Model-Based Analysis of Adoption Factors for Software Visualization Tools in Corrective Maintenance

Technical Report SVCG-RUG-10-2010
Scientific Visualization and Computer Graphics Group (SVCG)
Faculty of Mathematics and Natural Sciences, Institute Johann Bernoulli
University of Groningen, the Netherlands

Mariam Sensalire[1] and Patrick Ogao[1] and Alexandru Telea[2]

[1] School of Computing and IT, University of Makerere, Kampala, Uganda
[2] Institute Johann Bernoulli, University of Groningen, the Netherlands

Several classification models exist for software visualization (SoftVis) tools. Such models can be used to compare the features provided by several tools to the requirements of a given use case, in the process of selecting optimally fitting tools. However, it is not easy to predict how such models will perform when used to predict the adoption of SoftVis tools at large, especially for tools which were not considered during the model design.

Here, we consider an existing classification model that aims to provide generic guidelines for comparing SoftVis tools for corrective maintenance (CM) based of their features perceived as desirable by users. Although this model explicitly captures several such features, it is not evident that tools that fit the model will be found effective by users in practice.

This paper tests the above hypothesis by presenting a comparative evaluation of four software visualization (SoftVis) tools used in CM. The tools were selected to fit well the desirable criteria captured by the model under evaluation. Four independent groups of professional software developers participated in the evaluation, each group using a different tool to solve the same CM task on a real-world code base under typical industry conditions. The results show matches between the features described by the model as highly desirable and and those observed in practice to be essential for tool acceptance, *e.g.* IDE integration, extended search capabilities, multiple views, scalability, and the need for both dynamic and static visualizations; weakly relevant features, *e.g.* the commercial tool status; and features which do not influence acceptance, *e.g.* 3D and animation. Besides showing the correlation between the classification model and observed practice, our study further refines the model's criteria seen as important for industrial acceptance of software visualization tools.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; I.6.9 [**Computing Methodologies**]: Simulation, Modeling, and Visualization—*Visual programming and program visualization*

General Terms: Human Factors

Additional Key Words and Phrases: Software visualization, Software maintenance

1. INTRODUCTION

Corrective maintenance (CM) covers activities involved with correcting faults that occur in software [Swanson 1976]. Software visualization (SoftVis) is advocated as one of the important types of methods which assists typical CM activities by supporting the process of understanding the structure and behavior of large and complex software systems.

A variety of SoftVis tools that aid in CM have been developed. Despite this, an important *tool adoption* problem still exists [Storey et al. 2008; Tilley and Huang 2002]. For a tool to be adopted in the industry, it has to be effective and efficient in carrying out several specific tasks [Sensalire and Ogao 2007b; Zayour and Lethbridge 2001; Sun and Wong 2004]. Additionally, it is desirable if the tool matches the way professional software engineers (users) carry out their work [Storey et al. 2008]. This presents challenges for both users and developers of SoftVis tools for CM: *Users* want to efficiently assess if a given SoftVis tool fits a given set of desirable criteria in order to adopt (or reject) it, ideally without having to use the tool for a long period of time. Tool *developers* want to know which features they should concentrate on to maximize tool adoption chances so they do not waste effort implementing irrelevant features.

Many studies have been conducted in the past to support the claims of usefulness of specific SoftVis tools in industrial contexts [Storey 1998; Bassil and Keller 2001; Pacione et al. 2003; de Alwis et al. 2007; Subrahmaniyan et al. 2008; Telea and Voinea 2008; Hoogendorp et al. 2009]. Relatively fewer studies focus on making explicit the generic criteria that make a SoftVis tool useful and/or adopted [Bassil 2000; Bassil and Keller 2001; Sun and Wong 2004]. Among these are observational studies, controlled experiments, case studies, surveys, and empirical evaluations [Penta et al. 2007; Tonella et al. 2007]. Several such methods, notably controlled experiments and evaluations, have been used in the past with varying results. Several difficulties are mentioned when evaluating the effectiveness of SoftVis tools: the lack of a uniform professional user base of sufficient size to validate the claims [Storey 1998]; the reluctance of industry professionals to invest the necessary effort to learn the tools sufficiently and to invest time in the evaluation itself [Subrahmaniyan et al. 2008]; and the relative academic flavor or lack of maturity of some SoftVis tools which makes them unsuitable for industrial evaluation [Storey 1998; Bassil and Keller 2001].

Another difficulty in comparing and extrapolating the results of many evaluations is their *specificity* to a task or tool. This is particularly visible when we look at CM. Several SoftVis tools, and evaluations thereof, have been done for the more general tasks of program comprehension, reverse engineering, reverse architecting and redocumentation, in the context of adaptive and preventive maintenance. However, CM has its own specific goals, ways of working, and challenges, so it is not self-evident which of the results of SoftVis tool evaluations done for other maintenance areas are valid for CM.

There is very little work done in assessing the *generic* desirable features of SoftVis tools used for CM in a way that addresses effective tool selection for users and effective development focus on most relevant features for tool builders. In previous work, we gathered such desirable features based on developer interviews and proposed a tool classification model for matching tools with users and tool developers with user requirements [Sensalire et al. 2008; 2009]. In this paper, we take subsequent steps towards refining and validating this model. First, we use the model to select several industry-strength SoftVis tools which, according to the model, should be suitable for CM. Next, we measure the actual effectiveness and acceptance of these tools in a concrete CM task in the industry. Each tool is used by a separate group of professional software developers to solve the same CM task on the same software code base. The results of this study show that the features outlined as desirable by the model correlate well with those named as desirable during actual tool usage, and thus support the model's usefulness claims. Additionally, our study brings more insight in the challenges of adopting SoftVis tools for CM in the software industry.

The structure of this paper is as follows. In Section 2, we overview the challenges of evaluation and adoption of SoftVis tools, with a focus on CM. Section 3 details our aim: gathering evidence for the validity of the chosen classification model, using concrete tools and CM tasks. Section 5 presents the tools selected using the model, as well as the code base on which CM was performed. Section 6 details the user group, tasks, and evaluation procedure. Section 7 describes the results and discusses the various threats to validity. Section 8 discusses additional points observed during our study. Finally, Section 9 concludes the paper and outlines directions for future work.

2. RELATED WORK

SoftVis tools aim to help engineers during the software lifecycle by visual representations of the structure, behavior, and evolution of various types of software artifacts [Stasko et al. 1998; Zhang 2003; Diehl 2007]. Several challenges exist here. As early as 1993, it was seen that a ”significant gap exists between program visualization research and its practical application to programming environments and design tools” [Roman and Cox 1993]. Identical statements appear in later studies, *e.g.* [Storey 1998; Bassil and Keller 2001; Pacione et al. 2003; Telea et al.] This gap between SoftVis tool development and industrial SoftVis tool acceptance has risen strong questions from researchers [Reiss 2005], some even questioning an ”end of the line for Software Visualization” [Charters et al. 2003]. In the wider context of data visualization, a similar concern was formulated as the ”death of visualization” due to a lack of customers [Lorensen 2004].

These issues are well-known by many researchers. To better understand the factors influencing tool acceptance, many studies have been conducted. These can be classified in two groups:

(1) evaluations of desirable *features* for SoftVis tools, *e.g.* [Price et al. 1993; Bassil and Keller 2001; Sensalire et al. 2008; Kienle and Müller 2007]. Evaluations are effective in collecting general requirements that users find important, but do not measure how (much) a given SoftVis tool matches these. For example, Kienle and Müller identify seven quality attributes (rendering scalability, information scalability, interoperability, customizability, interactivity, usability, and adoptability) and seven functional attributes (views, abstraction, search, filters, code proximity, automatic layouts, and undo/history) [Kienle and Müller 2007]. Similarly, in our previous work, we proposed a tool classification model which outlines several so-called desirable features, *i.e.* rendering scalability, interoperability, interactivity, views, search, and filters [Sensalire et al. 2008]. In contrast to the more general work in [Kienle and Müller 2007], the work in [Sensalire et al. 2008] focuses specifically on CM and focuses on widely available tools rather than on research tools. More importantly, a model is proposed to translate desirable features into scores for selecting 'good' tools. The question in this paper is whether this model, and its score mechanism, effectively helps doing such a selection.

(2) evaluations of concrete SoftVis *tools*, *e.g.* [Pacione et al. 2003; Sun and Wong 2004; Subrahmaniyan et al. 2008; Marcus et al. 2005]. Such studies work conversely from desirable feature evaluations, *i.e.* decide upfront which features are desirable, implement them in a tool, and check the validity of the decision by testing the tool on a task.

Yet, such studies often do not detail how the features were chosen, and are often done in research settings which do not replicate all industrial challenges [Koschke 2003; Pacione et al. 2003; Reiss 2005; Marcus et al. 2005].

Maletic *et al.* classify SoftVis tools along five axes: task (why is visualization needed), audience (who uses the visualization), target (what is the data source), representation (how is data represented), and medium (where is data depicted) [Maletic et al. 2002]. This model is, however, not directly intended as a means to select tools to optimally match a given task or user group, but more as a general tool-and-task taxonomy. Our work here fits this model as follows (see also Sections 3 and 5):

—*task*: Program comprehension during code-level corrective maintenance;

—*audience*: Software developers involved in debugging a given code base;

—*target*: The code base maintained by the developers;

—*representation*: Implicitly given by the visual metaphors supported by the SoftVis tools selected by our classification model [Sensalire et al. 2008].

—*medium*: We focus solely on tools that use a typical PC setting (color monitor, keyboard, and mouse), as IT developers at large usually have no access to richer media such as stereo displays, tabletops, immersive virtual reality, or haptic devices.

Marcus *et al.* identify three types of SoftVis tool evaluations: interviews with users, case studies done in small groups (typically the tool builders), and usability studies with subjects that solve a concrete task [Marcus et al. 2005]. Our study here combines the first and third aspects: Twenty-one professional developers use four SoftVis tools to solve a concrete CM task. We measure the tools' usability implicitly and explicitly by checking the success of the CM activity and measuring the perceived usefulness of various tool features by questionnaires and interviews. However, in contrast to [Marcus et al. 2005], our ultimate aim is to assess the suitability of a tool classification *model* in selecting good tools rather than the suitability of *specific* tools.

