Classifying Desirable Features of Software Visualization Tools for Corrective Maintenance

Mariam Sensalire*
Faculty of Computing and IT
Makerere University, Uganda

Patrick Ogao†
Faculty of Computing and IT
Makerere University, Uganda

Alexandru Telea‡
Institute of Mathematics and Computing Science
University of Groningen, the Netherlands

Abstract

We provide an evaluation of 15 software visualization tools applicable to corrective maintenance. The tasks supported as well as the techniques used are presented and graded based on the support level. By analyzing user acceptation of current tools, we aim to help developers to select what to consider, avoid or improve in their next releases. Tool users can also recognize what to broadly expect (and what not) from such tools, thereby supporting an informed choice for the tools evaluated here and for similar tools.

1 Introduction

Several studies have advocated the use of software visualization tools for corrective maintenance (CM) [Baecker et al. 1997; Swanson 1976]. However, although several tool surveys exist on the Internet, it is still hard to answer the questions ”does this tool fit my user profile, context, and needs?” (for tool users) and ”what do the users most (dis)like in a tool?” (for tool developers). In a previous study [Sensalire and Ogao 2007b], we collected feedback from software engineers who used such tools for program understanding in general, and extracted several desirable features of SoftVis tools. In this paper, we refine and focus the set of desirable features on SoftVis tools for CM. We aim to guide users in selecting SoftVis tools for CM, based on their desirable features, either from the tools discussed here or from a larger, more general, set. Next, we aim to discover possible correlations between perceived tool acceptance levels and the usage of certain visual techniques, to further clarify what makes a tool accepted (or not).

This paper is organized as follows. Section 2 overviews related work. Section 3 presents our classification model for desirable tool features. Section 4 presents the evaluated tools and evaluation procedure (Sec. 5). Section 6 discusses our evaluation results. Section 7 concludes the paper.

2 Related Work

Price *et al.* [Price et al. 1993] compared 12 tools against 6 desirable features categories: scope, content, form, method, interaction and effectiveness. The tools were however not related to a single application area. Maletic *et al.* [Maletic et al. 2002] compared 5 software tools along 5 axes: task, audience, target, representation, and medium. Similar to Price *et al.*, the scope of this taxonomy and tools is quite broad. Since a tool’s audience strongly depends on its purposes [Maletic et al. 2002], evaluating similar-purpose tools would be more insightful [Koschke 2003]. Here, Storey *et al.* [Storey and German 2005] compared 12 tools that provide awareness of human activities during software development against the categories of intent, information, presentation, interaction and effectiveness. In corrective maintenance, Baecker *et al.* analyzed three classes of SoftVis techniques used for debugging [Baecker et al. 1997]: animation, improved typographic representations, and error sonification. Earlier, we evaluated ten general-purpose SoftVis software-understanding tools[Sensalire and Ogao 2007a]. This evaluation forms the basis of our extended study of SoftVis tools for CM.

3 Classification Model

We classify SoftVis tools using four categories of desirable features: Effectiveness, Tasks supported, Techniques used and Availability. These features and the scales their presence is measured on are derived from several user interviews [Sensalire and Ogao 2007b]. We use either a *low,medium,high* scale or a simpler *yes,no* scale, as described next.

3.1 Category A: Effectiveness

An *effective* tool arguably helps users to solve the problems it was designed to assist with. Effectiveness is task-specific, hence hard to measure in general. Yet, we identified three non-functional properties that effective SoftVis tools should have, as follows.

Scalability: A scalable tool supports CM tasks on systems of millions of LOC and/or thousands of classes and/or source files. Tools created for educational purposes, proof-of-concept, or research prototypes are given *low*; *high* if they support large-scale code (as defined above); and *medium* if falling in between.

Integration: This measures how easily several tools can switched and exchange data to complete a given task in an IDE or similar setup. A tool that input and output data from/to other tools, and also senses and displays data changes, is given *high*. Tools that coexist in an IDE but do not actively exchange data and/or events with peer tools are given *medium*. Standalone tools are given *low*.

Query support: Tools that work in batch mode are given *low*. Tools that support just lexical queries get *medium*. Tools that correlate queries and visualizations (query highlighting) or use complex queries(*e.g.,* syntax-aware or type-based) get *high*.

3.2 Category B: Tasks supported

We consider the following tasks: detecting code smells, code refactoring, trace analysis, and debugging support. While not exhaustive, these are typical for CM, and also well understood by our interviewed users.

