Not all Visualizations are Useful: The Need to Target User Needs when Visualizing Object Oriented Software

Mariam Sensalire and Patrick Ogao

"A picture is worth a thousand words". In the software field, this is justified by the increasing research into software visualization. Pictures are increasingly being used to represent software code with many studies establishing that they improve comprehension. This paper discusses results from observing expert programmers use 3 visualization tools. The results show that if tools are developed without user consultation, they may fail to be useful for the users. The necessity for developers of tools to target the needs of the users for whom the tools are aimed is further discussed.

1. Introduction

Understanding a program is usually of great importance to a software project. Whether a programmer is maintaining or developing a program, it is paramount that they understand the program being worked on [12]. In many cases, when trying to understand software, many programmers have to read the lines of code of a particular program. This approach is however not feasible as the program size increases [19]. As a supplement method, visualization has been used to increase program understanding [8]. Visualization is the process of transforming data into insight [26]. It can also be looked at as the technique for creating images, diagrams, or animations to communicate a message [6]. Software is complex, multifaceted, large, and contains many relationships between its component parts.

Therefore there are many aspects of software that may be appropriate for visualization [9]. Previously, different forms of visual presentations were used in understanding software, among which were the UML diagrams. While these mechanisms are good for representing a small component of the software, they are not scalable with regard to large software systems [24]. Software visualization aims to address these shortcomings and many designers have therefore embarked on developing software visualization tools to meet these demands. Despite this, not enough work is put into determining the desired features for such tools [27]. As such many of the designed tools are not used in practice. To be able to design a software visualization tool that is effective however, a proper needs assessment of the target users may need to be carried out.
2. Motivation

It has been noted in the past that very few of the software visualization systems developed are systematically evaluated to ascertain their effectiveness [18]. This means that many of the developed software visualization systems may not even be appropriate for their aims. Price [18], in his taxonomy, creates a scope for effectiveness of software visualization systems. It looks at the purpose of the system, the appropriateness and clarity, as well as empirical evaluation. It is further noted by [24] that a useful software visualization tool can only be produced if it is based on a study of program understanding.

It is important for software visualization tools to actually be able to increase understanding. The images that are generated have to be easily interpreted by the user in relation to the structure of the software being visualized [1] as effective visualizations can be of assistance even to expert programmers. In many cases, expert programmers join projects in whose initiation they were not involved [2], as a result, they are faced with the challenge of contributing substantially to a programming project whose low level and high-level structure they may not understand. Such programmers can use visualizations as a starting point to understanding the software.

There are other cases where programmers are recruited and given access to the program documentation with the aim of making small changes to already implemented projects. These programmers also face challenges. This is due to the fact that documentation for software is usually not updated in parallel with the software changes over time [2]. A result is documentation that may not reflect the actual status of the program being worked on. Due to this, many programmers consider source code to be the most trusted form of documentation making it critical to be understood [25].

Fig1: Visualizing form source code

An example of visualizing from source code is shown in figure 1 (adapted from [4]). The first part represents the source code, which can be received in its raw form or exported to a format that the tool expects. Once the tool receives the right format of the code, the visualizations are then generated.

A programmer who does not understand a programs source code can easily make changes which have negative effects in other parts of the software leading to
undesirable effects for the whole project ([2], [24]). If programmers had the ability to know how the different components in a program relate, such problems would be overcome. Even when programmers have access to the source code as well as UML diagrams of the system, it is still not easy to understand the system since these notations do not scale up well with respect to comprehension [11].

Source code becomes incomprehensible beyond hundreds of lines of code while UML decreases information density, which in turn limits scalability [1]. With both these systems, it is quite difficult to see an entire system. Visualization has been shown to improve the productivity and effectiveness of programmers, especially when working with complex systems, which can span the work of several people over extended periods of time [5]. Diagrams of a system are much easier to understand and they convey a lot of information in a standardized way [2]. It is also important to visualize from source code because in many situations, it may be the only source of information available about a given program [25].

