The Importance of Cognitive and Usability Elements in Designing Software Visualization Tools

Glauco de F. Carneiro

Software Engineering and Applications Group
(GESA/NUPERC)
Salvador University – Bahia/Brazil
glauco.carneiro@unifacs.br

Manoel Mendonça

Software Engineering and Applications Group
(GESA/NUPERC)
Salvador University - Bahia/Brazil
mgmn@unifacs.br

Keywords: POP-II.B program comprehension, POP-III.D Visualization POP-V.B observation

Abstract

Modern IDEs offer built-in support for developing plug-ins. More recently, we have seen a growing number of plug-ins that offer non-conventional software visualization interfaces. They usually aim to help programmers to understand unfamiliar source code by representing it in visual structures such as trees, scatter-plots or graphs. Although very attractive visually, we need to know more about the effectiveness of these interfaces in conveying information to software engineers. In this paper, we discuss some concepts and guidelines regarding the requirements of visualization tools for software comprehension as well as the set-up of an infrastructure to empirically evaluate how useful are those tools in supporting software comprehension activities.

1. Introduction

Software comprehension is the basis for software maintenance activities. Extracting information from industry strength software systems is difficult due to their size and complexity. As a single programmer is only capable of understanding a small portion of a large system, more and more design anomalies are introduced into a system as it evolves over time. As a result, the quality of a system maintained by different actors tends to decay over time (Lehman, 1996), mainly because changes are executed without an incomplete understanding of the whole (Parnas, 1994).

Visualization has been pointed out as a possible solution for supporting better understanding of complex systems. The cognitive process of human beings is more intuitive, effective and efficient when supported by visual resources such as images, drawings and signs (Tergan and Keller, 2005). Graphics communicate knowledge visually rather than verbally and, when well designed, they can transfer large amounts of complex information for programmers. As it goes the saying "a picture is worth a thousand words", graphics show rather than tell (Tidwell, 2005).

It is no surprise that the use of visualization for software comprehension is becoming a participant of interest in the software engineering community. It has been used by the scientific community and by the industry in different stages of the software cycle, as illustrated by the list of 58 tools surveyed in (SGC SmallWiki, 2008). However, most of the tools used nowadays uses single visual paradigm (Code Surfer, 2008) (Sotograph, 2008). This limits the full potential of the use of visualization in software comprehension, because software is usually understood from multiple perspectives.

Our work intends to explore how visual paradigms or multiple views (Baldonado et al, 2000) can be used in software visualization activities. In particular, we want to explore: (a) information visualization key principles that are not yet used in many of the software visualization tools (Card et al., 1999) (Shneiderman, 1992); (b) the usability and cognitive principles used in building a multiple visualization infrastructure and how it maps to software comprehension principles; (c) the design of an environment to capture the use (as opposed as to execution traces) of visualization infrastructures; and (d) empirical studies to characterize tool-supported comprehension activities.

In order to study (characterize and evaluate) how helpful visual interfaces are in supporting software engineering tasks, we developed a prototype experimental environment that integrates software

visualization interfaces into Eclipse and allows us to collect data directly from the interface use. This environment currently offers four non-traditional visualization interfaces and is able to log data from primitive operations executed on them. These data capture interactions with both the visual areas and filtering controls made available by the infrastructure.

We are now using this experimental infrastructure to execute a series of observational studies to characterize how effective and efficient are visual paradigms in supporting software comprehension activities. This paper describes our visualization infrastructure design, the experimental environment setup and reports the initial results we have obtained on our pilot studies.

The remainder of this paper is organized as follows. Section 2 briefly describes key design principles for visual interfaces with an emphasis on multiple views. Section 3 presents our cognitive model for software comprehension with multiple views. Section 4 discusses what we have done so far to set up an extensible visualization-based software comprehension environment. Section 5 explains the infrastructure we built to execute observational studies on how programmers deal with multiple views in supporting software comprehension activities. Section 6 presents our conclusions and future work.

2. Visualization for Software Comprehension

The use of software visualization allows humans to directly analyze multiple aspects of complex problems in parallel (Stako et al., 1998). This analysis takes place in the context of the understanding process and is often associated with cognition. It involves process such as learning, problem solving, perception, intuition, and reasoning (Les et al., 2008). All those elements constitute key design principles for building multiple views so that they can provide more effective software understanding.

2.1. Design Requirements for a Software Visualization Interface

Design principles for visual interfaces come from practical experience as well as psychological theory. People are far more effective thinkers when supported by appropriate cognitive tools. Combining a computer-based information system with flexible human cognitive capabilities, such as pattern finding, can be an effective way to aid the human cognitive process.

A software visual interface should address a set of requirements to match the human cognitive process. Shneiderman (1992) presents the "Eight Golden Rules of Dialog Design" to accomplish this goal:

(1) *Strive for consistency*: consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus, and help screens; and consistent commands should be employed throughout.

(2) *Enable frequent users to use shortcuts*: as the frequency of use increases, so do the user's desires to reduce the number of interactions and to increase the pace of interaction.

(3) *Offer informative feedback*: for every operator action, there should be some system feedback. For frequent and minor actions, the response can be modest, while for infrequent and major actions, the response should be more substantial.

(4) *Design dialog to yield closure*: sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the user the satisfaction of accomplishment, and an indication that the way is clear to prepare for the next group of actions.

(5) *Offer simple error handling*: As much as possible, design the system so the user cannot make a serious error. If an error is made, the system should be able to detect the error and offer simple, comprehensible mechanisms for handling the error.

