A Qualitative Assessment of the Efficacy of UML Diagrams as a Form of Graphical Documentation in Aiding Program Understanding

Scott Tilley
Department of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

Shihong Huang
Department of Computer Science
University of California, Riverside
shihong@cs.ucr.edu

ABSTRACT
Graphical documentation is often characterized as an effective aid in program understanding. However, it is an open question exactly which types of graphical documentation are most suitable for which types of program understanding tasks (and in which specific usage contexts). The Unified Modeling Language (UML) is the de facto standard for modeling modern software applications. This paper describes an experiment to assess the qualitative efficacy of UML diagrams in aiding program understanding. The experiment had participants analyze a series of UML diagrams and answer a detailed questionnaire concerning a hypothetical software system. Results from the experiment suggest that the UML's efficacy in support of program understanding is limited by factors such as ill-defined syntax and semantics, spatial layout, and domain knowledge.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *documentation*.
D.2.10 [**Software Engineering**]: Design – *representation.*

General Terms
Documentation, Experimentation, Human Factors

Keywords
Assessment, graphical documentation, program understanding, Unified Modeling Language (UML)

1. INTRODUCTION
The continuous evolution of large and complex software systems is a constant challenge. To make changes in response to shifting business requirements, the software engineers charged with the task must first understand the system from multiple perspectives: application-domain functionality, high-level architecture, and myriad implementation details. Gaining such an understanding from a legacy system (a system that has been deployed in the field for many years) is particularly difficult. The software may be written in an obsolete or unfamiliar programming language; the system may run on a hardware platform that is different from today's development environments; and the many "hacks" made to the code base during its lifetime may have obfuscated an originally clean design. The result is a program that is nearly impossible to comprehend.

The situation is not much better for modern software applications. They may not suffer from the same detrimental effects of long-term maintenance as legacy systems. However, they are engineered in (and for) a more complex environment, making them more difficult to manage right from the beginning of their lifecycle. For example, it is common for today's applications to be written in several programming and scripting languages, to be deployed across a network (possibly with a Web interface), and to rely on complex third-party infrastructure for part of their functionality. For the software engineers charged with maintaining such heterogeneous systems, even systems that are relatively young, the problems associated with program understanding are in fact greater than those of the mainly monolithic, homogenous legacy systems of an earlier era.

In both situations, evolving old legacy systems and developing new complex applications, software engineers turn to program documentation to help them understand the system. The necessary documentation can be produced in several ways: by the original engineers, by maintainers recording change rationale and affected components in a configuration management system, or dynamically using reverse engineering technology. The documentation can also take many forms, such as inline commentary, linked information, or graphical visualizations.

Software visualization tools are often advocated as an effective means of aiding program understanding by creating graphical representations of the subject system "to reduce complexity of the exiting software system under consideration" [7]. While there is no doubt that such visual images can be artistically pleasing to the eye, there is little scientific evidence that such forms of graphical documentation are superior to textual documentation in effectively aiding program understanding. There are many unresolved research issues related to which types of diagrams are most appropriate for aiding program understanding. More specifically, it is unknown exactly which forms of graphical documentation are most suitable for which types of program understanding tasks, and in which specific usage context.

SIGDOC'03, October 12–15, 2003, San Francisco, California, USA.

This paper presents preliminary results from an experiment that assessed the qualitative efficacy of the Unified Modeling Language (UML) as a form of graphical documentation in support of program understanding. We focused specifically on graphical documentation since it is commonly assumed to be superior to other forms of program documentation—an assumption without a sound foundation. We focused on the UML because it is the de facto standard for modeling modern object-oriented applications.

The next section presents background information on the underlying issues of program understanding, graphical documentation, and the UML. Section 3 outlines the experiment conducted to assess the validity of the assumption regarding the efficacy of graphical documentation in support of program understanding, using the UML as a standardized visual representation. Section 4 highlights the qualitative results from the experiment, and discusses the three key findings related to UML syntax and semantics, spatial layout, and domain knowledge. Section 5 summarizes the paper and outlines possible directions for future research in this area.

2. BACKGROUND

This section provides a brief background on three areas related to the experiment. The first area is program understanding, which is what the software engineer is trying to achieve so that maintenance changes can be made in a systematic and predictable manner. The second area is graphical documentation, a form of documentation commonly used in support of program understanding, and the focus on extensive information visualization research. The third area is the UML, the most widely used object-oriented design representation and arguably an emerging standard format of graphical documentation.

2.1 Program Understanding

The goal of program understanding is to acquire sufficient knowledge about a software system so that it can evolve in a disciplined manner [17]. The essence of program understanding is identifying artifacts, discovering relationships, and generating abstractions. This process is essentially pattern matching at various abstraction levels. It involves the identification, manipulation, and exploration of artifacts in a particular representation of a subject system via mental pattern recognition by the software engineer and the aggregation of these artifacts to form more abstract system representations.

