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,rprates,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 a comprehension tool may support or hinder the strategy used by the programmer. Furthermore, which and how information is presented to the programmer may influence the effectiveness of the comprehension process. Within this context, this paper shows the assessment of the user interface of selected program comprehension tools which support different comprehension strategies. A communicability evaluation method is conducted with users executing typical comprehension tasks using SHriMP and Understand for Java©. The communicability evaluation method captures communicative breakdowns during the user-system interaction, i.e., it identifies difficulties experimented by users in understanding the comprehension tool during the interaction. Our goal is to collect indicators on the quality of user-system interaction based on relevant breakdowns that took place during the evaluation. We show some findings that may help design~~ing~~ better tools.

Keywords: Program comprehension tools, user interfaces, communicability test.

(Received July 18, 2008 / Accepted September 18, 2008)

1 Introduction

Understanding an already built software system is necessary because those systems are in permanent maintenance and evolution. Mayer 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 [15]. 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 Furthermore, large systems and outdated documentation usually add to the complexity of program comprehension task.

Several researches have been developed to minimize this problem. Tools and theories are being developed to better support the program comprehension task. Storey [11] presents a study about theories and tools around the program comprehension activities. The study also presents several avaliable features of tools with a projection of several necessary features the tools should have.

Program comprehension tools are designed according to the 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 [14].

The functionalities supported by a tool, e.g., navigation, queries and multiple views are commonly implemented within a graphical user interface. It is well-know 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 [12] and *Understand for Java* [7]. 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 evaluation was done using the Communicability Evaluation Method [6], a qualitative method that identifies user-system communicative breakdowns during interaction.

The remaining of this paper is organized as follows. Section 2 presents some background on program comprehension. Section 3 reviews the comprehension tools that will be used in the experiments. Section 4 presents the *Communicability Evaluation Method* used to appreciate the quality of the selected tools' interfaces. Section 5 describes the experiment design and section 6 discusses the experiment results. Finally, ~~the~~ section 7 presents our final remarks.

2 Program comprehension

The program comprehension process requires the developer to produce mental models about the source code organization (high-level abstractions), about how the functionalities were implemented (low-level abstractions), about the implemented requirements, etc.

Understanding an already built software system is necessary because those systems are in permanent maintenance and evolution. Mayrhauser and Vans [15] 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.

2.1 Comprehension strategies

There are many ways to comprehend a program. Some strategies guide the way a programmer proceed with the comprehension process [14]. We review here some strategies.

Top-down comprehension is a process to understand by reconstruction the knowledge domain and mapping it to the source code [1]. The process starts with a general view of the system, which is successively refined.

Bottom-up comprehension is an understanding process that starts from the source code, and successively, software artifacts are grouped into higher-level abstractions. The process continues until a global view of the system is obtained [8].

Systematic comprehension is a process to read the code following the control and data flow. The as-needed approaches focus on control and flow for specific parts one intends to understand. These approaches seem to be hard to work effectively on very large systems. On the other hand, they may seem to be widely used for focused parts of the system [4][10].

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. The programmer repeatedly asks a question, conjectures an answer and searches throughout the code to validate or to reject that conjecture. During this inquiry process the programmer gains incremental understanding of the software [3].

Integrated comprehension is a process that combines top-down, bottom-up and knowledge based approaches into a single strategy. Comprehension tasks can be performed freely at any abstraction level, and hence any strategy can be used depending on the current level[15].

A programmer can understand an implementation using any of these strategies, either manually or using some comprehension tool to facilitate the process. However, there are some factors that may influence the choice of the strategy: diversity of types of programs being comprehended, specific aspects of a comprehension task and the programmers' preferences. Experienced programmers find it easy to determine which strategy is better for specific programs and tasks. Indeed, these factors explain the diversity of strategies. This helps to explain why program comprehension tools are difficult to design and why they may facilitate the strategy choice or even impose an strategy not always useful for the task at hand.

The next section describes the selected tools that support code navigation for program understanding or maintenance.