(6) *Permit easy reversal of actions*: This feature relieves anxiety, since the user knows that errors can be undone; it thus encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.

(7) *Support internal locus of control*: experienced users strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.

(8) *Reduce short-term memory load:* The limitation of human information processing in short-term memory requires that displays be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.

These principles were derived heuristically from experience and are applicable in most interactive systems after properly refined, extended and interpreted.

2.2. A Reference Model for Information Visualization

Figure 1 presents a reference model for information visualization and provides a high-level view of the (information) visualization process. The model assumes a repository of raw data. This data has to undergo a set of transformations to produce a meaningful visualization scenario for a user. Data transformations comprise filtering of raw data, computation of derived data as well as data normalization. These steps result in a set of transformed data in a unified structure. Visual transformations map the pre-processed data onto a corresponding visual structure. From this visual structure, a set of views can now be generated and explored by the user. A key point of this model is that data transformations, visual mappings and view transformations should be as interactive as possible. In other words, it is not only necessary that a visualization tool produces a visual scenario, but also that a user can interactively interfere in all aspects of this scenario with just a few mouse clicks. The response time between interactions and rendering should be instantaneous for all practical purposes. This interactivity is essential to support previously listed requirements such as easy reversal of actions and internal locus of control.

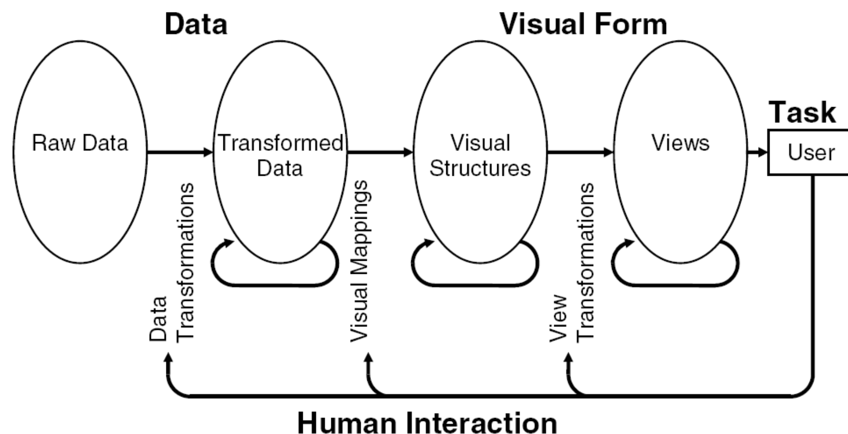

Figure 1: Reference Model for Visualization (Card et al., 1999)

This model has been used as a reference by information visualization tools such as Spotfire (2008) and TouchGraph (2008). Figure 2 shows a snapshot of a visualization scenario made available on-line by Spotfire (2008). The regions marked as A and B present data views selected by the user. Region D presents widgets like checkboxes and range sliders to filter what should be shown in the views, implementing the data transformations illustrated in Figure 1. At any moment the user can change the variables associated with visual attributes like colours, x-axis, y-axis, and bar or circle sizes in the views. This accomplishes the visual mapping illustrated in Figure 1. Range bars, such as the one on the bottom of Region A, can be use to zoom or pan over a given diagram, resulting in the view transformation illustrated in Figure 1. In all those cases, the time overlapped between a user action and a view update is practically instantaneous, much like those in a video game.

Other features enrich the user cognitive experience. Region C shows a text to contextualize the information presented in the overall scenario. Region D permits easy reversal of actions and offers informative feedback about operations done by the programmer.

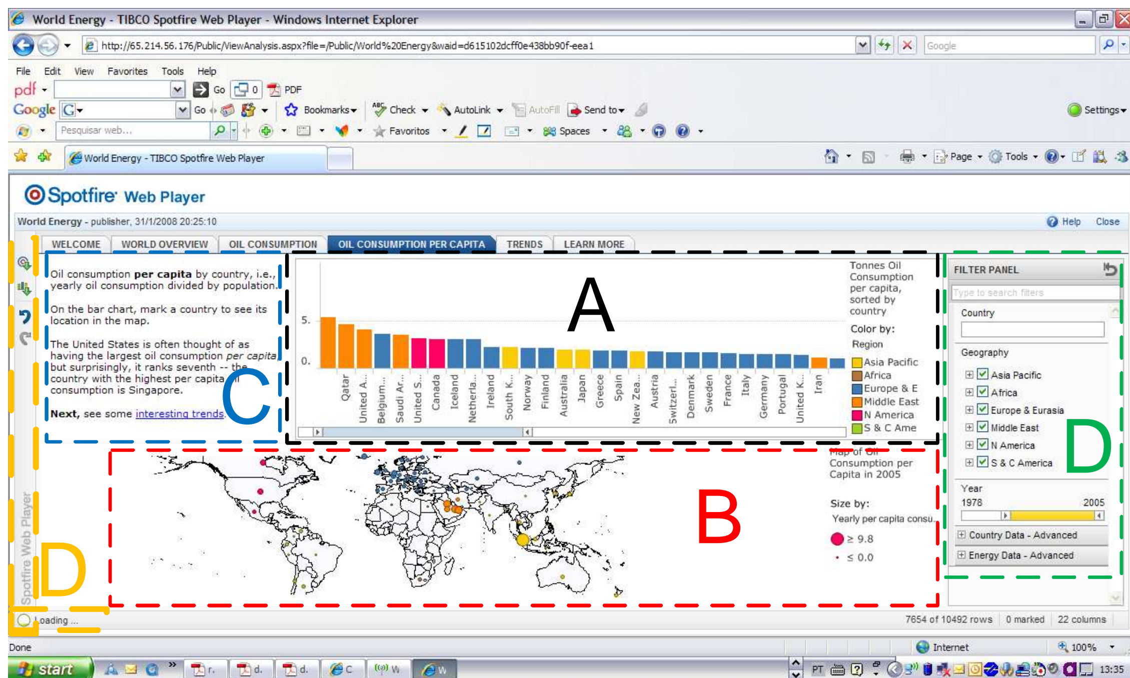

Figure 2: Information Visualization Snapshot (Spotfire, 2008)