The program understanding process depends on several factors, including one's cognitive abilities and preferences, one's familiarity with the application domain, and the set of support facilities provided by the software engineering environment. These factors often determine the approach taken to understanding a complex application. For example, someone familiar with the implementation domain, but unfamiliar with the application domain, may adopt a bottom-up approach to program understanding. This involves analyzing low-level code constructs and iteratively building higher-level mental models of the subject system.

There are a variety of support mechanisms for aiding program understanding. They can be grouped into three categories: unaided browsing, leverage corporate knowledge and experience, and computer-aided techniques like reverse engineering. By far the most advantageous technique for large and complex software systems is reverse engineering, because it provides automated support for tedious and error-prone tasks such as code analysis, cross-referencing definitions and uses of software artifacts like types and variables, and the creation of abstract representations of the subject system that are closer to the application domain—a necessity when it comes to properly understanding change requests in support of evolution.

Reverse engineering is a process of analysis, not a process of change [3]. The output from reverse engineering is typically program documentation, which can take many forms. One of the most common forms of documentation is graphical.

2.2 Graphical Documentation

There are two broad categories of documentation that are used as an aid in program understanding: textual and graphical [20]. Examples of textual documentation include printed manuals and user guides. A more flexible form of textual documentation is electronic, such as HTML or XML files, which permit activities such as automated indexing and the creation of hypertext links between document fragments.

The second category of documentation is graphical, exemplified by charts and graphs. Like textual documentation, a more flexible form of graphical documentation is also electronic, which permits actions such as interactive layout and user-directed editing of the graphs. Graphical documentation relies on a variety of software visualization techniques to make complicated information easier for the developer to understand (e.g., highlighting complex dependencies in a large application by drawing nodes, arcs, and related artifacts on a computer display).

Graphical documentation itself can be further categorized according to levels of user interaction [19]. The first level is static, which permits no interaction at all. An example of static documentation is a call graph suitable for inclusion in a printed manual. The second level is interactive, which permits the user to interact with the graphical documentation in a manner analogous to surfing the Web. The third level is editable, which permits the user to play both roles of reader and the author. Editable graphical documentation requires sophisticated tool support to "complete the cycle" between the documentation produced as a result of reverse engineering, and the design documents (such as UML diagrams) used in the normal engineering process.

2.3 The Unified Modeling Language

The Unified Modeling Language (UML) is the de facto standard for modeling modern software applications [5]. It is typically used during the design phase of the software lifecycle to graphically represent different aspects of the system's high-level architecture [10]. Engineers rely on the familiar syntax and semantics of the language's visual building blocks of "things," "relationships," and "diagrams" to unambiguously represent key artifacts of the software system. In this respect, UML diagrams can serve as a universal communication medium used by all members of the project team.

