A Framework to Evaluate the Effectiveness of Software Visualization Tools in Maintenance Activities

by

MARIAM SENSALIRE
Reg. No:2005/HD18/4004U
BSc (IT)(USIU), MSc Computer Science (Mak)
msensalire@cit.mak.ac.ug,(+256-712-933630)

A Dissertation Submitted to the School of Graduate Studies
in Partial Fulfillment for the Award of the Degree of Doctor of
Philosophy in Software Engineering
of Makerere University

November, 2009

I Mariam Sensalire do hereby declare that this Dissertation is original and has not been published and/or submitted for any other degree award to any other University before.

Signed———————————— Date—————————————

Mariam Sensalire
BSc(IT), MSc(Comp Sci),
Department of Networks
Faculty of Computing and Information Technology

This Dissertation has been submitted for Examination with the approval of the following supervisors.

Signed:——————————-Date:——————————-
Associate Prof. Patrick Ogao
Department of Information Systems
Faculty of Computing and Information Technology
Makerere University, Uganda

Signed:——————————Date:——————————
Prof. dr. Alexandru C. Telea
Institute of Mathematics and Computer Science
Faculty of Mathematics and Natural Sciences
University of Groningen, the Netherlands

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, software engineering is a full-grown industry, with many processes, methods, and tools in place. As a consequence hereof, the lifecycle of a software product, consisting of phases such as requirements analysis, architecting, design, development, testing, releasing, and maintenance, is very long, complicated, and expensive. Various analyses estimate that over 80% of the entire lifecycle costs of a software product are in maintenance, and that 50% of this cost relates to program comprehension [68, 164].

Software maintenance (SM) refers to the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment [52]. Maintenance can be further classified into four main sub categories *i.e.*, preventive maintenance, corrective maintenance, adaptive maintenance and perfective maintenance [127].

To reduce a part of the maintenance costs, several things can be done. A non-exhaustive list hereof includes reducing development, testing and architecting costs; using better, more effective programmers; using more cost-effective process management methodologies such as agile or lean development [106]; and reducing the cost of comprehension. From this palette of instruments, we focus in this dissertation is on the cost reduction process via better program comprehension techniques.

There are many ways to support program comprehension. To be able to reason about them, we have to understand why comprehension is difficult in the first place. A set of causes are involved here: Software is large, complex, abstract, changes in time, and involves many people in its construction. All these aspects, both taken individually and even more so in their interplay, make comprehension difficult. To support better, more effective ways to understand large, complex, abstract, and changing programs, several types of techniques have been proposed. Examples hereof source code and dynamic behavior analyses, formal methods, quality metrics, reverse engineering techniques, and visualization techniques. In this thesis, we will examine the last type of technique: Software visualization (SoftVis).

In brief, software visualization supports program comprehension by making the abstract concrete; reducing the scale of an entire software (sub)system to one or more compact images; and making the dynamics of software, *e.g.,* its behavior and also its changes in time, visible in space. To this end, software visualization maps the many attributes of a software system, such as structure, behavior, and evolution attributes, to visual elements, such as shape, position, spatial order, color, size, texture, and animated behavior. As such, software visualization reduces the exploration of a large, abstract, complex and changing software system to the hopefully intuitive task of visual exploration and navigation of an image.

Specific program comprehension tasks are assisted by specific visualization techniques, which are in turn implemented by specific visualization tools. A comprehension-related visualization tool is therefore 'effective'when, at the end of the day, it helps actual users (software engineers) to accomplish their comprehension-related tasks, *e.g.,* perform their maintenance activities, better [69]. A visualization tool is 'efficient'when it enables its users to perform their maintenance within a desired time frame.

Hence, the efficiency and effectiveness of visualization tools that support program comprehension are intimately related to a specific test of tasks. To identify whether a given visualization tool is indeed efficient and/or effective, or to build such a new tool, there is a need to identify the requirements that such tools are to satisfy [64, 79, 117, 120]. Furthermore, we need to be able to easily quantify whether a given tool, or set of tools, does indeed satisfy this identified set of requirements.

1.1 Background

Many visualization tools have been produced, to support many activities which are part of the maintenance process. Originally promoted within research and academic environments, such tools have, in the last years, been increasingly developed and advertised by commercial sources. However, the actual effectiveness of a given software visualization tool, or even of the software visualization discipline at large, is still debated in the software industry. It is not yet clear which aspects of one or more visualization tools are perceived as truly useful by industry practitioners, and which not. Considerably more software visualization tools are developed by researchers than the number which gets actually used, on a wide scale, by industry developers. Several complementary reasons for this situation may be named, including limited support and advertising, limited user awareness to the benefits of visualization, and lack of appropriate training. In this thesis, we however focus on a different class of reasons: We argue that there exists a disconnect between the actual *needs* of the industrial users and the actual *features* provided by software visualization tools.

To assess the effectiveness of visualization tools, or the lack thereof, researchers have conducted user studies. However, on the wide scale of all software visualization tools in existence, such studies are not frequent. Moreover, many of them are specific to a task, user group, and tool. This makes it difficult to extrapolate from these studies to the entire field in order to get a better understanding of the above-mentioned disconnect between users and tools. In the end, this situation affects both the tool users, as they are offered tools which are perceived as suboptimal from their perspective, but also the tool builders, as they may design tools which are not addressing the core requirements of actual users.

We cannot expect to do an exhaustive study of all tools and all needs. What we hope, however, is that there are consistent 'aspects' in the way that industrial users perceive software visualization tools in general. In particular, we are interested in understanding the process of *early adoption*[1], *i.e.* the phase in which users discover new, potentially useful, SoftVis tools, and the criteria they use to determine, after very short usage periods, whether a tool has usefulness potential or not.

After a tool successfully passes the early adoption phase, however, it does not mean that the tool will indeed be useful. The tool assessment enters a second phase, where users look at more fine-grained details, having concrete tasks and specific requirements in mind. To understand this process further, we decided to zoom in on a subset of maintenance activities (and related SoftVis tools), namely corrective maintenance. Corrective maintenance (CM) is a branch of software maintenance that deals with correcting errors in software so as to ensure that the software operates in the ways it was designed to [135]. CM is an intensive activity that involves many sub-activities. These include, but are not limited to, checking for problematic areas in source code; correcting the problems identified; and carrying out regression tests to ensure that the correction has not created further problems within the software.

In order to carry out CM tasks successfully, maintainers need a clear understanding of the software system under maintenance [148]. This is crucial as failure to understand the various dependencies and interactions within code can lead to the introduction of new errors after carrying out a CM task [54]. Many aids can be used in order to gain such knowledge of a software system. System documentation is, or should be, usually the first option. Unfortunately, software documentation is frequently inadequate or missing altogether, which leads to confusion for the maintainer that relies on it [59, 54].

In a recent study on documentation practice during corrective maintenance, it was found that only 16% of the organizations studied update their software system documents at all granularity levels, while only 21% provide their employees with guidelines on how to write system documentation [59]. It would therefore be risky for a maintainer carrying out CM tasks to base their system knowledge and understanding solely on documentation.

Due to the difficulty of CM coupled with the inconsistency of software documentation, tools can aid in the various comprehension-related activities of CM [54]. SoftVis tools are an example of such tools. Despite this, there has not been extensive research that correlates software visualization tools with corrective software maintenance. As such, tool developers targeting this area do not have a lot of resources on the requirements for these tools. This makes the understanding of the challenges related to acceptance in the industry of CM-related SoftVis tools even more difficult than understanding such challenges for SoftVis tools that attempt to support program comprehension in other types of software engineering activities. This is the

[1]The term *early adoption* is sometimes used with a different meaning, *i.e.* the fact that a tool or method gets quickly adopted after it has been created. Here, by early adoption, we understand a user's decision of adopting a tool after a limited, short, evaluation phase.

gap we aim to further investigate in this research.

1.2 Statement of the Problem

Many designers have embarked on developing software visualization (SoftVis) tools. Despite this, builders of SoftVis tools for the general task of software comprehension, as well as for corrective maintenance in particular, often see that their tools are not extensively used by their ultimate target audience: developers working in the software industry. One reason hereof is a gap between the features that such tools offer and the actual features that their target audience requires [91].

This situation is complicated further by the relatively limited number of usability studies and empirical evaluations of tools that are developed [56]. There are several cases where such tools are evaluated on an audience that significantly differs from the target audience for the tool being developed. We argue that this is one of the implicit reasons that leads to the fact that many of the existing SoftVis tools are not widely used in practice. Refining this idea, we argue that, in the absence of a guide for tool developers that is backed by a user study and that highlights which features are truly desirable (and which not), tools that are not in line with the targeted user requirements are likely to be developed.

General Objective

The aim of this thesis is to provide a framework for the study of the match, or lack thereof, between the needs of the software engineering community (with a focus on maintenance) with respect to visualization tools, and the provided facilities in existing visualization tools.

Specific Objectives

The specific objectives of this thesis are to:

i Determine and analyze the desirable properties of software visualization tools from the perspective of expert programmers in the industry;

ii Design a Unified Requirements Categorization (URC) for modeling the early adoption process of SoftVis tools;

iii Refine and evaluate the URC for software visualization tools for corrective maintenance;

iv Validate the refined URC against software developers carrying out actual corrective maintenance tasks.

Scope

In this work, emphasis was put on expert programmers and developers, as these users are a good reflection of the requirements that a SoftVis tool should meet to gain acceptance in the real-life software industry. Also, emphasis was put on the corrective maintenance of object-oriented software systems such as written in languages as Java, C++ or C#. The reason hereof is that such systems arguably exhibit a more complex structure, *e.g.,* in terms of the types of relations, hierarchy, and types of software attributes, than plain procedural language software such as C or Pascal, and therefore are more interesting targets for the added value claimed to be delivered by SoftVis tools. In the remainder of this thesis, the software being targeted by the studied SoftVis tools will be implicitly assumed to be object-oriented, unless otherwise specified.

To address the various steps of the goals outlined above, we have conducted several studies, as follows.

A first study was done to elicit several so-called general *desirable features* that developers have for SoftVis tools involved in program comprehension during maintenance. The aim of this step is to get a first-level understanding of what developers feel that such tools should provide in general, regardless of the specific type of maintenance task that is to be supported.

From the insights of this first study, we noticed that tool adoption is a multiple phase process. Like in other adoption processes involving humans, there appears to be a first, early, phase in which users decide, based on limited information and investigation time, whether a tool has the potential to be *usable* or not. A second phase involves a more in-depth analysis of the tool's provisions in order to decide whether the tool is *useful* or not (for a specific task). Based on this insight, we attempted to capture the types of desirable features prominent in the early adoption phase in a so-called Unified Requirements Classification (URC). Based on the compatibility of these URC elements, it may be possible to understand the types of decisions that users take to decide whether the tool has potential to be usable (and thus examine it further in deeper detail), or not (and thus reject the tool and possibly continue searching for a different tool).

The third step in our work is a study of way zooms in on the second phase of adoption, where a tool's usefulness is assessed in connection to a specific activity. To better understand this phase, we have chosen to focus on a more specific type of software activity, namely corrective maintenance, and SoftVis tools that aim to support it. The outcome of this step, based on a study of several SoftVis tools for corrective maintenance, is a refined requirements classification that models desirable features for such tools. Just like our first-level model, this second-level model can be used in several ways: to understand *a posteriori* why a given tool does match a given usage context; to check *a priori* which tools, from the large palette of available tools, are best suited in a given usage context; and to assist tool developers in the shaping of their tool development for high chances for adoption by a specific target audience.

The fourth step in our work attempts to validate the previous model for desirable features of SoftVis tools for corrective maintenance by means of an actual case study. Tools were selected for a given corrective maintenance task, based on the model's previsions. Next, the actual selected tools were used by actual developers to accomplish a given task related to the domain of application of the tools. The developers were next interviewed in the effectiveness of using the tools, and the results were correlated with the model predictions.

1.3 Contributions

- A Universal Requirements Classification (URC) that documents desirable features of software visualization tools that are used for the general task of software comprehension.
- A Requirements Classification for software visualization tools for Corrective Maintenance (CMRC) which refines the URC to present desirable features for software visualization tools used in corrective software maintenance.
- A guide in the setting up and executing of experiments involving software visualization tools.
- Finally, this thesis has also resulted into the publications shown below:

i M. Sensalire, P. Ogao and A. Telea. Evaluation of Software Visualization Tools: Lessons Learned. In Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), Edmonton, Canada, September 2009.

ii M. Sensalire, P. Ogao and A. Telea. Requirements for Software Visualization Tools for Corrective Maintenance: How Effective is the Classification Model? Submitted to the Springer Journal of Empirical Software Engineering, 2009. *Currently under a second round review with the journal.*

iii M. Sensalire, P. Ogao, and A. Telea. Classifying desirable features of software visualization tools for corrective maintenance. In Proceedings of the 4th ACM symposium on Software Visualization (SOFTVIS), Germany, September 2008.

iv M. Sensalire and P. Ogao. Tool users requirements classification:how software visualization tools measure up. In Proceedings of the 5th ACM International Conference on Computer Graphics, Virtual Reality, Visualization and Interaction in Africa (AfriGraph), Grahamstown, South Africa, 2007.

v M. Sensalire and P. Ogao. Visualizing object oriented software:towards a point of reference for developing tools for industry. In Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), Banff, Canada, 2007.

1.4 Thesis Format

The rest of this thesis is organized as follows (see also Figure 1.1 which presents a schematic overview of the workflow).

- **Chapter 2 - Literature Review**
  In this chapter, previous work in software visualization, as well as evaluations of software

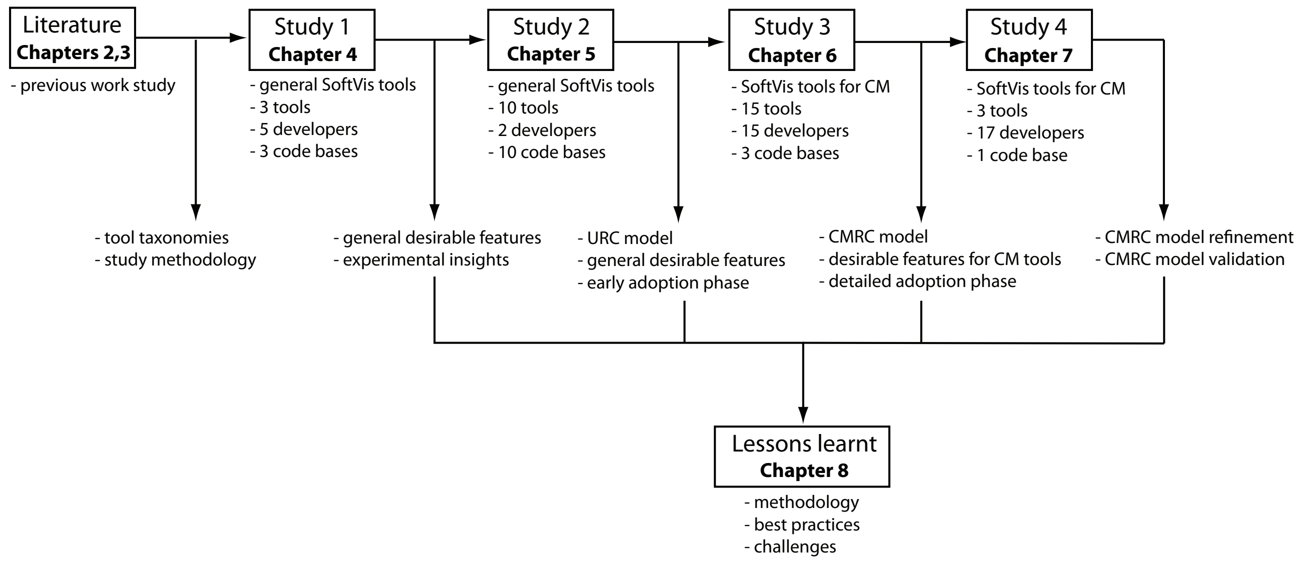


Figure 1.1: Workflow of the research presented in this thesis (Chapters 2 to 8)

visualization tools, is discussed. The link between the previous work and what has been covered in this thesis is detailed, outlining the gaps that still necessitate additional study.

- **Chapter 3 - Methodology**
  In this chapter, empirical research methods commonly used in tool studies similar to the studies described in this thesis are discussed, along with the advantages and disadvantages of each. The triangulation research methodology, which was central to our work, is further elaborated on with a justification on why it was suitable for our research.

- **Chapter 4 - Software Visualization Needs For General Software Comprehension**
  In this chapter, a first exploratory study is presented that aims to elicit the views of expert programmers on what should be incorporated in a software visualization too for it to be usable. These views are determined after exposing the programmers to three SoftVis tools that support the general task of software comprehension and allowing them to use the tools for a given short period of time. The results from these observations form the basis of further work aimed at producing a model of desirable features of SoftVis tools identified by developers in an early examination phase, which is described in the next chapter. The potential need for a point of reference for developing tools for the industry is also discussed.