Only few specific studies exist on the effectiveness of SoftVis tools in CM. Baecker *et al.* present early results in using animation, dense pixel displays, and auralization for debugging [Baecker et al. 1997]. Ko *et al.* observed expert programmers doing corrective and preventive maintenance using IDEs [Ko et al. 2005], aiming to elicit design requirements for future IDEs. They stressed the need for dependency visualization in IDEs to increase programmer effectiveness. Our work implicitly tests their hypothesis as we analyze how user performance is enhanced by several types of dependency visualizations. Alwis *et al.* compared three SoftVis tools supporting program modification tasks [de Alwis et al. 2007]. They interviewed professional developers and found that the studied tools had no significant impact on their efficiency. Reiss *et al.* analyzed two SoftVis tools for understanding the behavior of Java buggy code [Reiss and Renieris 2005]. They felt that the studied, and many other, tools did not address the reality of program understanding and made recommendations for improving SoftVis tools, *e.g.* the need for scalability and making the offered benefits explicit [Reiss 2005]. Sensalire *et al.* discussed the factors that influence case studies on SoftVis tools for CM [Sensalire et al. 2009].

3. CONTEXT AND AIM OF CURRENT WORK

3.1 Classification Model

In earlier work, we proposed a classification model for desirable features of SoftVis tools for CM [Sensalire et al. 2008]. The model emerged from a three-part formative longitudinal study carried out over a period of two years (see Fig. 1). In the first study, expert developers were given three SoftVis tools and asked to carry out several comprehension and maintenance tasks [Sensalire and Ogao 2007a]. Their feedback produced a preliminary model of tool desirable features. In the second study, this model was compared against ten SoftVis tools to validate and refine the initially elicited desirable features in a larger context [Sensalire and Ogao 2007b]. In the third study, 16 programmers used 15 SoftVis tools for several CM tasks: analyze source code for errors, find classes and class interactions involved in a given error, generate and analyze traces, and debug. The subjects expressed the features seen as desirable for the given CM tasks and graded the tools according to how they provided these features [Sensalire et al. 2008]. This delivered a SoftVis tool classification model focused on CM tasks which organizes desirable features in a two-level hierarchy. This model is shown in Table 3.1 (for full details, see [Sensalire et al. 2008]).

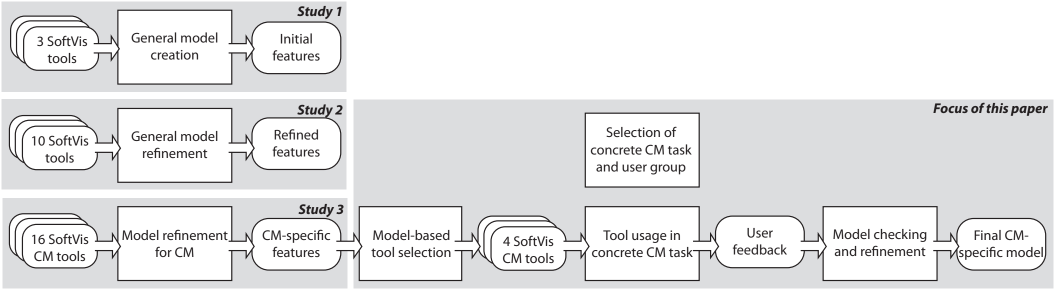


Fig. 1. Context of current study in refining the SoftVis tool classification model

3.2 Aims

Following the case study design theory of Yin [Yin 1994], our three formative studies above lead to the formation of a *hypothesis*: If a SoftVis tool for CM scores high on most of the model's desirable features, it is likely to be effective in addressing the targeted CM tasks. Yet, this hypothesis must be tested from various angles. In this paper, we consider two such angles:

—*correctness:* Is a SoftVis tool that scores high on the model's desirable features indeed useful for actual users in a CM task?

—*completeness:* Do other desirable features, beyond those covered by the model, exist in practice?

In the remainder of this paper, we study the proposed model's correctness and completeness. We proceed as follows (see also Fig. 1):

(1) use the model to select four SoftVis tools for CM that fit well, but up to different degrees, the model's description of a 'good' tool;

Category	Subcategory	What it describes:
Effectiveness	Scalability	The size of the software code bases the tool can handle efficiently
	Integration	How the tool interacts data- and control-wise with other tools in a workflow
	Query support	Genericity, flexibility, and customizability of provided queries on software
Tasks supported	Code smells detection	Ability to detect a set of potentially hazardous coding constructs
	Trace analysis	Ability to visualize program execution information (traces)
	Debugging support	Ability to visualize information generated during debugging
	Refactoring	Provision of semi-automatic visual code refactoring functions
Availability	Languages supported	Set of programming languages understood by the tool
	Licensing model	Requirements on tool usage (fee-based, shareware, open source)
	Platform	Requirements the tool places on its running environment (platform)
Techniques used	2D/3D metaphors	Usage of two- and three-dimensional visual representations
	Animation	Usage of animation to depict the operation of algorithms
	Color usage	Usage of colors to depict data attributes
	User interaction	Type of direct interaction (selection, visual manipulation, navigation)
	Multiple views	Presence of linked multiple views for showing cross-correlations
	Information density	Amount of software artifacts shown simultaneously on a screen
	Dynamic visualization	Ability to visualize dynamic (execution) information

Table I. Tool classification model consisting of four main categories with several sub-categories

(2) select a specific CM task and expert user group;

(3) ask the users to perform the CM task using the selected tools;

(4) gather actual quantitative and qualitative data on the tools' effectiveness in solving the task;

(5) interpret the data to reason about the model's completeness and correctness.

For the correctness analysis (step 4 above), we first assess the tools' effectiveness by measuring the ability to complete the tasks successfully, the task completion duration, and also gather detailed user feedback on the tool's usefulness (Section 6.2). Next, we test how these data correlate with the classification model. Specifically, we check if the developers who were able to solve the given task (or not) perceived that the tool helped them in doing so (or not). For the completeness analysis (step 4 above), we analyze the written and oral qualitative feedback from users in a post-study phase and check if the users required additional tool features which were not captured by the model (Sections 6.3 and 7)

4. STUDY STRUCTURE

Our study follows the *single explanatory case study* design, following Yin [Yin 2003; 1994]. Case studies are the method of choice when the phenomenon under study (in our case: the effectiveness of SoftVis tools for CM) is not readily distinguishable from its context (in our case: the CM process itself); the research question is of the type ”why” or ”how” (in our case: ”why and how is a SoftVis tool effective for CM?”); the investigator has little or no possibility to control the studied phenomenon (in our case: how developers use SoftVis tools for CM); and when the study object is a contemporary phenomenon in a real-life context. Our case study is of the explanatory (causal) type, as wetest the hypothesis ”if a SoftVis tool fits the model's requirements, then it is highly likely to useful for CM”.

The used methods involve both quantitative (measured) variables, *e.g.* developer experience, ability to complete task, task duration, accuracy, and correctness; and qualitative variables, *e.g.* the answers to the questions in the post-study phase (Sec. 6.3).

Following [Yin 1994] again, our study is of a *single-case* design, the case being the usage of SoftVis tools in CM. The *units of analysis* are the individual experiments in which developers solve a CM task using one of the SoftVis tools selected to fit the model (Sec. 6). The *theoretical proposition* is that a SoftVis tool which fits the classification model is highly likely to effectively support CM. The *logic linking data to propositions* states that, if a tool fits the model (by experiment construction), and it is effective in CM (as measured by the experiment data), then it likely supports our proposition. The *criteria for interpreting the findings* use qualitative textual and oral user feedback to interpret the hard measurements (task completion time and accuracy), and thereby elicit the reasons of *why* a SoftVis tool, or tool feature, was useful (or not) for a certain CM task.

Our study can be classified as an *expert review* study. In contrast to user studies which require tens of participants, expert reviews involve a relatively small number of expert participants (software professionals with significant industrial experience, in our case) and do not require sophisticated test software or strict performance measures [Tory and Möller 2005]. Expert reviews are increasingly advocated as an effective evaluation instrument for visualization applications [Preece et al. 2002; Freitas et al. 2002; Gabbard et al. 1999].

In the following sections, we detail the actual experiment: the selected SoftVis tools, the source code to be maintained and actual tasks, the participants, and the post-study questionnaire used to collect feedback.

5. TOOLS

We describe first the evaluated SoftVis tools. We restricted the selection to tools which comply with the following:

—the tools are usable for supporting CM activities;

—the tools score medium to high on most features deemed desirable by the classification model. These cover the most tool usability and effectiveness factors mentioned in the literature (Section 2);

—the tools are widely available, well known, and mature. This was done to remove any potential bias present in *e.g.* not fully operational, or research-grade, tools;

—the tools target the same programming language, run on the same platform, and integrate with the same IDE (Eclipse). This was done to diminish further bias caused by technical factors which are not relevant for our hypothesis testing. We specifically restrict selection to tools that work as IDE plugins, given the many studies that stress the crucial impact of tool integration in adoption and effectiveness *e.g.* [Tilley and Huang 2002; Koschke 2003; Charters et al. 2003; Schafer and Menzini 2005; Telea and Voinea 2008].

Practical constraints on our access to professional programmers willing to invest time in the study made us choose a Java code base (see Sec. 5.5). This further restricts the tool selection to those which seamlessly support Java.