Figure 1 reference model and Shneiderman's eight rules present an excellent set of criteria to evaluate any of today's software visualization tools. Most of them would not pass <u>all</u> those criteria. We add to those general <u>information visualization</u> criteria the principles summarized by Hundhausen et al. (2002) specifically for <u>software visualization</u>:

a) Epistemic Fidelity (Roschelle, 1990) (Hundhausen, 1999) emphasizes the value of a good denotation match between the graphical representation and the expert's mental model. The higher the fidelity of the match, the more robust and efficient is the transfer of that mental model to the viewer of the visualization, who decodes and internalizes the target knowledge. Its key assumption is that graphics have an excellent ability to encode an expert's mental model of an algorithm in a visual metaphor, leading to a robust and efficient transfer of that mental model to the viewer.

b) Dual-coding (Mayer and Anderson, 1991) proceeds from the assumption that cognition consists largely of the activity of two partly interconnected but functionally independent and distinct symbolic systems. One encodes verbal events (words) and the other encodes non-verbal events (pictures). According to Mayer and Anderson (1991), visualizations that encode knowledge in both verbal and non-verbal modes allow viewers to build dual representations in the brain, and referential connections between those representations. As a consequence, such visualizations facilitate the transfer of target knowledge more efficiently and robustly than do visualizations that do not employ dual-encoding.

c) Individual Differences theory (Cooper, 1997) asserts that measurable differences in human abilities and styles will lead to measurable performance differences in scenarios of software visualization. For example, within the scope of epistemic fidelity theory's knowledge transfer model, individual differences with respect to learning style (Riding and Rayner, 1998) might enable some individuals to decode visualizations more efficiently and robustly than other individuals.

d) Cognitive Constructivism (Letovsky, 1986) asserts that individuals actively construct new understanding by interpreting new experiences within the context of what they already know. Its emphasis on active learning has important implications for the effective use of software visualization.

In particular, it suggests that individuals do not stand to benefit from the technology by merely passively viewing visualizations, no matter how high the level of their epistemic fidelity. Instead, software visualization users must interact with the visual scenario in order to benefit most from it. The scenario is not only a conveyer of knowledge, but also a tool for knowledge construction.

2.3. Multiple View Systems

In a multiple view system, two or more distinct views can be used to support the investigation of a given conceptual entity. The example in Figure 2 uses a dual view system. Multiple view systems – systems that use two or more distinct interfaces – have been proposed to support the investigation on a wide range of information visualisation topics (Baldonado et al., 2000). North and Shneiderman (1997) observe that multiple view systems offer the following advantages: improved user performance, discovery of unforeseen relationships, and unification of the desktop.

Two or more views are distinct if they allow the user to learn more about different aspects of the conceptual entity. Three important issues for multiple views are: selection of views, presentation of views, and interaction among views.

Selecting an appropriate set of views is indeed an important step of the comprehension process. It should be driven by the peculiarities of a given maintenance task. For example, the appropriate set of views to evaluate a mediator pattern (Gamma et al., 1995) implementation would probably differ from the one to support the detection of the bad smells Long Method (Fowler, 1999).

Given an appropriated set of views to support a task, it is time to decide the types of presentation that are useful to gather information related to a specific task. It depends on the programmer to analyze the views sequentially or simultaneously. This choice is strongly influenced by his knowledge of the domain and how much of the program's functionalities he already knows. This is based on the Letovsky's (1986) cognitive constructivism theory already mentioned in the Section 2.1.

Each single view may have independent affordances, e.g. selection capabilities or navigation functionality such as pan and zoom. These affordances can be tied together so that actions in one view may have an effect in another view (Baldonado et al., 2000). Linked interactions between the views consist of the navigational slaving. It involves synchronizing associated views when a navigation action is performed on any one of a set of linked views.

The use of multiples views is a promising approach for software comprehension. Software is complex and usually understood from multiple perspectives. However, multiple view systems are highly challenging to design. They often use sophisticated coordination mechanisms and layout and, in addition, subtle interactions among the many dimensions of the design space complicate design decisions (Baldonado et al., 2000). Deciding when and how to apply multiple views to information visualisation problems involves balancing a set of design tradeoffs. On the one hand, multiple views can provide utility in terms of minimising some of the cognitive overheads engendered by a single, complex view of data. On the other hand, multiple views can decrease utility when added to a system, both in terms of higher cognitive overheads (e.g. for context switching) and in terms of increased system requirements.

These challenges are very much present, in the case of software visualization. We foresee the following requirements for a multiple view software visualization system:

1. One shall have a small but significant set of views;
2. The views shall match some typical software comprehension needs such as visualization of hierarchical structures, relationships between software entities and artefacts, and entities attributes (e.g. size and complexity).
3. The views shall complement each other fulfilling typical software comprehension needs.
4. The views shall be coordinated in a way that actions in one view is reflected in the others,
5. The presented views shall be easily configurable into typical or user preferred software comprehension scenarios.