- **Chapter 5 - Unified Requirements Classification**
  Various ways of categorizing software visualization tools have been developed in the past, as discussed in Chapter 2). This chapter presents a classification of tool requirements aimed at modeling the early adoption phase of SoftVis tools, based on the results from Chapter 2 as well as on previous taxonomies and research results. Ten software visualization tools that differ in their functionalities are then measured against this categorization in order to show the extent to which they fulfill the requirements that are desired by

tool users at a first glance. We do not attempt here to do a tool-against-tool comparison in terms of effectiveness of solving a specific problem on a given code base, but rather compare and assess the tools against a set of global requirements perceived as desirable, in an effort to address the early tool adoption issue, introduced earlier in Chapter 1, on a more general level.

- **Chapter 6 - Software Visualization for Corrective Maintenance**
  This chapter provides an evaluation of 15 software visualization tools aimed at corrective maintenance from the perspective of usefulness. In this phase, different types of requirements are important as compared to the early adoption phase, modeled by the URC introduced in Chapter 5. As such, the classification model is further refined for specific properties and requirements of SoftVis tools aimed at corrective maintenance. The result is a set of Corrective Maintenance Requirements Classification for SoftVis tools (CMRC). By observing trends of current SoftVis tools, tool developers can gain additional insight on what to consider, avoid or improve in their tools. Tool users can also recognize what to broadly expect from software visualization tools for corrective maintenance, before actually using the tool. The CMRC can function as a basis for an informed choice of a tool to use both for tools that have been evaluated in this chapter, but also extrapolated to other tools that share requirements and/or features discussed in the unified requirements classification.

- **Chapter 7 - Validating the Classification Model**
  This chapter presents a comparative evaluation of three software visualization tools used in the context of corrective maintenance which were selected with the aid of the CMRC classification model presented in Chapter 6. Two tools were chosen to fit the model's requirements well, whereas a third one was chosen to have a lesser fit. Three groups of professional software developers participated in the evaluation with each group using a different tool to solve the same concrete corrective maintenance tasks. The aim of the experiment was to validate the effectiveness of the proposed model as well as seek further recommendations for improving SoftVis tools used in the context of corrective maintenance.

- **Chapter 8 - Lessons Learned**
  This chapter presents a compilation of the lessons learned during our four studies involving SoftVis tools and industrial users. Several aspects are outlined related to methodological points involved in setting up studies, executing the actual study, and interpreting the results. The aim of this discussion is to outline important challenges and difficulties involved in such experimental work, in the hope of helping further researchers who wish to carry out such studies.

- **Chapter 9 - Conclusion**
  This chapter presents the conclusions of our research described in this thesis and outlines areas where future work can be carried out related to evaluation of SoftVis tools with respect to features deemed to be desirable by actual users.

Chapter 2

Literature Review

Significant work has been done in many of the areas related to the research questions addressed by this thesis. This chapter reviews the literature related to these areas, with the aim of outlining existing results and positioning our work with respect to existing gaps between the research questions, stated in Chapter 1, and these results.

In section 2.1, we review the basics of software comprehension. Section 2.2 presents an overview of previous work related to software visualization aimed at program comprehension. Section 2.3 discusses the need for software visualization to support a given application domain. The importance of empirical studies and validation of the effectiveness and efficiency of software visualization tools is discussed in Section 2.4. Related work aimed at practically comparing various software visualization tools is discussed in Section 2.5. Finally, Section 2.6 discusses requirements for software visualization tools that have been identified by earlier scholars.

2.1 Software Comprehension

Software comprehension refers to the process of understanding and reasoning about software [155]. Software practitioners need to understand the various artifacts of the development process, including source code, documentation, and execution log data, in order for them to carry out their various development and maintenance tasks effectively. In the following, our main focus is going to be on the understanding on source code and related structural data, *e.g.* design and architecture information which can be extracted from the source code. The tasks referred to above range from actual software design to corrective, perfective, and preventive maintenance [81].

In an effort to document the software comprehension process, several studies have been carried out in the past [153, 108]. These studies indicate that, when analyzing software for comprehension, several models can be used depending on the size of the software artifact dataset under study, as well as on the intention of the programmer [95]. These include the bottom-up model, the top-down model, as well as the integrated comprehension model. These models are outlined next.

The Bottom-Up Model

When this approach is used, the source code under observation is studied incrementally from a small section upwards, until the whole program is understood [95, 131]. In situations involving unfamiliar (new) code or limited programmer knowledge, this approach is predominant. It is also seen to provide the best cognitive results when studying medium to small-scale programs [154][1]. However, this approach is harder to use as the program size increases since the small sections under study become too many to correlate, and/or the dependencies between such sections become too numerous to store in the 'working memory' of the person that performs the analysis [108].

The Top-Down Model

This model proposes that a hypothesis is first formed, which is then proved or disproved accordingly until the whole program or desired segment is understood [154]. This process is achieved with the aid of beacons, or clues (denoted (1) in Figure 2.1)), that guide in the breaking of a larger hypothesis into smaller ones (denoted (2)..(4) in the same figure) [131]. This approach is more scalable than the bottom-up approach, and can also be used when the programmer is experienced either in the subject domain or is familiar with the source code itself [154, 153]. One of the reasons hereof is that a knowledgeable programmer is more likely to make an informed hypothesis, which arguably leads to a shorter overall time to successful hypothesis validation [95, 108].

The Integrated Comprehension Model

This model proposes a combination of both the top-down and the bottom-up methods of software comprehension. When the software being studied is large, a combination of both methods is found to be more effective for comprehension than using each method in isolation [131, 148]. The combination entails several repeated passes of bottom-up and top-down comprehension analyses during which information is gathered, hypotheses are made and tested, and the accumulated knowledge gets incrementally refined.

[1]Giving an exact quantitative definition of the *scale* of software systems is a difficult task. Here and in the following, we qualitatively distinguish between small-scale, medium-scale, and large-scale programs mainly from a source code size perspective. Small-scale programs range around a few hundreds to thousands lines of code (LOC). Medium-scale programs range from thousands to tens of thousands of LOC. Large-scale programs range from tens of thousands of LOC upwards.

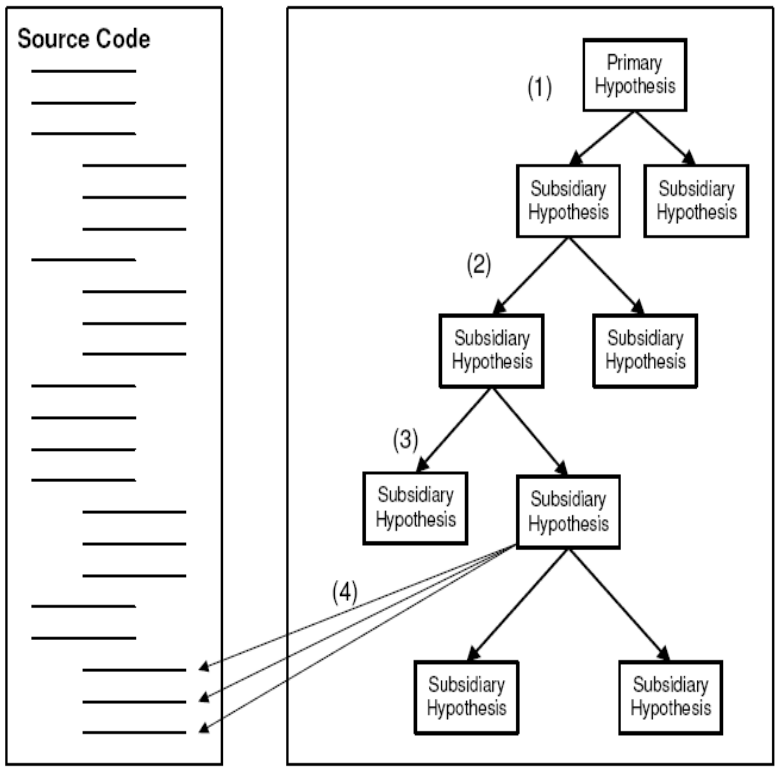


Figure 2.1: Hypothesis generation process [16]

2.2 Software Visualization and Software Comprehension

Software visualization is the use of interactive computer graphics, typography, graphic design, animation, and cinematography to enhance the interface between the software engineer or the computer science student and their programs [107]. Similar definitions worded slightly differently are used in the literature, *e.g.* the use of graphics techniques to support program comprehension [29]). In the context of this thesis, we refer to a tool as being a software visualization (SoftVis) tool if it is implemented to serve the aims, and using the technology, outlined in the definition above [2].

Software visualization and software comprehension are related procedures. Software visualization can be seen generically as an instrument (in the methodological sense), supported by actual tools, for supporting the various processes involved in program comprehension. The overall claim of software visualization is that it can reduce the time and effort required by the person that aims to understand a software system, both in providing answers to given questions (hypothesis validation) but also in helping the formulation of questions (hypothesis creation) [57, 156]. As such, software visualization relates to the general aims of data and information visualization of 'confirming the known and discovering the unknown' [118].

To support these claims (and aims), it is important for tools that claim to support human

[2]In the remainder of this thesis, the unqualified term *tool* will always refer to software visualization tools

cognition to support, or match, the theories on which these claims were based [155]. However, we argue that there exists a certain gap between the comprehension theories outlined in Section 2.1, *i.e.* bottom-up, top-down, and integrated comprehension, and the way in which existing SoftVis tools provide features that support (or not) these theories. In the following, we support this argument by examples from a (non-exhaustive) list of existing work in evaluating SoftVis tools with respect to a set of program comprehension tasks.

Storey *et al.* had earlier noted the importance of program comprehension theories and advocated for the use of results from program comprehension studies to support the design of tools [131]. Related to this background, a cognitive framework that considers the maintainers' comprehension needs was designed. As an outcome hereof, several software exploration tools have been developed, such as the SHriMP tool. Our work builds onto Storey's results as we expose software maintainers to SoftVis tools and extract the developers' views on how these tools can be improved, based on their comprehension activities. In relation to this point, one of the tools which we have explored was Creole [21], which implements the SHriMP visualization metaphor.

According to Petre *et al.*, there are several cases in which a mismatch between software visualization and software comprehension can lead to undesirable results [103]. An example hereof are animation tools that use code-steppers [57]. This suggests that not all software visualization tools have the usefulness that their creators originally claim, or imply, directly or indirectly. Petre *et al.* further suggest bridging the gap between desirable features and tool provisions by seeking the tasks that the programmers carry out and then deciding how the support for these tasks can be provided. This is one of the points explored further in this thesis, with a focus on corrective maintenance.

Sim developed a search tool called `grug` (**gr**ep **u**sing **G**CL) whose main aim was to support program comprehension [120]. Two studies were carried out and the results hereof were used as supporting evidence for the tool construction. In the first study, the Portable Bookshelf (PBS) tool was given to software maintainers and observations were carried out during tool use. Results indicated that a search facility would be useful, which was later added to the tool. The second study observed users with the modified tool and the feedback led to the development of `grub`. Our work relates to that of Sim [120] as our users were also exposed to tools with the aim of eliciting both the desirable and undesirable features . The difference is that we did not develop a new tool, but rather gathered a set of requirements and desirable features that a potential tool developer can use from existing tools.

Knight proposed customizable views and multiple browsing as the best way to support comprehension in SoftVis tools [67]. This was done with the guide of comprehension studies involving the SHRIMP technique proposed by Storey *et al.* [131]. As a proof of concept, the so-called 'broker' comprehension tool was developed. A user evaluation to test the tool's features was however not carried out. Our work differs from Knight's as several existing tools are compared and evaluated as opposed to developing a new tool [120].

In another study, the three-dimensional Source Viewer (sv3D) tool used three-dimensional (3D) visualization metaphors to aid in the software comprehension process, both at structural and evolution levels [84] (see also Figures 2.2 and 2.3). In order to justify the tool's development, the developers analyzed two-dimensional visualization metaphors and outlined shortcomings thereof, arguing that several such shortcomings could be addressed by the use of 3D visualizations. The link between the sv3D tool features and existing comprehension models was however not detailed to a large extent. An evaluation of the sv3D tool was later carried on with the aim of getting feedback from the users as well as determining how helpful the tool would be in software comprehension [82].

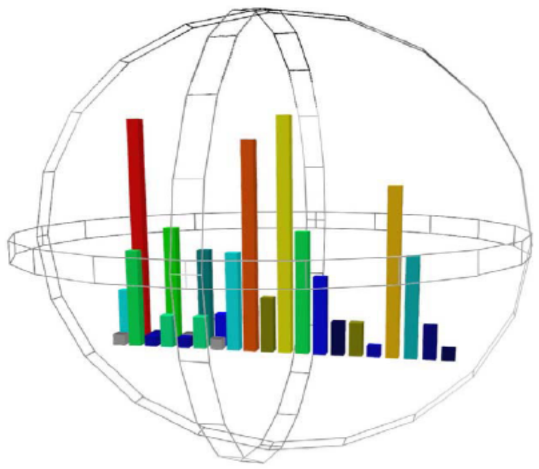


Figure 2.2: Classes as shown by sv3D [82]

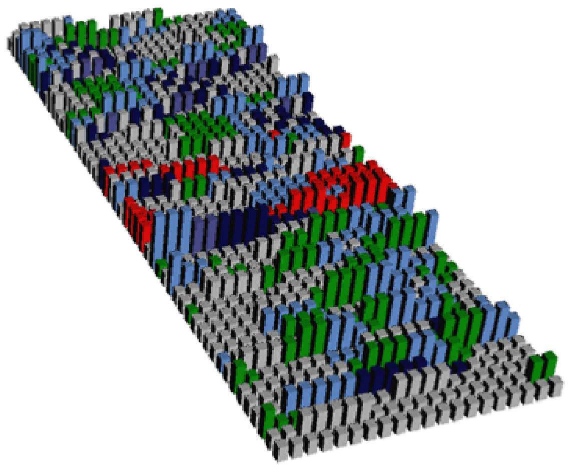


Figure 2.3: A sample image of sv3D's evolution view [80]

Undergraduate and graduate students participated in the study described in [84]. These involved answering questions related to the software code under study. One group of students used the tool as an aid while a second group addressed the same tasks without the tool's use. The authors noted that the evaluation results were *surprising* as the tool users took longer to answer the questions in relation to the group without the tool. Although arguably present, the ability to aid in software comprehension was therefore not immediately evident.

In a recent publication, two types of SoftVis metaphors for visualizing large call graphs of C and C++ code bases were compared [47]. These were classical node-link metaphors and the newer hierarchical edge bundles metaphor. The task addressed by these visualizations is understanding and assessing the modularity of large C and C++ systems. Several tool implementations of the node-link metaphor were compared by users with one implementation of the hierarchical edge bundles. A sample of this comparison is shown in Figure 2.4. A number of desirable SoftVis tool properties emerged from this study, most notably the need of layout stability and predictability and scalability, minimal occlusion, and search facilities. However, this study does not further detail the way in which the studied SoftVis layouts do directly map on finer-grained comprehension tasks beyond a global assessment of a system's modularity.

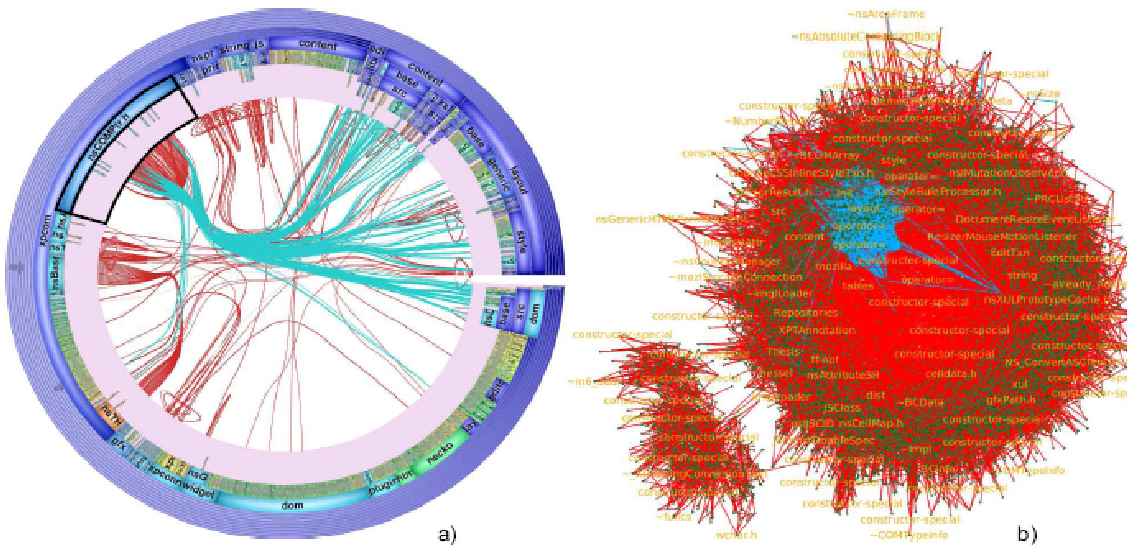


Figure 2.4: Comparison of hierarchical edge bundles (a) and force-directed layouts (b) for visualizing large call graphs [47].

