Assessing Program Comprehension Tools with the Communicability Evaluation Method

Denis Pinheiro[1], Marcelo Maia[2], Raquel Prates[1], Roberto Bigonha[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais

[2] Faculdade de Computação – Universidade Federal de Uberlândia

{denis,raquel,bigonha}@dcc.ufmg.br, marcmaia@facom.ufu.br

Abstract. *Most program comprehension tools present information extracted from the source code in a visual way. The user interface of such a tool may support or hinder the strategy used by the programmer. In this paper a communicability evaluation method is conducted in two different comprehension tools and provide indicators on aspects that are relevant for the quality of the interface of such systems.*

1. Introduction

Understanding an already built software system is necessary because those systems are in permanent maintenance and evolution. Mayrhauser and Vans [1995] point out five basic tasks related to software maintenance and evolution: system adaptation, perfective and corrective maintenance, source code reuse and restructuring. The program comprehension activity is necessary nearly in all of the above tasks. Furthermore, large systems and outdated documentation usually add to the complexity of program comprehension task.

Storey [2005] presents a study about theories and tools around the program comprehension activities. Program comprehension tools are designed according to several possible strategies used by a programmer while performing comprehension tasks. It is known that comprehension tools can better support or even hinder the comprehension strategies used by developers [Storey et al. 2000]. The functionalities supported by a tool, e.g., navigation, queries and multiple views are commonly implemented within a graphical user interface. It is well known in the human-computer interaction community that the success or failure of an interactive system is determined not only by the amount of delivered functionality, but also by how the designer chooses to present it to the user at the interface.

This paper presents indicators on how the comprehension strategy chosen by the tool designer and its presentation to users affect users´ experience with two comprehension tools that support different strategies, namely SHriMP [Storey et al. 2002] and *Understand for Java* [scitools.com]. Both are visual tools with graphical user interfaces showing diagrams and views built from Java source code analysis. However, SHriMP adopts an integrated comprehension strategy, whereas *Understand for Java* adopts a bottom-up one. We have performed a communicability evaluation with users interacting with both tools, and collected indicators on the tasks better supported by each tool, as well as on how the way these strategies were presented to the users impacted their experiences.

The next secition presents some background on program comprehension strategies and reviews the comprehension tools used in the experiments. Section 3

presents the Communicability Evaluation Method used to appreciate the quality of the selected tools' interfaces. Section 4 describes the experiment design and the results are discussed in section 5. Finally section 6 presents our final remarks.

2. Program Comprehension Strategies and Studied Tools

Some strategies guide the way programmers work in a comprehension process [Storey et al. 2000]. *Top-down comprehension* is a process to understand by reconstructing the knowledge domain and mapping it onto the source code [Brooks 1983]. *Bottom-up comprehension* is an understanding process that starts from the source code, and successively, the software artifacts are grouped into higher level abstractions [Shneiderman and Mayer 1979]. *Systematic comprehension* is a process to read the code following the control and data flow [Soloway and Ehrlich 1986]. *Knowledge-based comprehension* is a process based on the idea that the programmer can evolve the comprehension either with bottom-up or top-down strategies [Letovsky 1986]. *Integrated comprehension* is a process that combines top-down, bottom-up and knowledge based approaches into a single strategy [von Mayrhauser and Vans 1995].

The two systems used were SHriMP/Creole and *Understand for Java*. SHriMP (Simple Hierarchical Multi-Perspective) [Storey et al. 2002] adopts an integrated strategy, providing mechanisms to browse and to explore complex information spaces. It provides visualization of nested graphs to present hierarchically structured information. It introduces the concept of nested interchangeable views that allows users to explore information in multiple perspectives and multiple abstraction levels. SHriMP may be used to explore and to browse in the Eclipse IDE using the Creole plug-in.

SHriMP has several functionalities organized in windows encapsulated within tabbed panes. The main window shows the system architecture built with colored nodes and arcs. Nodes represent the program's entities and can be expanded/collapsed in order to show/hide the nested entities. The arcs represent the relationship between the entities, and they can encapsulate several relationship occurrences of the same type between two entities. Some visualization options are method call diagrams, dependency diagrams and class hierarchy diagrams. Some algorithms are available to organize the diagrams. A search tool is available to perform queries on entities in the program.