Detecting code smells / Refactoring: Code smells signal bad programming, design, or code, and are often used to drive refactoring. Tools that support neither refactoring nor code smells detection, *e.g.* debuggers, get *low*. Tools that can detect smells but do not suggest

*e-mail: msensalire@cit.mak.ac.ug
†e-mail: ogao@cit.mak.ac.ug
‡a.c.telea@rug.nl

SOFTVIS 2008, Herrsching am Ammersee, Germany, September 16–17, 2008.

how to correct these, *e.g.* static analyzers, get *medium*. Tools that detect and suggest how to correct smells get *high*.

Trace analysis: Traces are defined as data gathered during a program's execution. Tools that can neither generate nor show traces get *low*. Tools that generate but cannot display traces, *e.g.* profilers, or display but not generate traces, *e.g.* data viewers, get *medium*. Tools that generate and display traces get *high*.

Debugging support: Tools that do basic debugging, *e.g.* breakpoints, code stepping, and simple text watches, get *low*. Tools that show higher-level debugging facts, *e.g.* bug or testcase data, get *medium*. Tools that show both low and high-level debugging data, or relate bugs to actions carried out to remove them, get *high*.

3.3 Category C: Availability

Availability relates to the programming languages a tool supports, whether it is free or commercial, and the platform on which it runs. A tool must be available to evaluate it, so we do not grade this aspect, but just provide a textual description.

3.4 Category D: Techniques used

Assessing the types (and presence degree) of techniques used in a SoftVis tool can shed light on usability and ease of learning. Following previous taxonomies [Maletic et al. 2002] and user feedback [Sensalire and Ogao 2007b], we consider the following techniques:

2D or 3D Graphics 2D graphics (present: *yes-no*) is sometimes seen as less expressive than 3D, but can be easier to learn and use. 3D graphics, measured on the same scale, can display more data, but may come with additional usability costs.

Animation: Animation can be used to show dynamic data, *e.g.* traces or code evolution. Effects such as smooth zooming and panning are here not considered animations.

Color usage: Tools that do not use color, *e.g.* command-line ones, are given *low*. Using a few saturated hues, *e.g.,* to display categorical values, gives a *medium*. Tools that use blending for multivariate data display or color mapping of continuous value ranges get *high*.

User interaction: Tools that show static views get *low*. Tools that support a single task via user interaction, *e.g.,* adjust a layout with the mouse [Source-Navigator-Team 2007], get *medium*. Tools that support two or more tasks by two or more medium-graded techniques, *e.g.* direct mouse manipulation and rubberbanding, get *high*.

Multiple views: Tools that use a single view get *low*. Several not-linked views are gives a *medium*. Several linked views get a *high*.

Information density: Quantifies the amount of artifacts (*e.g.* code lines, classes, functions, or files) drawn per screen unit. For example, a text editor has *low* density, a UML browser *medium* density, and a dense-pixel treemap *high* density.

Navigation: We measure navigation options (panning, scrolling, zooming and viewpoint change) using a *yes,no* scale.

Dynamic visualization: Dynamic visualizations can display data generated on-the-fly while a program is running, *e.g.,* debugging or tracing data. We use here a *yes,no* scale.

4 Evaluated Tools

We evaluated our desirable features on the following tools.

Allinea DDT debugs scalar, multi-threaded large-scale parallel applications. Visualization is used mainly to control program execution.

CodeRush with RefactorPro is a Visual Studio add-on that enables code refactoring by providing change options with hints visually within the code.

CodePro AnalytiX is an Eclipse plugin that provides code coverage analysis, dependency analysis, and visual report generation.

Code Coverage is a Java NetBeans plugin that colors code based on low coverage.

CVSgrab uses customizable dense pixel displays to correlate project activity from code repositories and bug data from a Bugzilla database.

Gammatella does remote monitoring by combining execution-metric-colored treemaps and SeeSoft-like views.

JBIXBE simplifies debugging of multi-threaded code, combining classical breakpoints, step-mode execution, and watches with execution flowcharts and UML-like class diagrams.

JIVE supports runtime analysis and code debugging using forward and backward stepping and visualizes execution history using interactive UML sequence diagrams.

JSwat provides debugging by a combination of standard IDE-like breakpoints, watches, and visualizations.