6. The views shall meet the general purpose information and software visualization requirements established by Shneiderman (1992), Hundhausen et al. (2002) and Baldonado et al. (2000), as previously presented in this Section.

3. A Cognitive Model for Software Comprehension with Multiple Views

We aim to build a system with multiple views, multiple coding and interactions mechanisms to allow the user to configure the most appropriate visual scenario to a given software comprehension task. This deals with individual differences and fosters cognitive constructivism, matching Hundhausen's guidelines on the use of software visualization for code understanding (Hundhausen et al. 2000). This infrastructure enables the programmer to select views that are most suitable to gather information from source code and that can be gradually adjusted during the comprehension process.

The key challenge of this study is to capture, identify and understand the heuristics applied by programmers during the comprehension process. For that, we implemented a functionality to log data from primitive operations executed by the programmers on the system. These data capture interactions with both the visual areas and filtering controls (widgets) made available by the infrastructure. We believe that this logging functionality can be used as an experimental platform to - at least partially - capture the heuristics applied by programmers while performing specific software comprehension tasks. The heuristics used by experienced or successful programmers – those who performed well in controlled environments - could then be used as reference cognitive models.

These cognitive models can them be reused to guide novel, inexperienced and unsuccessful programmers. Due to individual differences, programmers performing the same activity will apply adjustments to the scenarios. We aim to identify the most successful heuristics and the corresponding scenarios that were used by them.

Figures 3a and 3b present the approach that we envision, where visual scenarios are built by experienced programmers by combining and gradually adjusting views and code metrics (Figure 3a). The heuristics applied during this process is captured and can be passed on as a sequence of suggested visual scenarios to novel programmers (Figure 3b).

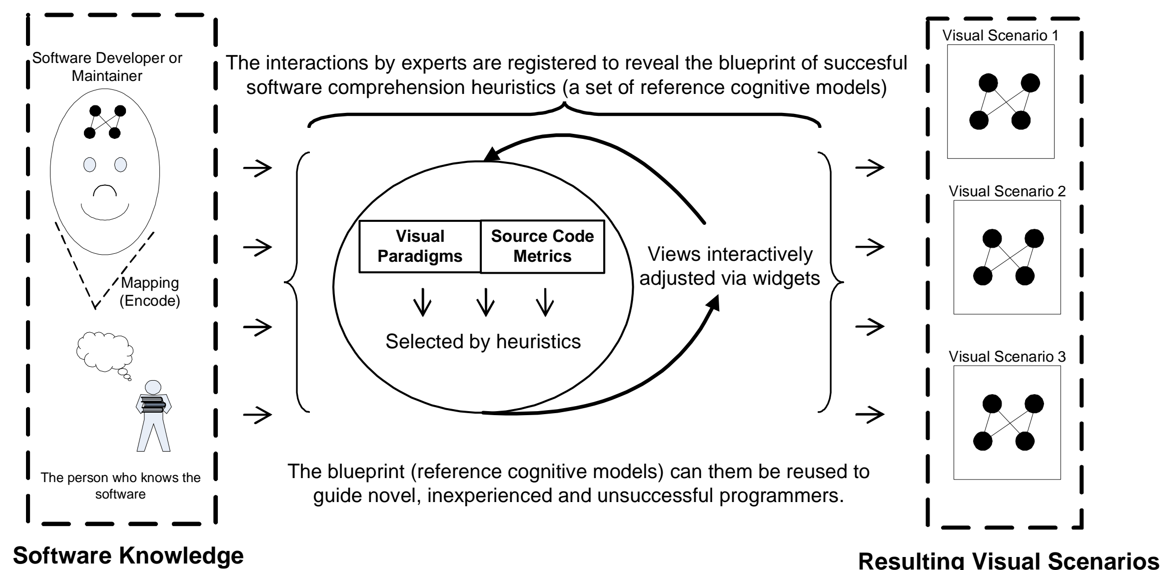

Figure 3a: A Cognitive Model for Software Comprehension - visual scenarios built by experienced programmers

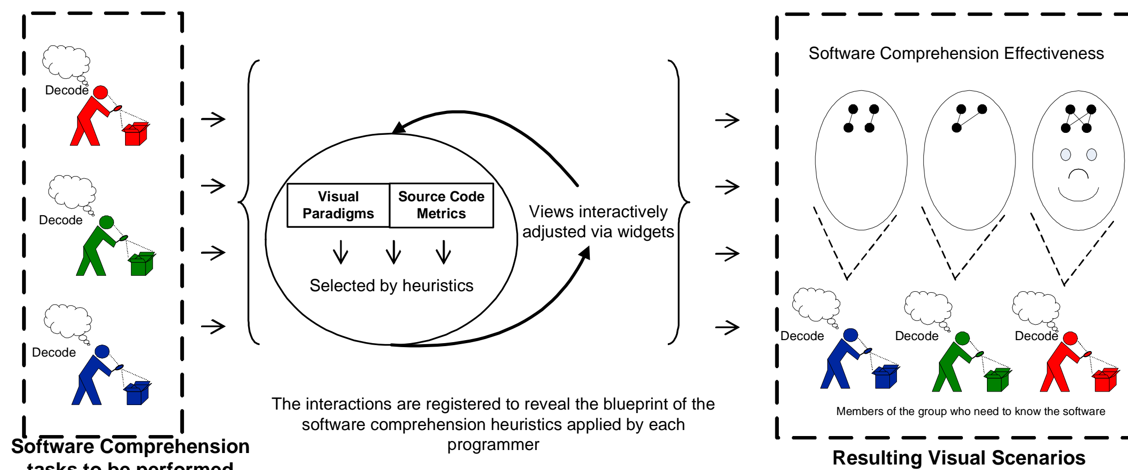

Figure 3b: A Cognitive Model for Software Comprehension - visual scenarios built by novel programmers