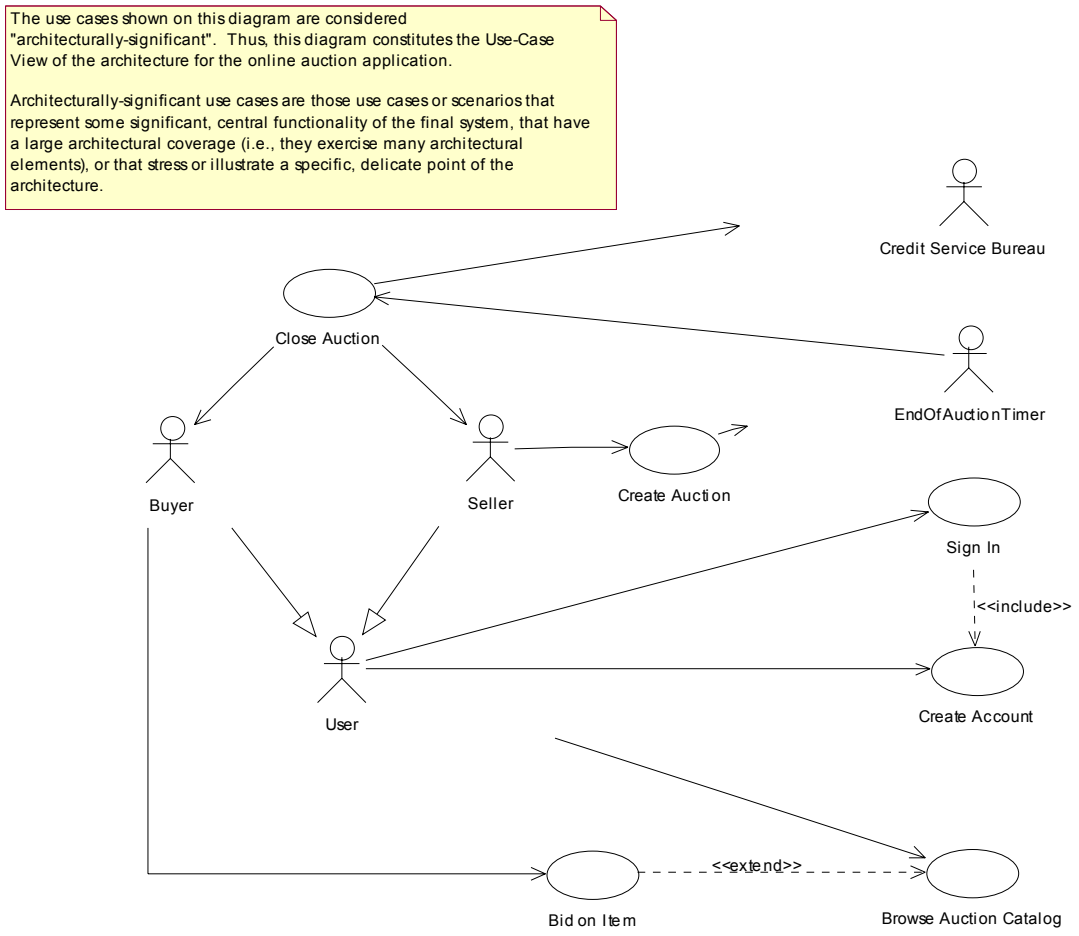

Figure 1: Architecturally Significant Use Cases from PearlCircle

Figure 1 is a UML diagram of a sample system, showing two of the most commonly used icons in system models: stick figures for actors, and ellipses for use cases. The overly simplistic representations used in the UML were chosen for ease of drawing by hand. Past experience with more formalized notations had highlighted significant barriers to adoption from software engineers to anything more complicated than "boxes and arrows."

The UML can also be used in other phases of the software lifecycle. For example, UML diagrams can be automatically recreated from existing source code through reverse engineering [14]. In this case, the resultant diagrams can serve as a form of graphical documentation suitable for use by software engineers who are tasked with carrying out modifications to the system. Disciplined and systematic evolution mandates that the engineers must first gain a sufficient understanding of the system before the modifications can be implemented, so that they know where the changes should be made, and the impact of the changes are limited and predictable.

The UML is developed under the auspices of the Object Management Group (OMG) [13]. The first release of UML 1.0 was in January 1997. Since that time, the UML has undergone several revisions. UML 1.4 is perhaps the most commonly used version in industry. However, UML 2.0 was recently ratified in Paris by the OMG in June 2003 [12]. The new 2.0 standard promises many welcome improvements that have the potential to affect the efficacy of UML diagrams as an aid in program understanding.

3. METHODOLOGY

This section presents the experimental methodology and its rationale. We are interested in assessing the qualitative efficacy of the UML as a form of graphical documentation in aiding program understanding. The idea for the experiment has its genesis in a series of related workshops that took place at SIGDOC 2001 [23] and SIGDOC 2002 [22]. The focus of each of these workshops was broadly on graphical documentation for programmers, but each specific instance had a slightly different focus.

The goal of this experiment was to determine how much a set of UML diagrams could inform a professional developer (who has reasonable knowledge of the UML) about an existing system – one that the developer didn't design or construct. In other words, assess how useful the UML is in helping to understand a legacy system. The UML diagrams in question might exist from earlier

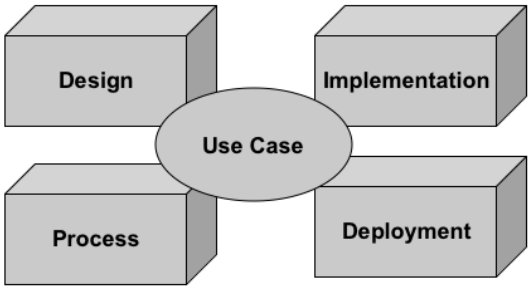

Figure 2: The 4+1 View of Software Architecture

designs, or they may be created after the fact through sophisticated reverse engineering.

3.1 Participants

The experiment was conducted during a weeklong seminar on software architecture recovery and modeling [18]. The seminar was by invitation only, with most people coming from academic positions (either Ph.D. students or Professors) in North America and Europe. All participants were experts in (one or more of) the areas of software architecture, reverse engineering, and program understanding.

More relevant for the experiment, all participants were extremely well versed in the UML. This is a rather unique situation, and one that was critical to the success of the experiment. Our interest was not in assessing the level of UML expertise of the participants, but rather how much the UML could tell them. To make this assessment, a baseline level of UML knowledge had to be assumed. Nevertheless, to ensure that all participants had a common understanding of the basic elements of the UML, we ran a "UML Boot Camp" two days before the experiment. The boot camp consisted of a 90-minute lecture that provided an overview of the UML and a review of the most common diagram types. Most participants reported that there was very little presented that they were not already familiar with.