3. Previous Work

In the past, there has been a lot of research on visualizing Object Oriented Software. Jerding and Stasko [8], were among the first to advocate for the use of visualization to foster the understanding of O-O software. Those events that need to be visualized to enable comprehension of O-O software were specified. Jerding and Stasko [8] however acknowledged the difficulty in gathering the necessary information to construct useful visualizations. Based on these difficulties, a framework was proposed which recommends that: Visualizations should require no programmer intervention once developed and they should present the aspects of a program that are important and will be of use to programmers. The framework also specifies that visualizations should be able to cater for the programmers’ needs in a timely manner, and lastly should be able to handle real world problems. Jerding and Stasko [8] however did not mention carrying out a user study to justify their conclusion despite the fact that a user study is critical for producing useful visualization tools [8]. Maletic et al, [11] also made a case for visualizing O-O software in virtual reality. The subsequent Imsovision tool proposed was based on Shneiderman’s [22] Task by Data Type for Information Visualization which specified a visual design guideline of overview first, zoom and filter, details on demand, relate, history and extract. Imsovision was however not based on any existing software comprehension models nor did it have a previous validation procedure to prove that virtual reality enables comprehension better than 2D or 3D methods.

In terms of proximity, this work closely relates to that of Pacione [17] who supported the use of a multifaceted three-dimensional abstraction model for software visualization. In that study, effective presentation techniques for visualization were presented. These were specified as the use of diagrams for describing software as well as the use of views for software comprehension. Diagrams were noted to be more effective if used in an interrelated hierarchical manner that addresses all levels
of abstraction. It was also specified that the use of multiple interdependent views was the best arrangement in relation to comprehension. Pacione however noted that identifying the views that were appropriate for particular comprehension tasks was still a challenge. It was also stated that the usefulness of the multifaceted three-dimensional model in software comprehension would be evaluated in future. This therefore means that the procedure proposed was yet to be proven as effective since it was not yet evaluated. Ihantola [7] on the other hand looked at algorithm visualization and specified a taxonomy for effortless creation of algorithm visualizations. The justification for this taxonomy was the identified lack of use of algorithm visualization tools beyond the labs, a problem that also applies to software visualization tools.

The view taken by that study was that the difficulty of creating visualizations using the developed tools led to their lack of reception in the classrooms where they were needed the most. This view was arrived at after consulting the different teachers of algorithms and getting their views on the source of the problem. Ihantola [7] also noted that many developers create systems based on their own needs or beliefs about others’ needs.

This problem was also noted to be prevalent in software visualization tools ([28],[21]) thus justifying the need for user consultation before tool development. Maletic at al. [11] also build the case for identifying the most appropriate visualization techniques for given software visualization tasks. It is proposed that this is done after considering why the visualization is needed, the people that will be using it, the kind of information it is going to be representing as well as the medium of representation. This is the same procedure that will be used in this research with specialization put on O-O software.

4. The experiment

This section looks at the design and conduct of an experiment observing five expert programmers use three visualization tools. The following section discusses the programmers’ use of the tools, in an effort to show what useful visualizations should be like as well as the process of generating them.

4.1 Tools and Source Code

Three tools were used for the study. These were Code Crawler ([20], [10]), Creole ([3], [14]), and Source Navigator [23]. These tools can all visualize Object oriented software. In particular, they can all visualize Java, which was the language that the source code used was in. The tools are all freely available and use different techniques for visualizing.
Fig 2: Display from Code-Crawler

Figure 2 above shows the system complexity diagram display from Code-crawler. It is the default diagram generated by Code-Crawler when visualizing code.

Fig 3: Display from Source Navigator

```
BaseTestRangeFilter ───── TestConstantScoreRangeQuery ───── TestRangeFilter
TestMultiSearcher ───── TestParallelMultiSearcher
TestSpansAdvanced ───── TestSpansAdvanced2
_TestHelper
Callback
CheckHits
CustomSearcher
DocHelper
DumbQueryParser
DumbQueryWrapper
EmptyTokenStream
English
IndexerThread
IndexTest
IndexThread
IntegerQueue
LockMock
MockFilter
MockIndexInput
```
while figure 3 and figure 4 display the sample class diagrams from Source Navigator and Creole respectively.

The source code displayed represents one package from the total packages included in the Lucene search engine. That package has 147 classes with 687 methods.

Three different sets of source code were used for the three different tools. Code Crawler was evaluated using the Lucene search engine source code; Creole was tested using Apache beehive code whereas Source Navigator was tried using Apache tomcat code. The source code was in the same range in terms of size.