Understand for Java© (*UJ*) [scitools.com] is a source code analyzer that supports programmers in understanding software projects written in Java. The source code is analyzed and a repository with structures and relationships extracted from the code is created. The repository is used to produce control flow diagrams, method call graphs, inheritance hierarchy, and used to provide navigation and queries on source code. The strategy adopted is bottom-up.

The tool interface is divided in three main areas: filter, information, and document area. Filters enable the user to select specific entities in the source code based on their types. The presentation of the entities' information includes location, metrics and relationships in the information browser. In the document area, several windows are presented, including either a specific diagram or the source code. Diagrams and entities' information are presented after selecting the entity name, or directly in the diagrams, or activating a right click menu. Control flow diagrams, method call diagrams, inheritance hierarchy diagrams, relationship between entities are presented using its own notation.

3. The Communicability Evaluation Method

The Communicability Evaluation Method (CEM) is a qualitative method for user interface evaluation, based on Semiotic Engineering (SemEng) theory [Prates et al. 2000, de Souza 2005, Prates and Barbosa 2007]. The SemEng perceives the system´s interface as a designer-to-user communication. In this message the designer conveys to users who the system is meant to, what problems it can solve and how to solve them [de Souza 2005]. Communicability is a distinctive quality of interactive systems that communicate efficiently and effectively to users their underlying design intent and interactive principles.

The CEM involves users, who perform specific predetermined tasks in a controlled environment, such as a user testing lab. Users receive scenarios describing the tasks to be executed with the system being evaluated. The tasks are executed one at a time and recorded using a screen capture software for further analysis. During the test, evaluators instruct users, observe and take notes on users behavior or comments that may strike them as relevant, but they do not interfere with task execution. At the end, an interview with the user may be carried out in order to better understand some of the actions and communicative breakdowns that may have occurred, and also to collect data on user experience and satisfaction. The data analysis is comprised of 3 steps:

Tagging. Evaluator plays back the user interaction and identifies communicative breakdowns, represented by specific interaction patterns. The evaluator then associates an utterance to this breakdown, as if "*putting words in the user´s mouth*". For instance, if the user stops the cursor over an interface element trying to see the hint associated to it the evaluator would tag it with the utterance ***What's this?*** or if the user was browsing the menus looking for a specific option it would be tagged with a ***Where is it?***. The other utterances the comprise the predefined set are: ***What now?***, ***Oops!***, ***Where am I?***, ***I can't do it this way.***, ***Why doesn't it?***, ***What happened?***, ***Looks fine to me.***, ***I give up!***, ***I can do otherwise.***, ***Thanks, but no, thanks.***, ***Help!***.

Interpretation. In this step, the evaluator tabulates the problems identified by the breakdowns. During the interpretation process the evaluator should consider 4 factors: (1) classification of the utterances according to the type of communicative failure they represent in the system-user communication; (2) the frequency and context of occurrence of each type of utterance; (3) the existence of patterned sequences of utterance types; and (4) the level of goal related problems signaled by the occurrence of utterance types. This step depends on the evaluator's experience with the method and knowledge of SemEng.

Semiotic profiling. The evaluator reconstructs the overall designer to user message.

Our analysis focuses on the tagging step and their impact to the users experience. The complete interpretation and semiotic profiling are not discussed.

4. Experiment

The experiment was conducted with four users, each of them using both tools. The goals of the experiment were: (1) collect information on the impact of the comprehension strategy used by the system and that requested by the task; (2) identify communicability problems in the selected tools that could hinder program comprehension; and (3) based on (1) and (2) identify some relevant aspects of the

interface designers should consider carefully during the project of a program comprehension system.

Participants were chosen through a pre-test applied to graduate students from the Computer Science Graduate Program at UFMG, who volunteered to participate in the study. Two of the four selected participants had intermediate knowledge of Java and the other two had advanced knowledge of Java. They all had familiarity with the Eclipse development environment and none of them had interacted with the selected comprehension tools before.

4.1. Experiment Organization

One of the factors that affected the choice of the comprehension tools was the different comprehension strategies they adopted. SHriMP adopts an integrated comprehension strategy, and the way the visualization is organized emphasizes top-down comprehension. *UJ* adopts a bottom-up comprehension strategy. A communicability test was conducted with the 4 participants performing 6 typical program comprehension tasks. The tasks were divided into two groups: 3 tasks that required analysis of high-level abstractions (e.g. models and diagrams) and 3 focused on low-level abstractions (e.g. source code, flow control, method calls). Each participant used both tools, performing 3 tasks with one tool and other 3 tasks with the other tool. The table below shows how tasks (T), participants (P) and tools were divided. Tasks 1 to 6 were performed in that order.