A total of 15 subjects participated in the experiment. All participants were volunteers. They were asked to rank themselves according to their level of expertise in the UML: six people indicated they were "Beginner"; eight chose "Intermediate"; and one said "Expert." This selection process was based on their practical knowledge of real-world use of the UML in actual software projects, along with their educational use of the UML in the classroom. Although only person self-ranked as "Expert," it is felt that in another context at least half of the participants would be considered to be in this top category.

3.2 Experiment Design

All participants were given the same experiment package. The package consisted of a consent form (which they had to sign before starting the experiment), a 24-page document consisting of a series of UML diagrams for a hypothetical system, and a questionnaire. As shown in Figure 2, the UML diagram documentation was divided into five categories, according to the well-known "4+1" decomposition of software architecture, into use case, design, implementation, process, and deployment views [9].

Rational Software Corporation donated the hypothetical system, called PearlCircle [16], for the experiment. PearlCircle is an online auction system, functioning much like the well-known eBay Web site. PearlCircle relies on the J2EE platform, and has an architecture that follows the recommendations of the Rational Unified Process [6] guidelines. It is a complex application, making it an excellent example of a modern net-centric application, and as such was a good choice to use as a sample system undergoing rapid evolution.

The questionnaire had 12 questions that were directly related to likely changes that might be made to the PearlCircle application. The rational behind the questions was that the participants, acting like proxy software engineers for purposes of the experiment, would need to gain a sufficient understanding of the system before they could carry out enhancements. Some of the questions were quite general and were designed to evaluate the participants' general application knowledge as gleaned from the UML. For example, "Describe the major functional features that the system realizes." Other questions were related to software engineering issues that were directly linked to the stated purpose of the UML: high-level design. For example, "What design patterns are present in the sample system's architecture?" A third set of questions was related to implementation details of the PearlCircle application. For example, "Who sets the 'end of auction' timer?" The delineation of questions into these three categories was not made explicit to the participants.

There was one question that was not answerable using only the information available in the UML diagram. This was included to reflect a common occurrence in real-world maintenance scenarios: attempting to answer a question using all available information, only to find out that providing an informed response is impossible. Unfortunately, it often takes a considerable amount of time and effort to come to this conclusion.

The participants were given 60 minutes to complete the questionnaire. All questions required free-format prose replies; the participants were told to use as much space as they needed to answer each questions. Participants were also told that they could answer the questions in any order, since there was no dependency in the question ordering.

3.3 Threats to Validity

All empirical studies can have possible threats to the validity of the experiment and the conclusions drawn from the analysis of the results. This is true for this experiment as well, although since we are focused on qualitative analysis (as opposed to quantitative analysis), the threats can be managed somewhat more easily. Of the many possible sources of contention for the experiment, the following three warrant special attention: the experiment design, the participants, and the sample system.

3.3.1 Experiment Design

The first threat to validity is the nature of the experiment itself. Some would argue that relying on graphical forms of documentation alone is not realistic: in a real-world situation the software engineers would have access to other sources of documentation as well, such as text, online help, and system source code. This may be true, but we were mainly interested in assessing the information content in the UML alone. There is little doubt that having other sources of information, in addition to the

UML, would help most people with common program understanding tasks.

Others might say that a better assessment of the efficacy of UML as a form of graphical documentation would be gained by using a comparable suite of textual documentation as a baseline. The difficulty here is in creating a "comparable" suite. In some ways, directly comparing text with graphics is like comparing apples and oranges: the conclusions drawn from such a comparison are suspect. It was for this reason that a graphical-only test was conducted. Efficacy was qualitatively measured according to the perceived correctness and thoroughness of the participants' responses on the questionnaire (according to our best judgment when reading their answers), and the time they took to complete the entire experiment.

The other aspect related to the experiment that could be contentious is the makeup and structure of the questionnaire. The participants were given a finite amount of time in which to complete as many questions as they could. It could be argued that there were too many questions to answer in 60 minutes, or that in a true maintenance setting such time restrictions would not be as inflexible. This is quite possibly true. However, many critical maintenance fixes are made in a hurry, under extreme duress, with little time for proper contemplation. For example, patches issued by vendors for security vulnerabilities found in their products—vulnerabilities that could have widespread and devastating consequences if not fixed in a very short time.

3.3.2 Participants

The second threat to validity is the selection of participants in the experiment. Recall that the experiment's focus was assessing the efficacy of the UML as a form of graphical documentation in aiding program understanding. We were not particularly interested in assessing the participants' knowledge of the UML itself.

In this case, we were especially fortunate to have participants whose knowledge of the UML was world-class. It is doubtful that we could find such a group in a more traditional academic experimental setting, such as with a group of students. The only place we would be likely to find equally capable participants would be a mature software development organization that uses modern tools and disciplined techniques. Therefore, we feel comfortable that the threat of participant expertise is minimal.

