *alt* fragment represents a set of mutually-exclusive alternatives. Interaction fragments present additional challenges for diagram visualization. The presence of many fragments makes a diagram difficult to comprehend. For example, fragment box boundaries can easily be confused with message lines. Furthermore, if a fragment is large, zooming in on that section of the diagram will not look any different than if it were outside the fragment. Finally, if there is any sort of complicated flow of control, the resulting diagram could become visually cluttered.

3 Interactive Diagram Exploration

Section 2 discussed the visual characteristics of sequence diagrams which make their comprehension hard (or even impossible) for software engineers. This section describes several techniques that allow filtering and exploration of the diagrams in a useful manner.

3.1 Overview and Zooming

A complex diagram visualized in its entirety (i.e., at an *overview scale*) can provide some useful information to the user. For example, few (if any) details of the interaction shown in Figure 2 can be determined at the scale shown; however, there is still information to be gleaned from such a scale. For example, objects that are created during the interaction time interval are shown at the level corresponding to their creation time, as opposed to pre-existing objects which are shown at the top of the diagram [10]. At an overview scale, it can be easily seen that the execution of the methods from Figure 2 creates several new objects. Furthermore, one of these newly-created objects is the initiator of a large number of calls (its lifeline is the source of many messages). In general, a programmer familiarizing himself with a system can quickly start to associate different high-level aspects of the code behavior with the overall "shape" of the diagram shown at a low zoom level. Although we provide the ability to zoom and translate, a diagram does not necessarily convey more information simply by zooming in on parts. Without the techniques described below, higher zoom levels often cull information and produce a useless image.

3.2 Filters

All of the filters described below allow the user to either "gray out" filtered messages and objects, or to actually cull them from the diagram thus creating a simpler structure. Both techniques are useful in determining different

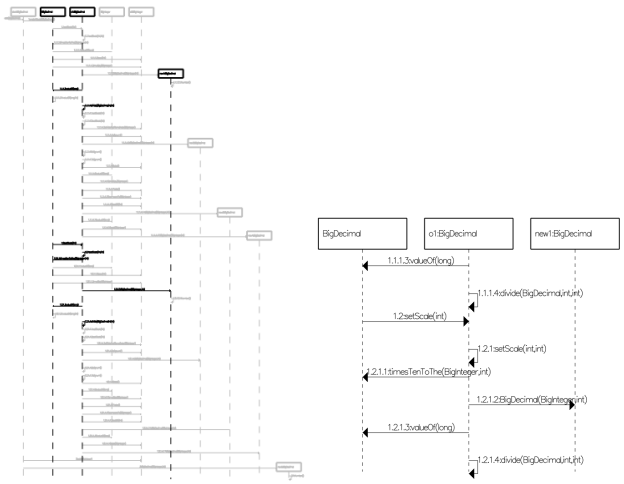


Figure 5. Graying out (left) and culling (right).

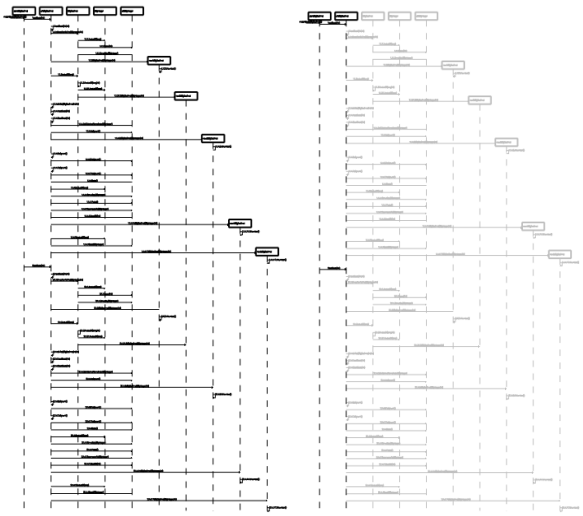


Figure 6. Graying out calls at depth ≥ 2.

types of information. Graying out allows the user to observe diagram-level properties, such as regions of complexity, while culling allows the user to examine a smaller, more tractable "slice" of the diagram in detail. Figure 5 shows an example of graying vs. culling. The diagram has been filtered by a range of messages and by call stack depth, as described below.