2.3 Application Domains for Software Visualization

In this thesis, we argue that studying SoftVis tools that target the same application (sub)domain is generally more insightful from our perspective of eliciting desirable requirements and how well tools fit these requirements than comparing tools that address different domains. For instance, it makes more sense to compare SoftVis tools (and their features) that address corrective maintenance only, rather than comparing SoftVis tools that address program comprehension in general. Two main reasons are outlined below in support of this claim.

First, *program comprehension* (when used without further qualifications) is a rather vague term, as it does not refer to a set of specific tasks that the programmer wants to solve beyond 'understanding' a program[3]. As such, it is hard to compare SoftVis tools purely from the perspective of how much they support comprehension in general, since they may actually support different finer-level, more specific, comprehension tasks, which are best addressed by different visualization features. It is more relevant to compare tools that support comprehension-related tasks within a subdomain of software engineering, like corrective maintenance for example. This will necessarily limit and focus the types of actual tasks that the users want support for.

Secondly, for a comparative analysis of the effectiveness of visualization *techniques*, used in different SoftVis tools, in supporting concrete tasks, the link between these techniques (or features) and the actual tasks to be supported must be made as clear as possible. For this, we need first a clear delimitation of the types of tasks targeted. Note that, from this perspective, such tasks can be part of the same or different activities in software engineering. For example, the

[3]This problem is not unique to software visualization. Several data and information visualization papers propose techniques that are useful in 'getting insight' in datasets with only a limited quantitative or qualitative qualification of the term 'insight'.

task of correlating the hierarchical structure of a software system with its function call pattern, may be seen as part of corrective maintenance (when tracing cause-effect relations between failures and bugs), part of perfective maintenance (when refactoring a system to improve its modularity), or part of adaptive maintenance (when determining how to change a system to remove or add new components). For this task, different techniques can be used, *e.g.* different graph layouts.

To further proceed with our analysis, we need a classification of SoftVis tools with respect to the application domains within software engineering that they address. Several such classifications exist, as follows.

Price *et al.* proposed a taxonomy for SoftVis tools which compares twelve tools against six categories of desirable features [57]. The categories used were scope, content, form, method, interaction and effectiveness. The level to which each of the tools fulfilled the categories selected was shown. The twelve tools compared were chosen based on the diversity of the approaches that they used as well as their historical relevance in the area of SoftVis [57]. The tools were however not related to a single application area.

Another general taxonomy of SoftVis tools is proposed by Maletic *et al.* [79]. This classification compares five software tools along five axes: task, audience, target, representation, and medium. As in the case of Price *et al.*, the scope of the considered taxonomy and tools is quite broad.

The audience for different SoftVis tools strongly varies depending on the tools' exact purposes [79]. From the audience (users) perspective, too, it can be hypothesized that a comparative evaluation of tools in the same (sub)domain is more useful for users in that area, and also for designers of those specific tools, than a comparison of tools targeting different areas [69]. In this direction, Storey *et al.* performed a survey of tools supporting software development [132]. Twelve tools were compared to show how each of them measure up against the identified categories of intent, information, presentation, interaction and effectiveness. For a different field (corrective maintenance), and from a technique rather than tool perspective, Baecker *et al.* analyzed three classes of visualization techniques that are useful for debugging [11]. These were animation, improved typographic representations and sound in relation to program errors. The evaluation presented in Chapter 6 relates to that of Baecker *et al.* as they both cover visual tooling supporting corrective maintenance. However, Baecker's work addressed only how certain visual *techniques* can help debugging activities, whereas we consider techniques, their 'instantiations' in concrete tools, and desirable features identified by users.

In Chapter 5, we present the evaluation of ten SoftVis tools. The tools address the quite general task of software understanding without focusing specifically on one activity, similar to [57, 79]. In this evaluation, we noticed the earlier mentioned difficulties of comparing desirable features of tools that address different activities. In Chapter 6, we narrow the scope of tool evaluation by looking only at tools that support corrective maintenance. Moreover, we also classify and analyze the set of visualization techniques used in the tools' implementations, and attempt to correlate user acceptance with these techniques in order to detect which techniques seem to yield the most accepted tools.

2.4 Empirical Studies and Validation for Tools

It is important for SoftVis tools that are developed to be *evaluated*. In the past, extended work has been done in both tool evaluations as well suggestions on how to improve the evaluation process itself. While a variety of evaluations have been carried out, some of which are outlined in the previous sections, we note that the evaluation procedure used is not always documented. As such, the lessons learned from previous tool evaluations are not often used by new researchers. The relatively low availability of such information can potentially hinder new evaluations as researchers may deem them too difficult to carry out. In the following, we detail a few general aspects related to the evaluation of SoftVis tools, with a focus on the evaluation process itself.

Among the previous tool evaluations, Pacione evaluated five SoftVis tools in relation to comprehension questions on program dynamics (behavior) understanding [100]. In his study, a lot of emphasis was put on the tools as well as the results of the evaluations, clearly showing how each tools performed in relation to the criteria used. Despite this, there was little information on the evaluation procedure, the challenges faced as well as recommendations on how to carry out a similar evaluation.

Bassil and Keller [121] also carried out an evaluation of seven commercial and academic tools using the taxonomy of Price *et al.* [107]. In this work, the level up to which the evaluated tools measured against the desirable features captured by the taxonomy was discussed. The main evaluators were the actual authors of [121], rather than independent users from the application domain. Also, relatively little information was given on the actual challenges faced during the evaluation process.

Working from another angle, other researchers have made various suggestions related to improving evaluation experiments for visualization tools. Ellis and Dix suggested many ways in which information visualization evaluations can be improved [160]. These included evaluating for the sake of discovery as opposed to proving that a given system is optimal; measuring variables that contribute to the aim of the evaluation; evaluating the whole tool as apposed to only its best features; and balancing between qualitative and quantitative evaluations. Information visualization is tightly related to software visualization [29]. Hence, the recommendations made by Ellis and Dix are largely applicable to SoftVis evaluation studies.

Interestingly, the recommendations made in the above-cited work were made after reviewing 65 papers that described new information visualization techniques and found that less than 20% of the authors had carried out evaluations. A similar trend was observed earlier by Tory and Möller who noted that relatively few papers published over a seven year period in the IEEE Transactions on Visualization and Computer Graphics (TVCG) had a so-called 'human component' (see Figure 2.5). By definition, humans are important both in the design, as well as the evaluation, of visualization tools. One of the reasons for this low percentage could be the lack of guidelines on how such evaluations can be carried out, an aspect which we mentioned earlier.

Recently, Kraemer *et al.* outlined several lessons learned in carrying out various evaluations and empirical studies related to SoftVis tools, specifically algorithm animation and engineering

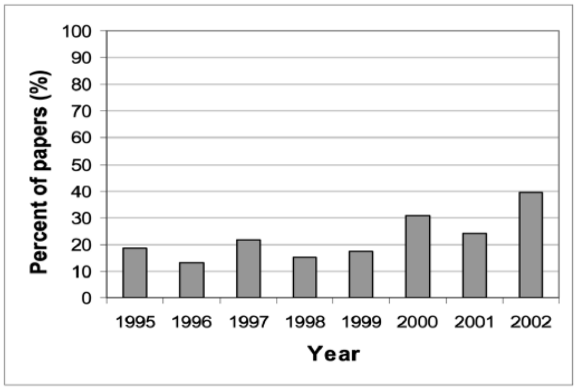


Figure 2.5: Papers from IEEE Transactions on Visualization and Computer Graphics which contain a human component [145]

diagram visualization [70]. Some of these included encouraging the use of interviews and think-aloud studies to obtain feedback on site as opposed to surveys, a technique also used earlier in [152]; and advocating several advantages of observational studies such as think-aloud studies, notably the wide variety of unexpected information that is generated by this method.

In an early study, Shneiderman *et al.* suggested multi-dimensional, in-depth, long-term case studies as the way forward when carrying out information visualization tool evaluations [117]. This procedure involves identifying three to five domain experts that can take part in the evaluation which can run from several weeks to months. After these users are exposed to the tools, feedback is collected and the tools under consideration are to be improved. The work presented in this thesis converges on the above-presented idea of using feedback gathered during tool evaluation to further improve the tools, albeit in the narrower context of SoftVis tools. In terms of study duration, the work covered by this thesis covers both relatively long studies (months) as well as shorter studies (one weeks).

Marcus *et al.* discuss the aspect of user training, or in broader terms, user suitability for performing user studies of SoftVis tools [80]. They observed that students performed poorly when doing SoftVis tool evaluations, due to their low exposure level to such tools, but also due to an implicit low exposure to the actual tasks that those tools were aimed to support. As such, to obtain results with a high relevance from such studies, they recommend selecting users for studies from the same category as the final users whom the tools target. For corrective maintenance, for example, those would be experienced programmers actively involved in maintenance activities in the software industry. This observation is in line with the recommendations of [28].

O'Brien *et al.* took a broader perspective and observed how to empirically study software practitioners, and outlined certain so-called 'negative trends' which should be discouraged during evaluation studies [165]. Some of these included setting up experiments that were not natural in nature (*e.g.* asking users to perform tasks that they were not fundamentally interested in), as well as not combining qualitative as well as quantitative evaluations. The emphasis of O'Brien's observations was program comprehension in general, whereas in this thesis we focus more specifically on the way in which SoftVis tools support program comprehension for maintenance activities.

In a keynote presentation, Basili gives an overview of the past, present, and future trends in empirical software engineering research [12]. Although not specifically aimed at software visualization, most of his arguments are directly applicable to SoftVis. First, hidden context variables in the phenomena under study must be better understood in order to be able to interpret, and generalize, a study's findings. However, this is a challenging task by definition. Secondly, the dimensionality of the space of a study is very large (*e.g.* threats, studied artifacts, study population, study duration, experimental setup). Ideally, all these dimensions should be densely sampled. However, this requires considerable effort and resources, typically not available within a single research group. Correlating the results of several studies by means of so-called meta-analyses or meta-studies is a promising solution, but it requires detailed information on all parameters of such studies, as already mentioned.

In Chapter 8, we present our own 'lessons learned' during the evaluation of a wide range of SoftVis tools, with a focus on the evaluation process itself. The evaluation results themselves are discussed in earlier chapters.

2.5 Comparison of SoftVis Tools

As already mentioned, a specific class of evaluation studies focuses on comparing several concrete SoftVis tools against each other, with the aim of finding out which tool is best for supporting a certain task. Several studies in this class are outlined below.

Pacione compared five dynamic visualization tools and evaluated them with one user in relation to specific comprehension questions [100]. The study found that no one tool was able to answer both the small-scale and large-scale program comprehension questions that were presented. Subsequently, a multiple faceted abstraction model was proposed that combines statically and dynamically generated data for better comprehension. The Vannessa tool was later built based on this model proposed as a proof of concept [101]. Despite this, Pacione noted that the usefulness of the multifaceted three-dimensional model in software comprehension as well as the developed tool was yet to be evaluated.

The work presented in this thesis differs from Pacione [100] as we study only SoftVis tools. This is done with the aim of eliciting user requirements and so-called 'desirable features' of such tools as opposed to comparing the presentation (visualization) techniques employed by the tools, *e.g.* different graph layouts. In comparison with Pacione, the number of users in our evaluations is also different: Up to 16 expert programmers took part in the software comprehension experiments presented in this thesis, while only one user was involved in [100] .

Storey *et al.* performed a survey of tools in a single application area, *i.e*, tools that provide awareness of human activities during software development [132] . Twelve tools were compared clearly showing how each of the tools measure up against the identified areas of intent, information, presentation, interaction and effectiveness. Among the observations made from the comparison was the lack of desirable requirements for the tools as well as the low levels of evaluation that the tools were exposed to. As such, it was difficult to determine the success of a given tool in the absence of this information. While the evaluation presented in [132] was in a different area, the same problems exist for SoftVis tools targeting (corrective) maintenance, which is the main focus of our work.

Bassil and Keller [14] evaluated seven tools, three of which were commercial and four academic, against the taxonomy of Price *et al.* [107]. This was done with the aim of showing how each of the tools compared to the features captured in the taxonomy. As opposed to our work, their evaluation did not involve participants (users) from the industry. As such, it is arguable that features which are (un)desirable from the specific perspective of industrial users may have been insufficiently covered.

In another study, Charland *et al.* carried out a comparison of three software visualization tools with the aim of identifying the value that they offered to programmers during program comprehension activities [19]. Five experienced programmers participated in the study and examined the three tools which covered a combination of both dynamic and static visualization techniques. All the three tools were standalone ones, that is, not embedded within Integrated Development Environments (IDEs). In the experiment setup, the users first employed IDE's in order to familiarize themselves with the source code under study before they were exposed to the SoftVis tools. The limitation noted with this setup was the difficulty in determining the exact tool that contributed to adding value, *i.e.* IDE or SoftVis tool. In Chapters 6 and 7, we evaluate both SoftVis tools that are integrated with IDEs and tools that work standalone, and explicitly ask users to assess the desirability of tool-IDE integration.

A recurring problem in tool comparison studies is the appropriate selection of users, or evaluation subjects. Several constraints have been outlined in this respect. Tools that are compared from the perspective of measuring their effectiveness in supporting a given task should be used, during comparison, by similar (or ideally, identical) categories of users, *e.g.* professional developers with similar experience or students, but not mixed groups thereof [35, 18]. User availability is another factor of concern, especially for longitudinal studies [117]. Although students are easier available for longer studies, they may need longer training phases, and the results of such studies cannot be easily extrapolated to *e.g.* industrial users [82]. The fact that tools that work well with academic users are not automatically effective in the industry was also noted by Cordy [22]. Finally, involving the actual tool developers in studying their own tool has advantages (reduced learning) but also clear limitations (bias).

In Chapters 4-7, we detail on these points in the specific context of SoftVis tools aimed to be used in the industry. Our general approach is to consider tools developed by third parties (that is, not the study subjects or the researchers involved in this thesis), and focus on eliciting desirable features from the viewpoint of the target group (industrial users). Our aim is not develop new SoftVis tools, but understand which are the features of existing tools which best serve our target user group, and why, with the aim of helping the tool adoption process and also potentially helping tool developers to better match their target group.

2.6 User Requirements for SoftVis Tools

One of the core areas that we are interested in is understanding the way user requirements map to features provided by SoftVis tools. The core question here is: What is the best way to determine which are the essential requirements that a given user category has on a given type of SoftVis tool? Previously, various forms of specifications for tool development were proposed. These specifications covered a broad area of what could be needed in tools both in the development process, the functionality required, as well as the necessities after the development phase. We outline below related work in these directions.

Coulmann's work was among the first to suggest requirements for SoftVis tools [24]. Among the suggestions made were the need for the tools to display just the needed information while hiding that which may not be required for each user session. This is perfectly in line with the well-known information seeking mantra of Ben Shneiderman "overview first, zoom, then details on demand" [118]. Among the other requirements suggested were the ability to show multiple views as well as the need for users to manipulate what has been visualized. In similarity to our work, Coulmann also noted that evaluation of tool functionality was critical especially for adoption purposes [24]. It is from this background that we advocate tool evaluation as a (first) basis for eliciting user requirements.

In another study, Zayour noted that, when developing a reverse engineering tool, focus should be put on designing it in a way that increases its chances for adoption [165]. It was proposed that a study aimed at discovery needs to be carried out in realistic settings, *i.e.* settings which parallel the actual usage settings as closely as possible. This can be complemented by a study of program understanding by programmers so as to know which particular tasks are essential for tool support. After this is done, a tool can then be developed based on earlier requirements, after which the tool is evaluated to ensure that it actually overcomes the problems. While these recomendations were made for reverse engineering tools, many are still applicable for SoftVis tools. As such, several of the suggestions made by Zaynour were extensively followed in our work.

Kienle dedicated a whole chapter of his thesis on requirements for reverse engineering, with the majority being tool requirements [63]. His work includes a comprehensive literature review focused on non-functional requirements or quality attributes. Some functional requirements were mentioned as well. The identified quality attributes were scalability, interoperability, customizability, usability, and adoptability.

Reiss also mentioned questions that should be asked when approaching software visualization research or the development of a new tool [112]. The questions can be summarized as follows

- Is the approach used realistic and scalable?
- Is the approach easy to use?
- Does the system abstract to the needed information?
- Lastly, does the new approach offer any advantage to the programmer that can be easily

seen.