The final tool selection, as well as the scores of these tools against the classification model is shown in Table II. We evaluated the tools' fitness to the model by reading their documentation and using them on several small-scale, example-like, programs. This replicates the typical way in which users assess new SoftVis tools in practice [Tilley and Huang 2002; Subrahmaniyan et al. 2008; Koschke 2003]. All tool scores were obtained based on

	EFFECTIVENESS			TASKS SUPPORTED			AVAILABILITY		
TOOLS	Scalability	Integration	Query support	Detect smells	Trace analysis	Debugging	Languages	Licensing	Platform
CodePro	High	Medium	High	High	Low	Low	Java	Comm.	Linux, Win
Ispace	High	Medium	High	High	Low	Low	Java	Free	Linux, Win
SonarJ	Medium	Low	Medium	High	Low	Low	Java	Comm.	Linux, Win
SolidSX	High	High	Medium	Medium	Low	Low	Java, C++, .NET	Comm.	Win

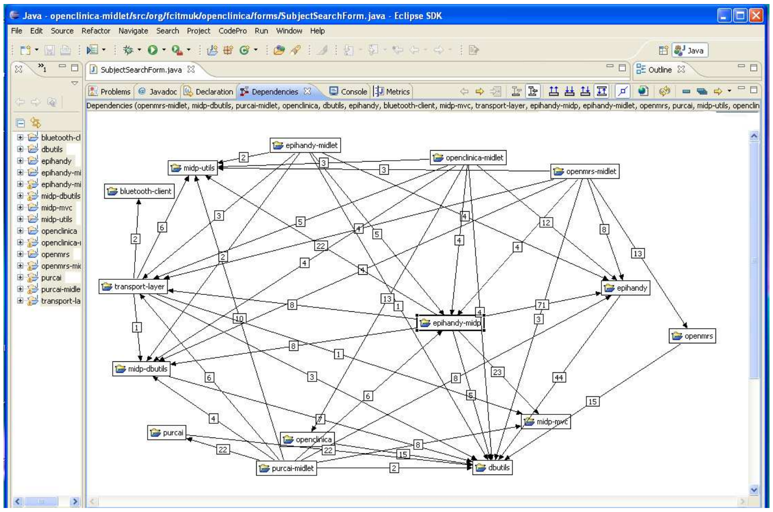
	VISUAL TECHNIQUES USED							
TOOLS	2D	3D	Animation	Color usage	User interaction	Multi views	Info. density	Navigation	Dynamic viz
CodePro	Yes	No	No	Medium	Medium	High	Medium	Yes	No
Ispace	Yes	No	No	Medium	Medium	High	High	Yes	No
SonarJ	Yes	No	No	Medium	Low	Medium	Low	Yes	No
SolidSX	Yes	No	Yes	Medium	High	Medium	High	Yes	No

Table II. Tools against desirable features of the classification model

the classification model *solely*, and with *no* prior discussion with the developer group from this studyt or actual fitness testing for the chosen CM tasks (described further in Sec. 6.2). CodePro, SonarJ, and SolidSX are commercial tools with fully functional trial licenses. Ispace is an open-source tool. All tools show the same level of maturity, documentation support, ease of installation, and overall ease of use. To remove further bias, all tools were used on comparable PC machines running Windows XP and Eclipse. The four tools are briefly presented next.

5.1 CodePro Analytix

This tool plugs into the Eclipse environment and adds several visual functionalities to the Java IDE that can be used during CM [Instantiations, Inc. 2008]. Given a Java project loaded within Eclipse, the plugin adds several menus to the folders and files in the project, containing options for code auditing, computing code metrics, code coverage, and analyzing dependencies. Visualizations include various types of force-directed and hierarchical graph layouts for extracted containment and dependency relations, optionally annotated by metrics, such as the example shown in Figure 2.



Java - openclinica-midlet/src/org/fcitmuk/openclinica/forms/SubjectSearchForm.java - Eclipse SDK

Dependencies (openmrs-midlet, midp-dbutils, purcai-midlet, openclinica, dbutils, epihandy, bluetooth-client, midp-mvc, transport-layer, epihandy-midp, epihandy-midlet, openmrs, purcai, midp-utils, openclin

Fig. 2. CodePro Analytix visualization plugin (actual screen snapshot from the tested code base)

5.2 Ispace

Ispace is also an Eclipse plugin targeted at visualizing Java code [I. Aracic 2008]. The tool enables drilling down from displayed dependency graphs to the code level, and also allows code changes to be synchronized with the displayed views. The visualization options are roughly analogous with the ones available in CodePro, but based on a different implementation of layouts, interaction and navigation options, and a slightly different look-and-feel. Figure 3 shows a snapshot of an Ispace view showing containment and dependency relations between classes.

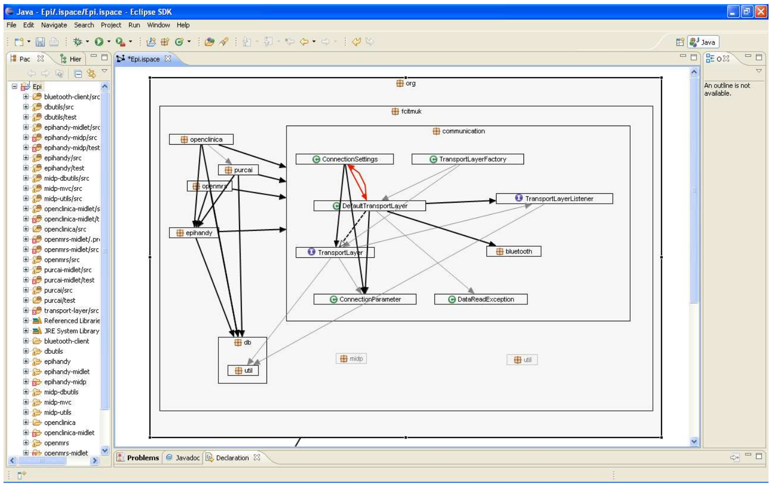


Fig. 3. Ispace visualization plugin (actual screen snapshot from the tested code base)

5.3 SonarJ

SonarJ is a program comprehension tool that provides functions for navigating the structure, dependencies, and source code of Java programs [hello2morrow 2008]. The tool provides multiple views on a system's structure, including architectural views, code metrics, hierarchical aggregation of dependencies, and selection of software entities and dependencies that violate a set of user-specified architectural or design rules. SonarJ can be used as a standalone tool or integrated with the Eclipse IDE. Figure 4 shows a visualization of dependencies between the packages of a Java system produced by SonarJ, with dependencies highlighted based on a user query.

5.4 SolidSX

SolidSX can work both as a standalone tool and as an Eclipse plug-in, similar to SonarJ. It provides static source code analysis, dependency and structure extraction, metric computation, and visualization [SolidSource 2009]. Several views are available: classical tree views, treemaps [Shneiderman 2010], table lenses [Rao and Card 1994], and a new visualization for hierarchy-and-dependency graphs which uses the highly scalable, clutter-reducing, hierarchically bundled edges layout [Holten 2006]. Static analysis covers Java (using the Recoder framework [Ludwig 2009]), .NET (including C# and Visual Basic), C,

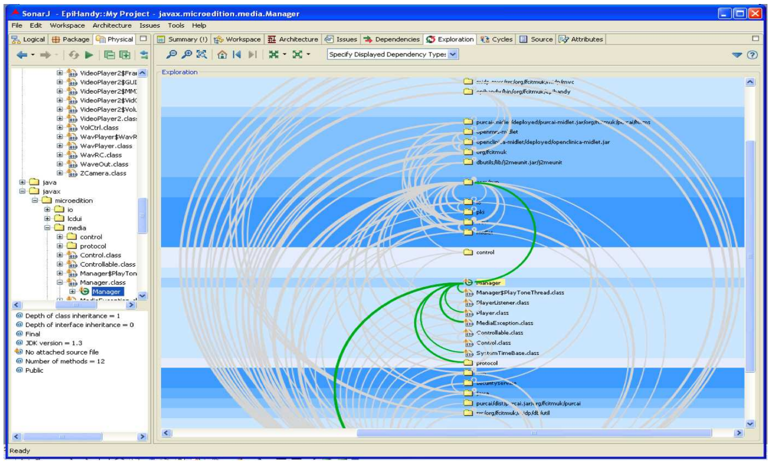
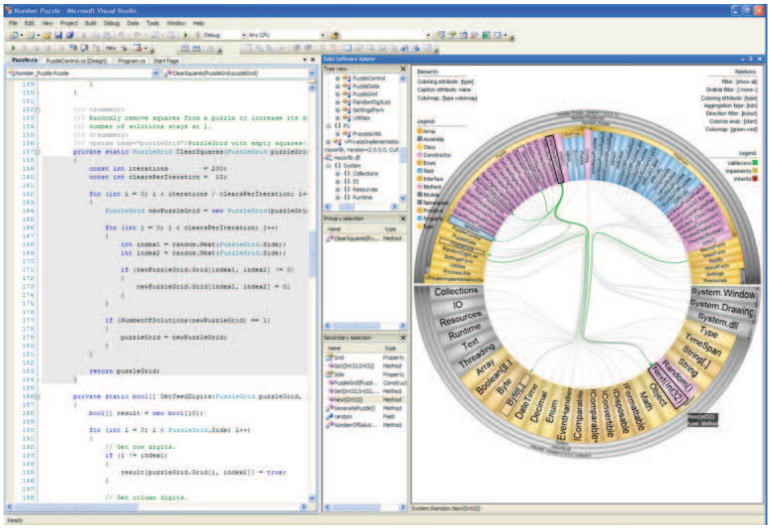


Fig. 4. SonarJ visualization plugin (actual screen snapshot from the tested code base)

and C++, covering parsing, syntax and semantic analysis, structure and dependency extraction, and quality metrics computation. All views can be correlated with text editors in a target IDE (Visual Studio or Eclipse) by means of socket-based plug-ins. Navigation mechanisms include selections, queries, and visual layout parameterization based on metrics. Figure 5 shows a snapshot of SolidSX displaying the structure, dependencies, and code quality metrics of a Java system.



```
            }
        }

        /// <summary>
        /// Randomly remove squares from a puzzle to increase its d
        /// number of solutions stays at 1.
        /// </summary>
        /// <param name="puzzleGrid">PuzzleGrid with empty squares
        private static PuzzleGrid ClearSquares(PuzzleGrid puzzleGri
        {
            const int iterations        = 200;
            const int clearsPerIteration =  10;

            for (int i = 0; i < iterations / clearsPerIteration; i+

            {
                PuzzleGrid newPuzzleGrid = new PuzzleGrid(puzzleGri

                for (int j = 0; j < clearsPerIteration; j++)
                {
                    int index1 = random.Next(PuzzleGrid.Side);
                    int index2 = random.Next(PuzzleGrid.Side);

                    if (newPuzzleGrid.Grid[index1, index2] != 0)
                    {
                        newPuzzleGrid.Grid[index1, index2] = 0;
                    }
                }

                if (NumberOfSolutions(newPuzzleGrid) == 1)
                {
                    puzzleGrid = newPuzzleGrid;
                }
            }

            return puzzleGrid;
        }

        private static bool[] GetUsedDigits(PuzzleGrid puzzleGrid,
        {
            bool[] result = new bool[10];

            for (int i = 0; i < PuzzleGrid.Side; i++)
            {
                // Get row digits.
                if (i != index1)
                {
                    result[puzzleGrid.Grid[i, index2]] = true;
                }

                // Get column digits.
```

Fig. 5. SolidSX visualization plugin (actual screen snapshot from the tested code base)

5.5 Source Code

The code base used for the CM activities of this study is EpiHandyMobile [Kayiwa et al. 2008], an extension of EpiHandy, which is a mature Windows Mobile data collection tool. EpiHandyMobile extends the base application, EpiHandy, to support Java-enabled phones using Java Mobile Edition (J2ME). The code base is relatively large, containing 5886 methods in 207 classes, with an average of 8.59 lines of code per method, and a total of 12051 lines of source code (excepting system packages and libraries).