3.3.3 Sample System

The third threat to validity is the choice of PearlCircle as the sample system. Many experiments choose to use a very small system. We chose instead a rather large system, mainly because we felt that it was a more realistic choice, better encapsulating the challenges facing modern software engineers. We also felt that PearlCircle, being based on emerging standards such as J2EE and relying on tools that directly support the UML for its design, was a good choice for a system that might undergo significant enhancement to support future evolution.

Moreover, we did not create the PearlCircle application; experts at Rational Software Corporation did. Unlike the experiment participants, we had access to forms of documentation other than the UML diagrams, such as requirements documents and textual descriptions of the system's architecture. However, the UML diagrams used in the experiment were taken directly from the Rational documentation suite for PearlCircle, and were used unaltered by the participants. We did not change or reformat the diagrams in any way.

4. QUALITATIVE RESULTS

Preliminary qualitative results from the experiment suggest that UML diagrams can indeed help software engineers understand large-scale systems. However, the diagrams' efficacy is limited by several factors. This section summarizes three areas that were identified as cross-cutting issues for most of the experiment's participants: the syntax and semantics of the UML itself, issues related to spatial layout of the diagrams, and the role of domain knowledge in system understanding.

4.1 Syntax and Semantics

As a visual modeling language, the UML provides a convenient mechanism for software engineers to represent high-level system designs and (to a lesser extend) low-level implementation details. It is a standard in the sense that it is backed by a large consortium (the OMG), it is well supported by commercial tools (albeit unequally), and it is generally agreed what each of its constituent diagrammatic constructs mean. As such, it is a definite improvement over the traditional "boxes and arrows" diagrams used by developers before the UML, since these earlier diagrams could have vastly different meanings associated with the same boxes and arrows, depending on who was doing the drawing and who was doing the reading.

However, the problem is that some of the syntax, and much of the semantics, of the UML is still not specified very clearly. Many parts of the standard remain somewhat open to interpretation, resulting in different tools implementing the same thing in different ways. This is not true for the majority of the UML's main building blocks, such as class diagrams and common relationships. But it is true for some of the UML's more advanced features. At the very least, such syntactic differences can be annoying. However, for complex applications that rely on nuanced UML to capture essential aspects of the subject system, such syntactic ambiguity can lead to misunderstanding and confusion. This is particularly troublesome when UML diagrams are used as a form of graphical documentation, since there may not be any other information associated with the system, forcing the software engineer to rely entirely on the possibly erroneous visual model.

For reverse engineering tools that are used to recreate UML diagrams, the problem is actually worse than for normal design tools. The reverse engineering tool must select a collection of UML diagrams that best represent the system being analyzed. Since the selection procedure depends on both the input (which is incomplete), and the output (which is ill-defined), the result is that two tools can analyze the same source code yet produce different UML diagrams that represent the code. Clearly this has significant implications for program understanding.

The problem is even more vexing when it comes to UML semantics. The current UML draft is already quite large and complex. Many of its more advanced features take considerable time to learn to use properly, and even then few people have a complete mastery of the entire language. This impacts both the reverse engineering tool developers, who must select the

appropriate collection of related UML diagrams to represent the subject system, and the software engineers using the resultant tools to help them understand the same system. Hopefully, the UML 2.0 standard will address these issues.

In the experiment, several participants pointed out possible errors in the UML diagrams. In most cases, they were not sure if the errors were real, or if their judgment was incorrect. When one considers that many of these same people were world-class UML experts, and that the sample system was created by extremely capable people with a deep knowledge of UML, it is obvious that the "average" software engineer might experience considerable difficulty in differentiating correct but complex diagrams from errors introduced by the designer or by the reverse engineering tool.

4.2 Spatial Layout

A significant body of research into information visualization focuses on trying to improve comprehension of complex data sets. Improvement techniques used include layout algorithms that minimize edge crossings, layered graphs that represent subsystem hierarchies, and careful selection of stencil shapes and colors (and relationships arcs) to better reflect the users' mental model of the system under scrutiny. There is no doubt that the physical layout and spatial relationships between entities in a large graph can play an important role in program understanding [10].

There are numerous classical problems related to graph layout and detail visibility when trying to display complex diagrams in a limited area (e.g., on a standard-sized sheet of paper or a typical computer screen) [8]. For complex UML diagrams, which can have many dozens of artifacts and an equally large number of relationship arcs, the problem is particularly acute. Several of the UML diagram types (e.g., sequence diagrams) are quite dense and do not lend themselves to space compression while remaining understandable.