In our work, these questions were considered to be user requirements. From a tool developer perspective, these requirements can be used as checkpoints during the tool development process.

In a related study, O'Shea studied the types of information that are most important to the experienced programmer during maintenance, with the aim of finding out the requirements that would lead to supportive software visualization tools in the area of source code comprehension [99]. O'Shea's results showed that high importance was attached to meta data and informal code comments, descriptions of the code itself, *e.g.* class and package level documentation, as well as information related to the control structure of the code. Other types of information, like functionality-related information, were mentioned. Interestingly, their level of importance was deemed to be less than 5% [99].

Using a different approach, Tjortjis *et al.* surveyed software maintainers in order to determine their needs and in turn provide tools to support them [143]. Among the needs noted were the ability to view both low-level and well as high-level abstractions automatically in order to ease the comprehension process . It was also noted that alternative means other than consulting the code experts were needed in order to faster comprehend the code. Visualization was one of the means proposed. Considering that many of the changes made in the source are documented in the comments, it was also deemed important that these comments be easily seen and accessed (*e.g.* searched for).

Overall, the above previous work addresses a complementary approach to the tool evaluation studies described earlier. While tool evaluations measure the effectiveness of an *already* implemented tool, in whose construction a number of requirements have been already considered, direct consultation of users in order to elicit requirements is an earlier, more general, step of gathering insight into desirable features. In our work, we follow a mixed approach: On the one hand, we expose users to existing tools. On the other hand, we ask users to comment on the desirable features of these tools both from a general perspective ('would these features be useful to you in general', Chapters 4 and 5) as well as from a task-oriented perspective ('which features of this specific tool help you in solving this specific task', Chapters 5, 6 and 7). The elicited requirements by us, as well as other requirements pointed out by different researchers, are structured in a Unified Requirements Classification (URC) (Chapter 5). This URC can be used as a reference point for future SoftVis tool developers, but also for practitioners interested in quickly pre-selecting existing SoftVis tools for further in-depth evaluation.

2.7 Summary

Summarizing our review of previous work, several points become apparent.

First, there are many axes, or dimensions, along which evaluations can take place, *e.g.,* evaluating tools from the perspective of end-users in a focused domain; comparing tools addressing different domains; comparing tools against a set of technical visualization requirements deemed to be generally important; comparing tools against a set of desirable features deemed to be important; and executing empirical studies to actually measure the effectiveness of tools for a given task.

However, it is quite hard to draw *upfront* conclusions from existing studies as to whether a given SoftVis tool is going to be appropriate for a given software engineering task. Most studies present a posteriori gathered information from actual tool usage. It would be highly useful to have a way in which we could assess whether an existing SoftVis tool is good for a given task or usage context, a priori, that is, before actually using the tool. This would help developers in choosing from the multitude of existing tools. Also, this would help tool designers in fine-tuning their designs to better serve a given user group. Achieving this without actually using the tool is probably impossible in general, as there are many hidden context variables that no single requirements classification model can capture [12]. However, a unified requirements classification can serve as a pre-selection instrument for quickly narrowing down the tool search scope.

The problem of visualization tools evaluation becomes increasingly important and is acknowledged as such by many prominent researchers. Without good evaluations, specific tools, but also the general claims of (software) visualization being useful, may be endangered [112, 69, 77].

In the remainder of this thesis, we aim to fill this gap, *i.e*, establish a way to extract desirable features of software visualization tools for a given usage domain from both the tools themselves and feedback of the users, build a classification model for these features, and show how this model can be used to predict the suitability of SoftVis tools in new usage contexts.

Chapter 3

Methodology

As outlined in Chapter 2, there are many types of studies involving SoftVis tools, related to various dimensions such as the techniques used, the tasks addressed, the requirements fulfilled, and concrete types of evaluation. There is no simple common denominator along which all these studies can be related. As a consequence, it is currently hard to establish how a user study for the effectiveness of a new software visualization tool should be set up. Since we are going to perform such studies next, as described in the following chapters, we need to make choices in the ways, or methodology, used when setting up a study.

This chapter overviews the different methodologies that contribute to the construction and execution of a user study, with a focus on studies on software visualization tools. In the following chapters, we shall combine different ingredients of the different methodologies presented here to set up our own studies. Hence, the purpose of this chapter is to overview the different applicable methodologies and also outline their advantages and limitations in our context. Moreover, this chapter defines the methodology-related terminology which will be used in the following chapters.

3.1 Empirical studies

Empirical studies aim to compare what is believed to be true about a product or process (the *hypothesis*) with what that product or process actually is (the *measurement*). As such, empirical studies can validate or invalidate the whole or parts of a hypothesis, and also help in creating new hypotheses.

According to Perry *et al.*, empirical studies usually involve four main steps *i.e.*, generating the hypothesis, testing it, analyzing the generated data and getting a conclusion in relation to the hypothesis earlier generated [102]. Empirical studies can either be qualitative or quantitative depending on the quantitative or qualitative nature of the hypothesis, and whether the measurement done involves extracting quantitative or qualitative data. Below is a brief description of these methods. For a more detailed analysis, we refer to Yin [163, 162].

3.1.1 Qualitative research

This type of empirical research aims at understanding the reason for a given type of behavior. Qualitative studies, as opposed to quantitative ones, do not aim to produce actual measurements on a metric scale of their results, but limit themselves to comparative or qualifying statements, *e.g.* ‘technique A is better than B’ or ’technique A is suitable for task B’. However, this does not mean that such studies have a lower value than quantitative ones. It is not always possible or easy to set up a metric scale along which to compare some attributes. The difficulty of establishing a metric scale is related to two different dimensions:

- it is generally hard to devise a metric scale to measure qualitative properties such as ’insight’. It can be argued that the difficulty relates to the fact that such properties like insight are acquired during a complex process that is also highly person-specific;
- even when the metric is clear, *e.g.* comparing whether accomplishing a task using tool A is faster than tool B, the actual task may involve attributes which are hard to quantify, for example measuring how ’modular’ or ’well organized’ a certain software system is.

Related to the above, several points can be made about the particular measurement of effectiveness of SoftVis tools in supporting program comprehension tasks. One difficulty here is to measure how much understanding has a user obtained on a software system. This reflects the difficulty of quantifying insight. A second difficulty relates to measuring how much of that understanding has been obtained because of the use of a given SoftVis tool, and how much is independent on that.

In cases when quantitative studies are hard to perform, qualitative answers can be more appropriate and correct. Qualitative research involves observation of the relevant study group in order to establish both their views as well as actions taken as a basis for generating research findings. Two types of qualitative studies are case studies as well as action research.

Case Studies

Case studies are a qualitative research approach that attempts to understand the workings, shortcomings, as well as advantages of a given setting [119]. This is done using actual users carrying out tasks in a realistic environment [119]. Case studies are among the first types of instruments used in empirical software engineering research [12].

Case studies are explorative in nature. As such, they present the opportunity to discover new results as the study progresses. Among the other advantages of case studies are the effectiveness that they offer when choosing between two competing technologies [65]. Due to their small scale nature, they can also be used to detect early failures or shortcomings in a technology before it is deployed on a large scale. For researchers, case studies also provide a good chance for developing tools and techniques that are relevant to the studied audience [38].

The deep level of understanding of the subjects required for this approach however leads to a high investment cost which may hinder the level to which they are used. Another challenge faced by case studies is the small sample that they typically target, which raises questions on whether the results can be generalized. Cases where the person carrying out the study has an interest in the results can introduce bias in the way that the results are interpreted. Even in cases where no bias exists, the non-specific nature of the results in case studies leads to many different ways of interpretation of the findings.

In some sense, case studies convey ‘insight’ about the working of a process or product which is conceptually very similar to the function of providing ‘insight’ of a visualization tool or method. That is, just as one of the aims of visualization is not to measure things but to discover the unknown, case studies also try to discover unknown facts about a given process or product. As opposed to this, quantitative studies try to measure concrete metrics of a process or product, just as another role of visualization in general is to offer a way to quantitatively compare data values.

Action Research

Action research is a qualitative method that aims at balancing both action and reflection using a participatory as well as democratic process [110]. Action research is centered around the practitioner who gets a chance to question their environment and find ways to improve it with the researcher playing an advisory role [86]. A sample cycle of action research is shown in Figure 3.1.

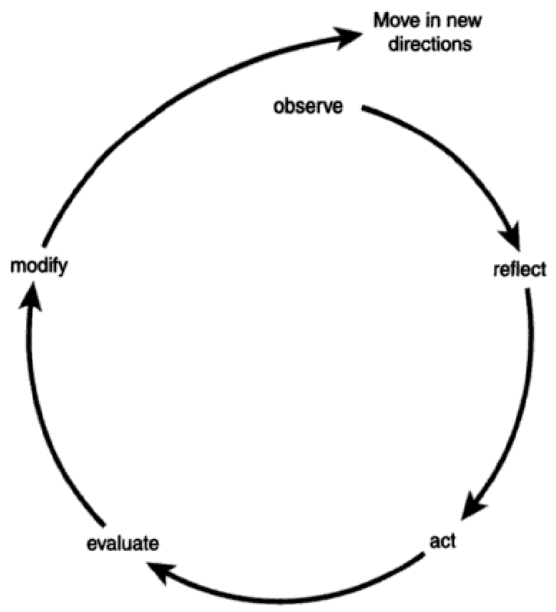


Figure 3.1: Action reflection cycle. Adapted from [86]

Action research is good for cases where the problem may not be immediately obvious. As such, a hypothesis does not have to exist before one starts the research. The action is taken, after which a hypothesis is formed which can be further tested using *e.g.* quantitative methods.

Again, this is similar to the role of 'discovering the unknown' that a data visualization tool or method has.

Action research may, however, not be useful in cases where a comparison is being made between two competing technologies. Other cases where it may not be relevant is when a precise statistical measurement is needed, *e.g.* the time users take to finish a given task or the percentage of programmers that are unable to use a given technique. In such cases, quantitative studies are a better instrument.

3.1.2 Quantitative research

This type of empirical research relies on statistical methods in order to analyze the numerical results of studies [123]. Quantitative research can also be applied when the measured data is non numerical, as long as such data can be described along a metric scale or one has, at least, a similarity (distance) function defined on such data. Surveys, experiments and historical data are all types of qualitative studies. For statistical methods to be applicable, sufficient measurements (samples) need to be gathered. As such, quantitative studies are powerful instruments that can determine if a given process or phenomenon is 'statistically significant', *e.g.,* detect correlations between different variables. However, they pose more constraints on the type, quantity, and quality of the gathered data. For example, in order to draw statistically relevant conclusions from a set of measurements, a minimal number of samples, measured within a maximal error, must exhibit a certain correlation. For such reasons, statistical studies are relatively less frequent in practice in the software visualization world.

While quantitative research is not primarily aimed at developing hypotheses, it can be used to test an already established hypothesis. This method also has other advantages. For example, the chance of diverging from an established experiment path is less high than with qualitative studies, given that quantitative experiments require fixing the studied input and output variables [85]. Interpretation of the quantitative numerical results is also less exposed to subjectivity of the researcher. Drawing higher-level qualitative conclusions from such results is, of course, still potentially affected by wrong generalizations.

The closed nature of quantitative research method, however, presents certain shortcomings. Among these is the inability to get broad insights on the participants due to the specific nature of the requested answers. This limits chances of identifying other areas of research as well as getting a deeper understanding of the issue being studied [85].

Surveys

Surveys are a quantitative research method that is used to generate data from a group of respondents who are considered to be representative of a given population. Several data collection methods exist, such as paper questionnaires, Internet surveys, or telephonic interviews. Surveys are used in many disciplines, but are arguably less appropriate for assessing software visualization tools. Reasons hereof include the difficulty of identifying upfront a significant group of developers using comparable SoftVis tools in the same area.

Experiments

Experiments are a quantitative research method that is used to explain causal relationships between variables by applying different measures and determining the effect of the change [89]. In software visualization, experiments start by identifying a number of variables of interest, such as various parameters of the visualization methods used *e.g.,* number of colors, icon sizes, type of layouts, and so on. Next, these variables are changed, and a given task is performed using the resulting visualization. Relevant output variables are measured, such as time or accuracy of completion. Following this, a quantitative relation is inferred that tries to link the input and output variables, *e.g.* using statistical correlation.

3.2 Data Collection Methods

When any of the research methods mentioned above is being used, there are usually several instruments involved in the collection of the data. These include questionnaires, interviews as well as observations.

3.2.1 Questionnaires

This method consists of a group of questions which are prepared for the participant to answer. The questions are usually tailored to a particular audience with the inquiry being made for a specific subject.

Questionnaires are cheap to use yet they can also cover a wide geographical area. They provide convenience on the part of the participant as they do not require the researcher to be present when they are being filled. When designed properly, the identity of the participant can be protected which in turn may lead to more genuine responses.

The design of the questionnaire is however vital to the success of the study [163, 162]. Participants may be discouraged by requests for personal details as well as too many closed or open questions. Also, the design of quantitative or qualitative scales has to be carefully done to remove potential bias, and to sample uniformly the answer value range in relationship to the statistical distribution of the answers. Typical quantitative scales used in questionnaires include the 5-point and 7-point Likert scales on which subjects state their level of (dis)agreement with a given statement [87]. Qualitative answers can be gathered by means of free-form answers. In comparison with other data collection methods, such as interviews, questionnaires have however the weakness of a relatively low response rate, as participants may not feel obliged to respond.

3.2.2 Interviews

Using this method, a set of pre-defined questions are presented the participants to answer one at a time. This can be done via face-to-face discussions, or interviews where the researcher meets with the participant, or via the telephone, where the participant is called and an interview date and time agreed upon. Interviews have a high response rate and also enable the researcher to get a deeper understanding of the issues. Effective combinations of interviews and questionnaires are possible to gather both qualitative and quantitative data.

Interviews can however be expensive to use both in terms of money as well as time. Geographically, this method may not be able to cover a very wide area. Also, the attitude of the interviewer during the interaction with the subject has the potential of biasing the answers.

3.2.3 Observations

Using this technique, the behavior of participants during an experiment are observed as a basis for feeding the research. The observations can either be disguised or undisguised. This method is able to capture behavior that may not otherwise be possible using other methods. One important challenge is that participants may change their behavior after knowing that they are being observed. Observations are often complemented by video recording of the activities of the users, followed by a post-mortem analysis of the video. This technique is frequently used in studies interested in user interfaces, *e.g.,* Kagdi *et al.* [58].

A variant of this technique asks the users to think aloud expressing the questions they have, what they see, and the actions they take. This is a very valuable and cost-effective way for eliciting otherwise unseen cause-effect relationships [152].

3.3 Selected Methodology and Justification

In order to achieve the objectives of this work, the triangulation research method was used. Triangulation refers to the combination of various research methodologies, *e.g.* quantitative and qualitative studies, with the aim of addressing a given research problem [55].

When triangulation is used, a more solid argument with broad sources of evidence is presented making it easier to tackle a given research problem [157, 89].

The first step of our work encompassed the exploration of the study area, in order to form a hypothesis. A hypothesis is a conditional statement that proposes an explanation to some phenomenon or event [2]. In our case, the hypothesis involves the set of features that industrial users perceive as desirable in a SoftVis tool. This part is mainly described in Chapter 4. For this stage, case studies were the main method used. Due to their explorative nature, they were helpful in creating a basis from which a hypothesis could be formed. In our case, this regards the initial gathering of desirable features of SoftVis tools.

After the hypothesis was generated, there was need to test and refine it. This was done in Chapters 5 to 7 using a combination of qualitative and quantitative methods. This included case studies as well as experiments which were combined for better analysis. Testing the hypothesis involved examining additional SoftVis tools with additional subjects in order to see whether the initial set of desirable features is indeed broadly applicable. Refining the hypothesis involved refining several of the initial desirable features, as well as zooming in on the more specific area of SoftVis tools for corrective maintenance.

Among the data collection methods used in this work were questionnaires which aided in the selection and filtering of the participants for the studies. This instrument was also used after each study to get the participants views of the studies carried out. Questionnaires were supplemented by interviews depending on the particular task being covered.

Chapter 4

Software Visualization Needs For General Software Comprehension

This thesis focuses on evaluating SoftVis tools in maintenance activities. Software comprehension is however needed before, as well as during the execution of software maintenance. As such, this chapter analyzes the user needs for SoftVis tools that support the general task of software comprehension. This is done by observing five expert programmers using three visualization tools. Their output is discussed in the attempt of formulating a first-level hypothesis on which are general desirable features for SoftVis tools, with a focus on industrial users.