6. EVALUATION PROCEDURE

6.1 Participants

In total, twenty-three programmers participated in the SoftVis tools evaluation, distributed in two roles. First, two developers who had advanced knowledge of the code base (Section 5.5) were selected as control users. One of them was actually the main developer of the code base. They assisted in evaluating the correctness of the CM task completion, *i.e.* determined if the activities carried out by the other group members did solve the given task or not. Next, twenty-three professional developers were assigned to the evaluation, six for CodePro (CP), six for Ispace (IS), three for SonarJ (SJ), and six for SolidSX (SX), as shown in Table III.

All participants were sought from software development companies. A pre-study questionnaire was used to ensure that they had all needed skills, *i.e.* very high fluency in Java, competence in software development with emphasis on CM, knowledge of J2ME, and familiarity with Eclipse and other modern IDE's. Familiarity with any SoftVis tool was also queried, but not used in the selection.

In several tool evaluations, participants get only very limited time to learn the tool to evaluate (under one hour). For example, in [Sensalire et al. 2008], users got a 15-minutes training for the tools they had to evaluate; over half of those users, however, felt that a developer needs a few days to really grasp the features of a new tool in practice, and advised this for further studies. We followed this point and provided the subjects of our current study with 3 days of tool study time and a compact user manual explaining the tool's main features. The subjects could call back on a 'trainer' (one of the authors) several times during this training phase, in order to receive additional help with the tools. To get a clear feeling that the tools were well understood, we had the subjects use the SoftVis tools on their own code during this training phase. Before starting the actual CM task, we confirmed with the subjects that they did not have unanswered questions on the SoftVis tools that they had to use.

The participants were clearly told that the study organizers were not related in any way to the evaluated tools. Finally, following the recommendations in [Sjøberg et al. 2007] on improving experiments that involve industrial developers, the participants were paid nominal fees in order to increase both their dedication and lower the risk of drop-out.

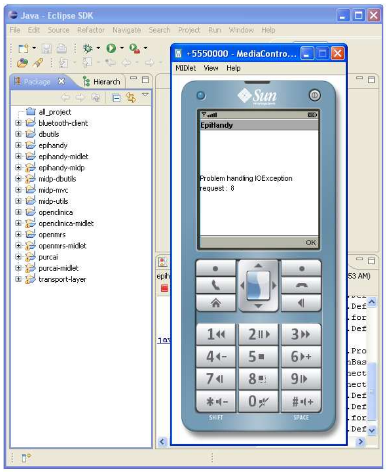
6.2 Tasks

All four developer groups received the following use case: A user would like to download study lists from a server. For this, he opens the Phone Emulator, clicks the *Run* option, starts the EpiHandy midlet, and logs on using the default username and password. Next, using the wireless connection emulator, he selects a given connection type, goes back to the main screen, and selects the *Download Study List* option. At this point, the application

User	Sex	Languages known	Programming experience (years)	CM experience (years)	IDE used for daily work	Ever used SoftVis tool	Prior code exposure
$CP1$	M	Java, C#, VB	$5 \ldots 10$	< 5		No	Yes
$CP2$	M	Java, PHP, VB	$5 \ldots 10$	< 5	NetBeans	Yes	No
$CP3$	M	Java, Python, HTML	$5 \ldots 10$	$5 \ldots 10$	command line developer	No	No
$CP4$	M	Java, VB, C#	$5 \ldots 10$	< 5	VS	Yes	No
$CP5$	F	Java	< 5	< 5	Eclipse	Yes	No
$CP6$	M	Java, C++, VB	$5 \ldots 10$	$5 \ldots 10$	Eclipse, VS	No	No
$IS1$	M	Java, C#, VB, .NET	< 5	< 5	VS, Eclipse	No	Yes
$IS2$	M	Python, Java, C	$5 \ldots 10$	$5 \ldots 10$	VS, Eclipse	No	No
$IS3$	M	C++, C#, Java	$5 \ldots 10$	$5 \ldots 10$	VS, Eclipse	No	No
$IS4$	M	Java, C, HTML	$5 \ldots 10$	$5 \ldots 10$	Eclipse	No	No
$IS5$	M	Java, VB	< 5	< 5	NetBeans	No	No
$IS6$	M	Java, C++	$5 \ldots 10$	>10	Eclipse	No	No
$SJ1$	M	Java, C, C++, Python	> 10	> 10	Eclipse, VS	Yes	No
$SJ2$	M	Java, C++	$5 \ldots 10$	$5 \ldots 10$	Eclipse, VS	Yes	No
$SJ3$	M	Java, C++, Python	< 5	< 5	Eclipse, VS	Yes	No
$SX1$	M	Java, C, C++, VB	$5 \ldots 10$	5	Eclipse, VS	Yes	No
$SX2$	M	Java, Python, VB	$5 \ldots 10$	$5 \ldots 10$	Eclipse, VS	Yes	No
$SX3$	M	C, C++, Java, C#	> 10	$5 \ldots 10$	Eclipse	Yes	No
$SX4$	M	Java, C, C++	5	5	Qt Creator, Eclipse	No	No
$SX5$	M	C#, .NET, Java	5	5	Eclipse, VS	No	No
$SX6$	M	C, C++, Java	5	< 5	Eclipse, KDevelop	No	No

Table III. Developers evaluating the CodePro Analytix (CP), Ispace (IS), SonarJ (SJ) and SolidSX (SX) visualization tools

throws an exception. A screenshot of this situation is shown in Figure 6.



Problem handling IOException
request : 8

Fig. 6. Application use-case showing a user provoked error

Given this use case, we asked each developer to individually perform corrective maintenance on the code base, *i.e.* find the problems and repair the code to ensure that the exception thrown is replaced by a user-friendly message telling what is happening and what the user should do to correct the problem. For this, they could use the Eclipse IDE, including the standard Java compiler and debugger, and one additional SoftVis tool (Section 5). The set-up of this CM task is very similar, both in context and complexity, to typical CM activities performed by mobile Java developers in their daily work.

The challenging part of the CM task is actually locating the code component (and its dependencies) that had to be modified to correct the problem. Although only one component had to be modified, finding it involved a thorough checking of the exception stack, after the exception was thrown, as well as the code itself, to identify the precise code section that needed to be changed in order to alter the displayed message. Considering the size of the code, and the fact that the developers were not familiar with it, except for the two control persons mentioned above in Sec. 6.1, the entire process is far from trivial.

6.3 Post-Study Questionnaire

After carrying out the CM task described in the previous section, which we also timed, we collected feedback from each developer by means of a post-study questionnaire. The questionnaire consisted in multiple choice and free-text answers, as shown below. The questions were explained in detail to the participants, and we sought their confirmation to be sure that they were understood fully and correctly.

a Was the SoftVis tool suitable/helpful for the required debugging tasks?
Answer: yes / no

b Would the tool have been more helpful if it was detached from the IDE
Answer: yes / no

c To what level did querying/ searching assist in fixing the problem
Answer: single choice:
—it did not assist at all
—it was a bit helpful but I wish it was more precise
—it was very helpful in its current form and would need no further improvements

d Which of the following factors would positively influence your decision to adopt this or a similar visualization tool to assist you in doing CM?
Answer: multiple choices possible:
—the tool displays its results using 2D graphics
—the tool displays its results using 3D graphics
—the tool provides animation
—the tool has a high use of color to show data
—the tool allows users to interact with the visualization
—the tool supports multiple simultaneous views of the source code
—the tool displays as much information as possible on one screen
—the tool supports dynamic visualizations of the program as it runs
—the tool could automatically point where there are potential errors and offer suggestions on how to overcome them
—the tool is free (alternative: I would not mind paying if it catered for my needs)

e What functionality did you miss in the visualization tool which would have helped in completing the given CM tasks?
Answer: Free text

f Any other comments / suggestions
Answer: Free text

Besides the questionnaire, we also engaged the participants in discussions to explain their answers in detail and provide any additional feedback, and noted down the comments made.

7. RESULTS

The results presented below are compiled from the two questionnaires filled in by the participants, *i.e.* pre-study (Section 6.1) and post-study (Section 6.3), as well as the results and timings of the CM tasks, which are presented in Table IV. We discuss these results along several axes: Familiarity with SoftVis tools, task completion and duration, used tool features, query facilities, IDE integration, and adoption factors.

7.1 Familiarity with SoftVis

The pre-study questionnaire (Table III) shows that most participants had never used a SoftVis tool before despite the fact that most of them had been programming at professional level as part of a software company for more than five years, a large part of which was spent in CM. This further emphasizes the point that SoftVis tools are not yet well adopted in the industry. Some of the reasons become more clear as we further analyze the other results, as follows below.

7.2 Task completion and duration

All users of CodePro, Ispace, and SolidSX successfully completed the assigned task, as measured by the two expert developers (see Section 6.1). All these users but one explicitly stated that the SoftVis tool was helpful (Table IV). Ispace and SolidSX users had lower completion times on the average (Figure 7), and Ispace was also perceived as more useful than CodePro (Table IV). Since both user groups did not know any of the tools prior to the study learn phase (Section 6.1), we may conclude that Ispace was easier to use than CodePro. This point is detailed further in Sec. 7.3 below.