In the experiment, the participants were looking at a series of UML diagrams printed on standard A4-sized paper. This resulted in parts of some of the more complicated diagrams being scaled down to nearly 6-point type, making it extremely hard to read and hence understand. This is illustrated by Figure 3, which represents the time-based interactions among components in browsing an auction catalog. For purposes of this paper, the figure is shown at 90% magnification. However, even at 100% scale (the version the experiment participants had), the figure is very difficult to read and to understand.

It might be argued that developers would typically be viewing such diagrams using large 21" cinema-style displays, not on regular paper. However, the counter argument is that UML diagrams (when treated as a form of graphical documentation) must be capable of inclusion in a printed (or PDF online equivalent) document. More importantly, the document must still be legible and usable in this form; otherwise, the diagrams will help program understanding very little.

For reverse engineering tools that produce UML diagrams, spatial layout (and related graphical attributes) is key to fostering understanding. Indeed, the recent book "The Elements of UML Style" by Scott Ambler [1] speaks at length about the importance of layout issues as a matter of good diagramming style. Reverse engineering tools can create UML diagrams that are suitable for processing by other tools, such as layout engines, by representing the UML in the XMI format. XMI is an XML-based representation of the UML graphical diagram. Unfortunately, in the current version of UML standard in widespread use (UML 1.4), the XMI model interchange format is deficient in the sense that it does not specify layout information in the saved data. The result is that spatial layout information of the UML diagrams is not recorded. As with the issues related to the syntax and semantics of the UML, hopefully this problem will be addressed as the community adopts UML 2.0.

4.3 Domain Knowledge

The UML is well suited to representing high-level design concepts, some lower-level implementation guidelines, and dynamic information related to message passing and method invocation. This can be viewed as a mixture of application domain knowledge, coupled with an object-oriented design philosophy.

However, mastering a complex system requires knowledge of more than just its high-level design; several types of domain knowledge are essential. Complimentary knowledge of the implementation domain (e.g., Java), the development and deployment infrastructure (e.g., J2EE), and related technologies is needed in order to gain a sufficient understanding of the program to make informed decisions before changes are made. Without this multi-faceted domain knowledge, the representation of the subject system is incomplete and does not convey enough information to fully assist the software engineer in understanding the application. This is information that the UML (and graphical documentation in general) cannot easily provide.

In the experiment, the sample system used was an online auction. To properly understand it requires extensive expertise in Java, J2EE, Sun Microsystems design patterns, distributed systems architecture, databases and persistent storage, networking and security, the business rules of a typical auction, Internet and Web technologies, and so on. In other words, one needs to be a renaissance software engineer [21]. The UML by itself is insufficient to represent this sort of broad and deep information. Several participants commented that they were unable to answer several of the more challenging questions because they did not know enough about the server-side deployment platform (J2EE), or they did not know enough about how an auction worked, etc. In other words, their own domain knowledge was lacking, and the UML diagrams were insufficient to make up such deficiencies—one of the main purposes of quality program documentation.

Even if the UML were rich enough to represent this level of multiple domain knowledge, a reverse engineering tool would need to be capable of producing the information upon which the knowledge is based. For example, the tools would need to be able to include data from sources other than implementation domain artifacts such as source code. Unfortunately, very few tools permit the incorporation of domain knowledge into the repository.