Temporal Filtering. Since the number of messages in a diagram could be large, a programmer would often restrict her attention to understanding a specific time interval inside the diagram, and ignore the messages happening outside of that interval. Thus, the user should be able to choose a contiguous section of the diagram; we provide this functionality by enabling the user to change the "start" and "end" message of interest. As shown in Figure 5, the user can then choose to either gray out the messages (and the all objects they are related to, except for objects that are also senders/receivers of unfiltered messages), or remove the messages and objects completely.

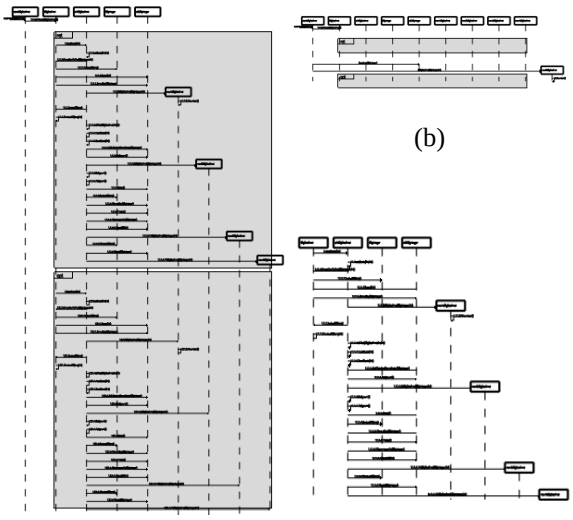
Filtering Based on Call Stack Depth. Each message in the diagram exists at a particular *call stack depth*. The greater the depth of a message, the "further away" it is conceptually from the starting method of the diagram. We allow the tool user to interactively define (and change quickly) a threshold value for message depth. All messages whose depth exceeds this value are filtered out. Through this mechanism the programmer can increase or decrease the overall complexity of the diagram. In "gray out" mode the user can see an overall view of which sections of the diagram are at which levels, while in "cull" mode the diagram's complexity will be reduced, thus allowing the details to be seen easily. Figure 6 shows a diagram which has two calls at depth 1. Each call itself triggers a complex sequence of other calls. This behavior is not immediately obvious at overview scale until the user grays out all calls of depth greater than one.

The level of adjustable visual complexity based on depth corresponds naturally to the conceptual complexity of the filtered messages. Messages at deeper levels are typically harder for programmers to reason about, because they are executed in the calling context created by the earlier messages on the call stack. If a programmer wanted to understand the top-level behavior, he would restrict the call depth to 1 (as illustrated in Figure 6). After obtaining satisfactory understanding at this level, he would "dive in" the deeper levels by changing the depth threshold value.

Filtering of Interaction Fragments. The introduction of interaction fragments to UML (e.g., opt, alt, and loop fragments, as described in Section 2) presents additional visualization challenges. Fragments are often disliked by programmers and UML experts (e.g., [4]) due to the visual clutter they can add to the diagrams. This is unfortunate since they could be otherwise useful for expressing the variations in the flow of control among interacting objects.

We have investigated several techniques to address the visual deficiencies inherent in interaction fragments. First, as defined in the UML 2.0 standard [10], we show fragments as boxes with a label in the upper left corner indicating the type of fragment (opt, loop, etc.). Rather than simply drawing a box outline as usually done for UML sequence diagrams, the boxes are shaded gray. This makes the fragments stand out in a low zoom overview level, where normally the boxes would not be seen clearly or at all.