In contrast, none of the three SonarJ users was able to complete the CM task successfully. The times reported in Table IV and Figure 7 for these users indicate the time until they gave up the task. The potential reasons for this outcome of Ispace are detailed separately in Section 7.6.

Within the same tool group, there are not very large variations of the required time, and we do not see a speed difference in favor of the two users having prior code exposure (Figure 7). Note that completion times include a combination of using the IDE *and* visualization plug-in tool for combined code understanding and error correction. It is hard to precisely quantify the amount of time spent in the 'visualization proper' (SoftVis tool) and the IDE itself, due to the tight integration and intertwining of the two. However, our qualitative observation of the users' behavior suggests that roughly a third up to half of the total time was spent in the SoftVis tool. Given this, we argue that the usage of the SoftVis tools had a relevant impact in the success, respectively failure, of the CM task.

The completion time within each tool group seems, further, to be correlated with the developers' experience (Table III vs Figure 7): The two users that have the highest completion times, $IS1$ and $CP5$, both within their group and globally, are also the only ones having under 5 years of programming and CM experience.

User	Completion duration (minutes)	Task completed	Prior code exposure	SoftVis tool was useful	IDE integration desirable
$CP1$	49	Correct	Yes	No	No
$CP2$	55	Correct	No	Yes	Yes
$CP3$	43	Correct	No	No	Yes
$CP4$	65	Correct	No	Yes	Yes
$CP5$	70	Correct	No	Yes	Yes
$CP6$	54	Correct	No	Yes	Yes
$IS1$	78	Correct	Yes	Yes	Yes
$IS2$	42	Correct	No	Yes	Yes
$IS3$	44	Correct	No	Yes	Yes
$IS4$	47	Correct	No	Yes	Yes
$IS5$	50	Correct	No	Yes	Yes
$IS6$	45	Correct	No	Yes	Yes
$SJ1$	40	Failed	No	No	Yes
$SJ2$	45	Failed	No	No	Yes
$SJ3$	60	Failed	No	No	Yes
$SX1$	50	Correct	No	Yes	Yes
$SX2$	40	Correct	No	Yes	Yes
$SX3$	42	Correct	No	Yes	Yes
$SX4$	55	Correct	No	Yes	Yes
$SX5$	48	Correct	No	Yes	Yes
$SX6$	52	Correct	No	Yes	Yes

Table IV. Post-study results for the CodePro Analytix (CP), Ispace (IS), SonarJ (SJ) and SolidSX (SX) tools

7.3 Used visualization features

The Ispace plugin provides only one type of view to show package relationships in a project, with drill-down functions enabling the user to move from packages to classes and then to code, which can be edited. In comparison, CodePro offers more advanced features like code auditing, metrics computation, generation of factory and test classes, analyzing dependencies, and finding code clones. SonarJ offers structure and dependency visualizations and drill-down navigation which is roughly similar to Ispace. Besides these, SonarJ offers a powerful set of architectural views and ways to check architectural rules. However, for our CM task, these latter features are less relevant. Finally, SolidSX offers several types of views: hierarchy-and-dependency layouts, treemaps, and table lenses, all linked with the code views. Its navigation features are roughly identical to SonarJ and Ispace, though implemented in different ways in its user interface. Its code analysis options are, however, much more limited than *e.g.* CodePro and SonarJ, being limited to simple code quality metrics like complexity, code size, comment density, fan-in, and fan-out.

All four tools provide a set of standard visual navigation features: a folder explorer to navigate the system hierarchy (see Figures 2 - 5), linked views, and source code views. There are few implementation differences at this level.

Apart from these features, all four tools provide ways to visualize *dependencies* such as calls, type uses, and inheritance relations. However, the tools use quite different techniques for this. Ispace and CodePro use a nested layout for containment relations and a spring embedder for dependencies, similar to the well-known SHriMP layout [Storey et al. 2002]. Navigation-wise, Ispace uses a drill-down method which enables both detail and overview: one can see both parent and children nodes in the same view. In contrast, CodePro shows only one hierarchy level while expanding or collapsing nodes. As such, the nested layout implemented by Ispace was perceived as delivering a higher information density than CodePro. Both CodePro and Ispace enable users to customize the dependency layout by manually rearranging nodes. In contrast, SonarJ and SolidSX use predefined, fixed, layouts: SonarJ lays out nodes along a fixed grid and draws dependencies as arcs, similarly to the technique in [Neumann et al. 2007] (see Fig. 4). This layout is reasonably effective in navigating hierarchically layered systems [Neumann et al. 2007], but is less effective, produces more edge clutter, and has a lower visual scalability, than the force-directed layouts used by CodePro and Ispace. Finally, SolidSX uses the novel hierarchical edge bundles (HEB) layout: the node hierarchy is drawn as a radial icicle plot and dependencies are drawn as splines bundled to match their end nodes' hierarchical positions. In SolidSX, node positions are determined by a user-chosen sort order and manual rearrangement is not possible.

During our study, we noticed that all users of all four tools mainly relied on dependency visualization and navigation to support their CM task, which is not surprising. Navigating hierarchy-and-dependency software structures follows the well-known information visualization principle of combining overviews with zooming and details-on-demand [Shneiderman 1996]. Given this, we argue that the above-mentioned technical differences in implementing this functionality strongly contribute in explaining the difference in user performance and completion success (Sec. 7.2). As one CodePro participant put it

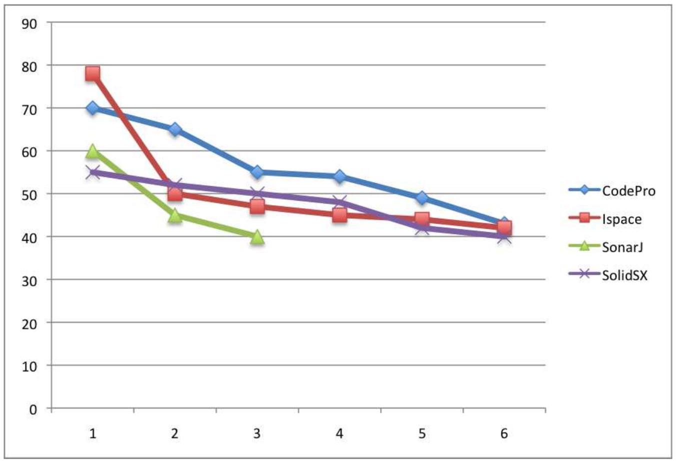


Fig. 7. Comparison of tool utilization duration until task completion of failure (y axis) for users of different tools. For each tool, users are sorted in decreasing completion time order (x axis) (Sec. 7.2)

> "The tool [CodePro] cannot drill-down to the methods. The dependency view provides just high-level information for classes and interfaces. This is of little value to the developers as class dependencies can be anyway traced more easily than methods. CodePro is good and its usefulness can not be ignored, but another view of dependencies is needed, apart from the tree. Maybe it could be a table enhanced with drill-down functions. This is important in case of large volumes of dependencies".

A similar comment, albeit in a stronger form, was made by the SonarJ users. In contrast, Ispace allows users to drill down up to the level of individual methods, thereby providing a quicker link and navigation to the source code itself.

All in all, there were several complaints of various degrees about all three dependency visualization (scalability, flexibility, visual presentation, navigation). Just as one example, one Ispace user required to hide/unhide the arrows (relations) between classes to reduce clutter. This feedback actually maps to the scalability points made when determining the desirable features: A scalable and clutter-free dependency visualization was, indeed, a main factor for tool acceptance degree [Sensalire et al. 2008]. The concrete feedback obtained in this study, both in the form of textual and timing results, proves the importance of this desirable feature for the tool success.

7.4 Query facilities

Figure 8 shows responses on how important the tools' query facilities were to the participants. While most users found this facility to have been helpful, preciseness was found to be lacking. Also, the tools' inability to integrate with the finer-grained Eclipse query functions was complained about, the weakest tool here being SonarJ. In particular, the fact that the various SonarJ views are not tightly linked by queries and selection makes their usage inefficient for our task of locating and following code elements (Section 6.2). One Ispace user felt that it would have been helpful if the tool could show all classes implementing an interface and including classes in other packages. This is a typical example of a complex query, strengthening our claim, also reflected in the classification model, that extensive query mechanisms are crucial to SoftVis tools in CM. Very similar limitations were reported for SolidSX.

One single user found the query facilities not useful at all, this being the expert in the CodePro group, who also had the highest code exposure (see Sec. 8.2 below). This user actually also did not find the SoftVis tool useful at all, so he used the built-in text query facilities of Eclipse.

Apart from limitations on the flexibility of specifying complex queries, over half of the users complained about the *presentation* of the query results. SonarJ, Ispace and CodePro do this by showing query hits as a list of hyperlinks to actual code fragments. It was found that this makes it hard to actually understand where these hits appear in the overall context of a software system. In contrast, SolidSX highlights hits directly on its radial HEB view, which shows a complete overview of the software, so enables easier reasoning about the hits' contexts.

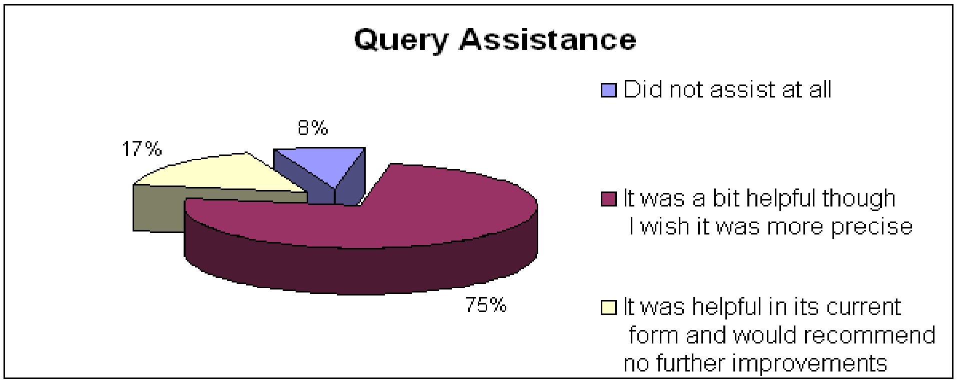


Fig. 8. Response to question 6.3c

7.5 IDE integration

All participants strongly agreed that the SoftVis tools would work better attached to the IDE than standalone (Table IV). Some users mentioned explicitly the need to have the visualizations integrated with the debugger so they could quickly locate classes or methods throwing exceptions. If the buggy areas could be highlighted, at runtime, with a different color for clearer identification, it would be better, one user suggested; another user suggested the ability to auto-highlight the path of an exception in the code so as to make it easier to troubleshoot. Such paths could be highlighted in sync in both the textual views as well as the dependency views.