	Higher-level tasks			**Lower-level tasks**		
	T1	T2	T3	T4	T5	T6
P1, P2	SHriMP			UJ		
P3, P4	UJ			SHriMP		

4.2. Participant Instructions and Training

On arrival participants heard an explanation about the evaluation and its goal. They then were given an overall explanation about the functionalities of the selected tools. They were informed that the interaction needed to be recorded and gave their consent. They were also informed that their identities would remain anonymous; they could interrupt their participation at any time if they wished to and that they would have access to the data collected during their tests at any time. The basic functionalities of the selected tools were quickly demonstrated to the participants in order to provide an initial knowledge about the tools to them. The demonstration took place through the visualization of a small Java program implementing a banking application.

4.3. Execution and Pos-Test Interview

The tasks executed by the participants during the test were typical tasks of program comprehension. A typical scenario of software maintenance requiring a comprehension phase was presented to all participants. The participants performed the comprehension tasks with a Battleship game program, implemented in Java. There are 15 classes implemented in 625 lines of code, organized in 3 packages (*grid, uinterface* and *util*). None of the participants had any knowledge about the implementation. Task execution was recorded and an evaluator was present and took notes. Each participant's session lasted approximately an hour and a half.

The 6 tasks were chosen based on the benchmark proposed in [Sim 2003] and were: (1) Navigate the views created from source code by the tool and try to get an overall architecture of the program. Draw a diagram for the architecture; (2) In which blocks of the previous diagram were the game rules implemented?; (3) In the Battleship game, is the mode "*one user against the computer*" available?; (4) What is the size of the grid that defines the "*Sea*" of the game? Is the size of the grid fixed or can it be redefined? If the size is fixed, how can the implementation be changed to support the redefinition before the game starts, and if the grid is not fixed?; (5) How many ships can be located in the grid (*Sea*)? If we want to add another kind of ship, which changes need to be made to the program and where?; (6) What is your evaluation about the program structure? Do you consider the Battleship game program was well-designed?. The first three tasks required the participants to get an overall view of the program, but the third task, also required source code inspection. The last three tasks required a lower-level view of the program, that is, required visualizing the source code. There were tasks that required understanding how to change the program.

At the end of the experiment the evaluator interviewed each participant. The interview intended to allow evaluators to better understand some of the actions observed; collect data on user experience.

5. Results

The first step of the analysis intended to identify strategies adopted by users at each task, next the CEM tagging was performed. All participants completed all tasks but with some variation on the detail level of the answers. Advanced users were more objective in their answers, as opposed to intermediate users who tried to better explain their answers. Each participant's comprehension strategy was identified observing the records. The participants used a bottom-up strategy with *UJ* because it was the only one supported. They browsed the code and selected diagrams related to the entities in the visualized code. Although SHriMP supports an integrated comprehension strategy, participants adopted a top-down strategy. They kept navigating throughout the diagrams to find and verify relationships. Even when they needed to visualize the code, they navigated throughout the diagrams until they could locate points they could explore in the source code, and only then did they visualize the code. It is possible that this choice of strategy was motivated by the initial diagram presenting the overall architecture of the program, from which the participants started the exploration of other diagrams.

During the tagging step users interactions were associated to utterances. Figure 1 shows the tags distributed along the timeline for each participant performing each task. Notice that three tags occurred much more often than others, mainly ***Where is it?***, ***What´s this?*** and ***Oops!***. The ***Where is it?*** tag could be observed frequently at the beginning of a task execution. This breakdown occurred when users navigated throughout the several views provided by the system looking for the necessary information about the Battleship program to perform the task. It is interesting to notice that although users were not looking for a function in the interface (usual interaction pattern associated with ***Where is it?*** tag) they were trying to identify which view would provide the desired information about the code. The occurrence of ***Where is it?*** breakdowns are expected when first interacting with a system. However, the fact that its frequency did not decline in time, as users learned the comprehension tool being used,

as well as the Battleship program, may indicate a lack of clarity in the interface when expressing the information extracted from the source code to the user.