4.1 Motivation

Traditionally, before developing a software product, specifications and requirements are sought from the future users of the application. Many times though, software developers do not involve the final users as much as they should during the product development process [145]. This situation occurs also in the development of software visualization tools. This aspect of visualization tool design has the risk of underutilizing existing perceptual and cognitive theories on usability, with the ensuing risk of having tools that are not optimally suited to the users for whom they are intended [116, 124, 145]. As such, the effectiveness of many of the tools and techniques developed for visualizing software is yet to be proven [83].

In this chapter, we do a first step towards eliciting general desirable features of SoftVis tools by exposing several professional developers to several SoftVis tools and gathering their feedback on the types of features that they see in these tools which they identify as desirable. This work serves as an exploratory study in getting a first impression on which are the types of desirable features that users focus on when being exposed to a new SoftVis tool. In Chapter 5, we refine this insight within a larger study involving more SoftVis tools. Both the work here and in Chapter 5 focus on the so-called *early adoption phase*, *i.e.* the phase in which users attempt to quickly decide whether a given SoftVis tool has the potential to be useful, and thus should be further studied, or not. In Chapter 6, we refine the extraction of desirable features by focusing on a more specific domain (and tools supporting it): Corrective maintenance.

4.2 Tools

To start with our discovery of desirable features of visualization tools from their users, we first need to select both the tools and the users. This section outlines the procedure used for tool selection. The user selection is discussed further in Section 4.3.

4.2.1 Tools Selection

There are hundreds of SoftVis tools in existence which could be used to elicit a set of desirable features from their users' perspective. Clearly, performing such an analysis is prohibitive both in terms of effort, tool availability, and complexity. As such, we chose the following set of criteria to perform a pre-selection of the tools to study

- i The tools had to be able to visualize object oriented software;
- ii The tools had to be freely accessible/available;
- iii The techniques used to represent software by the different tools had to vary;
- iv The tool had to be well-known and used within the software visualization community.

These criteria were considered because of varying reasons, as follows.

In recent years, there has been a substantial move from procedural to Object Oriented (OO) programming in many sectors of the software development industry [25]. It can therefore be hypothesized that SoftVis tools that are able to visualize OO software would be more useful to software developers. Hence, the introduction of the first pre-selection criterion.

Free availability of a tool (either as open source or academic or trial license) can aid in the replication of the study or for users that may be doing research but lack the budget for buying the tools. Even in contexts where the financial constraints would not be a problem, obtaining the needed approvals for actually purchasing the tool can be a lengthy process. Moreover, as we have seen it ourselves in a number of industrial contexts, it is sometimes hard for some corporate users to present a negative evaluation of a tool once the buy decision has been taken (the implication is that a tool is bought because it is a good one). Choosing tools that are freely available was therefore considered important as these tools would be able to cover a wider audience in a shorter time and with less bias. Finally, a freely available tool has a higher chance to get a quick evaluation, potentially followed by adoption, than a tool whose evaluation involves a buy decision.

The third criterion was motivated by the need for the users to get a broad view of the techniques being used in the visualization field. Selecting a set of tools that cover different visualization

Tools	Availability	Visualize Object Oriented Software
ALMOST	Free	Yes
AVIS	Free	Yes
CC-RIDER	Commercial	Yes
Code Crawler	Free	Yes
Creole	Free	Yes
Imagix 4D	Commercial	Yes
Source Navigator	Free	Yes
Understand for C++	Commercial	Yes
Understand for FORTRAN	Commercial	No
Visualize It	Commercial	Yes

Table 4.1: Tools comparison

techniques was chosen so that we obtain an overall insight in a broad spectrum of such techniques, rather than deeper insight in the desirability of one single technique. This is in line with our overall goal of eliciting a set of *general* desirable features for software visualization tools. In Chapter 6, we refine this analysis by zooming in on more specific tools aimed at supporting a subset of program comprehension activities involved in corrective maintenance.

The fourth criterion was used for two purposes. First, a well-known SoftVis tool has higher chances of being a relevant (and useful) one, as it has already been tested, improved and used by many people. Secondly, well-known tools have a higher chance of being further considered by new users. Hence, additional information that we can find to support (or reject) the usage of such tools can be of higher value to the user community at large than if we considered a less-known tool.

Further on in Chapter 6, we shall zoom in on a subset of SoftVis tools (and their desirable features), namely tools that support corrective maintenance. At this level, however, we are interested how industrial users appreciate SoftVis tools in general, hence the broad and task-unspecific tool selection criteria presented above.

4.2.2 Chosen Tools

Ten tools were initially shortlisted and evaluated based on the criteria presented in Section 4.2, as shown by Table 4.1. After a further examination of the tools from the viewpoints captured in the preselection criteria, only three tools were retained. These were Code Crawler [125, 73], Creole [21, 88], and Source Navigator [126]. A brief description of each of these tools is given below.

CodeCrawler: This is a SoftVis tool that combines traditional 2D graph layouts such as trees, treemaps, and nested force-directed layouts with simple metrics that reflect static software properties [26]. It was developed based on the principle of simplicity, scalability and language independence. Examples of the metrics supported by CodeCrawler are the number of instance variables, number of methods, lines of code in a method and the number of children in a class. These metrics are extracted from source code projects using third-party static analyzers and stored in a language-independent database format. The metrics are displayed simultaneoulsy with nodes and arcs in software diagrams. The node colors, positions and sizes can encode each a different metric value, as outlined in Figure 4.1. This enables users to compare a few different metrics among themselves and also correlate their values with the system structure.

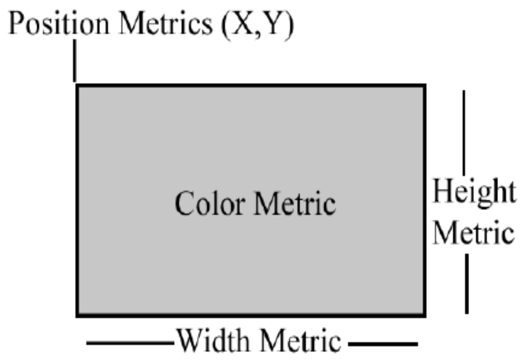


Figure 4.1: Metrics-and-structure visualization in CodeCrawler. Adapted from [26]

CodeCrawler is incorporated within Moose, which is an extensible language-independent environment for re-engineering object oriented systems [125, 32]. Figure 4.2 shows a sample system hierarchy diagram displayed by CodeCrawler.

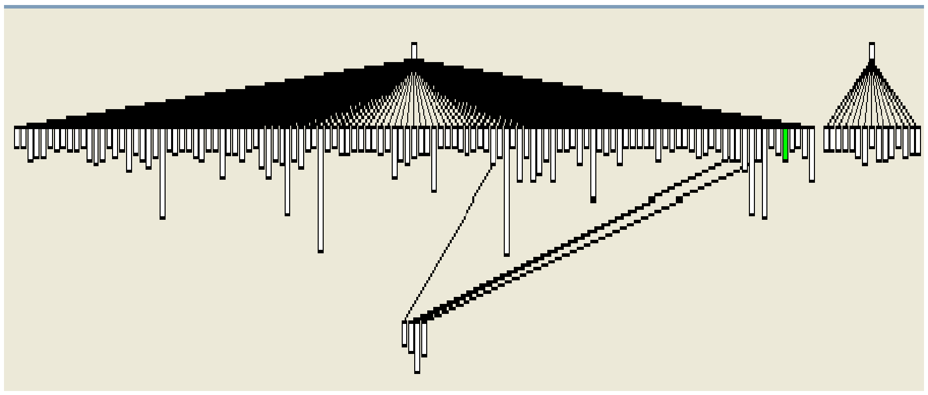


Figure 4.2: Code Crawler's hierarchy diagram on which complexity can be mapped as a metric

Creole: Creole is an Eclipse plug-in that integrates the SHriMP visualization metaphor [128] with the Java Development Tools (JDT) platform in the Eclipse environment [33]. The main purpose of Creole is to provide both high-level program views as well as visualization of different dependencies within the source code using multi-views. Various navigation and visualization techniques are used, *e.g.* the fish-eye technique, pan and zoom, as well as nested graphs. Creole makes comprehensive use of color and supports system browsing from the package level down to the source code. Examples of inheritance hierarchies classes visualized by Creole are shown in Figure 4.4 and Figure 4.5. The SHriMP views used in Creole were generated based on the cognitive framework that considers how maintainers understand software [128]. Part of the framework is shown in Figure 4.3.

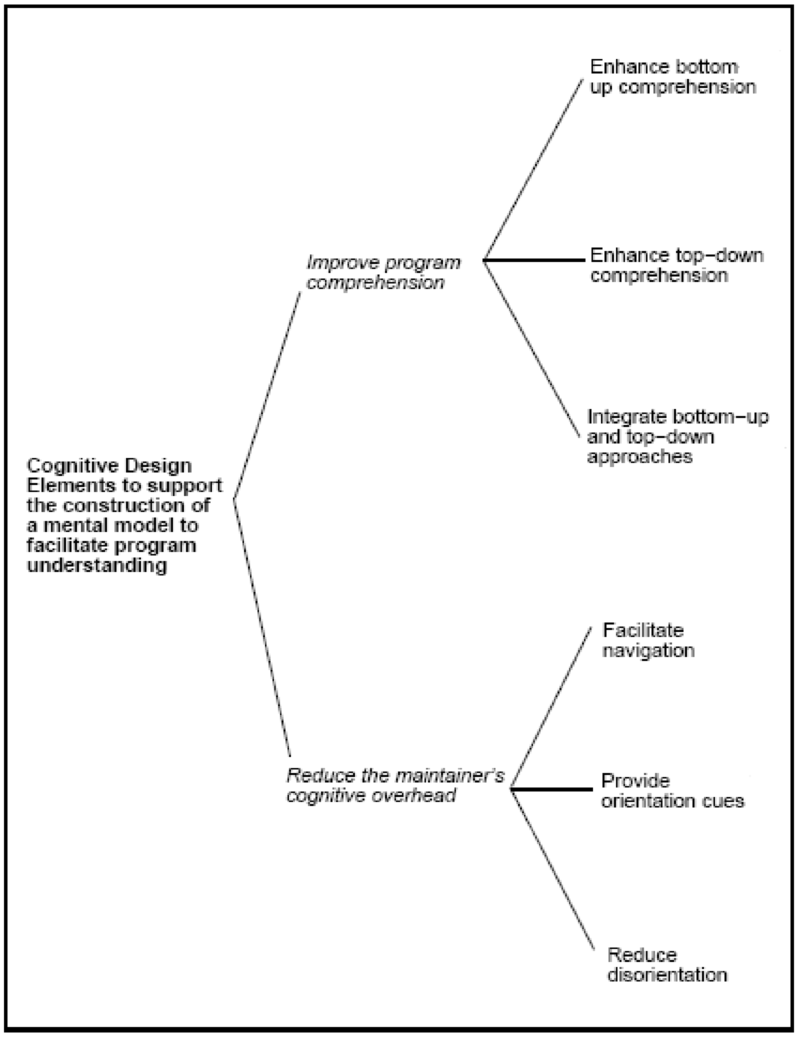

Cognitive Design Elements to support the construction of a mental model to facilitate program understanding
Improve program comprehension
Enhance bottom up comprehension
Enhance top–down comprehension
Integrate bottom–up and top–down approaches
Reduce the maintainer's cognitive overhead
Facilitate navigation
Provide orientation cues
Reduce disorientation

Figure 4.3: Classification of cognitive design elements (adapted from [131])

In order to use Creole, an import of a Java project needs to be made using the Eclipse IDE's menus. Once the project is in the IDE space, a drag and drop of elements from the project that are to be examined can be made to the Creole view.

Source Navigator: This is a source code analysis tool which enables code editing, display of classes and their relationships, as well as call trees. It uses a small set of quite simple visualization techniques, such as syntax coloring and tree views [126]. Overall, the visualization gives a strong 'text-oriented' display impression. Source Navigator loads information extracted from source code into a project database. This database stores facts about file names, program symbol elements *e.g.* functions and variables, and symbol relationships. Source Navigator then provides different browsers (graphical views) into the project database. An example hereof that displays function references is shown in Figure 4.6. The tool supports C, C++, Fortran,

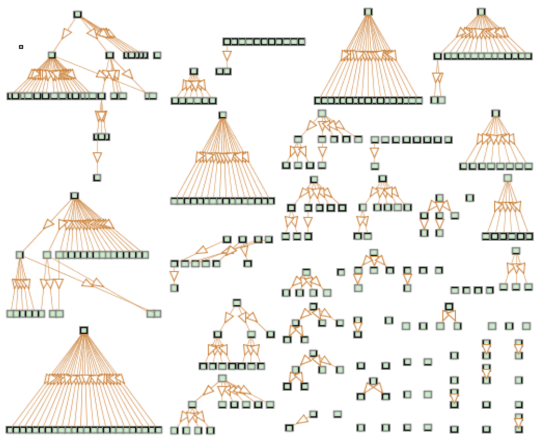


Figure 4.4: Creole visualization of inheritance trees

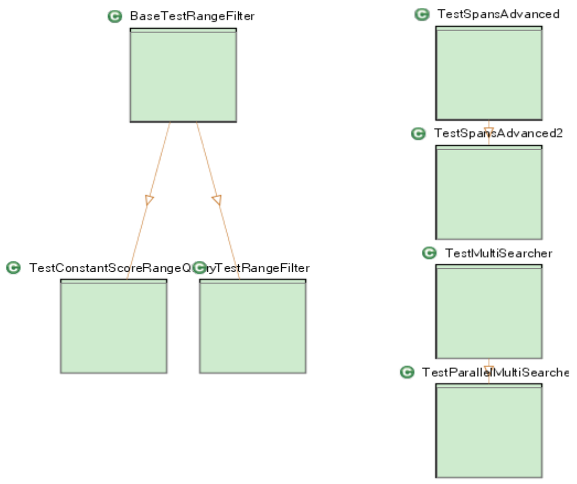


Figure 4.5: Zoom-in view of the Creole inheritance visualization

COBOL and Java and uses the `grep` tool to enable different searches within the source code at a lexical level.

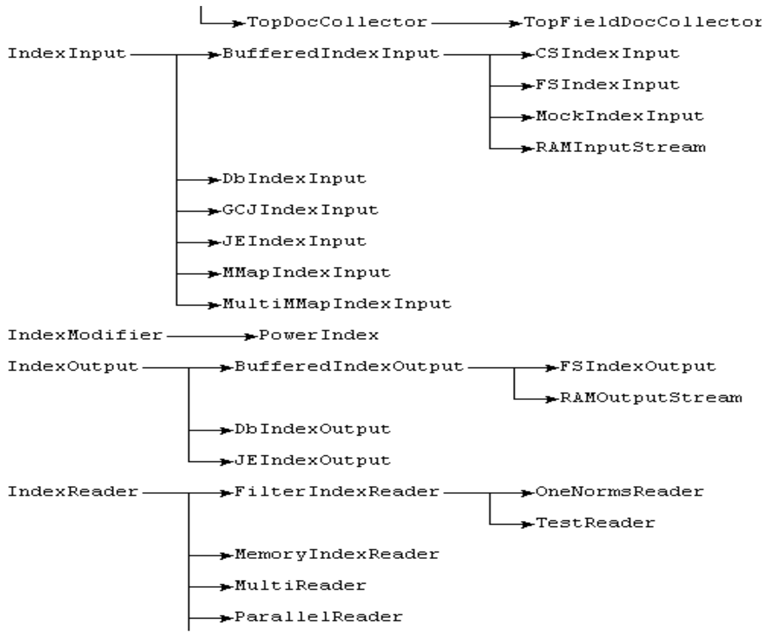


Figure 4.6: Source Navigator visualization of function references

4.3 Participants

In order to test the abilities of the selected tools, as well as gather feedback with respect to which visualization features are perceived as desirable, users needed to be selected to perform this test. The details of the participants as well as the selection process are explained below.

4.3.1 Selection Process

The study was aimed at getting participants who are expert programmers and are also employed in the software development and maintenance industry. As already mentioned, our aim is to understand how typical professional developers in the software industry reflect upon the usefulness of SoftVis tools. As such, all selected users were professional programmers with wide programming skills. The idea behind having experienced users was that they would generally find it easier to learn a new tool and their criticism would be based on an informed background rather than on their limitations. Furthermore, the collective opinions and advice on the various tools gathered from such users would provide relevant guidelines to future tool developers if their target audience is similar to such users.

In order to ensure that this type of user was selected, pre-study questionnaires were given to all the participants that were interested to engage in the study. The questionnaires sought knowledge about the participants gender, the number of years that they had used the computer (*i.e* general computer knowledge) as well as the number of years that they had spent programming. Other details asked were the object oriented languages that the participants had working knowledge of as well as the frequency that they used each of these languages. The participants were also asked if they had used a software visualization tool before.