SonarJ was perceived to have by far the weakest IDE integration of the four studied tools. Although SonarJ claims the capability of a smooth Eclipse integration, it appeared that many of its functions, such as its query facilities, hierarchy browser, and even the code views, are replicated functionalities rather than coordinated (linked) with the similar Eclipse features.

This high need for IDE integration matches observations in a wide variety of SoftVis tool evaluations and studies [Tilley and Huang 2002; Koschke 2003; Storey 1998; Charters et al. 2003; Schafer and Menzini 2005; Telea and Voinea 2008]. However, none of the previous references are actually from the context of SoftVis usage in CM. From our observations in the current study, compared with our own insight in earlier tool design [Telea and Voinea 2008; Telea et al. 2002; Voinea and Telea 2006; Hoogendorp et al. 2009], the IDE integration requirement is significantly stronger for SoftVis tools used in CM than for SoftVis tools used in other activities, due to the inherent tight communication needs with editors, debuggers, and compilers typical to CM.

7.6 SonarJ: Potential reasons for failure

If we summarize the apparent limitations of SonarJ, as indicated by its users, the following points stand out (in decreasing order of importance):

—*IDE integration:* the tool replicates several functions of Eclipse, such as browsing, code views, and queries, instead of (re)using the native functions. This makes users constantly navigate between the Eclipse and SonarJ views, thereby creating considerable usage overhead. This overhead was mentioned as the main element for not being able to complete the task;

—*Information density:* the dependency views use a fixed grid-based layout and arc-like dependencies, which do not scale to large systems, and create significant visual clutter;

—*Query support:* the limited amount of linkage between views upon selection and searching made the multiple views less collaborative, thus less useful in actually solving the given task;

The main issue here is that the above points are in line with the classification of SonarJ, which was performed independently on the actual CM task, user group, and code base (Section 5, Table II). That is, the classification model ranks SonarJ as 'weak' on the same desirable features which are mentioned by its actual users.

7.7 Adoption factors

Figure 9 shows the results to the question on which factors would positively influence an adoption decision for a SoftVis tool for CM. We asked this question separately in order to determine, first of all, whether the classification model dimensions are indeed seen as relevant by the users (note the similarity between question 6.3 d and the model's dimensions, detailed in [Sensalire et al. 2008]). The following facts were observed:

—*Dynamic visualization:* Over three quarters of the participants felt that dynamic visualization tools would be very helpful, which is also argued for in earlier studies [Baecker et al. 1997; Storey 1998; Koschke 2003]. Note that none of the tools evaluated here provided such features (Table II).

—*Show suggestions:* Two thirds of the participants also wanted tools to show suggestions on how to overcome errors while enabling simultaneous views into the source code. This was a less expected point, but nevertheless one that should be of interest to developers of SoftVis tools for CM. Such bug finding tools exist (see *e.g.* [Rutar et al. 2004]. However, we know of none which is integrated with a SoftVis front-end.

—*3D rendering and animation:* None of the participants felt that 3D rendering or animation would motivate adoption. In line with this, none of the evaluated tools provided such features, and there was no complaint for the lack thereof. We acknowledge that there is a certain split in the SoftVis research community between advocates and skeptics of the effectiveness of 3D software visualizations (see *e.g.* [Maletic et al. 2003; Wettel and Lanza 2008; Koschke 2003; Teyseyre and Campo 2009]). However, we suggest adapting the proposed classification model to remove 3D visualization and animation as 'desirable features', as these did not appear, in this study or our previous ones [Sensalire et al. 2008; Sensalire and Ogao 2007b], as being required by users. Animation needs a few clarifications. By animation, we mean here the use of tools that animate given *algorithms* at a high abstraction level, like *e.g.* in [Baecker et al. 1997]. This is in contrast to dynamic visualizations of executing code (mentioned earlier), where live data is extracted from the *runtime* object and visualized. The distinction is potentially important for tool success.

—*Commercial aspect:* Two thirds of the participants were willing to pay for the tool, while one third preferred a free tool. It is important to mention that our industrial users indicated *preference* for free tools, but this was not a strict requirement. Hence, non-commercial tool status was not seen as a decisive acceptance factor. The situation may be different for academic users.

8. DISCUSSION

In this section, we discuss additional points related to the organization and outcome of the presented study. For a more general description of the factors affecting the set-up of evaluations of SoftVis tools, we refer to [Sensalire et al. 2009].

8.1 Motivating participants

Although the participants were motivated using the fee payment, it was still highly challenging to get professional developers to invest enough time to do the complete round-trip of tool learning, studying the code base, using the tool within the Eclipse IDE to successfully perform the CM task, and fill in the questionnaire. This was clearly a non-trivial task, which required from the participants as much attention as they would invest when doing their regular work. However, this makes us believe that the obtained results are indeed valid in a real-world usage of SoftVis tools for CM. To increase the study's success, we

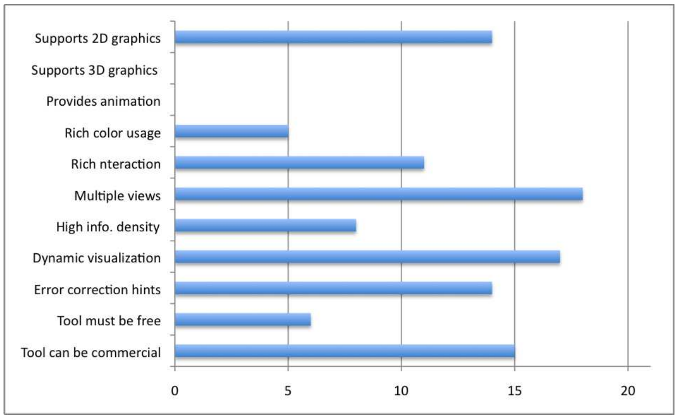


Fig. 9. Response to question 6.3d. For each subcategory of 6.3d, the number of respondents who responded positively is listed

allowed negotiations for higher fees than initially planned and accepted to carry out the study at the users' workplace at a time that was convenient for them. Had there been no payments, it would have been almost impossible to get this set of users to dedicate enough time and attention to do the study. This point on compensations is relevant when planning tool studies in industrial contexts, as also mentioned in [Sjøberg et al. 2007].

8.2 Adoption by experts

As mentioned earlier, there was a code expert in each of the CodePro and Ispace groups (Section 6.1). Interestingly, the Ispace group expert found the tool helpful in carrying out the task, while the CodePro expert, who had the highest code exposure, did not and eventually fixed the bug using only the source code and a text editor. The experts' detailed comments show a correlation between the usefulness of a SoftVis CM tool and the tool's ease of learn and use and specificity. The tool usefulness and user's familiarity with the code appear to be uncorrelated, as also noted in Sec. 7.2. Despite the fact that programmers are familiar with a given code base, they can still be assisted by a SoftVis tool, if the tool does not take up a lot of time to learn and use. However, due to prior knowledge about the code, such users are also very likely to be impatient with tools that do not quickly achieve what they desire. As such, it is very important to allow expert users to incorporate custom features into such SoftVis tools. Any wrong step with this group would lead to major adoption hurdles.

8.3 Evaluation constraints

As a general observation, the quality and generality of the results of tool effectiveness and acceptance studies are strongly dependent on the use of suitable study participants. Tools aimed at novice users, *e.g.* in education contexts, would be best evaluated using novices, whereas tools aimed at industry professional developers are best evaluated using such users [Sjøberg et al. 2007]. This is essential, and is noted that the role of human subjects in empirical evaluations is sometimes taken too lightly [Tonella et al. 2007; Storey 1998]. Similar observations are made by Koschke when distinguishing research and end-user contexts when evaluating future directions of software visualization [Koschke 2003].

In addition to this, there is also a need to elaborate on the procedure and aim of the evaluation in order to aid in future replication of the studies as well as usefulness of the results [Briand 2006; Penta et al. 2007; Tonella et al. 2007]. Although the efforts of setting up such evaluations are quite high, as described above, we do believe that the insight obtained is worth the price.

8.4 Model validation and refinement

As detailed in Section 3, we aimed to check the predictive abilities and completeness of a given tool classification model [Sensalire et al. 2008]. For the predictive aspect, we observed that all participants but one found the two tools that scored high in the classification model (CodePro and Ispace) as being suitable and useful for supporting the given CM task (see question 6.3a and Table IV). All participants using these tools completed the given task succesfully. We also noticed features explicitly named as desirable by users, and for which the tested tools score low in the model, *i.e.* dynamic visualization and showing debugging suggestions (Figure 9 vs the model's *dynamic viz.* and *debugging* dimensions (Table II). The participants using SonarJ, which scored lower on all three effectiveness dimensions and three of the visual techniques used (Table II) was not effective in supporting the given CM task. Furthermore, the textual and verbal comments on SonarJ indicated limitations mainly along these dimensions (Section 7.6).

Although not exhaustive, the information above suggests that the dimensions of the proposed classification model are, indeed, relevant for selecting effective SoftVis tools for CM. In other words, from the answers to questions 6.3b and 6.3d, as well as to the free-text comments (6.3e and 6.3f, discussed above, we observed that the users found the evaluated SoftVis tools useful (or not) *because* of the presence, or absence, of the features deemed as desirable by the classification model.

However, we also noticed dimensions of the classification models which seem less relevant, and as such, are possible candidates for elimination, *i.e.* the presence of animation and 3D visualization. To fully decide on the relevance of these dimensions, further study is needed *e.g.* by comparing SoftVis tools for CM that are equal along most other dimensions, except animation and 3D views.