3 The Comprehension Tools

Software comprehension could be an easier task if programmers had up-to-date and high-quality documentation. Unfortunately, source code is, frequently, the only source of reliable information that can help comprehension. In this situation, comprehension tools play an important role in the comprehension process. Tools may use graphical and textual representation to enhance the comprehension activity. Facilities to enhance navigability in the available information are generally also provided. Many tools present relevant information using structural graphs where nodes represent program entities and arcs represent relationship between these entities. Tools also use diagrams to present information about program dynamics, such as, control flow diagrams and method call graphs.

The tools *SHriMP* and *Understand for Java* are presented in the next sections.

3.1 SHriMP

SHriMP (Simple Hierarchical Multi-Perspective) [13] 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 ~~use~~ to explore and to browse Java programs in the Eclipse IDE using the Creole plug-in. It also may be used to visualize ontologies in Protégé with the Jambalaya plug-in, and used to visualize flow diagrams combined with the IBM Websphere Studio Workbench [12]. In this paper we use the Creole plug-in, shown in figure 1.

SHriMP has several functionalities organized in windows encapsulated within tabbed panes. The main window shows the system architecture built with coloured nodes and arcs. Nodes represent the program's entities and can be expanded to show the nested entities or collapsed to hide those entities. The arcs represent the relationship between the entities, and they can encapsulate several relationship occurrences of the same type between two entities. Nodes and arcs can be hidden or visible, depending on the state of the Node Filter and Arc Filter panes. Other visualization options and functionalities are available in the quick view bar and in the toolbar. 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 query for some specific entities in the program.

3.2 Understand for Java

Understand for Java©[1] [7] is a source code analyzer that helps programmers 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. Figure 2 shows the main window of the tool.

The tool interface is divided in three main areas: filter area (upper-left), information area (lower-left), document area (right). Filters enable the user to select, based on the entities' types, specific entities in the source code. The presentation of the entities' information also 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 either 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. Syntax highlighting is used to facilitate source code visualization.

4 The Communicability Evaluation Method

The Communicability Evaluation Method [6] is a qualitative method for user interface evaluation, based on Semiotic Engineering theory [6, 2, 5]. ~~The~~ Semiotic Engineering 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 [2]. ~~Communicability is a distinctive~~ quality of interactive systems that communicate efficiently and effectively to users their underlying design intent and interactive principles.

The Communicability Evaluation Method 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 [6]. The data analysis is comprised of 3 steps:

1. **Tagging**. Evaluator plays back the user interaction

[1]Commercial tool developed by Scientifc Toolworks, Inc.