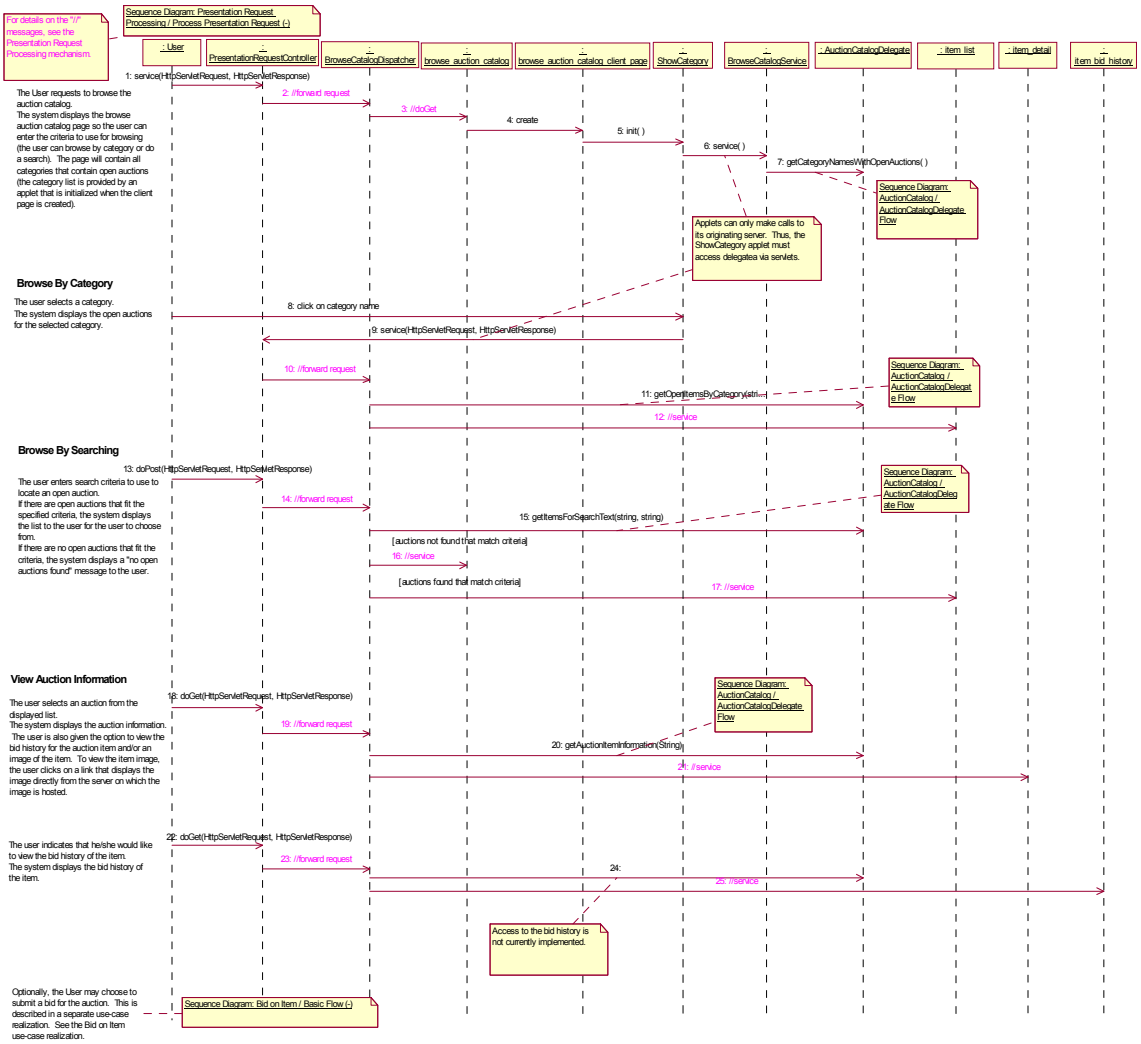

Figure 3: Browse Auction Catalog (Basic Flow) from PearlCircle

5. SUMMARY

This paper presented qualitative results from an experiment assessing the efficacy of the UML as a form of graphical documentation in aiding program understanding. The experiment was conducted during a seminar on software architecture recovery and modeling. The participants were very knowledgeable about the UML, ensuring that the experiment assessed the information content contained in the diagrams, and not their understanding of the basics of UML per se.

Based on the answers provided by participants to the questionnaire about the PearlCircle sample system, three distinct issues related to the efficacy of the UML were identified: (1) the ill-defined syntax and semantics of the visual modeling language itself; (2) the critical role that spatial layout plays in fostering program understanding; and, (3) the importance of domain knowledge in supplying necessary information to aid the software engineer in system evolution tasks. It is hoped that the recently approved UML 2.0 standard will address some of the UML-specific issues, such as syntax and semantics. Reverse engineering tools that recreate UML diagrams based on extensive source code analysis will have to be improved to better support effective layout. Perhaps most importantly, the incorporation of domain knowledge into the reverse engineering process is considered essential to augment a developer's lack of background experience related to the subject system, and this domain knowledge must be properly represented in the generated UML diagrams.

There remains a significant amount of work to be done in this area. Since the experiment in this paper was conducted, we have conducted a second, similar experiment at IWPC 2003 [4]. However, in the second case we selected a much-simpler sample system, and we used a textual document as a baseline for comparison. The results of this study will be published in a suitable venue once the results have been analyzed.

There is a growing awareness of the need to employ evidence-based arguments to support the practices of software engineering and technical documentation, rather than arguments based upon

advocacy [2]. One of the paradoxes of software engineering is that, although it extensively employs widely accepted concepts and practices that are drawn from experience and observation, we rarely possess any solid audit trail that can provide a validation of these ideas and that could link theory and concepts to observed practices. The qualitative experiment described in this paper is one step towards a more scientific and objective measure of the efficacy of the UML as a form of graphical documentation in support of program understanding. Further work focusing on quantitative results, such as that described in [15], would be a welcome next step.

REFERENCES

[1] Ambler, S. *The Elements of UML Style.* Cambridge University Press, 2003.

[2] Budgen, D.; Hoffnagle, G.; Müller, M.; Robert, F.; Sellami, A.; and Tilley, S. "Empirical Software Engineering: A Roadmap." *Proceedings of the 10th International Conference on Software Technology and Engineering Practice* (STEP 2002: Oct. 6-8, 2002; Montréal, Canada). IEEE Computer Society Press, 2003.