We should stress again that, during the entire study, the users were not aware of the existence of a classification model, or the existence of a predefined set of so-called 'desirable features'. As such, we argue that the proposed classification model is indeed valid in the sense that a SoftVis tool meeting the features deemed desirable by the model, has a high chance of being useful in CM practice *because* of having these features. Beyond this point, however, we cannot claim more predictive power for the proposed model. However, the fact that a SoftVis tool would be perceived as useful without meeting most of the model's desirable features would be quite improbable, due to the fact that many other independent studies outline similar features as our model, for a wide range of use-cases outside code-level CM [Price et al. 1993; Kienle and Müller 2007; Sun and Wong 2004; Subrahmaniyan et al. 2008; Marcus et al. 2005; Maletic et al. 2002].

8.5 Threats to validity

We acknowledge several threats to validity concerning our study. Following *e.g.* Huberman and Miles [Huberman and Miles 2002] and Trochim and Donnely [Trochim and Donnelly 2007], we can classify these as follows:

8.5.1 *Internal validity.* Internal validity would be threatened if one were to conclude that a SoftVis tool under study was helpful in solving the given CM problem when the problem was solved otherwise. In our case, however, we explicitly asked the users to say whether, and how the tools were helpful or not (questions 6.3a, 6.3c and 6.3f). Moreover, in the five cases when the tool was not helpful, this was explicitly reported and justified by the users (see also Table IV). History, testing, and instrumentation threats were not present, given the actual set-up of the study. Expectancy bias was arguably removed by explicitly stating that the authors had no stakeholding in a positive or negative outcome (Sec. 6.1). Differential subject selection is a possible threat, which we tried to minimize (though not eliminate) by having a uniform mix of participants with various experience in each group (Table III).

8.5.2 *External validity.* External validity would be threatened if the hypothesis that the classification model can predict likelihood of (un)suitability of a SoftVis tool for CM were not to hold for other SoftVis tools or CM tasks or users. The fact that a SoftVis tool scoring very low on the model would be suitable for a wide range of CM tasks and users, is, however, very unlikely, given that the model's dimensions are in line with attributes identified as desirable by many other researchers, as mentioned in Sec. 8.4. Moreover, the type of CM task, selection of the SoftVis tools (widely used and mature), and developers from the software industry, are quite typical, we believe, for what the usage of a SoftVis tool in CM should be in the industry. Pretest effects are arguably small, as we did not notice that the users had already formed opinions on the (un)suitability of the tested tools at the end of the training period, except that they understood their operation and were willing to test them further (Sec. 6.1). Reactive effects to experimental arrangements were minimized by letting the users work individually in their own familiar environment.

8.5.3 *Conclusion validity.* Conclusion validity would be threatened if there were no relationship between a tool fitting the model and its perceived usefulness in practice. We tested three different tools with seventeen participants, where each participant worked independently. The feedback from the participants on the reasons why they (dis)like the tool they studied was quite similar. Moreover, as already mentioned in Sec. 8.4, this feedback relates directly to most model features, except for the animation and 3D views presence. A strong threat to conclusion validity would have implied that the users perceived their studied tool as (not) useful for *other* reasons than the ones accounted for the model's dimensions. Such feedback, if present, would have been outlined by the answers to questions refsec:poststudye and refsec:poststudyf.

8.5.4 *Construct validity.* There are several construct validity threats to mention. Mono-operation bias is clearly an issue: we used only one code base, CM task, three tools, and seventeen participants. Finding more participants that would qualify the pre-selection requirements was, however, not possible. This is a recurring issue in SoftVis research, and many SoftVis tool industrial evaluations actually use even fewer participants than we did, and even wider tasks and more different tools to compare [Marcus et al. 2005; Storey 1998; Tilley and Huang 2002; Koschke 2003; Subrahmaniyan et al. 2008; Sun and Wong 2004]. Determining professional programmers to invest days (as in our case) to learn and evaluate a tool, and actually use it to measurably solve a CM task on a non-trivial code base, is quite challenging. Many SoftVis studies focus just on the tool builders or students, which has a high risk of introducing external validity threats such as pretest effects, reactive effects, and internal validity threats such as expectancy bias and history threats. Mono-method threats are present in the sense that the number of measured variables (the questions) that quantify tool usefulness are limited. We tried to reduce, though not eliminate, this by designing questions that measure 'usefulness' in different ways, *e.g.* question 6.3a (direct one), 6.3d (willing to buy a tool is another usefulness measure), 6.3e (if a missing feature that helps solving the problem were present, the tool were arguably more useful). Interaction of setting and treatment threats are arguably low, in the sense that the users experimented in their own environment (their own companies and workplaces), independently, using the tools installed on their own computers. The main factors which would differ from a real usage of the SoftVis tools for 'own work' would be the usage of an externally imposed code base and the presence of the silent observer monitoring the experiment. Concerning the first point, however, our developers are used to work on commissioned tasks and third-party code on a regular basis under relative delivery pressure, so this factor should not differ markedly from their actual work patterns.

8.5.5 *Questionnaire design.* A separate potential threat to validity is related to the design of the post-study questionnaire (Sec. 6.3). For instance, following the funneling method described by Oppenheim [Oppenheim 1998], the order of the questions should start with broader questions and then narrow down to more specific questions, one of which is actually question 6.3a. A further recommendation is to follow the answers to the general questions by requests for clarification. Although our paper questionnaire was not designed in this way, the verbal clarifications sought to the participants after handing in the questionnaires followed this funneling pattern and the probing mechanism. Further design points that may pose as validity threats are the lack of "not applicable" or "don't know" categories to the questions, and the relative bias of question 6.3b towards answering it negatively.

Following Yin [Yin 2003; 1994], our current study is a single-case, multiple units of analysis (*i.e.* SoftVis tools) design. The study's conclusions are based on analytic generalization (which is possible from one single case), rather than statistical generalization (which would require, indeed, more sample points). Probably the most feasible way to increase statistical generalization is to test more tools, as finding more programmers fitting our preconditions proved too difficult. Yet, it is quite hard to find a large group of tools that share preconditions that make them comparable, *i.e.* address the same CM task(s); are mature and well-accepted by the developer community; and share the same target language, IDE integration, and platform. For example, there are simply very few SoftVis tools for CM that are integrated with the same IDE. Comparing tools that share less aspects is possible, but would require (a) the refinement of the classification model with additional dimensions, which was not our goal here; and (b) a larger user group or (c) increased pressure on the user group up to levels where they become uncooperative or superficial, a factor we tried to avoid at all costs (Section 8).

8.5.6 *Tool selection.* Interestingly, Ispace, which was found to be most useful by its users, is an open-source tool, while CodePro, SonarJ, and SolidSX are commercial tools. Mentioning this point is important, as there are many tool evaluations which suffer from the bias of comparing a less-than-mature open-source tool with a professional commercial tool. Given the evaluation outcome, and also the fact that no user pointed in the verbal or written feedback to 'tool immaturity' as a limiting factor, we believe that our measurement of tool usefulness is not biased by having selected immature tools.

9. CONCLUSION

In this paper, we have presented an evaluation of four representative software visualization (SoftVis) tools in the context of corrective maintenance (CM). Our aim was to check whether the potential usefulness of a SoftVis tool for CM, as determined by an existing classification model, does match indeed the opinions of actual professional developers using the tool for a concrete CM task. We described the set-up and results of an evaluation study that targets the above question. The study results are in line with the model predictions. This brings additional justification to the claim that this classification model is helpful in determining the usefulness of a SoftVis tool for typical CM tasks such as debugging. Specifically, the features identified by the model as desirable are indeed necessary for a SoftVis tool to be useful in CM. However, due to the limited number of test points, we cannot yet argue that the presence of these features is also *sufficient* for predicting a tool's usefulness.

During this evaluation, several desirable points named as crucial for acceptance were observed: tight functional integration within a recognized IDE, multiple correlated views, visualization-to-code navigation, advanced search capabilities, and scalable automated dependency visualizations. These are in line with the main factors of SoftVis tool acceptance in the industry, independently identified by other types of studies [Price et al. 1993; Bassil and Keller 2001]. Additional points which we found are the important need of dynamic visualizations, error-correction suggestions, and the observation that there is no significant difference between a tool's availability (free vs commercial) as long as it is highly effective for the users' needs.

To conclude, a well-chosen SoftVis tool can, thus, be of high value in typical debugging activities in the software industry. However, the above-mentioned features required for acceptance are not typically those that foster creative visualization research, but those that require high implementation efforts. As a SoftVis survey also states: "Overcoming the problem of high [implementation] investments with unclear benefits for visualization researchers is the real challenge in integration and interoperability" [Koschke 2003]. We believe that this is one of the key causes of the gap between SoftVis research and its industrial acceptance.

Apart from this, there is still a group of programmers who believe in the traditional way of doing CM. Apparently, no amount of functionality would entice them to use a SoftVis tool. As one programmer in our study put it:

> I do not work with tools/IDE's because it is like spoon feeding. I can get by with basic code editors like `vi`. I would therefore not be motivated to adopt a Softvis tool, no matter how good the functionality. Debugging is fun; with all these tools, the fun is taken away leaving no incentive to debug.

In the future, we consider performing similar studies for other specific sub-tasks of corrective maintenance, as well as considering different SoftVis tools. Additionally, an interesting direction would be the elaboration of a large-scale review of lessons and experiences learnt in using SoftVis tools in CM, having as outcome a refined classification model which could influence both users and researchers in their strategic tool choices over time.

REFERENCES

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

BASSIL, S. 2000. Evaluation qualitative et quantitative des outils de visualisation logicielle. MSc Thesis, Univ. of Montreal, Canada.

BASSIL, S. AND KELLER, R. K. 2001. Software visualization tools: Survey and analysis. In *Proc. Intl. Workshop on Program Comprehension (IWPC)*. IEEE, 7–17.

BRIAND, L. C. 2006. The experimental paradigm in reverse engineering: Role, challenges, and limitations. In *Proc. of the* 13^{th} *Working Conference on Reverse Engineering (WCRE)*. IEEE, 3–8.

CHARTERS, S. M., THOMAS, N., AND MUNRO, M. 2003. The end of the line for Software Visualisation? In *Proc. of the* 2^{nd} *Workshop on Visualizing Software for Analysis and Understanding (VISSOFT)*. IEEE, 27–35.

DE ALWIS, B., MURPHY, G. C., AND ROBILLARD, M. P. 2007. A Comparative Study of Three Program Exploration Tools. In *Proc. of the* 15^{th} *Intl. Conf. on Program Comprehension (ICPC)*. IEEE, 103–112.