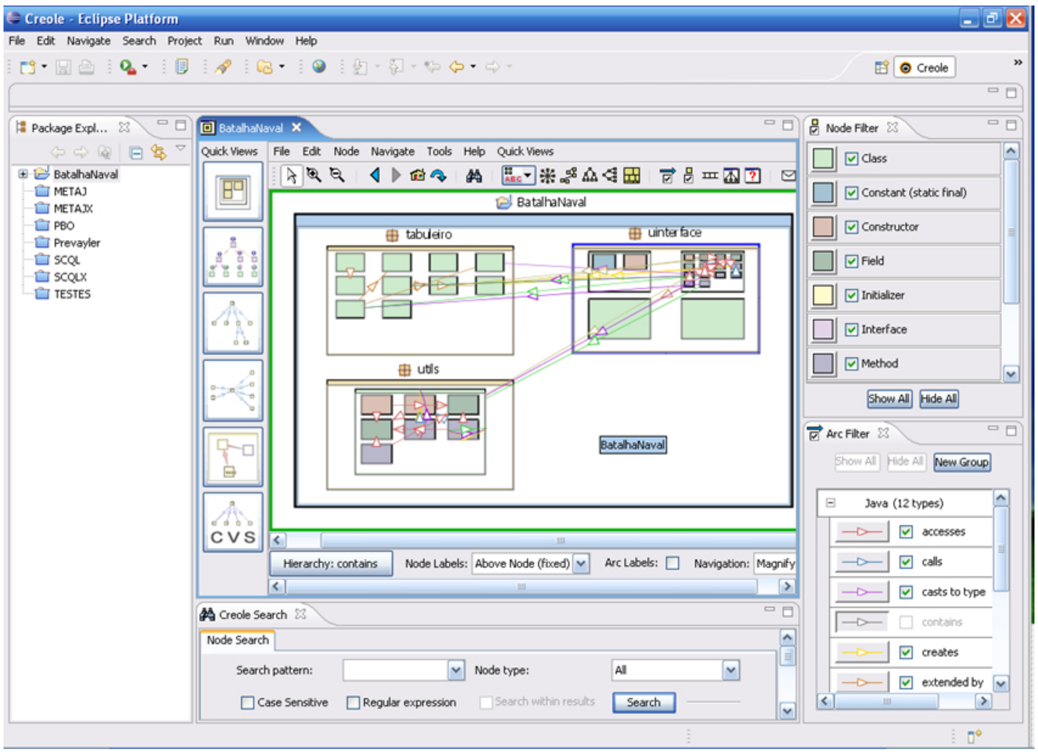
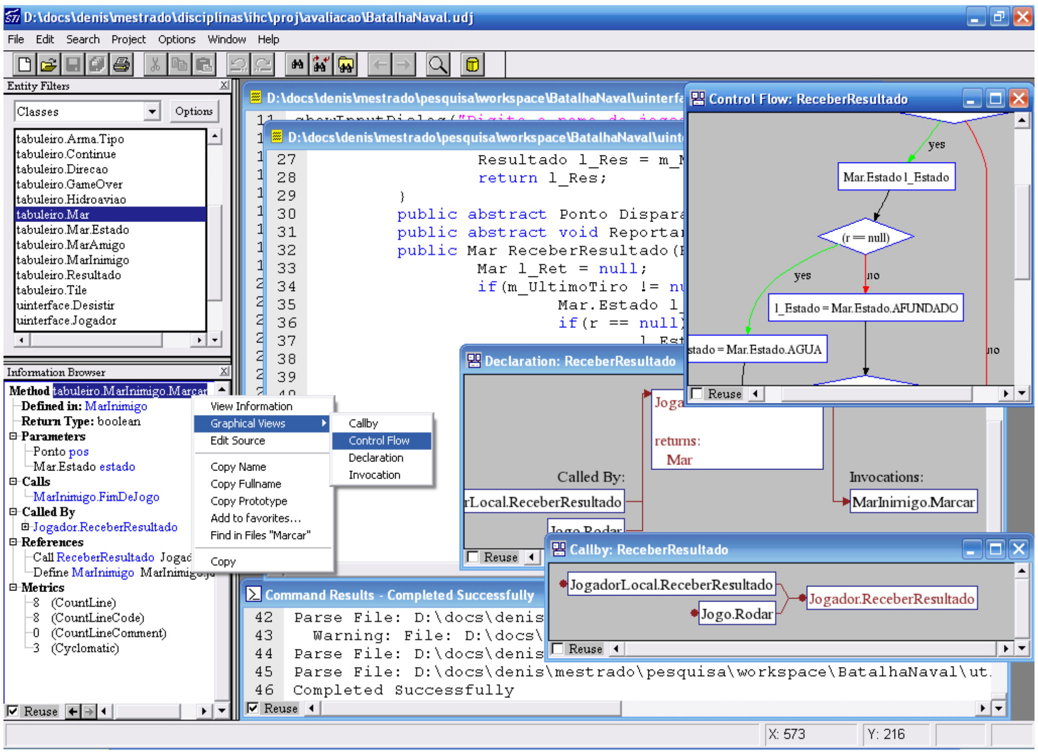


Figure 1: Creole window (SHriMP plug-in for Eclipse).



```
                Resultado l_Res = m_M
                return l_Res;
        }
        public abstract Ponto Dispara
        public abstract void Reportar
        public Mar ReceberResultado(P
                Mar l_Ret = null;
                if(m_UltimoTiro != nu
                        Mar.Estado l_
                        if(r == null)
```

```
Parse File: D:\docs\denis
  Warning: File: D:\docs\
Parse File: D:\docs\denis
Parse File: D:\docs\denis\mestrado\pesquisa\workspace\BatalhaNaval\ut.
Completed Successfully
```

Figure 2: Understand for Java window.

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!***.

2. **Interpretation**. In this step, the evaluator tabulates the problems identified by the ~~break downs.~~ 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 Semiotic Engineering.

3. **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.~~

5 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.