4.2 Participants
Five expert programmers participated in the study. They were all male with over ten years experience both in programming and computer usage. They were experienced with the object oriented paradigm with knowledge of at least two object oriented languages as shown in table 1.
Table 1. Summary of participants

<table>
<thead>
<tr>
<th>User</th>
<th>Computer Usage</th>
<th>Langs</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>&gt;10 Yrs</td>
<td>Java, C++, SmallTalk</td>
<td>&gt;10 Yrs</td>
</tr>
<tr>
<td>2.</td>
<td>&gt;10 Yrs</td>
<td>Java, C++, Python</td>
<td>&gt;10 Yrs</td>
</tr>
<tr>
<td>3.</td>
<td>&gt;10 Yrs</td>
<td>Java, C++, Python</td>
<td>&gt;10 Yrs</td>
</tr>
<tr>
<td>4.</td>
<td>&gt;10 Yrs</td>
<td>Java, C++, Ruby</td>
<td>&gt;10 Yrs</td>
</tr>
<tr>
<td>5.</td>
<td>&gt;10 Yrs</td>
<td>Java, C++</td>
<td>&gt;10 Yrs</td>
</tr>
</tbody>
</table>

Java was the language that was frequently used by all the expert programmers. The experiments were therefore carried out using Java source code.

Creole’s default display, shown in figure 4, is that of packages. A double click into a particular package shows the classes and methods inside the package as well as any links between each other.

Each participant was given a 5 minutes introduction to a tool, after which they had 5 extra minutes within which to familiarize with the tools themselves or seek any extra information.

After the familiarization stage, 2 tasks were given to the participant for each tool, one task at a time. The tasks given out were:

i) Describe the static structure of the system, ie the main classes and their relationships.

ii) What would be the effect of deleting the Hook class?

Those tasks were replicated for all the three tools, however the second task was modified according to the source code being analyzed.

i) For Creole, the second task was changed to “What would be the effect of deleting the org.apache.beehive.netui.tags”

ii) Source Navigator’s second task was naming the effect of deleting the DbStoreTest class.
There was a 1-minute break between the completion of tasks for each tool after which the second tool was evaluated and the third accordingly. The planned time for the experiment was 1 and half hours calculated as shown by figure 5.

Fig 5: Timing of the tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>Times Carried Out</th>
<th>Task Duration Per Task In Minutes</th>
<th>Total Duration In Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-study questionnaire</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Introduction to tool</td>
<td>3</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>Familiarize with tool</td>
<td>3</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>2 assignments</td>
<td>3</td>
<td>7</td>
<td>42</td>
</tr>
<tr>
<td>Post-study questionnaires</td>
<td>3</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>92 min</td>
</tr>
</tbody>
</table>

4.3 Experimental Analysis

The details of the experiment were analyzed as follows: The pre-study questionnaire aimed at establishing the computer knowledge of the experts in order to know how it would affect their performance. It also sought confirmation on whether the users had used a software visualization tool before and if indeed they were expert programmers.

Tool usage instructions were also availed to the candidates before the actual experiment and sample tasks given to them. These tasks were similar to the ones in the experiment only that the code used was different. This stage helped in clearing any questions that the users had about the tools and also ensuring that working knowledge of the tools had been established before the actual experiment.

During the actual experiment, Tasks about the makeup of the code that was being analyzed were given and answers written down for all the three tools. After this stage, the post study questionnaire was availed. This questionnaire aimed at establishing the shortcomings of the tools that were used during the experiment as well as their positive points. Questions also sought to know what extra features the users needed but felt were lacking in the tools. A combination of the answers given in this questionnaire as well as observations during the experiment led to the results for the study.

5. Results

In this section, the results from the study are discussed. The importance of targeting user needs when developing visualization tools is also shown based on these results.
Based on the results of the study, there was a lot that was still lacking from all the visualization displays as well as the tools used for the experiment. Some of the queries from the expert programmers are summarized below.

**The displays were too crowded.** The visualizations tended to display too much information in a small area rendering the display unhelpful for the programmers. The ability to search the visualization was another desired component that the users felt could have been addressed better. Even a good visualization that cannot be manipulated was not found to be useful.

**The speed** of achieving the visualization was also highly complained about. More than two users stopped the generation of call graphs because they felt it was taking too long. So even after the visualization has been generated if the timing is not right then it is not considered useful.