4. Building an Extensible Infrastructure

In 2007 we developed a software visualization interface called SourceMiner (Carneiro et al. 2007). This interface uses treemaps (Shneiderman 1992) to represent and explore large volumes of source code. The interface shows the code structure as a hierarchy of recursively nested rectangles representing software modules. Source code metrics such as cyclomatic complexity and module size can be visually represented by attributes like rectangle color and size.

More recently, we decided to implement this interface as an Eclipse plug-in and ceased the opportunity to integrate it with other source code visualization interfaces (Carneiro et al., 2008a). We ended up producing an extensible infrastructure that can integrate new open code plug-ins for source code visualization. Currently we have integrated SourceMiner´s treemaps with the University of Lugano's X-Ray (polymetric and graph dependency views) (Lanza e Ducasse 2003) (Malnati 2007). Our final goal is to study the use of static source code visualizations in software maintenance tasks.

Considering the elements, concepts and the proposed model, this Section explains the overall visualization structure design implemented in our extensible infrastructure (an Eclipse plug-in). We then map it to the requirements that it intends to address. Figure 4 gives a high level view of this infrastructure.

In accordance with Figure 4, we used Eclipse's Java Development Tooling (JDT) to build the proposed infrastructure. JDT provides APIs to manipulate Java source code, detect errors, perform compilations, and launch programs. Eclipse's JDT has its own Document Object Model (DOM) with the same purpose as the well-known XML DOM. The AST can be used to exam the structure of a compilation unit down to the statement level. Based on the information available from the AST, it is possible to build the model to make up the views. The user can than select the appropriate set of views to accomplish a given task.

Figure 5 presents a snapshot from SourceMiner. It exhibits a possible scenario that comprises two of the four plug-in views. We are still planning to implement, adapt and integrate a number of other source code visual paradigms to our infrastructure (Carneiro et al. 2008b). The views are arranged side by side and marked as (B) and (C) in the figure. It is up to the programmer to layout the views on the screen to cope with a specific maintenance task. As seen in the example, these views can be complemented by other views, such as Eclipse's editor (E) and package explorer (A). This fulfills Baldonado's (2002) selection and presentation criteria discussed previously.

Using controls like range sliders and checklists – region (D) from Figure 5, the user can configure and filter the information presented simultaneously by the views (Carneiro et al. 2008b). These controls

filter the modules based on their name or software entities attributes (e.g. LOC and complexity). Their response time is instantaneous by all practical purposes. With just a few clicks, one can select the modules that fulfil certain search criteria.

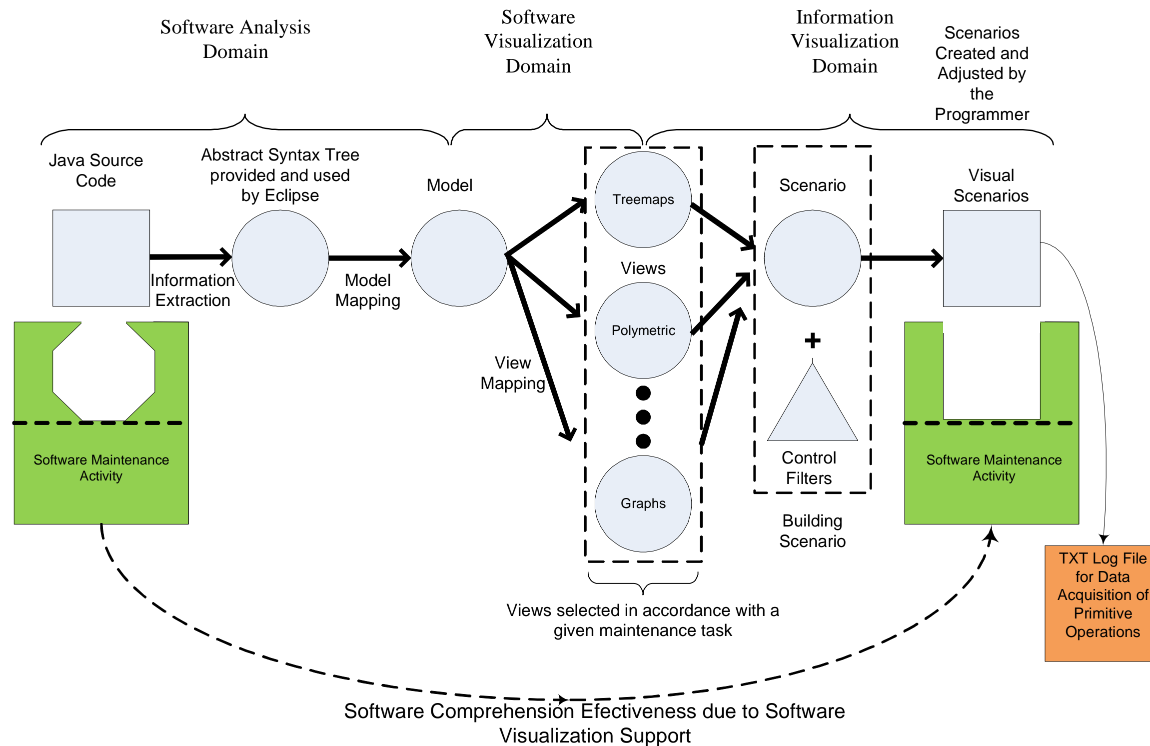

Figure 4: Multi-view Software Visualization Infrastructure

We call this dynamic filtering. In order to complement his mental model about the software, the user can select a specific module directly over the visual interface – regions (B) or (C) – to access its corresponding source code in the editor – region (E). The mapping works both ways, as modifications done in source code are automatically refreshed and updated in the selected views. These features fulfils Baldonado's interaction criterion.