4.3.2 Selected participants

The five expert programmers who best scored on the above-mentioned selection guidelines were further selected to participate in the study. They were all male, working in the software industry in Kampala, Uganda, and had over ten years experience both in programming and computer usage. They were experienced with the object oriented paradigm with knowledge of at least two object oriented languages. These aspects are summarized in Table 4.2.

4.4 Source Code

The third step after selecting the tools and participants is to select a code base to be examined (visualized). The selection of the source code used in this study was based on the following properties:

- *Scale*: We are interested in visualizing large-scale code bases having hundreds of components and thousands of methods. This criterion aims to replicate actual use-cases in the industry. As such, it aims to elicit desirable features of SoftVis tools which make them effectively and efficiently usable in understanding large code bases.

User	Computer usage	Languages known	Programming experience
1	> 10 years	Java C++ SmallTalk	> 10 years
2	> 10 years	Java C++ Python	> 10 years
3	> 10 years	Java C++ Ruby	> 10 years
4	> 10 years	Java C++ Ruby	> 10 years
5	> 10 years	Java C++	> 10 years

Table 4.2: Selected study participants

- *Language*: As our targeted visualization tools support object-oriented code, we selected object-oriented code bases for the study. Furthermore, we restricted the selected code bases to programming languages which were mastered well by the targeted participants (see Table 4.2).
- *Availability*: Just as for the tools, we selected freely available (open source) code. This favors potential interested researchers in replicating our study. Further on, none of the participants had prior knowledge on these code bases.
- *Ease of importing*: A fourth and final criterion for the code selection was that the code would be straightforward to load and examine in the targeted visualization tools. This was done so that the participants' efforts would be focused on the actual visualization actions and not the data management actions.

4.4.1 Selected Source Code

Three different source code bases were selected based on the criteria mentioned above. These were Lucene, Apache BeeHive and Apache Tomcat. All code bases are written in Java, which is one of the languages in which our subjects were proficient, and also is well supported by the studied SoftVis tools.

Lucene: Lucene is an information retrieval Application Programming Interface (API), originally implemented in Java by Doug Cutting [8]. It is free, open source and released under the Apache Software License [39] . The Lucene source code is arranged in 35 main packages and has 772 classes.

Software visualization tool	**Source code used**
Code Crawler	Lucene
Creole	Apache BeeHive
Source Navigator	Apache Tomcat

Table 4.3: Tools and source code

Other details of this code base, as measured by the MOOSE engine [32], are as follows: 4710 methods in the entire project, 6.1 average number of methods per class, and 1.43 average class hierarchy depth level. Lucene has been ported to other languages like C++ and C#. However, the implementation used for this study was the native Java one.

Apache BeeHive: Apache BeeHive is a Java Application Framework designed to make the development of Java Enterprise Edition (EE) based applications quicker and easier [6]. BeeHive is contained in 17 general packages with an average of 7 sub-packages within each main package.

Apache Tomcat: Apache Tomcat is a web servlet container that implements the servlet and the Java Server Pages (JSP) specifications from Sun Microsystems [7]. It provides an environment for Java code to run in cooperation with a web server.

4.4.2 Tools and Code Allocation

Each tool was evaluated using separate source code bases, as follows. Code Crawler was evaluated using the Lucene source code, Creole was tested using Apache BeeHive code, and Source Navigator was tried using Apache Tomcat code (see Table 4.3). The code-tool allocation was not done based on any specific considerations. We could have tried several other combinations, in particular using more than one SoftVis tool to examine the same code base. We limited the possible combinations in order to restrict the experiment duration to a level which was acceptable for our industrial developer subjects.

All participants used all the three SoftVis tools to study the respective code base associated with each tool. A different subject system was assigned for each tool in order to ensure that there was no familiarity with the code for each session. Using the same code could bias the participants as knowledge from the previous session could be carried forward.

4.5 Experiment

The five participants were all exposed to the three tools, one tool at a time. Each participant was given a five minutes introduction to a tool. After this, they had five extra minutes for familiarizing with the tools themselves or seek any extra information. Tool usage instructions and sample tasks were also given to the candidates during the familiarization phase before the actual experiment (study of the large code bases) was started. These familiarization tasks were similar to the ones in the experiment but the code used was different and much smaller. This stage helped in clearing any questions that the users had about the tools and also ensuring that working knowledge of the tools had been established before the actual experiment.

After the familiarization stage, all users noted that they understand the way the tools operated. At this point, two tasks for each tool were given to the participants, one task at a time. The tasks given out were as follows:

- i Describe the static core structure of the system, *i.e.* the main classes and their relationships;
- ii What would be the effect of deleting a particular class or method from the source code?

These tasks were replicated for all the three tools. However, the second task was modified according to the source code being analyzed, *i.e.* the specific class or method to be deleted was chosen differently for each code base. Specifically, the second task was outlined as follows:

- i What would be the effect of deleting the "hook class"? (CodeCrawler)
- ii What would be the effect of deleting "org.apache.beehive.netui.tags"? (Creole)
- iii Name the effect of deleting the "DbStoreTest" class (Source Navigator)

Overall, the combination of the two tasks given to each user was aimed at testing the tools' capacity to provide answers concerning both the larger-scale structure (task 1) and small-scale details (task 2) of the source code. The second task was able to test further the abilities to examine relationships and cross-references between given parts of the source code. Both types of activities are very common in a wide range of software maintenance tasks [100]. The answers to these tasks and further user comments were recorded on a pre-formatted, provided answer sheet that covered all the three tools.

After working with a particular tool, there was a two-minute break before proceeding to the next tool. This duration was kept short so as to ensure that the participants attention was not diverted from the experiments. Table 4.4 shows a summary of the tasks and the time that was planned for each.

After the actual task execution, the post study questionnaire was given to the participants to be filled in. This questionnaire was aimed at establishing the shortcomings of the tools that were used during the experiment as well as their positive points. Questions also sought to find out what extra features the users needed but felt were lacking in the tools. A combination of the answers given in this questionnaire as well as observations during the experiment led to the results of the study.

4.5.1 Analysis of Results

The main aim of the experiment was to expose several expert programmers to SoftVis tools so as to get their opinions on their desires and dislikes from the tools during typical software

Task	**Number of times the task was executed**	**Time per task (minutes)**	**Total time (minutes)**
Pre-Study questionnaire	1	5	5
Introduction to tool	3	5	15
Familiarize with tool	3	5	15
Two assignments	3	7	42
Post-study questionnaire	3	5	15
			92 minutes

Table 4.4: Planned tasks and their execution time

understanding activities performed in maintenance. The tasks chosen in this experiment aimed to replicate such typical activities, albeit on an unknown code base. Tool knowledge was acquired in a short pre-study phase in which sample tasks were accomplished on a small code base.

We did not measure, however, the task completion rate. The reason for this is related to the purpose of our evaluation. The aim here was to extract general desirable *features* that the users would require, as based on typical work with a SoftVis tool for a typical software maintenance task. Since we were interested in general features, we did not need to measure actual completion rates. Such rates would be needed when *e.g.* comparing the actual effectiveness of several tools against each other, or when being interested in finding out which specific tool features are effective in assisting a specific maintenance task (see Chapter 2). This type of detailed analysis is described separately in Chapter 7.

4.6 Results

During the course of the experiment, the participants made numerous comments about the tools in a 'think-aloud' fashion and also asked for extra functionality which was not always supported by the tools. The results from the questionnaires as well as the observations are combined to provide the experiment's results shown in Table 4.5.

Overall, the results of the study showed that the programmers had numerous desires with respect to features to be provided by the visualization tools, but such features were missing. In this respect, the main missing desirable features mentioned are outlined below.

4.6.1 IDE Integration

There was a strong desire to have the visualization tools integrated with an IDE for the target programming language. The reasoning was that when one visualizes data, it is usually for a purpose. If the desire is to add more code to existing software, or to modify existing code for a purpose (*e.g.* debugging) then it would be too much effort to switch between the visualization tool and the environment that is being used to program. Hence, the impression of the subjects was that even if a SoftVis tool is able to generate amazing visualizations, the effort and time spent switching between the two environments may have an effect on the knowledge preservation for programming and thus on the overall effectiveness. Creole and Source Navigator, which were both IDE integrated, were noted to be easier to use. On the other hand, users had challenges with Code Crawler due to its lack of IDE Integration.

4.6.2 Fast responses

The low speed of constructing the visualization was also highly complained about. Having programmed for a while, code was not as difficult for the experts to comprehend as it is for the novices. This means that the switch to using additional tools, like the SoftVis tools we studied here, should not decrease the overall working speed that these programmers are accustomed to. More than two users stopped the generation of call graphs because they felt it was taking too long and a third user said that they would have achieved better results even with a plain IDE as opposed to waiting for a visualization solution that takes too long.

The fast response requirement is especially important for corrective maintenance activities, where a fast analyze-code-test cycle is usually present. This requirement may be of a lesser importance for perfective maintenance and more specifically for higher-level tasks such as architectural recovery, modularity assessment, or quality assessment, which have a less dynamic pace and involve slower iterations from the supporting tools (whether visualization or code analyzers) to the actual code modification and back. Also, we argue that there may exist a correlation between the expected visualization speed and the speed of the native IDE in which one works. For example, Java programmers are used to extremely fast compilation cycles, due to the nature of the language and compiler. In contrast, we believe that C++ programmers may be more tolerant towards slower tools, as typical C++ compilers (*e.g.* `gcc` and Visual C++) take significantly longer to compile large code bases than similar code bases written in Java. However, we have not found literature studies that further shed light on this hypothesis.

Overall, the users' feeling was that the added value of an otherwise correct and complete visualization is highly reduced by having to wait too long for this visualization to be generated. At the opposite end of the spectrum, and in line with the above observation, a fast, responsive, integrated tool like Source Navigator was highly appreciated.

	Best features observed in tools			Complaints on tools		
	Creole	Code Crawler	Source Navigator	Creole	Code Crawler	Source Navigator
User 1	- Eclipse integration - Zoom	- Ease to see classes	- Good speed - Ease of use	- Slow response time	- Difficulty in use - Absence of IDE	-Better layout needed
User 2	- Good layout - IDE integration	- Scalable display	- Ability to see source code - Ease of use - Good speed - IDE integration	- Slow response time	- Difficulty in use - Inability to see source code	- Better layout needed
User 3	- Search function	- Scalable display	- Good speed - Ease of use - IDE integration	- Slow response time	- Difficulty in exporting code - Absence of IDE - Better search	- None
User 4	- Zoom - Search function	- Scalable display	- Good speed - Ease of use	- Speed	- Better search - Inability to see source code	- None
User 5	- Metaphors used - Zoom - Search function	- User-definable customizations	- IDE Integration - Access to code - Ease of use	- Reduce crowding	- Difficulty in use	- Better initial display

Table 4.5: Tool evaluation results

4.6.3 Minimal effort

The users stated their high preference for having visualizations that can be generated with a minimal amount of effort, *e.g.,* parameter settings, data conversions, and other manipulations. Having to export and import code in another format was found to be cumbersome for the participants. As much as possible direct code manipulation was preferred. In cases where that was not possible, close proximity to code was encouraged, *i.e.* have a visualization which can be easily and quickly cross-correlated with the source code by means of clickable linked views.

4.6.4 Good search abilities

The ability to search the visualization was another desired component that the users felt could have been addressed better. A good visualization (from the perspective of layout, clutter, and scalability) that can not be queried was not found to be useful. Searching was desired both at the displayed diagram level and at the source code level. Linked views were again mentioned as important in the search context, *i.e.* being able to search in one view and have other views focus and highlight the search hits. Creole is one of the tools that covered this element to a greater extent.

4.6.5 Clear display

Simple, clear, easy to read displays were prefered by the participants. If, for example, a class was shown, the participants desired its name (fully qualified or not) to be shown as well in close proximity to the class. The named class hierarchy of Code Crawler was especially mentioned as having been very clear.

4.7 Study Limitations

There are limitations to the approach used in this study that may hinder the level of generality of the results. The experiments carried out were in a controlled environment, a method that has been criticized in the past [119, 105]. We specifically examined only the area of static visualization, or more exactly put, visualizations that address questions related to the static code structure. However, keeping in mind our general aim of understanding the desirable features of SoftVis tools for corrective maintenance, visualizations that cover a combination of static and dynamic data should have been looked at. However, this was hard to address, as we also wanted to have tools which are simple to use and, if possible, well integrated within an IDE.

Another limitation was the relative small size of the user group and set of tested tools. Although both sizes are in line with the typical user group and toolset sizes considered by many user studies on SoftVis tools in the literature [14, 19, 100, 132], this is nevertheless a point of concern for our study in particular and for SoftVis and empirical software studies in general [12]. In particular, we note that some of the larger user studies involving SoftVis tools involve mainly academic participants [18, 71]. As already explained, our intention was to evaluate tools with their direct target audience, software development professionals. Organizing larger scale studies with such professionals is a much harder endeavor given their limited time and availability.

4.8 Conclusions

From this first study, the following general desirable features of SoftVis tools emerged: IDE integration, fast visualization generation, search facilities, and the possibility to set up and use the tool with minimal effort.

It can be deduced that better visualization tools and techniques can be achieved if more views of desired features within SoftVis tools are sought before hand, that is, before the tools are actually developed. This method would be most effective if the target group of the tool being developed is the one that is used in the evaluations. In order to develop tools that can be accepted in the software development and maintenance industry therefore, effort should be put on finding out the needs of the industry users first and then later developing tools to meet those needs.

Two main conclusions can be drawn as to the desirable features elicited in this chapter. First, these are *general* desirable features, meaning that a more specific study, *e.g.,* focusing on a subclass of software maintenance tasks, a subcategory of tools, or a more specific user group, would need to take into account how these general features are to be refined for that specific context. Secondly, by being general, the insight derived from the presence of these desirable features is also not sufficient for further tool *selection* (by users) or tool improvement (by tool designers). Although being a starting point for pre-selection that is able to filter out some tools, more fine-grained requirements of the users are not captured.

In the following chapters, we aim to address the above points. Chapter 5 refines these general desirable features in a more detailed classification, still aimed at understanding the early adoption process of a tool. Chapter 6 continues this refinement further, by focusing specifically on SoftVis tools for corrective maintenance and at the phase following early adoption. As such, chapters 5 and 6 can be seen as a top-down refinement of the general insight in desirable features presented here in Chapter 4. Finally, Chapter 7 presents an experiment in which the refined requirements are validated by actually testing whether tools that satisfy them are indeed appreciated by users.

Chapter 5

Unified Requirements Classification

In the previous chapter, a variety of preliminary top-level desirable features of SoftVis tools were identified after exposing the users to various such tools and a set of sample program understanding tasks. In this chapter, these results are combined with previous work in the same area in order to generate a Unified Requirements Classification (URC). Ten tools that use different techniques are then compared against the URC in order to see the level to which they meet those requirements. The selection process for the tools as well as the methodology used to generate the URC are further detailed. Comparisons and observations are made from the study data, and insight is derived which can be used to improve tool development. The aim of the URC introduced here is to capture the types of desirable features that users test for during the first steps of evaluating a new tool, or the so-called early adoption phase.

The rest of the chapter is organized as follows. Section 5.2 introduces the Unified Requirements Classification (URC). Section 5.3 gives an introduction to the tools selected for comparison. In Section 5.5, the selected tools are compared against the categorization and the observations from the comparison are drawn. Lastly, Section 5.6 concludes and provides areas for future research.

5.1 Proposal

As already discussed in Chapter 2, several dimensions exist along which one can evaluate the effectiveness of SoftVis tools. These dimensions have emerged during research in the SoftVis area from various perspectives: evaluating a given SoftVis tools against other tools; aiming to classify requirements of SoftVis tools for a given set of tasks; or the more general attempt of producing a set of universal (or unified) requirements that a large class of SoftVis tools would need to comply with to be usable.

As already mentioned in Chapter 1, tool adoption is a multiple-phase process. We distinguish an early adoption phase, in which users want to quickly test a tool's suitability, and thus do not have time to zoom in on fine-grained details; and a second phase, in which tools that pass the first phase are further examined with specific tasks in mind. In the first phase, desirable features are still tested at a general level. In the second phase, desirable features are linked to a specific set of actions to be supported.

However, such a unified classification of SoftVis tool requirements[1] for early adoption is still missing. By unified, we mean here a classification of requirements that

- captures a large part of the features seen as desirable by typical users of such tools in an industrial usage context;
- is instrumental in assessing the effectiveness of a new SoftVis tool that has not yet been used in a given context.

By early adoption, we mean the phase in which general desirable features are quickly tested for a rapid rejection or further adoption examination. As a simple example hereof, consider the elements one examines when finding a new tool on the internet: If the tool is well documented, available for download and quick install, and easy to start and try out, it has a higher chance to be examined in further detail.