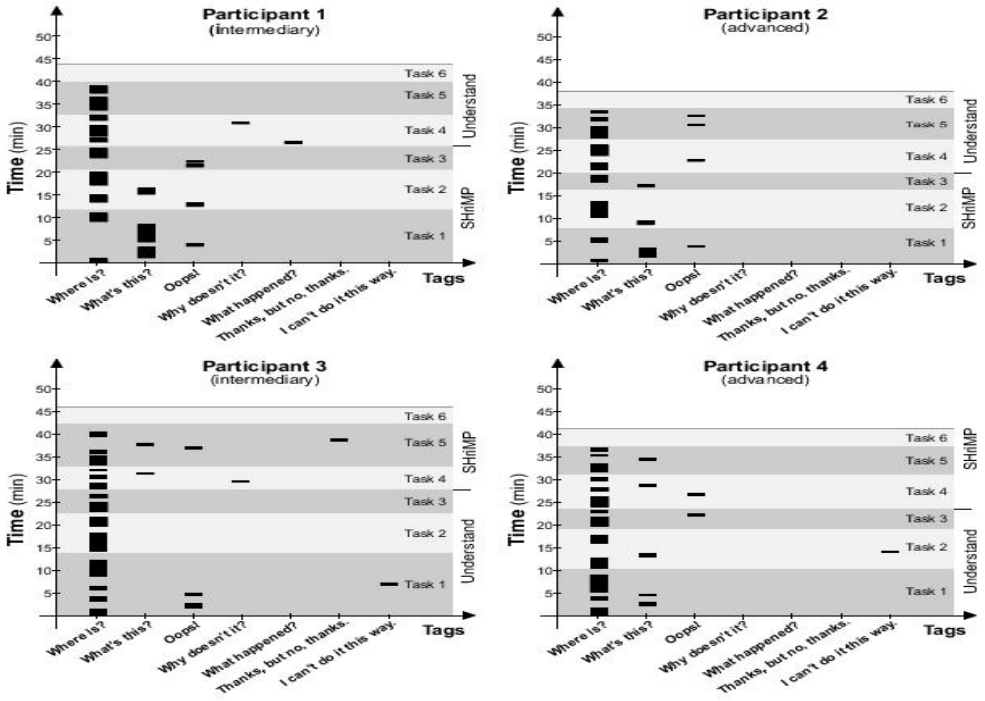

Figure 1. Tagging results

The ***What´s this?*** tags occurred, mainly when using SHriMP. However, there were 2 different situations that led to that tag. The first was the original one, that is, when the user stops the cursor over an interface element and waits for the explanation available at the hint. The other one represented the same interaction pattern, but with different purposes. In SHriMP the abstract views would present detailed information by use of the hint. For instance, in the view that showed the dependency between entities, once the cursor was put on the arc that represented the dependency, dependency details about which specific method depended on which other one was shown. In this case users understood how to interact with the system and what the underlying intent was and were able to use the system successfully. Thus, the request for detailed information should not be considered a communicative breakdown.

Another frequent communication breakdown was associated to the ***Oops!*** tag. It was mainly identified when users opened up a view, either a diagram or the source code, and rapidly noticed that there was no useful information related to the task at hand on that view, returning then to the previous view. This breakdown was more frequent when using *UJ*. One participant commented that the *Information Browser* of *UJ*, that presents textual information (e.g. metrics, declarations, uses, dependencies, etc.) about the entities was the only really useful functionality in that tool.

There was an instance of communicative breakdown that is interesting to discuss, even though it only happened in one of the tests. In SHriMP, the visualization of the source code is limited to a small window. Most users complained about this feature. The utterance ***Thanks, but no, thanks!*** occurred when participant 3 was executing task 5. This participant needed to navigate throughout the source code to understand specific details of the implemention, but he refused to use the SHriMP

visualization option and decided to visualize the code directly with the Eclipse editor. Even though it occurred only once, it is relevant because users are declining a visualization offered by the designer, one of the main features of comprehension tools. Thus, declining this specific feature could lead declining the system as a whole.

Breakdowns associated to tags ***What happened?***, ***Why doesn´t it?*** and ***I can´t do it this way.*** were also identified. However, since they happened with such low frequency they did not provide any conclusive indicators in this experiment.

Analyzing the results of the interview, it could be noticed that users considered both tools easy to use. However, they pointed out that the dependencies view of both presented interaction difficulties. SHriMP represented all the relationships in this view with arcs. It resulted in an overwhelming amount of information to users. *UJ* on the other hand, presents each kind of relationship on a different window, making it necessary for users to manage several windows. Most participants believe that the dependencies view is crucial for the effectiveness of a program comprehension tool.