Treemaps, a space-filling method of visualizing large hierarchical data sets (Shneiderman, 1992), are one of the four views (C) available in the plug-in (Figure 5). It shows packages, classes and methods as nested rectangles. Using this metaphor, classes that are in a specific package are presented together in the visual representation. In the same way, all the methods declared in a class are presented in the same rectangle area related to its class. The user can decide at any time, the association between rectangles colour and area with software entities attributes (e.g. LOC and complexity). New software entities attributes can be easily added to the tool.

The other three views integrated to the infrastructure are the polymetric, package and class dependency views. They were implemented originally in (Malnati 2007). The polymetric view (B) (Lanza e Ducasse 2003) is particularly efficient to spot disharmonies in the design and implementation of a system. It is easy to find and identify big modules or anomalies in the shape of the project (provided by the source code inheritance tree). The user is therefore able, with a single picture, to analyze and understand complex systems in terms of methods, lines of code and inheritance hierarchies without the need of reading source code (Malnati, 2007). The class dependency and package dependency views arrange classes and packages in a radial graph, linking them together by dependency links. Each of these links has a certain weight highlighting how strength is the dependency between entities (Malnati, 2007).

The data acquisition of primitive operations is a non-obtrusive way to capture user activities in a software interface. It complements traditional usability assessments like surveys and questionnaires.

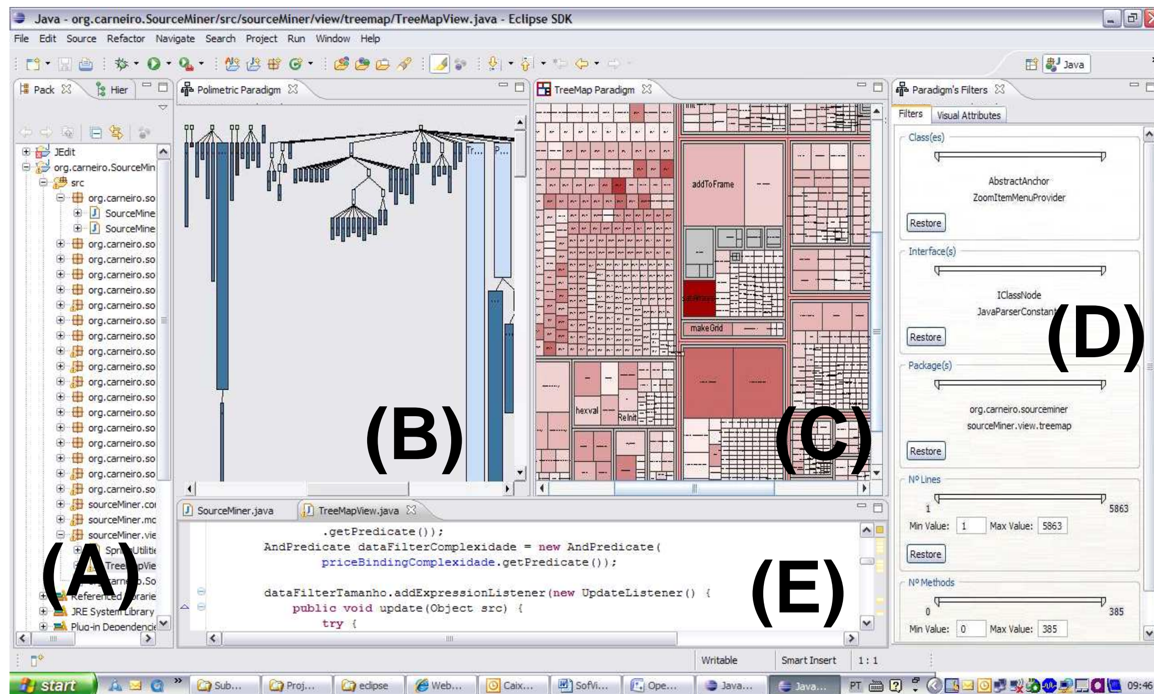

Figure 5: A snapshot from SourceMiner plug-in

We have implemented data acquisition in our plug-in. All user activity on the interfaces is recorded as ASCII text file. The file registers view selection, presentation and interaction events. Furthermore, the data log also registers the selection of other Eclipse resources activated to perform a given task.

The log is a dense source of information that can play an important role in an experimental environment. From these logs one can better understand how developers build a mental model from the available views complemented by source code and figure out what were the heuristics used to accomplish specific tasks.

This experimental environment and the data acquired from the programmers (participants) are then helpful to study (characterize and evaluate) how useful visual interfaces are in supporting maintenance tasks.

5. Experimental Environment

The existing studies on software visualization seem to focus on several aspects related to software comprehension, but there is a lack of environments specially built to characterize and evaluate how useful multiple visualizations are in supporting software comprehension tasks.