5.1 Experiment Organization

One of the factors that influenced 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. Understand for Java© (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 ~~needed to analyze~~ high-level abstractions (e.g. models and diagrams) and 3 tasks that ~~needed to analyze~~ low-level abstractions (e.g. source code, flow control, method calls). Each partcipant used both tools, performing 3 tasks of the first group with one tool and other 3 tasks of the second group 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 (intermed.)	SHriMP			UJ		
P2 (advanced)	SHriMP			UJ		
P3 (intermed.)	UJ			SHriMP		
P4 (advanced)	UJ			SHriMP		

The reason for this choice was to enable the identification and comparison of strong and weak points of each tool.

During the test, each participant´s session lasted approximately an hour and a half where each participant was instructed about ~~test's goals,~~ had a brief training on the tools, received the material for the test, performed the tasks and participated in a post-test interview.

5.2 Participant Instructions

The first ~~phase~~ of the experiment was ~~an explanation~~ to ~~the~~ participants ~~about all phases~~ of the evaluation. Initially, participants were advised that the only purpose of the test was to evaluate the effectiveness of the selected tools in the program comprehension process. The participants were given an overall explanation about the functionalities of the selected tools. They were ~~aware~~ their interaction was being recorded~~,~~ using a screen capture software and gave permission for it to be done. They were also informed their identities would remain anonymous and they could had access to the data collected during their tests at any time.

5.3 Training

Before conducting the test itself, the basic functionalities of the selected tools were quickly demonstrated to the participants. This demonstration aimed ~~to level the knowledge about the tools of all participants and to minimize the initial interaction problems~~ (previously observed in another pilot test). ~~So, the~~ participants could concentrate specifically in performing the comprehension tasks.

The demonstration took place through the visualization of a small Java ~~program implementing a~~ banking application. The program has 5 files containing 5 classes and 165 lines of code. During the visualization, the main functionalities of each tool were presented to the participants.

5.4 Tasks

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:

> You are a Java developer. You were recently hired by a software company to maintain a system already developed by another person no longer in the company. The referred program is the Battleship game. Before beginning the maintenance you need to understand the program. ~~As you know you have a~~ short time, you ~~should decide~~ to search for and test some program comprehension tools to facilitate your comprehension process. You ~~find~~ two tools: *Creole* (SHriMP plug-in for Eclipse - *freeware*) and *Understand for Java*© (a commercial tool developed by *Scientific Toolworks, Inc.* - evaluation version). In order to evaluate the tools, you will perform some comprehension tasks. At the end of the evaluation, you hope to have a better comprehension of the system, and you are expected to be ready to maintain the system as requested by your boss.

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 choice of the 6 tasks followed the ideas proposed in~~ [9], ~~where a benchmark of typical tasks during program maintenance was suggested.~~ The chosen tasks ~~were to answer~~ the following questions:

1) Navigate the views created from source code ~~by the tool~~ and try to get an overall ~~architecture of the program~~. Draw a diagram constituted of interconnected blocks that represents ~~the concrete~~ architecture.

2) In which blocks of the previous diagram, ~~where~~ were the rules of the game implemented?

3) In the implemented 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 before starting the game? ~~If the size is fixed, h~~ow can the implementation be changed to support the redefinition before the game starts~~, and vice-versa, 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 of how to change the program, for example, how new kinds of ships could be added to the program. However, the participants did not perform the changes, they only listed the necessary steps to implement the new requirement.~~

5.5 Post-test

At the end of the experiment, the participants were interviewed ~~to provide their own views on the tools: easiness of use, satisfaction with interface layout, and information absorption. The goal of the interview was to complement some issues observed by the evaluator during the task execution.~~ Participants were also asked about their interest in the tools, and ~~to verify~~ whether they would use or suggest ~~the tools~~ in ~~some~~ situation. And finally, ~~questions to verify if~~ they believed the tools met their goals.

5.6 Developed Material

Two documents were written to guide the test execution. The first document, delivered to each participant, contained a description of the evaluation goal, a short description of the tools, the execution scenario and the tasks. The other document, for the evaluator, contained all the experiment script, with all the steps the evaluator should follow to conduct the experiment as predefined, as well, all questions to be asked during the interview.

6 Results

The first step of the analysis intended to identify strategies adopted by users at each task, next the Communicability Evaluation Method 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 Understand for J~~j~~ava 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 3 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.