The interviews also showed that participants had a better experience with SHriMP diagrams because they resemble UML diagrams, and the icons representing the entities followed the Eclipse standard. Although, the diagrams of *UJ* were considered easy to understand, users pointed out that they felt it was easier to understand the information by using SHriMP. They believed the main reason for that was the use of standard known notation.

6. Final Remarks

This paper presented an evaluation of the user interface of two comprehension tools: SHriMP and *UJ*. The user interfaces were evaluated using the CEM. Being a qualitative method, the goal was not to obtain statistically valid results, but rather indicators of relevant problems related to the interactive program comprehension systems.

In order to collect information on the impact of the comprehension strategy, the experiment was planned to include tasks that were favored by system strategy (i.e. abstract tasks favored by top-down strategy), as well as tasks that might be hindered by the strategy (i.e get low level code information by use of a top-down strategy). The study showed that the strategy adopted by users was mainly imposed by the tool, and less influenced by the task. In *UJ* the abstract levels offered by the system were not enough to provide the user with the information needed by the high level abstraction tasks. Users had to create the abstractions based on existing views that favored a bottom-up strategy. SHriMP took an integrated perspective, but favored a top-down approach. When users performed the low level tasks that required a detailed view of the code, they used a top-down strategy to find and access it. This result corroborates the findings presented by [Storey et al. 2000] and points to the need to better understand the factors that influence program comprehension systems effectiveness.

In that direction, indicators show that it may be interesting for tools to support a quick understanding of the overall architecture of the system, as well as a good visualization of the source code. In other words, provide both top-down and bottom-up strategies that allowed users to decide on and adopt the best one for the task at hand. The participants also pointed out that one of the most important features for program comprehension tools is the visualization of system dependencies, since it often is a bottleneck in the comprehension process.

The analysis of the experiment showed that most of the communicative breakdowns were related to finding the desired information in one of the many possible views. It would be interesting to conduct a study of the use of these systems during a longer period of time to see whether these breakdowns would decrease and users would be able to understand what was depicted in the different views or if they would end up declining some of the views and using a subset of them. A study in a longer period of time would also allow for an observation whether other breakdowns that had only a few occurrences in this study would increase. In regard to the views that offered a more detailed level of information on demand, it would be interesting to investigate how well they supported users in navigating through different levels of abstraction.

References

Brooks, R. E. (1983). Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies, 18(6):543–554.

de Souza, C. S. (2005). The semiotic engineering of humancomputer interaction. The MIT Press, Cambridge, MA.

Letovsky, S. (1986). Cognitive processes in program comprehension. In 1st Workshop on Empirical Studies of Programmers, pages 58–79, USA. Ablex Publishing Corp.

Prates, R. O. and Barbosa, S. D. J. (2007). Introdução à teoria e prática da interação humano computador fundamentada na engenharia semiótica. In T.Kowaltowski e K. K. Breitman (Org.), Atualizações em Informática 2007, pp 263–326, SBC 2007.

Prates, R. O., de Souza, C. S., and Barbosa, S. D. (2000). A method for evaluating the communicability of user interfaces. ACM interactions, 7(1):31–38.

scitools.com. Java editor with reverse engineering, code navigation and automatic documentation. Scientific Toolworks Inc. http://www.scitools.com/uj.html.

Shneiderman, B. and Mayer, R. (1979). Syntactic/semantic interactions of programming behaviour: A model. Int´l Journal of Comp. and Information Sciences, 8(3):219–238.

Sim, S. E. (2003). A Theory of Benchmarking with Applications to Software Reverse Engineering. PhD thesis, Dept. of Computer Science. University of Toronto, Canada.

Soloway, E. and Ehrlich, K. (1986). Empirical studies of programming knowledge. Readings in artificial intelligence and software engineering, pages 507–521.

Storey, M.A. (2005). Theories, methods and tools in program comprehension: Past, present and future. In: Proc. of the 13th Int´l Workshop on Program Comprehension, pp 181–191, USA. IEEE Computer Society.

Storey, M.A., Best, C., Michaud, J., Rayside, D., Litoiu, M., and Musen, M. (2002). SHriMP views: an interactive environment for information visualization and navigation. In Extended Abstracts CHI 2002, pp. 520–521, USA. ACM Press.

Storey, M., Wong, K., and Müller, H. (2000). How do program understanding tools affect how programmers understand programs? Sci.Comp. Prog., 36(23):183–207.

von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. Computer, 28(8):44–55.