Paraver visualizes program traces using 2D dense pixel displays similar to CVSgrab.

Project Analyzer is an IDE that uses syntax highlighting and multiple code views to aid checking for error proneness as well as guiding on ways in which to improve the code.

Source Navigator provides syntax-highlighed and graph-based views to show code symbols and syntax relations, and also several `grep`-based queries.

STAN is an Eclipse plugin offering linked graph views, metric-colored treemaps, and metric histograms to understand and detect design flaws.

Tarantula helps finding faults or problems at code level, using a Seesoft-like code coloring to show test passes or fails.

VB Watch provides code testing and debugging using multiple text views, metrics histograms, and metric-annotated call graphs.

5 Tools Evaluation

Our study participants were 16 professional developers, 12 male and 4 female. Only two had used a SoftVis tool before (Table 1). We used a pre-study questionnaire to select users having good skills (over 4 years in maintenance) and being fluent in 2 or 3 programming languages. We assigned tool groups G_i to users matching the tools' programming language and platform to the ones best known by the respective user, as follows: VBWatch, CodeRush (G_1); JIVE, JBIXE, Source Navigator (G_2); Alinea, Paraver, Code Coverage (G_3); Project Analyzer, CodePro Anaytix (G_4); Tarantula, Gammatella (G_5); CVSgrab (G_6); and JSwat, STAN (G_7).

Four source code bases were studied: ArgoUML for Java tools, the network simulator *ns 2* [Henderson 2008] for C++ tools, a proprietary code base for VB tools, and the ArgoUML and Visualization Toolkit repositories for CVSgrab. After a 15-minute tutorial, the participants performed six generic tasks to assess the tools' capabilities for (a)*Scalability*, (b)*Integration*, (c)*Query support*, (d)*Detecting code smells*, (e)*Trace analysis*, and (f)*Debugging*, as follows:

a Analyze code for errors or error-proneness, from classes to files to packages to the whole project;

b Change code using a tool, analyze effects using other tools;

User	Languages known	Years of experience	Years in maintenance	Tool group evauated
1	VB, Java, C#	6	4	1
2	VB, C#	8	5	1
3	Java, C++	8	5	2
4	Java, C	9	7	2
5	C, C++, Java,	7	5	3
6	C++, Java	> 10	7	3
7	C++, VB, Java,	> 10	8	4
8	VB, Java, C#	7	5	4
9	C++, Java	8	5	5
10	C, VB	6	4	5
11	C++, Java	> 10	7	7
12	C#, Java	9	6	7
13	C, Java, C++	8	4	6
14	C, Java, C#	9	6	6
15	C, Python, Java	8	4	6

Table 1: Evaluation participants

c Find all classes having errors and display them to find how they interact with the rest of the code;

d Find two code smells and use the tool's refactoring hints to improve them;

e Generate traces and analyze them;

f Set breakpoints, step through code and find who has been involved in debugging activities over the last 2 years;

The users graded the perceived support level and visual techniques offered (Sec. 3.4) and also wrote down any additional remarks.

6 Evaluation Results

Tables 2 and 3 show the desirable features and supported techniques obtained by averaging the given grades for each tool.

Tool acceptance was heavily influenced by the amount of user interaction provided. Too little interaction impairs usability. Too much interaction makes learning hard, thus affecting usability too. This correlates with our earlier study [Sensalire and Ogao 2007b], where users disliked tools asking a lot of input before showing any output. Also, tools that did not enable users to tune at all the final images (layout, colors, annotations) were not desired.

Over half of the participants questioned the need for SoftVis tools, and mentioned their preference for the Eclipse IDE, feeling that it would suffice for their CM tasks. This suggests that (new) SoftVis tools should show clear advantages as compared to existing 'plain' IDEs. An evaluation such as the ours is relevant in proving the level to which a tool lives up to its claims. A lack of such studies may create skepticism on the users' side. Tool developers can use the users' IDE attachment to their benefit, by creating tools that plug into an IDE, or using the IDE metaphor in organizing visualizations. For example, the STAN tool was found natural as its visualizations follow the Eclipse displays. Five of the participants felt that education had a major role in industrial tool adoption, saying that they would have been more motivated to use SoftVis tools in their work if they had been exposed to them during their undergraduate training.