There was also concern about integrating the visualization tools with an IDE. The reasoning was that when one visualizes, it is usually for a purpose. If the desire were to add more code to existing software, then it would be too much effort to switch between the visualization tool and the environment that is being used to program. So even if a tool is able to generate amazing visualization, the effort and time spent switching between the two environments may have on effect

### Summary of Results

<table>
<thead>
<tr>
<th>Tool</th>
<th>Best Features</th>
<th>Additional Features needed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Creole</strong></td>
<td>The Zooming Visualization Interface</td>
<td>Reduced Crowding</td>
</tr>
<tr>
<td></td>
<td>Eclipse Integration</td>
<td>Better Speed</td>
</tr>
<tr>
<td></td>
<td>Search Abilities</td>
<td></td>
</tr>
<tr>
<td><strong>Code-Crawler</strong></td>
<td>Code-Complexity View</td>
<td>Direct Code Access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Less Steps for tasks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IDE Integration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Package views</td>
</tr>
<tr>
<td><strong>Source Navigator</strong></td>
<td>Simple Interface</td>
<td>Better Lay out</td>
</tr>
<tr>
<td></td>
<td>Speed</td>
<td>IDE Integration</td>
</tr>
<tr>
<td></td>
<td>Ease of Use</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ability to view code</td>
<td></td>
</tr>
</tbody>
</table>

Based on the results of the study, there was a lot that was still lacking from all the visualization displays as well as the tools used for the experiment. Some of the queries from the expert programmers are summarized below.
on the knowledge for programming. The expert programmers appreciated tools that were able to generate visualizations in minimal steps.

5.2: Interpretation

Fig 6: System complexity diagram of Lucene source code in CodeCrawler.

The results however showed that more time than had been planned was spent on the tasks. Figure 7. Class interface hierarchy displayed in Creole. As shown in figure 6 by the partial visualization from Lucene above, too much information in a single display may be complex to understand.

Fig 7: Shows Lucene source code as displayed by Creole.
The creole display was appreciated by the participants of the study due to the ability to zoom in and expand particular components as well. Speed was however still an issue.

**Fig 8: Lucene displayed by Source Navigator**

![Diagram of Lucene displayed by Source Navigator]

Figure 8 shows the same source code that was displayed in Creole as displayed in Source Navigator.

### 5.3. Effective Visualization

There are also different views from the literature about what an ideal visualization must be like. According to Knight [9] a visualization can only be considered effective if results can be achieved by users that use it without incurring any extra cost. It has to be able to reduce complexity while increasing understanding and must be able to capture the viewers attention [15]. Staples, [24] adds that a visualization which provides information that can be readily got by inspection of code, is not effective.

The presentation techniques also have to be chosen carefully as the style in which a visualization is specified affects its usability (Price [24], Schafer [21], Tory
If the visualization tool however requires too much effort to use, it would not be adopted in the real world regardless of the advancements of the techniques [2]. There is therefore need to know the kind of presentation that is suitable for the users and the task to be visualized [15]. This re-emphasizes the need to consult the group for which the visualizations are aimed before developing the tools.

5.4. Conclusions and Future work

From the results of the study carried out combined with literature from previous studies, it can be concluded that better visualization tools and techniques can be achieved if the views of the target group are sought before hand. Future work will re-visit program comprehension models in an effort to supplement the current work with the aim of better visualizations. It is not feasible to enforce processes on programmers when they are not supported by validated cognition models [13]. Despite this, some cognition models developed in the past were established after carrying out experiments using very few lines Of code [13]. Future work would include carrying out a comprehensive study on object oriented software comprehension to supplement the existing comprehension models.

References


Eng, D. Combining static and dynamic data in code visualization. PASTE02 Chalseton, SC, USA, 2002.


Schafer T. Towards more flexibility in software visualization tools. Department of Computer Science Darmstadt University of Technology,, 2005.

Shneiderman B. The eyes have it: A task by data type taxonomy for information visualizations. Department of Computer Science University of Maryland, College Park Maryland,, 1996.


Staples S. and Bieman J. 3-d visualization of software structure. School of Computing Colorado State University, BC, Canada,, 1999.

Storey M. On integrating visualization techniques for effective software explorations. School of Computing Simon Fraser University, Burnaby, BC, Canada, 1997.