The proposed classification should also be extendable with new desirable features, and features existing in the classification should be refinable into sub-features once more insight is available.

The aim of this chapter is to propose a classification of desirable features of SoftVis tools along the guidelines mentioned above. The qualification of 'unified' to our classification should be seen along these guidelines too, *i.e.*, applicable to a large class of SoftVis tools, tasks, and users, and extendable and refinable. However, we do not claim the classification to be unified in the sense of complete. We are definitely aware of its limitation and needs of refinement with more tools, users, and tasks. Yet, such refinement should be doable within the framework proposed here.

5.2 Generating the Unified Requirements Categorization

In order to generate the URC, insight from tool users is considered to be essential. The context of the URC is identical to the one outlined in Chapter 4: The users taken into account are not novices but experienced developers that have considerable knowledge in programming and computer usage, and work in an industrial context. As such, the level to which our results are applicable to novice users and/or academic users has therefore not been further elaborated in this thesis. This point is in correlation with the differences between early adoption constraints of industrial and academic users: An industrial user is arguably under higher time pressure than an academic user, thus (s)he is interested to quickly decide whether a new tool has potential or not.

[1]Here and in the following, we use the terms 'requirements' and 'desirable features' interchangeably. A strongly desirable feature can be seen as a user requirement.

There are many types of requirements identified by other researchers that provide useful insight into the subject of what are the desirable features of software visualization tools. These may include specifications that make the development process for the tools easier, for example, development of tools in such a way that their code is re-usable [142, 64]. While these requirements are usually needed by the tool developers, they are not always needed by the tool users. Here, we make an explicit distinction between the two roles, and argue that there is a large segment of industrial developers who do not have the time (or freedom) to take up the tool developer role. As such, only user requirements have been considered for our URC. The categories of the URC presented below thus narrows down to those requirements that are presented by the tool users that directly impact of the way that they use the tool, and more specifically on the requirements that can be evaluated during a quick, early adoption phase.

The URC are structured along the following main axes, or dimensions, each of them indicating a separate class of user requirements for software visualization tools:

- Search enabled
- Meta-data display
- Simplicity/lightweight
- Added advantage
- Integrated
- Realistic specifications

A detailed explanation of each of the sections of the URC is given below:

5.2.1 Search Enabled

This requirement refers to the ability of a tool to support query and analysis facilities both within the source code as well as the visualization displayed.

This requirement was among those generated in our initial experiment presented in Chapter 4 which involved five users from industry using three SoftVis tools to perform typical comprehension tasks on three large code bases. Among the other studies that advocated for the same are Bassil and Keller's survey which had over 100 participants, in which 74% of the respondents considered search facilities for both textual and graphical displays to be important [13].

Independently, Sim carried out two user studies and concluded that a search facility was necessary in the SoftVis tool they studied (the Portable Book Shelf) so as to ease the user's movement through the code [120]. Petre *et al.* also made an analysis of what experts demand from visualization tools in which search and navigation tools featured highly [103]. In addition, in their comprehensive review on software visualization tools, Lemieux and Salais included search and query facilities among the essential requirements of SoftVis tools [76].

5.2.2 Meta-data Display

Meta-data display describes the ability of a SoftVis tool to show textual data inside the visualization, like code fragments, documentation snippets, URLs, or other meta data related to the graphical entities being visualized. In our initial study shown in Chapter 4, the participants were displeased with tools that did not allow the viewing of the source code in a linked way with the visualizations. The inability to view code prevents access to the code comments which would greatly increase program understanding. O'Shea's observations from open source Java mailing lists also indicated that meta-data was considered important by expert programmers [99]. A tool that supports this feature would therefore be more appealing to a this group of users. Moreover, being able to quickly correlate visualizations with code is important during an early adoption phase, when users are not yet familiar with the visual semantics of the new tool.

5.2.3 Simplicity/Lightweight

When a tool is simple or lightweight, it can be easily installed and used. Users from our study complained about tools whose usage procedure was complicated with preference being for tools that are easy to install and use [117]. Clearly, a tool that is simple to install and start has higher chances for early adoption than a tool that requires a long time investment to create the first images.

The ease of using and installing a tool is relative, yet there are established procedures within the computing industry that installations and user interfaces usually follow *i.e.*, Graphical User Interface based installers or portable makefiles. When a tool deviates too much from these procedures, it is considered difficult to use.

Other researchers who advocated for simplicity include Reiss who emphasized that software developers can only use a SoftVis tool if it does not require too much work before output is got from it [112]. Schafer further noted that a tool can only be accepted if its ease of use as well usefulness are obvious [116]. Müller *et al.* additionally noted that tools that require too much time to learn and use would be difficult to be adopted in industry thus the need for simplicity [92]. Lanza first introduced CodeCrawler as a 'lightweight software visualization tool', arguing for the importance of simplicity of use [72]. Simplicity and ease of use are especially important in a cost-aware environment such as the software industry where users can be reluctant to invest significant amounts of time in installing, configuring, and fine-tuning visualization tools.

5.2.4 Added Advantage

This requirement addresses the need for tool designers to observe the existing tools before investing efforts in new tools. Evaluating existing tools and techniques helps in showing the added advantage of a new idea before too much money and time is invested in a new design. This is one of the reasons that we opted for a review and experimental analysis of existing tools as opposed to inferring desirable features from designing yet another software visualization tool and studying its usage pattern in a given context. Moreover, if the added advantage of a tool is quickly evident, then its chances for early adoption increase.

In this context, Reiss also noted the failure by many researchers to show the superiority of their novel techniques over the existing simpler representations [112]. Muller *et al.* and Coulmann recommended the need for tool developers to explicitly show the added benefits of their tools. Among the methods proposed in the aid of this activity include expert reviews, user studies, field observations, case studies, and surveys. When a tools benefit is easily seen, software developers are more willing to take it up [92, 24]. It is therefore important that new tools show their improvements in comparison with a workflow in which no such visualization tools are present, like using a traditional IDE or command-line development tool only.

5.2.5 Integrated

This requirement refers to the tools ability to be coupled together with the development environment as well as with other tools that support the task being worked on. Users from our tool exposure experiment presented in Chapter 4 showed preference for tools that were integrated with an IDE as the stand alone tools have a switching effect on the understanding of code. Integrated tools also have a higher chance for early adoption, especially for users who routinely work with IDEs and do not want to change this way of working for a quick evaluation.

Many other studies have also identified integration as being important. While selecting a tool to be used in an industrial setting, Tilley *et al.* noted that companies prefered tools that their developers were more familiar with. A tool that is integrated with the available development environment would therefore be best as it would provide a smaller learning challenge to the programmers [141]. After a review of over 140 papers and nearly as many tools, Lemieux and Salois identified essential requirements of SoftVis techniques in which integration featured highly [41]. Müller *et al.* observed that SoftVis tools can increase their adoption within the industry only if they integrate with the development environments as well as the platforms that the developers use [92]. Similar observations are made by Koschke [69], Storey [130], and in a larger software development context, by Wuyts and Ducasse [161].

5.2.6 Realistic specifications

This requirement refers to having a SoftVis tool that can be practically used in *real-life* scenarios. The intention is to quantify whether a tool is able to address problems of real interest to the users in a given environment, including platform, type of software under scrutiny, type of activity to support, programming language, and interchange data formats. As such, a realistic tool is a tool that can operate under a given real-life environment as outlined above. In particular, a realistic SoftVis tool should be able to run on the typical computer configuration used by mainstream software developers in the industry, *e.g.* a PC having a mainstream operating system and an average graphics card and memory. Having the tool readily run on a wide range of platforms increases the chances of early adoption, as the tool can be tested right away, without further modifications to the host environment.

The word 'realism' is sometimes used with a different connotation with respect to SoftVis tools, *i.e.* from the perspective of generating images which look natural. In our work, we associate realism with the targeted working environment of the tool and not the type of generated imagery.

Gilmore mentioned the need to design tools that support activities that people really engage in, *i.e.*, realistic tools, as this would increase the tool's chances of being adopted [43]. When discussing the potential reasons for the failure of software visualizations, Reiss also advised developers of new systems to ask themselves whether the new systems they are developing addresses real, relevant and important problems for the programmer [112]. Tilley *et al.* in addition noted cases where relatively advanced tools were not selected because the operating system that they run on was different from that of the company selecting the tool [141]. Wang *et al.* recognize the strong need for SoftVis tools to operate in complex real-world environments and approach this goal by advocating an extensible multi-layer and multi-tier software visualization framework [158].

5.2.7 Unified Requirements Categorization

The proposed URC hierarchy is shown in Figure 5.1. As indicated in this figure, the five main URCs are further refined into sub-requirements, along the lines already explained in the previous section. It is already easily visible that our proposed URC hierarchy differs notably from other classifications of requirements for SoftVis tools, *e.g.* that of Bassil and Keller [13] (which focuses on functional aspects) or the taxonomy of Price *et al.* [107] (which focuses on a mix of functional and non-functional aspects). Other taxonomies are discussed in Chapter 2. The main difference between our URC and related taxonomies is that we our central (and actually only) focus is on what *users* perceive as desirable in a tool in an early study phase, rather than on what functions or mechanisms a tool may offer (which become apparent only after one uses the tool for a while). Although the two perspectives intersect, we argue that a first step in tool adoption (or rejection) hinges on a user *perception* of desirable features identified in an *early* usage phase. After a tool passes this early adoption phase, more fine-grained feature classifications can be used to discriminate between different tools. This is the context that we try to capture with our URCs.

5.3 Visualization tools

In the first part of this chapter, we have presented a proposal for an unified requirements classification (URC) for SoftVis tools. In the remainder, we shall evaluate the URC against a set of concrete SoftVis tools. The aim of this evaluation is twofold. First, it serves to check how the URC model actually performs in being able to describe the capabilities of a set of concrete

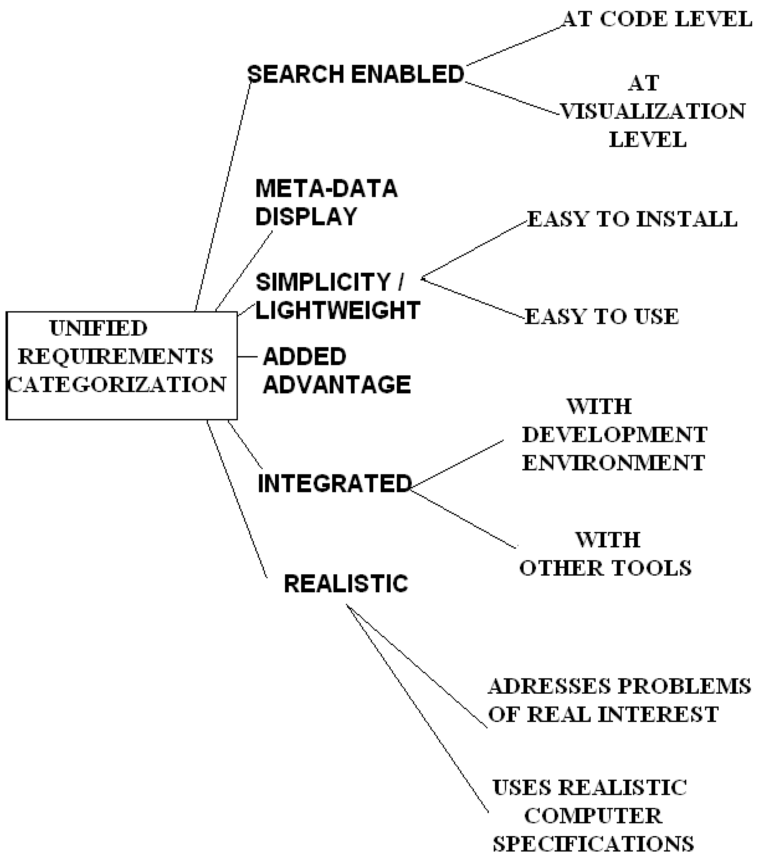


Figure 5.1: Unified Requirements Categorization

tools. Secondly, we aim to exemplify how the URC model can be used in practice in pointing out how they can be used to determine advantages and limitations of concrete visualization tools in an early phase, and thereby serve as to assess the suitability of such tools, or point to directions in which such tools can be further refined.

In order to evaluate the URC against concrete software visualization tools, ten such tools were selected and compared against the URC. The comparison procedure aimed at establishing the level up to which these tools satisfied the URC requirements. This study is similar in structure to the study described in Chapter 4, but is performed using the more refined URC model, and against a larger set of tools.

The tool selection procedure is described next.

5.3.1 Selected tools

To assess the level of satisfaction of the URC requirements, we selected ten SoftVis tools which complied with the selection requirements described in our first study in Chapter 4(Sec. 4.2.1). We did not select these tools based on their perceived compliance with the URC requirements, as the assessment of this degree of compliance was precisely the desired output of the study.

The three tools that had been used for the study in Chapter 4, *i.e.*, Creole, Code Crawler and Source Navigator were among the tools selected, as these tools appeared to be interesting candidates according to the first study. The remainder of seven tools were selected based mainly on their wide audience in the software engineering world. Widely known tools have a higher chance to be candidates of an early adoption test than less known tools, simply because they are easier to be found. The first three tools were explained in detail in Chapter 4 and are thus excluded from the brief tool introduction of the other seven tools which is given below.

ArgoUML

This is a Computer-aided software engineering (CASE) tool that is able to reverse engineer Java code into UML. It supports an input format of both Java and XML and uses the XMI format which enables exchange between other UML tools. It is good for design and modeling especially where previous knowledge of UML exists [115]. Figure 5.2 shows ArgoUMLs display of a class, its child classes as well as the operations (methods) within that class.

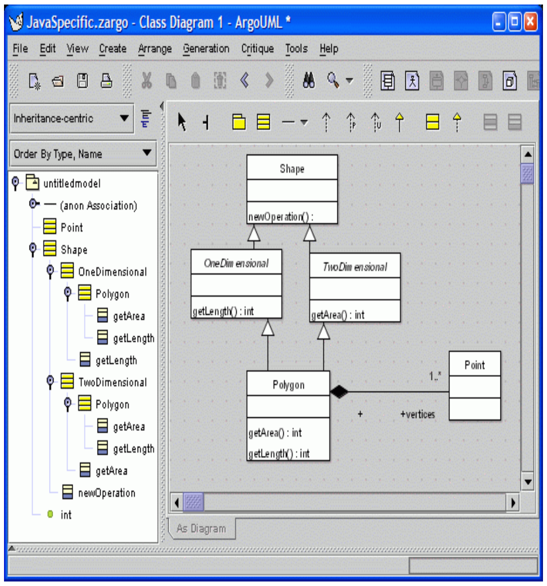


Figure 5.2: ArgoUML display [9]

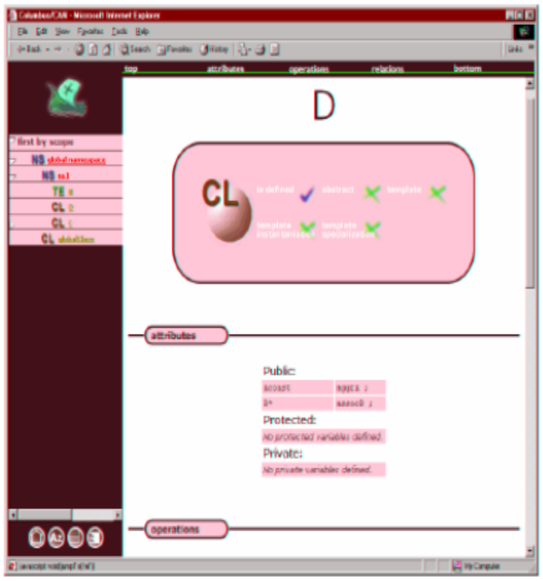


Figure 5.3: Sample output view from Columbus CAN [40]

Bauhaus

Bauhaus is a tool-set that is supported on GNU/Linux, Microsoft Windows and Sun Solaris and is able to work with programs that are written in Ada, C, C++, and Java [109]. Depending on the information that one needs to represent, two main formats are used in Bauhaus. The InterMediate Language (IML) is used to represent the low level data while the Resource flow graphs (RFG) are used to show the general picture of the analyzed system. Some of the capabilities of this tool are dead code detection, protocol analysis as well as feature analysis.

Columbus/CAN

Columbus/CAN is currently a commercial package that can be freely downloaded for academic purposes [40]. It is aimed at heavyweight, detailed static analysis of large-scale C and C++ programs. Columbus is the main framework that enables the analysis filtering and exporting of results to various formats, while CAN is the core tool that parses C++, performs semantic analysis, and creates the central database on which Columbus operates further. Currently, various C and C++ dialects are directly supported by Columbus/CAN. A third tool, the linker, combines the information extracted from independent source files, or translation units, to deliver whole-program results such as call graphs and class inheritance diagrams. Many export formats are supported including HTML, FAMIX, GXL and RSF (Rigi Standard Format). Figure 5.3 shows a sample output from Columbus/CAN exported in HTML format.