Among the considered tools, there is more support for *static* than *dynamic*visualization. *Animation* and *3D*, on the other hand, are not frequently utilized. Also, 2D visualizations were much better accepted by nearly all users, as previously noticed

The preciseness of answering *queries* was ranked as very important. A tighter integration between analysis and presentation, as well as too integration, were deemed as extremely desirable by all users. Tool integration has a direct relation to the tool's user interface and as such influences the tool's usability. When a tool is integrated with an IDE, a familiar user interface is provided to the software developer as the menus for the new tool are incorporated within the existing IDE. Standalone tools, on the other hand, present a new learning challenge to the user, thereby reducing the ease of use. The *medium* to *high* levels of integration observed in our evaluation certify the importance of this factor to the success of the tools. Tool developers would therefore benefit from integrating their tools in order to increase their acceptance by users.

Rich *color usage*, *multiple views*, rich *user interaction* and *navigation* were all features deemed extremely desirable, and also found to be well supported by most users.

As programming *language*, Java is well supported by most tools, probably due to the ease with which Java engines enable data extraction and analysis. Hence, future tool developers can choose to support Java or may choose instead to exploit the lower support of other languages, *e.g.* C++, so as to develop tools for them.

Visual *scalability*, or information density, is not high in most tools. This can limit user acceptance. High information density allows one to monitor more at a given time, which is important for high-volume dynamic data such as traces, but also large code bases. Yet, very high density tools, *e.g.* CVSgrab, were quite hard to learn. This may prove an initial acceptance blocker. Overall, we believe tool developers should use *multiscale methods* to allow users to flexibly select the data amount shown at a given time.

6.1 Comparison with program comprehension studies

We now correlate our results with [Vans et al. 1999] who observed developers during industrial CM tasks. The CM actions involved chunking and hypothesis generation at different abstraction levels. Hence, SoftVis tools that support CM should let users view more abstraction levels to enable hypothesis generation. To (dis)prove a hypothesis, this further requires tool query support and query-view integration. Inability to query large code also makes it hard to do efficient chunking. Additional factors crucial in showing many abstraction levels are the multiple view support and IDE integration for inter-view navigation. Six of the 15 tools evaluated had *high* query support, while 8 tools had *medium* support. Most tools therefore tend towards *high* query support which is the prefered mode for CM [Sensalire and Ogao 2007b; Vans et al. 1999]. Maintenance professionals had varying information needs depending on expertise [Vans et al. 1999], most experienced ones tending to favor precise searches. This emphasizes the need for refined, *high* query support in SoftVis tools for CM. As program comprehension is an essential part of (corrective) maintenance, a further comparison of SoftVis tools in this area with studies of program comprehension tools would bring additional information.

7 Conclusions

We evaluated fifteen SoftVis tools that support corrective maintenance. Our aim was to detect patterns that can guide tool users and developers in their choice of tool to use or techniques to support. The patterns observed from the evaluation were further correlated with results on a program comprehension study in CM. Several features (IDE integration, scalability, multiple views, and query support) were strongly required by all users and provided by increasingly many tools. Some other features (3D, animation) were less present. This raises several open questions:

a Do users reject 3D and animation in SoftVis tools for CM or are developers reluctant to provide them?

b Would developers exposed to SoftVis tools in their early training be more willing to use them in their professionalf work?

c When developing SoftVis tools for CM, when should the views of practicing professional be *mainly* involved: at the point of idea conception; during tool development; or when

TOOLS	EFFECTIVENESS			TASKS SUPPORTED			AVAILABILITY		
	Scalability	Integration	Query support	Detect smells	Trace analysis	Debugging	Languages	Licensing	Platform
Allinea DDT	High	Low	High	Medium	Medium	Medium	C, C++, Fortran	Comm.	Linux
CodeRush	Medium	High	Medium	High	Low	Low	VB, C#	Comm.	Win, Linux
CodePro Analytix	High	Medium	High	High	Low	Low	Java	Comm.	Linux, Win
Code Coverage	Medium	High	Medium	Medium	Low	Medium	Java	Free	Linux, Win
CVSgrab	High	High	Medium	Medium	Low	Medium		Free	Linux, Win
Gammatella	Low	Low	High	Medium	High	Medium	Java	Free	Linux, Win
JBIXBE	Medium	Low	Medium	Low	Low	Medium	Java	Comm.	Linux, Win
JIVE	Medium	Medium	High	Medium	High	Medium	Java	Free	Linux, Win
JSwat	Medium	High	Medium	Medium	Low	Low	Java	Free	Linux, Win
Paraver	High	Low	High	Medium	High	High	Java	Free	Linux
Project Analyzer	Medium	Low	Medium	Medium	Low	Low	VB	Comm.	Windows
Source Navigator	High	High	Medium	Medium	Low	Low	Java, C/C++, Fortran	Free	Windows
STAN	High	Medium	High	Medium	Medium	Low	Java	Comm.	Linux, Win
Tarantula	Low	Low	Low	Medium	High	Medium	C	Free	Linux
VB Watch	Medium	Low	Medium	Medium	Medium	Medium	VB 6	Comm.	Win