[3] Chikofsy, E.; and Cross, J. "Reverse Engineering and Design Recovery: A Taxonomy." IEEE Software 7(1):13-17, January 1990.

[4] Huang, S. and Tilley, S. "Workshop on Graphical Documentation for Programmers: Assessing the Efficacy of UML Diagrams for Program Understanding." Held in conjunction with *The 11th International Conference on Program Comprehension* (IWPC 2003: May 10-11, 2003; Portland, OR).

[5] IBM Corp. "UML Resource Center". Online at http://www.rational.com/uml.

[6] IBM Corporation. "The Rational Unified Process (RUP)." Online at http://www.rational.com/products/rup/index.jsp.

[7] Knight, C.; Munro, M. "Comprehension with[in] Virtual Environment Visualizations". Proceedings of the 7th International Workshop on Program Comprehension (IWPC 1999: May 5-7, 1999; Pittsburgh, PA, USA), pp. 4-11. Los Alamitos, CA: IEEE Computer Society Press, 1999.

[8] Koning, H.; Dormann, C.; and van Vliet, H. "Practical Guidelines for the Readability of IT-Architecture Diagrams." Proceedings of the *20th Annual International Conference on Systems Documentation* (SIGDOC 2002: October 20-23, 2002; Toronto, Canada), pp. 90-99. ACM Press: New York, NY, 2002.

[9] Kruchten, P. "Architectural Blueprints—The '4+1' View Model of Software Architecture." *IEEE Software* 12(6):42-50, November 1995.

[10] Medvidovic, N.; Rosenblum, D.; Redmiles, D.; and Robbins, J. "Modeling Software Architecture in the Unified Modeling Language." *ACM Transactions on Software Engineering Methodology* 11(1):2-57, January 2002.

[11] Müller, H.; Tilley, S.; Orgun, M.; Corrie, B.; and Madhavji, N. "A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models." *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments* (SIGSOFT '92/SDE 5: Tyson's Corner, VA; December 9-11, 1992), pp. 88-98. New York, NY: ACM Press, 1992.

[12] Object Management Group (OMG). "UML 2.0 Standard Officially Adopted at OMG Technical Meeting in Paris." June 12, 2003. Online at http://www.omg.org/news/releases/pr2003/6-12-032.htm.

[13] Object Management Group. Online at http://www.omg.org.

[14] Pierce, R. and Tilley, S. "Automatically Connecting Documentation to Code with Rose". *Proceedings of the 20th Annual International Conference on Systems Documentation* (SIGDOC 2002: October 20-23, 2002; Toronto, Canada), pp. 157-163. ACM Press: New York, NY, 2002.

[15] Purchase, H.; Colpoys, L.; McGill, M.; and Carrington, D. "UML Collaboration Diagram Syntax: An Empirical Study of Comprehension." *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002: June 26-27, 2002; Paris, France).*

[16] Rational Software Corp. "PearlCircle." Online at http://rdatux.rational.com/auction/index.jsp.

[17] Tilley, S. "The Canonical Activities of Reverse Engineering." *Annals of Software Engineering* 9:249-271, 2000.

[18] Tilley, S. and Huang, S. "Assessing the Efficacy of Software Architecture Visualization Techniques for Recovered Artifacts." *Dagstuhl Seminar 03061: Software Architecture Recovery and Modeling* (Feb. 2 – 7, 2003; Schlöss Dagstuhl, Germany).

[19] Tilley, S. and Huang, S. "Documenting Software Systems with Views III: Towards a Task-Oriented Classification of Program Visualization Techniques". *Proceedings of the 20th Annual International Conference on Systems Documentation* (SIGDOC 2002: October 20-23, 2002; Toronto, Canada), pp. 226-233. ACM Press: New York, NY, 2002.

[20] Tilley, S. and Huang, S. "On Selecting Software Visualization Tools for Program Understanding in an Industrial Context." *Proceedings of the 10th International Workshop on Program Comprehension* (IWPC 2002: June 26-29, 2002; Paris, France), pp. 285-288. Los Alamitos, CA: IEEE Computer Society Press, 2002.

[21] Tilley, S. and Huang, S. "On the Emergence of the Renaissance Software Engineer." *Proceedings of the 1st International Workshop on Web Site Evolution* (WSE'99). Atlanta, GA: October 5, 1999.

[22] Tilley, S. and Wong, K. "Workshop on Graphical Documentation for Programmers." Held in conjunction with *The 20th Annual International Conference on Systems Documentation* (SIGDOC 2002: October 21, 2002; Toronto, Canada).

[23] Tilley, S.; Smith, D.; and Thomas, B. "Documentation for Software Engineers: What is Needed to Aid System Understanding?" Held in conjunction with *The 19th Annual International Conference on Systems Documentation* (SIGDOC 2001: Santa Fe, NM; October 21-24, 2001).