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. This utterance occurred mainly in the interaction with the SHriMP interface, in the begining of ~~the~~ Task 1 executed by ~~the~~ Participants 1 and 2, and during the execution of the Task 4 and Task 5 by ~~the~~ Participants 3 and 4.

The second situation of the ***What's this?*** tags 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. This interaction pattern occured mainly in the ongoing execution of the Tasks 1 and 2 by the Participant 1 and during execution of the Tasks 2 and 3 by the Participant 2, when it was requested to the participant a high level information about the program implementation.

Another frequent communication breakdown was identified with the utterance ***Oops!***. It was mainly identified when the user opened up a view, either a diagram or the source code, and rapidly identified that there was no useful information for its task on that view, and then he went back to the previous view. This breakdown was more frequent when using *Understand for Java*, when the user ~~frequently~~ was tempted to open a view expecting something useful, and ~~he~~ realized that either the diagram or the source code was not related with the task at hand. The less frequent is this breakdown, the more directly users get desired information. There was a user that commented that the *Information Browser* of *Understand for Java*, that presents textual information (e.g. metrics, declarations, uses, dependencies, etc.) about the entities was the only really useful functionality in that tool.

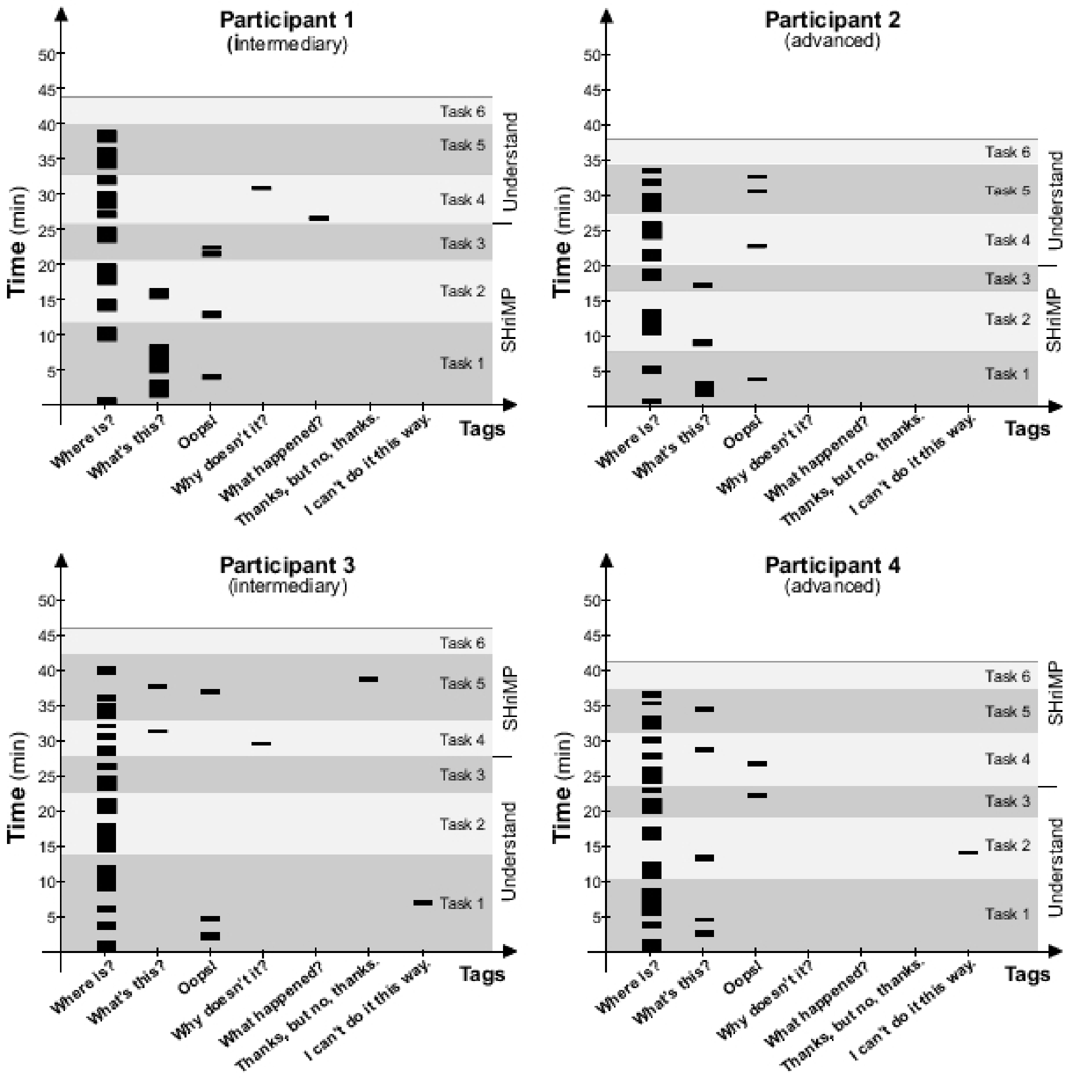


Figure 3: Tagging results

Communication breakdowns, such as, ***What happened?***, ***Why doesn't it?*** and ***I can't do it this way*** happened when the users did not understand or did not realize~~d~~ how a functionality actually worked. Since, they happened with very little frequency ~~and~~ they did not point to any conclusive indicators. These breakdowns are very important to be analyzed because if they were more frequent, the program comprehension would be severely hindered ~~caused by the focus of the user possibly moving from the comprehension task to the interaction with the tool.~~

It is interesting to notice a communicative breakdown identified during 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 the participant 3 was executing task 5. This participant needed to navigate throughout the source code to understand specific details of the implementation, 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, and one of the main features of comprehension tools are the visualizations they provide the user with. This operational breakdown, could lead to a strategic breakdown leading the user to decline the system as a whole.

Based on these observations we can discuss some features of the evaluated tools:

- SHriMP has shown to have an effective schematic visualization of the program architecture (top-down), nonetheless, it has shown deficiencies to provide a better view of the source code.