In order to assemble such an experimental environment, we selected a set of software maintenance tasks and artefacts that could be used in in-vitro experiments with students, our typical experimental participant. The goal is to produce a meta-experimental package that can be instantiated for specific experimental designs (Carneiro et al., 2008c).

The central part of this package is the object of the study and the tasks to be performed on it. We selected a program called Paint (Ko et al., 2006). This is a Java Swing application, implemented with nine Java classes across nine source files with 503 non-comments lines of code. The application allowed users to draw, erase, clear and undo colored strokes on a white canvas.

The tasks to be performed on the programs were adapted from (Ko et al., 2006). They compose a sequence of perfective and corrective maintenance tasks to be performed by the participants: a) Task 1 – The scroll bars do not always appear after painting outside the canvas, but when they do appear, the canvas does not look right. Participants should fix the program so that the scroll bars appear immediately when painting outside the visible canvas; b) Task 2 – Users can not select the yellow

color. Participants should fix Paint so that users can use it; c) Task 3 – The undo last stroke button does not always work. Participants should fix the Paint so that the button undoes the last stroke or clear the canvas; d) Task 4 –There is a radio button for line drawing, but it does not work. Participants should create a functionality that allows users to draw a line between two points; Task 5 – Participants should create a thickness slider that controls the stroke thickness for all drawing resources; Task 6 – Participants should apply the Model View Control (MVC) Pattern to the Paint program.

The tasks represents software comprehension experimental scenario that follows the principles presented in (Knodel et al., 2006). Task 1 is a simple task that everyone should be able to solve. If there are persons that do not solve this task they should be taken out of the experiment as an outlier that probably did not understand the instructions or lack basic programming skills. Tasks 2 to 5 can be solved by analyzing the code and extracting facts from the visualization and enhancing the source code accordingly. Task 6 is a complex task that can be used to recognize outliers on the end other end of the spectrum. It can also be assigned to longer experimental studies.

5.1. Characterization Pilot Studies

Our first studies aim at testing the experimental artifacts and to baseline the dependent variables for a future controlled experiment. The dependent variables measured were the number of tasks concluded correctly, the time to perform each task, the views resources utilized by the programmers. The independent variables to be considered are: the experimental object (in this particular case the Paint program) and the participant's experience, captured by a questionnaire.

We adopted a very simple design, intended to fit a single 3.5 hours lab session and to account for a limited number of participants. The participants took part in a 30 minutes training session on how to use the interfaces and a description of the tasks to be performed, followed by a 3.0 hours for the execution of the tasks. To conclude, we asked the participants to answer a feedback questionnaire on the use of the interfaces and the task execution.

5.2. Pilot Studies Results

We execute two pilot studies. The both studies used junior professionals taking a post-graduate course at the university. The students were split in teams of three that worked together to solve the tasks. The student participation was required but no grading was associated with their performance.

Both studies involved eight teams. In the first study, the teams executed on average 3.0 tasks and two teams manage to finish five activities. All teams did in fact use the visualization interfaces. In the second pilot study, the teams executed on average 1.1 tasks and two teams did not even finish the filter task. Our logs shows that those two teams also did not even used the interfaces.

The results for the first pilot study indicated that our experimental environment was consistent and could move to a full scale controlled experiment. The second pilot study showed exactly the opposite.

Although the experimental set ups were quite similar, the groups involved in the second study were slightly less experienced than the first. However, this difference was not enough to cause the disparity observed in the results. Analyzing the results and the questionnaire answers more closely, we concluded that motivation was the key issue.

6. Conclusions and Future Work

This paper presents some of the key usability and cognitive principles to build a multiple view software visualization infrastructure. We present a multi-view software visualization infrastructure model and instantiated it to build an Eclipse plug-in. This plug-in currently offers four views that fulfil typical software comprehension needs such as visualization of hierarchical structures, relationships between software entities and artefacts, and entities attributes (e.g. size and complexity).

In order to explore how specific sets of visual paradigms or multiple views can be applied in the context of software comprehension activities, we also developed an experimental infrastructure to support empirical studies. This infrastructure characterizes tool-supported comprehension activities by logging all user actions on the IDE and was already used in two pilot studies. The results of the first

pilot study indicated that our experimental environment was consistent and could move to a full scale controlled experiment. The second pilot study showed the opposite. Analyzing the results and the questionnaire answers more closely, we concluded that motivation was a key issue. We decided to execute another pilot study to further characterize the use of the interfaces, now grading the performance of students. We hope that this will motivate them better and help us to gather trustworthy data on the interface usage. We are currently developing data analysis and mining techniques to evaluate the extensive log files we obtained in the pilot studies.

At this time, we have a complete experimental design to evaluate the IDE with the visualization interfaces against the regular IDE. Our goal is to map high level comprehension tasks with sequences of events on the log files. The plug-in can be downloaded from http://www.nuperc.unifacs.br/tools.