DIEHL, S. 2007. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer.

FREITAS, C., LUZZARDI, P., CAVA, R., WINCKLER, M., PIMENTA, M., AND NEDEL, L. 2002. Evaluating usability of information visualization techniques. In *Proc. ACM CHI*. 36–45.

GABBARD, J., HIX, D., AND SWAN, J. 1999. User-centered design and evaluation of virtual environments. *IEEE Comp. Graphics & Applications 19,* 6, 51–59.

HELLO2MORROW. 2008. SonarJ Visualization Plugin. website. `www.hello2morrow.com/products/sonarj`.

HOLTEN, D. 2006. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE TVCG 12,* 5, 741–748.

HOOGENDORP, H., ERSOY, O., RENIERS, D., AND TELEA, A. 2009. Extraction and Visualization of Call Dependencies for Large C/C++ Code Bases: A Comparative Study. In *Proc. VISSOFT*. IEEE.

HUBERMAN, A. AND MILES, B. 2002. *The Qualitative Researcher's Handbook*. Sage Publications.

I. ARACIC. 2008. Ispace Visualization Plugin. website. `http://ispace.stribor.de`.

INSTANTIATIONS, INC. 2008. CodePro Analytix. website. `www.instantiations.com/codepro`.

KAYIWA, D., TUMWEBAZE, C., AND NKUYAHAGA, F. 2008. EpiHandy Mobile Code Base. website. `http://code.google.com/p/epihandymobile/`.

KIENLE, H. M. AND MÜLLER, H. A. 2007. Requirements of software visualization tools: A literature survey. In *Proc.* 4^{th} *Intl. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 2–9.

KO, A. J., AUNG, H., AND MYERS, B. A. 2005. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Proc. of the* 27^{th} *Intl. Conf. on Software Engineering (ICSE)*. IEEE, 126–135.

KOSCHKE, R. 2003. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *J. of Software Maintenance and Evolution* 15, 87–109.

LORENSEN, B. 2004. On the death of visualization: Can it survive without customers? In *Proc. of the NIH/NSF Fall Workshop on Visualization Research Challenges*.

LUDWIG, A. 2009. Recoder java analyzer. `recoder.sourceforge.net`.

MALETIC, J., MARCUS, A., AND COLLARD, M. 2002. A task oriented view of software visualization. In *Proc.* 2^{nd} *Intl. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 32–39.

MALETIC, J., MARCUS, A., AND FENG, L. 2003. sv3D: A framework for software visualization. In *Proc. of the* 25^{th} *Intl. Conf. on Software Engineering (ICSE)*. IEEE, 812–818.

MARCUS, A., COMORSKI, D., AND SERGEYEV, A. 2005. Supporting the evolution of a software visualization tool through usability studies. In *Proc. Intl. Workshop on Program Comprehension (IWPC)*. IEEE, 307–316.

NEUMANN, P., SCHLECHTWEG, S., AND CARPENDALE, M. 2007. ArcTrees: Visualizing Relations in Hierarchical Data. In *Proc. EuroVis*. IEEE, 53–60.

OPPENHEIM, A. 1998. *Questionnaire Design, Interviewing and Attitude Measurement*. Continuum.

PACIONE, M., ROPER, M., AND WOOD, M. 2003. Comparative evaluation of dynamic visualisation tools. In *Proc.* 10^{th} *Working Conf. on Reverse Engineering (WCRE)*. IEEE, 80–89.

PENTA, M. D., STIREWALT, R. K., AND KRAEMER, E. 2007. Designing your Next Empirical Study on Program Comprehension. *Proc. of the* 15^{th} *Intl. Conf. on Program Comprehension (ICPC)*, 281–285.

PREECE, J., ROGERS, Y., AND SHARP, H. 2002. *Interaction Design: Beyond Human-Computer Interaction*. J. Wiley & Sons.

PRICE, B. A., BAECKER, R., AND SMALL, I. 1993. A principled taxonomy of software visualization. *J. of Visual Languages and Computing 4,* 3, 211–266.

RAO, R. AND CARD, S. 1994. The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *Proc. ACM CHI*. 318–322.

REISS, S. 2005. The paradox of software visualization. In *Proc Intl. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 59–63.

REISS, S. P. AND RENIERIS, E. M. 2005. Tool demonstration: JIVE and JOVE: Java as it happens. In *Proc. of the* 25^{th} *Intl. Conf. on Software Engineering (ICSE'03)*. IEEE, 820–821.

ROMAN, G. C. AND COX, K. C. 1993. A taxonomy of program visualization systems. *IEEE Computer 26*, 11–24.

RUTAR, N., ALMAZAN, C., AND FOSTER, J. 2004. A comparison of bug finding tools for java. In *Proc. Intl. Symp. on Software Reliability Engineering (ACM ISSRE)*. 245–256.

SCHAFER, T. AND MENZINI, M. 2005. Towards more flexibility in software visualization tools. In *Proc. Intl. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 20–26.

SENSALIRE, M. AND OGAO, P. 2007a. Tool usrs requirements classification: How software visualization tools measure up. In *Proc. Intl. Conf. on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa (Afrigraph)*. ACM, 119–124.

SENSALIRE, M. AND OGAO, P. 2007b. Visualizing Object Oriented Software: Towards a Point of Reference for Developing Tools for Industry. In *Proc.* 4^{th} *Intl. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 26–29.

SENSALIRE, M., OGAO, P., AND TELEA, A. 2008. Classifying desirable features of software visualization tools for corrective maintenance. In *Proc. of the* 4^{th} *Symposium on Software Visualization (SOFTVIS)*. ACM, 87–90.

SENSALIRE, M., OGAO, P., AND TELEA, A. 2009. Evaluation of software visualization tools: Lessons learned. In *Proc.* 5^{th} *Intl. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 156–164.

SHNEIDERMAN, B. 1996. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. Symp. on Visual Languages*. IEEE, 336–343.

SHNEIDERMAN, B. 2010. Treemaps for space-constrained visualization of hierarchies. Website. `http://www.cs.umd.edu/hcil/treemap-history`.

SJØBERG, D., DYBÅ, T., AND JØRGENSEN, M. 2007. The future of empirical methods in software engineering research. In *Proc. Intl. Conf. on Software Engineering (ICSE)*. IEEE, 358–378.

SOLIDSOURCE. 2009. SolidSX software explorer. `www.solidsourceit.com/products/SolidSX-source-code-dependency-analysis.html`.

STASKO, J. T., DOMINGUE, J. B., BROWN, M. H., AND PRICE, B. A. 1998. *Software Visualization - Programming as a Multimedia Experience*. MIT Press.

STOREY, M. A. 1998. A cognitive framework for describing and evaluating software exploration tools. Ph.D. thesis, Simon Fraser University, Canada.

STOREY, M. A., BENNETT, C., BULL, R. I., AND GERMAN, D. M. 2008. Remixing visualization to support collaboration in software maintenance. *Proc. Frontiers of Software Maintenance (FoSM)*, 139–148.

STOREY, M. A., BEST, C., MICHAUD, J., RAYSIDE, D., LITOIU, M., AND MUSEN, M. 2002. SHriMP views: an interactive environment for information visualization and navigation. In *Proc. ACM CHI*. 520–521.

SUBRAHMANIYAN, N., BURNETT, M., AND BOGART, C. 2008. Software visualization for end-user programmers: trial period obstacles. In *Proc. SoftVis*. ACM, 135–144.

SUN, D. AND WONG, K. 2004. On understanding software tool adoption using perceptual theories. In *Proc. Intl. Workshop on Adoption-Centric Software Engineering (ACSE)*. IEEE, 51–55.

SWANSON, E. B. 1976. The dimensions of maintenance. In *Proc. of the* 2^{nd} *Intl. Conf. on Software Engineering (ICSE)*. IEEE.

TELEA, A., ERSOY, O., AND VOINEA, L. Visual analytics in software maintenance: Challenges and opportunities. In *Proc.* 1^{st} *European Symposium on Visual Analytics (EuroVAST*.

TELEA, A., MACCARI, A., AND RIVA, C. 2002. An open toolkit for prototyping reverse engineering visualizations. In *Proc. VisSym*. ACM, 241–249.

TELEA, A. AND VOINEA, L. 2008. An integrated reverse engineering environment for large-scale C++ code. In *Proc. SoftVis*. ACM, 67–76.

TEYSEYRE, A. AND CAMPO, R. M. 2009. An overview of 3d software visualization. *IEEE TVCG 15,* 1, 87–105.

TILLEY, S. AND HUANG, S. 2002. On selecting software visualization tools for program understanding in an industrial context. In *Proc. Intl. Workshop on Program Comprehension (IWPC)*. IEEE, 285–288.

TONELLA, P., TORCHIANO, M., BOIS, B. D., AND SYSTÄ, T. 2007. Empirical studies in reverse engineering: state of the art and future trends. *Empirical Software Engineering 12,* 5, 551–571.

TORY, M. AND MÖLLER, T. 2005. Evaluating visualizations: Do expert reviews work? *IEEE Comp. Graphics & Applications 5,* 25, 8–11.

TROCHIM, W. AND DONNELLY, J. 2007. *The Research Methods Knowledge Base (3^{rd} ed.)*. Atomic Dog Publ. `http://www.socialresearchmethods.net/kb`.

VOINEA, L. AND TELEA, A. 2006. CVSgrab: Mining the History of Large Software Projects. In *Proc. EuroVis*. IEEE, 187–194.

WETTEL, R. AND LANZA, M. 2008. Visually localizing design problems with disharmony maps. In *Proc. SoftVis*. ACM, 155–164.

YIN, R. K. 1994. *Case study research: Design and methods*. Sage Publications.

YIN, R. K. 2003. *Applications of Case Study Research (2^{nd} ed.)*. Sage Publications.

ZAYOUR, I. AND LETHBRIDGE, T. C. 2001. Adoption of Reverse Engineering Tools: A Cognitive Perspective and Methodology. In *Proc. Intl. Workshop on Program Comprehension (IWPC)*. IEEE, 245–258.

ZHANG, K. 2003. *Software Visualization: From Theory to Practice*. Kluwer.