- Understand for Java is a complementary to SHriMP. It provides effective views of the source code, but does not provide good overall views of the system.

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. Understand for Java 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 Understand for Java 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.

7 Final Remarks

This paper presented an evaluation of the user interface of two comprehension tools: SHriMP and Understand for Java. The user interfaces were evaluated using the Communicability Evaluation Method. 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 Understand for Java 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 [14] 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

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

[2] de Souza, C. S. *The semiotic engineering of human-computer interaction*. The MIT Press, Cambridge, MA, 2005.

[3] Letovsky, S. Cognitive processes in program comprehension. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 58–79, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[4] Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E. Mental models and software main-

tenance. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 80–98, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[5] Prates, R. O. and Barbosa, S. D. J. Introdução à teoria e prática da interação humano computador fundamentada na engenharia semiótica. In *T.Kowaltowski e K. K. Breitman (Org.)*, Jornada de Atualização em Informática do Congresso da Sociedade Brasileira de Computação, pages 263–326, Rio de Janeiro, Brazil, Julho, 2007. SBC 2007.

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

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

[8] Shneiderman, B. and Mayer, R. Syntactic/semantic interactions of programming behaviour: A model. *International Journal of Computer and Information Sciences*, 8(3):219–238, 1979.

[9] Sim, S. E. *A Theory of Benchmarking with Applications to Software Reverse Engineering*. PhD thesis, Department of Computer Science. University of Toronto, Toronto, Canada, October 2003.

[10] Soloway, E. and Ehrlich, K. Empirical studies of programming knowledge. *Readings in artificial intelligence and software engineering*, pages 507–521, 1986.

[11] Storey, M.-A. Theories, methods and tools in program comprehension: Past, present and future. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Storey, M.-A., Best, C., Michaud, J., Rayside, D., Litoiu, M., and Musen, M. SHriMP views: an interactive environment for information visualization and navigation. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 520–521, New York, NY, USA, 2002. ACM Press.

[13] Storey, M.-A. D. and Müller, H. Manipulating and documenting software structures using SHriMP views. In *Proc. of the International Conference on Software Maintenance (ICSM '95)*, pages 275–284, Oct. 1995.

[14] Storey, M.-A. D., Wong, K., and Müller, H. A. How do program understanding tools affect how programmers understand programs? *Sci. Comput. Program.*, 36(2-3):183–207, 2000.

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