Table 2: Tools against desirable features

	VISUAL TECHNIQUES USED								
TOOLS	2D	3D	Animation	Color usage	User interaction	Multi views	Info. density	Navigation	Dynamic viz
Allinea DDT	Yes	Yes	No	Medium	Low	Medium	Low	Yes	No
CodeRush	Yes	No	Yes	Medium	Medium	High	Low	Yes	No
CodePro Analytix	Yes	No	No	Medium	Medium	High	Medium	Yes	No
Code Coverage	Yes	No	No	Medium	Medium	Medium	Low	Yes	No
CVSgrab	Yes	No	No	High	High	High	High	Yes	No
Gammatella	Yes	No	No	High	Medium	High	High	Yes	No
JBIXBE	Yes	No	No	Medium	Medium	High	Medium	Yes	Yes
JIVE	Yes	No	No	Medium	Medium	High	Medium	Yes	Yes
JSwat	Yes	No	No	Medium	Medium	High	Low	Yes	No
Paraver	Yes	Yes	No	High	High	High	High	Yes	Yes
Project Analyzer	Yes	No	No	Medium	Medium	Medium	Medium	Yes	No
Source Navigator	Yes	No	No	Medium	Medium	High	Medium	Yes	No
STAN	Yes	No	No	Medium	Medium	Medium	Medium	Yes	No
Tarantula	Yes	No	No	High	Low	Medium	High	Yes	Yes
VB Watch v2	Yes	No	No	Medium	Medium	Medium	Low	Yes	No

Table 3: Tools against techniques used

the tool is fully functional? Involvement at all points, although desirable, may not be practically feasible.

Future work includes refining the analysis on a set of specific tasks and sample code bases in order to provide a more quantitative evaluation of the tools' abilities to address their tasks. A second direction is to extend our evaluation of SoftVis tools to adaptive, perfective and preventive maintenance.

References

BAECKER, R., DIGIANO, C., AND MARCUS, A. 1997. Software visualization for debugging. *Commun. ACM 40*, 4, 44–54.

HENDERSON, T., 2008. ns-allinone-2.33 release. Website. `http://http://sourceforge.net/project/showfiles.php?group_id=149743&package_id=169689&release_id=588643`.

KOSCHKE, R. 2003. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintainance. Evol.: Res. Pract 15*, 87–109.

MALETIC, J., MARCUS, A., AND COLLARD, M. 2002. A task oriented view of software visualization. *Proceedings of IEEE Workshop of Visualizing Software for Understanding and Analysis Paris, France,*, 32–40.

PRICE, A., BAECKER, R., AND SMALL, I. 1993. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing 4(3):211-266*.

SENSALIRE, M., AND OGAO, P. 2007. Tool users requirements classification:how software visualization tools measure up. *AFRIGRAPH '07' Proceedings of the 5th International Conference on Computer graphics, virtual reality, visualization and interaction in Africa, Grahamstown, South Africa*.

SENSALIRE, M., AND OGAO, P. 2007. Visualizing object oriented software:towards a point of reference for developing tools for industry. *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, Banff, Canada*.

SOURCE-NAVIGATOR-TEAM, 2007. Source code analysis tool. Website. `http://sourcenav.sourceforge.net/`.

STOREY, M.-A. D., AND GERMAN, D. M. 2005. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, ACM, New York, NY, USA, 193–202.

SWANSON, E. B. 1976. The dimensions of maintenance. In *ICSE '76' Proceedings of the 2nd International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA.

VANS, A., VON MAYRHAUSER, A., AND SOMLO, G. 1999. Program understanding behavior during corrective maintenance of large-scale software. *International Journal of Human-Computers Studies 51*, 1, 31–70.