7. References

1. Parnas, D. L. (1994) *Software Aging*. ICSE, 1994, 279-287.
2. Baldonado, M., Woodruff, A., Kuchinsky, A. (2000) *Guidelines for Using Multiple Views in Information Visualization*. Proceedings of ACM AVI 2000; Palermo, Italy. 110-119.
3. Hundhausen, C., Douglas, S., Stasko, J. (2002) *A Meta-Study of Algorithm Visualization Effectiveness*. Journal of Visual Languages and Computing, pp.259-290.
4. Mayer, R., Anderson, R. (1991) *Animations need narrations: an experimental test of a dual-coding hypothesis*. Journal of Educational Psychology.
5. Cooper, C. (1997) *Individual Differences*. Oxford Illustrated Press. Oxford.
6. Letovsky, S. (1986) *Cognitive Process in Program Comprehension*. In Empirical Studies of Programmers. Pages 58-79. IEEE Computer Society Press.
7. S. O. Tergan, T. Keller (Editors). (2005) *Knowledge and Information Visualization: Searching for Synergies* (Lecture Notes in Computer Science). Springer.
8. Shneiderman, B. (1992) *Tree Visualization with Tree-Maps: 2-d Space-Filling Approach*. ACM Transactions on Graphics, Vol. 11, No. 11. January. Pages 92-99.
9. Lanza, M., Ducasse, S. (2003) *Polymetric Views - A Lightweight Visual Approach to Reverse Engineering*. In IEEE TSE, Vol. 29, No. 9, pp. 782 - 795, September.
10. Malnati, J. (2007) *X-Ray: An Eclipse Plug-in for Software Visualization*. Bachelor Project. Lugano University. July.
11. G. Carneiro, A. Orrico, M. Mendonça. (2007) *Empirically Evaluating the Usefulness of Software Visualization Techniques in Program Comprehension Activities*. In JIISIC, Lima, Peru, January.
12. Carneiro, G.; Magnavita, R.; Spinola, E.; Spinola, F.; Mendonça, M. (2008a) *An Eclipse-Based Visualization Tool for Software Comprehension.* In 22nd Brazilian Symposium on Software Engineering 2008. Campinas, Brazil.
13. Carneiro, G.; Magnavita, R.; Mendonça, M. (2008b) *Combining Software Visualization Paradigms to Support Software Comprehension Activities*. In ACM Symposium on Software Visualization. Herrsching am Ammersee, Germany. (Poster).
14. Carneiro, G.; Magnavita, R.; Spinola, E.; Spinola, F.; Mendonça, M. (2008c) Evaluating the Usefulness of Software Visualization in Supporting Software Comprehension Activities. In 2nd ESEM. Kaiserslautern, Germany. Accepted as Short Paper.
15. Ko. A. J., Myers, B.A., Coblenz, M. and Aung, H. H. (2006) *An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks*. IEEE TSE, 32(12), 971-987.
16. Jens Knodel, Dirk Muthig, Matthias Naab. (2006) *Understanding Software Architectures by Visualization--An Experiment with Graphical Elements*. WCRE : 39-50.

17. North, C., and Shneiderman, B. (1997) *A Taxonomy of Multiple Window Coordination*. Univ. Maryland Computer Science Dept. Technical Report #CS-TR-3854.
18. Fowler, M. (1999) *Refactoring: Improving the Design of the Existing Code*. Addison-Wesley.
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.
20. Tidwell, J. (2005) *Designing Interfaces*. O'Reilly. November.
21. Lehman, M. (1996) *Laws of Software Evolution Revisited*. Proceedings of the 5th European Workshop on Software Process Technology.
22. SGC SmallWiki. (2008) *A non-exhaustive list of Software Visualization tools*. Available at http://smallwiki.unibe.ch/codecrawler/anon-exhaustivelistofsoftwarevisualizationtools.
23. *CodeSurfer - a maintenance, understanding, and inspection tool*. (2007) Available at http://www.grammatech.com/products/codesurfer/overview.html.
24. *Sotograph. Analysis of large-scale object-oriented software systems, reverse engineering, architectural verification, code smells, trend analysis*. (2006) Available at http://www.sotograph.com/.
25. S.K. Card; J.D. Mackinlay; B. Shneiderman (eds.). (1999) *Readings in Information Visualization – Using Vision to Think*. Morgan Kaufmann, San Francisco, CA.
26. Ben Shneiderman. (1992) *Designing the User Interface - Strategies for Effective Human-Computer Interaction*. Second Edition. Reading, MA: Addison-Wesley Publishing Company.
27. Deborah J. Mayhew. (1992) *Principles and Guidelines in Software User Interface Design*. Englewood Cliffs, NJ: Prentice Hall.
28. S.K. Card, J.D. Mackinlay, and B. Shneiderman. (1999) *Readings in Information Visualization Using Vision to Think*. Eds. San Francisco: Morgan Kaufmann.
29. TouchGraph. (2008) Available at http://www.touchgraph.com.
30. SpotFire. (2008) Available at http://spotfire.tibco.com/index.cfm.
31. J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price (Eds.). (1998) *Software Visualization — Programming as a Multimedia Experience*. MIT Press.
32. Roschelle, J. (1990) *Designing for Conversations*. AAAI Symposium on Knowledge-Based Environments for Learning and Teaching. Stanford, CA.
33. Hundhausen, C. (1999) *Toward Effective Algorithm Visualization Artifacts: Designing for Participation and Communication in an Undergraduate Algorithms Course*. PhD Dissertation. Department of Computer and Information Science. University of Oregon.
34. R. Riding, S. Rayner. (1998) *Cognitive Styles and Learning Strategies.* David Fulton Publishers, London.
35. Les, Z., Les, M. (2008) Shape Understanding System: The First Steps toward the Visual Thinking Machines. Springer-Verlag Berlin Heidelberg.
36. Li, Q., Bao, X., Song, C., Zhang, J., North, C. (2003) Dynamic query sliders vs. brushing histograms. Conference on Human Factors in Computing Systems (CHI'03). Florida, USA.