It is possible that as a user explores the diagram, she may find the messages in fragments distracting even after temporal and call depth filtering. Consider for example a user who is examining a section of the diagram which contains a loop fragment. If the programmer is familiar with this particular



(a) (b) (c)

Figure 7. (a) Interaction fragments stand out in an overview level zoom (b) Collapsed fragments (c) Focusing on the first fragment.

loop, she may want to avoid seeing the messages inside the fragment. Alternatively, it is possible that the programmer wants to focus on the high-level flow of control, and to ignore the details about the behavior inside the fragment. In both cases the use of a different background color is helpful, but if the loop is sufficiently complex, the messages inside the fragment will still dominate the diagram. Another example of a plausible scenario is when the user is interested *only* in the fragment itself: for example, after understanding the higher-level flow of control which creates the execution context for this fragment.

To address these issues, we allow the user to collapse a fragment, leaving only a placeholder in its position. Furthermore, with a click the user can zoom on the fragment (filtering everything outside the fragment) to examine its contents in detail. Figure 7a shows an example of an overview with fragment boxes shaded in gray. The user may collapse down the boxes to reduce the complexity of the diagram (Figure 7b), or could click on a fragment to zoom on only the messages contained in that fragment (Figure 7c).

3.3 Details on Demand

As discussed earlier, zooming on a diagram does not necessarily reveal any more detail to the user. For example, if a particular message of interest needs to be explored, several key questions are usually asked: What is the source object? What is the destination object? What is the state of the call stack that led to this message? To answer these questions, a tool user would be forced to waste time by tediously walking along message and object lines.

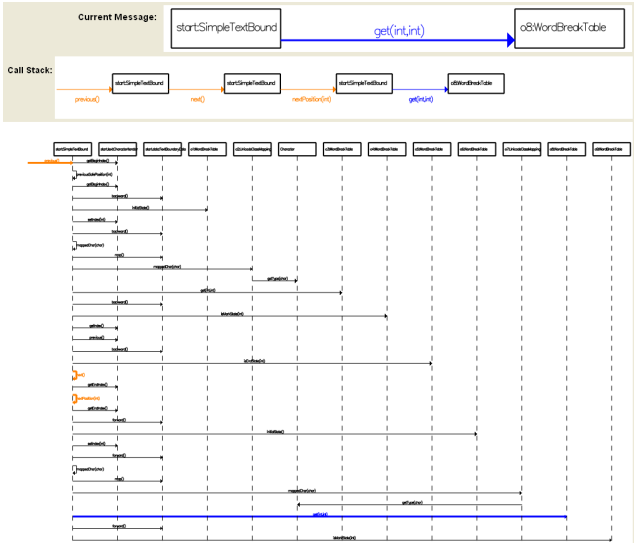


Figure 8. Diagram with a selected message.

To solve these problems, we allow the user to select a message by clicking on it. We then highlight the selected message in the main diagram, and in a separate subwindow show the message, the source object, and the destination object in an abbreviated readable form. We also highlight the messages on the corresponding call stack in the main window. Finally, the call stack is also shown at a comfortable zoom level in a separate subwindow. Figure 8 shows a diagram with a selected message. We found this approach to be essential for not getting "lost" in a large diagram, something that can happen easily in a standard diagram.

4 Experience

We implemented the above techniques in a prototype visualization tool which takes as input a textual representation of the diagrams generated by RED. We then used the tool to explore a variety of diagrams extracted from several packages from the standard Java libraries.

Our initial experience with about two hundred diagrams led to several observations. First, the reverse-engineered diagrams presented valuable abstractions that made the comprehension process significantly faster and easier, compared to simply reading the source code. For non-trivial comprehension tasks, it was very easy to get "buried under" hundreds of lines of code. The visual nature of the diagrams and their higher level of abstraction made such tasks much more tractable. Second, the ability to abstract away parts of the diagram was critical in dealing with complexity. The same way it was easy to get lost in the source code, it was also easy to get lost in a large and complicated diagram. All of the techniques described in Section 3 proved valuable in

managing complexity, and the synergy between the different techniques provided even further comprehension benefits. Third, the interactive nature of the visualization process was crucial, because it allowed us to quickly examine various scenarios based on questions of the form "what happens when ...?". The ability to have a rapid succession of "what if" questions and answers was of fundamental importance for making the tool useful.

5 Related Work

There is a large body of work on reverse-engineering analyses for object interactions [2, 3, 5, 9, 11, 12, 15]. Some of these approaches construct sequence diagrams or similar representations. In most of this work, the focus is primarily on the analysis techniques used to extract the diagrams, rather than on effective approaches for visualizing and exploring the produced diagrams; some notable exceptions are [7, 5, 15, 3, 9]. Several of these approaches consider abstraction mechanisms to ease diagram comprehension. However, such mechanisms are typically not designed to be part of highly-interactive diagram exploration. Based on our experience, we believe that the visualization of reverse-engineered sequence diagrams should be highly interactive, and the use of abstraction mechanisms should provide a programmer with immediate feedback on the effects of the corresponding diagram changes.

6 Open Questions and Future Work

Using techniques similar to the ones presented in this paper, reverse-engineering tools can take advantage of the expressive power of UML sequence diagrams in order to improve programmer productivity and software quality. If such techniques for interactive exploration are *not* used, the resulting diagrams can easily become useless in real-world software development.

The builders of reverse-engineering tools need to address two open questions. First, what are the appropriate abstraction mechanisms that should be provided by a visualization tool for sequence diagrams? Even though our initial work provides some simple answers to this question, a more extensive investigation is needed to determine the abstractions that programmers need and the appropriate interface for defining and visualizing these abstractions. Secondly, what are the actual benefits of the visualization techniques? Experiments with programmers or university students should be used to quantify the benefits of individual tool features. Furthermore, implementations of various techniques should be made publicly available. In the case of our work, an extended version of the prototype will be a part of the RED reverse-engineering tool, and will be made publicly available for use by programmers and researchers.

References

[1] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[2] L. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *Working Conference on Reverse Engineering*, pages 57–66, 2003.

[3] W. DePauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of Java programs. In *Software Visualization*, LNCS 2269, pages 151–162, 2002.

[4] M. Fowler. *UML Distilled*. Addison-Wesley, 2003.

[5] D. Jerding, J. Stasko, and T. Ball. Visualizing interactions in program execution. In *International Conference on Software Engineering*, pages 360–370, 1997.

[6] R. Kollman and M. Gogolla. Capturing dynamic program behavior with UML collaboration diagrams. In *European Conference on Software Maintenance and Reengineering*, pages 58–67, 2001.

[7] K. Koskimies and H. Mössenböck. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In *International Conference on Software Engineering*, pages 366–375, 1996.

[8] C. Larman. *Applying UML and Patterns*. Prentice Hall, 2002.

[9] R. Oechsle and T. Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using JDI. In *Software Visualization*, LNCS 2269, pages 176–190, 2002.

[10] OMG. *UML 2.0 Infrastructure Specification*. Object Management Group, `www.omg.org`, Sept. 2003.

[11] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *International Conference on Software Maintenance*, pages 34–43, 2002.

[12] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *International Conference on Software Engineering*, pages 254–263, 2005.

[13] A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *Fundamental Approaches to Software Engineering*, LNCS 3442, pages 282–297, 2005.

[14] A. Rountev, O. Volgin, and M. Reddoch. Static control-flow analysis for reverse engineering of UML sequence diagrams. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2005.

[15] T. Systä, K. Koskimies, and H. Muller. Shimba—an environment for reverse engineering Java software systems. *Software–Practice and Experience*, 31(4):371–394, Apr. 2001.

[16] P. Tonella and A. Potrich. Reverse engineering of the interaction diagrams from C++ code. In *International Conference on Software Maintenance*, pages 159–168, 2003.

[17] E. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.