Goose

Goose was developed by the Research Center on Information Technologies at the Karlsruhe University in collaboration with the FAMOOS project for reengineering object-oriented systems [31]. The GOOSE tool set is capable of source code analysis, visualization, abstraction and problem detection. It is supported on the SUN Solaris/SPARC operating system and is able to analyze source code written in C, C++ and Java.

Once code is loaded onto the tool, several analyses can be carried out. This stage does not give any output for examination by the user but helps gather the information to be used at the later stages of the tool. The analyzer can also give pointers as to why the chosen analysis task could not be completed. After the analysis stage, several types of visualizations can be carried out. The output for this step is generated in three main formats: simple relational database format, graph-based (GML and VCG) and logic (Prolog facts). Given the large size of the generated datasets, GOOSE offers several abstraction and hierarchical clustering mechanisms. An example snapshot of a GOOSE visualization at two levels of detail is shown in Figures 5.4 and 5.5.

Tarantula

Tarantula is a system developed at the Georgia Tech School of Computing that helps users find faults or problems with their systems using a line-oriented source code visualization [57]. Color is used to differentiate between statements that have passed a given set of tests and those that have failed, *e.g.* using green and red respectively. Yellow is added to represent cases that have both passed and failed parts within the selected texts. The visualization metaphor builds atop of the pixel-oriented mapping technique introduced in Seesoft [34] which reduces each line of source code to a (colored) line of pixels, thereby achieving visual scalability. The system is configured together with a Java environment to acquire the source code and test results. A sample display of Tarantula is shown in Figure 5.6.

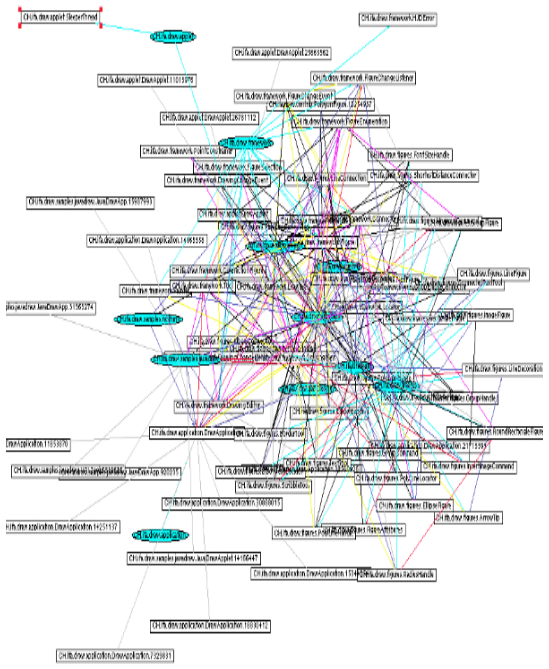


Figure 5.4: GOOSE visualization - fine level of detail [36]

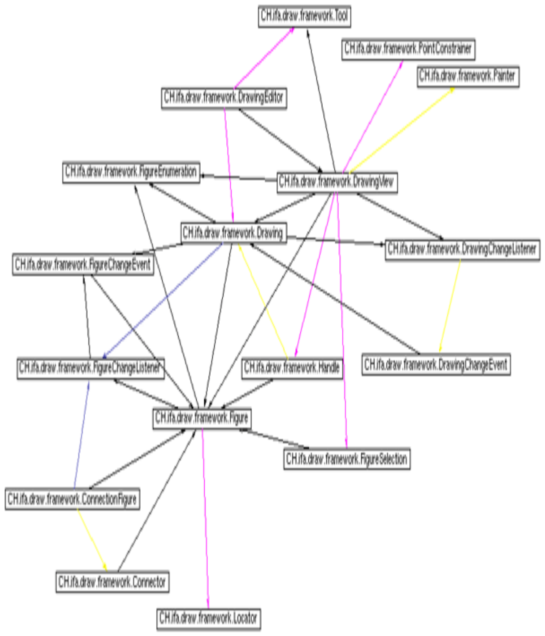

CH.ifa.draw.framework.Tool
CH.ifa.draw.framework.PointConstrainer
CH.ifa.draw.framework.Painter
CH.ifa.draw.framework.DrawingEditor
CH.ifa.draw.framework.FigureEnumeration
CH.ifa.draw.framework.DrawingView
CH.ifa.draw.framework.Drawing
CH.ifa.draw.framework.DrawingChangeListener
CH.ifa.draw.framework.FigureChangeEvent
CH.ifa.draw.framework.FigureChangeListener
CH.ifa.draw.framework.Handle
CH.ifa.draw.framework.DrawingChangeEvent
CH.ifa.draw.framework.Figure
CH.ifa.draw.framework.FigureSelection
CH.ifa.draw.framework.ConnectionFigure
CH.ifa.draw.framework.Connector
CH.ifa.draw.framework.Locator

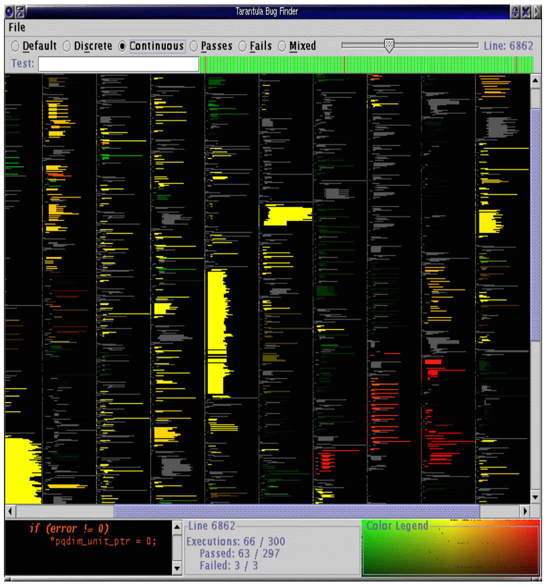
Figure 5.5: GOOSE visualization - coarse level of detail

TkSee

TkSee is a tool that enables browsing of source code while maintaining a history of the tasks carried out [46, 62]. Search is a particular strong point of this tool, in line with its targeted user group. Figure 5.7 shows one of the views of TkSee. This tool is currently supported on the UNIX platform and maintains a close link with the source code. Its look and feel is strongly similar to that of a classical IDE, making heavy use of syntax coloring and highlighting techniques.

VizzAnalyzer

VizzAnalyzer is a prototype tool developed based on a framework at the Software Technology Group at the Växjö University [78]. The idea behind the VizzAnalyzer prototype is to integrate different visualization techniques and enable the viewing of both low-level and high-level representations. The workflow, like for TkSee, is centered around an IDE metaphor, with several visualization plug-ins such as different 2D and 3D graph layouts of static software structure, and analyses such as static software metric computations, filters, and queries. This tool receives source code and binary code using various data importers. The received code is then changed to a relational (graph) format which becomes the input for the so-called High Level Analysis and metrics engine (HLA). Finally, different types of graph visualizations can then be generated from the output of the HLA module.



```
if (error != 0)
  *pqdim_unit_ptr = 0;
```

```
Line 6862
Executions: 66 / 300
  Passed: 63 / 297
  Failed: 3 / 3
```

Figure 5.6: Tarantula visualization [10]

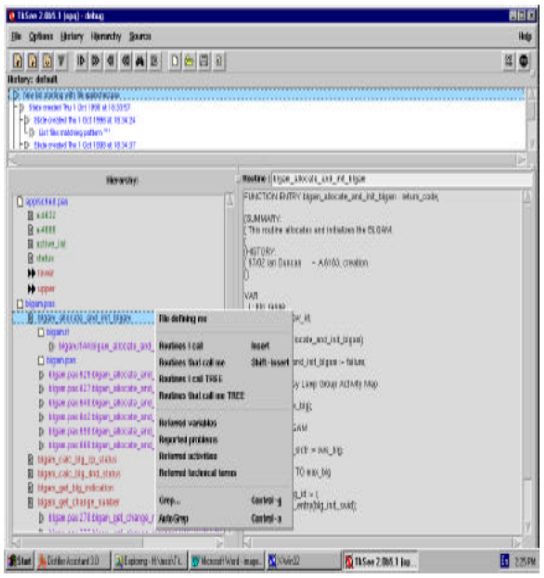


Figure 5.7: TKsee's Graphical User Interface [62]

5.4 Evaluation procedure

Two professional software developers assisted in the evaluation of the ten selected tools. Documentation for each tool was collected along with a functional tool copy. The developers were then allocated the ten tools and asked to grade them along the categories of the URC. The format of the study was similar to the typical evaluation one would do when finding a tool on the internet and deciding to have a first look at it to assess its usability. Independently of the developers, we also carried out a separate evaluation. The results from both the developers and our own results were compared, and a final grading of the analyzed tools was produced.

As already mentioned earlier in this chapter, the focus of the URC model is to capture an 'early impression' of developers interested in using a given SoftVis tool. As such, the exact capability of the studied tool to complete a given task, or the relative speed or accuracy of task completion, is not important in determining the scores within this classification. Hence, the evaluation procedure used generic code bases and generic program comprehension tasks such as searching for a given software component in a system and producing a high-level overview of the static system structure. The main aspect measured in this evaluation was whether the users find the examined tools *usable*. In a positive case, further, more refined, evaluations based on specific tasks and functions can be executed to determine the finer-grained level of tool effectiveness.

In terms of the tasks addressed by each evaluated tool, the measurement related the expectations of the users with respect to the tool's features as advertised by the tool's *own* documentation, and the prior expectations of the users as to what they would perceive as being desirable features of *any* SoftVis tool.

5.5 Tools against Categorization

Table 5.1 shows a comparison of the tools against the categorization (URC). The way in which each examined tool scores with respect to all categories of (sub)requirements captured by the URC model is described in the table rows, one row per tool. Each tool is assessed against each of the components of the URC to see whether that component is fulfilled or not. When a URC requirement has sub-requirements, separate grading is given for each of these sub-requirements. In this way, one can assess whether a given top-level URC requirement is fulfilled fully or only partially.

Tools	URC CATEGORIES									
	Lightweight/Simplicity		Search enabled		Meta-data display	Added advantage	Integrated		Realistic	
	Install	Use	Code	Visualization			IDE	Other tools	Computer specifications	Problem addressed
ArgoUml	√	X	X	X	X	√	X	X	√	√
Bauhaus	√	√	√	X	√	√	X	√	√	√
Code Crawler	√	X	X	√	√	√	X	√	√	√
Creole	√	√	√	X	√	√	√	X	√	√
Columbus/Can	√	√	√	X	√	√	X	X	√	√
Goose	X	√	X	√	√	√	X	X	√	√
Source Navigator	√	√	√	X	√	√	√	X	√	√
Tarantula	X	√	√	X	√	√	X	√	√	√
TkSee	X	X	√	X	√	√	X	√	√	√
VizzAnalyzer	√	√	X	X	√	√	X	√	√	√

√ = tool fulfills requirement up to a sufficient degree from the user's perspective
X = tool does not fulfill requirement sufficiently from the user's perspective

Table 5.1: Results of the URC evaluation

Observations
Using the URC, we can summarize and compare the examined SoftVis tools in a compact manner. Further details of this comparison are presented below, with a focus on tool aspects which were identified as suboptimal and which may be of priority for tool developers in the future.

Simplicity: It was observed that majority of the tools were lacking in at least one of the two simplicity subsections, *i.e.*, installation and use, with only four of the evaluated being found both easy to install as well as use. This presents a challenge for the potential tool adopters as difficulty in any of these areas may prevent them from using otherwise good tools and giving up on further tool examination from an early phase. Tool developers therefore need to ensure that their tools do not divert too much from widely accepted installation procedures if they are to be adopted.

Search Facilities: None of the tools had search enabled at both the code level as well the visualization display stage. Considering that the graph displays shown by the tools can become very complex, search at both these levels is deemed important. As such, tool developers need to ensure that the visualizations displayed by the tools can be searched at both levels, and that the search hits presented in both code and visual displays are correlated.

Meta-Data: Meta data display relates to the proximity of the visual elements to the source code being evaluated, or the ability to recognize code-level constructs in the visualizations by means of annotations. Based on the study's results, the majority of the tools supports meta data display, mainly by small textual annotations and tooltips. Several tools even enable the user change the code and automatically update the visual displays to reflect the changes. This relates well to the users' general requirements of having an easy to follow mapping from the visual metaphors to the code, a trend that should be encouraged by future tool developers. The availability of such annotations was deemed important by the users in understanding the visual displays during the early adoption study, and motivating for continuing them to further study the tools.

Added Advantage: The added advantage was assessed by reading the tools' documentation. In all cases, the documentation contained a fairly explicit description of what the tool should help with. However, from actual tool usage, it was not always observed that the advertised added advantage was in sync with the impressions of the users. This points at the fact that there is potential for credible added advantage in many SoftVis tools, but that not all tools live up to all of the created expectations. Having (at least) a large parts of the advertised added advantage

Integration: Integration was measured at two levels, *i.e.*, integration with an IDE as well as integration with other tools that support program comprehension. A general trend of supporting tool interoperability at various levels was found in all tools that were studied. The main integration mechanism was by means of data exchange in widely accepted data formats. However, the tighter level of integration, namely IDE integration by means of shared data and linked views, was still lacking in many of the studied tools. There is ample evidence that such integration is highly desirable for the targeted user group of professional developers (see Chapter 2). As such, more effort seems to be needed in the future in this direction to increase SoftVis tool acceptance in an industrial context. We noticed that IDE integration strongly helps in an early adoption phase: An integrated visualization tool is simply easier to quickly test, as users can always fall back on the known, trusted, functions of their IDE than in case they have to understand a fully new tool.

Realistic: Many of the tools presented in this chapter address a realistic problem experienced by programmers. Moreover, the computer specifications on which the tools are to be run were also found to be realistic, *i.e.* in line with the average specifications of typical computers used in the software engineering industry. As such, there were no tools in the studied set that would require specific software or hardware configurations to be used.

5.6 Conclusion

Comparing SoftVis tools from the perspective of desirable features is a difficult task. In this chapter, we have proposed a set of Unified Requirements Classification (URC) that can serve to structure the comparison of such tools (and the outcomes of such a comparison) with a focus on the early adoption phase. The URC consists of six main categories further refined into eight subcategories, and can be further refined to accommodate a more detailed type of tool comparison, *e.g.,* for a specific set of tasks, user group, or tool type.

To demonstrate the way in which the URC can be actually applied to summarize the comparison of SoftVis tools, we have evaluated ten such tools against the URC requirements and discussed the results of the evaluation comparatively. The evaluation process followed the typical steps users take when quickly examining a new tool during the so-called early adoption phase. As a side effect of this study, we have observed that many features described in the URC are still only partially implemented in the studied tools. However, the users who participated in the study mentioned that these features are indeed highly relevant from their perspective of quickly accepting (or rejecting) a given tool. We believe that this holds true also for other SoftVis tools at large, beyond the studied sample of tools, and identifies directions in which tool design can be improved.

The URC presented here can be used by future developers of SoftVis tools, especially those targeting experienced programmers. The comparison of existing SoftVis tools against the URC can be used by typical industrial users during the early acceptance phase of a SoftVis tool, specifically by actively looking for the degree of fulfillment of these requirements. Since these requirements are relatively easy to test for, they can act as an easy filter that can be applied on a large set of tools to obtain a smaller subset that has higher chances of being usable in an industrial context. On the other hand, developers that aim to penetrate the industrial development community with their new tools can steer efforts in the directions outlined by the URC model to maximize their chances for early tool adoption.

In this study, the focus has been on general requirements of SoftVis tools during their early adoption phase. However, for a better, more detailed, assessment of desirable features of such tools, we need to have a narrower field of analysis. In the next chapter, we analyze the desirable features of SoftVis tools aimed at helping a more specific field of activity: Corrective maintenance.

Chapter 6

Software Visualization for Corrective Maintenance

In this chapter, we focus on the support that SoftVis tools offer for a specific area of software engineering: Corrective maintenance. To support evaluation and comparison of such tools, we refine the URC classification of desirable features introduced in Chapter 5 to focus on corrective maintenance. Next, we illustrate the way that such a classification can be used by evaluating 15 SoftVis tools that support corrective maintenance. The evaluation procedure used a set of sample corrective maintenance tasks to be executed by a group of 15 developers on a given code base. Finally, we compare and discuss the evaluation results using the proposed classification model.

6.1 Introduction

Corrective maintenance (CM) refers to the branch of software maintenance concerned with correcting faults that occur in software [135]. Corrective maintenance, as a part of software maintenance, can also benefit from SoftVis tools. The use of software visualization (SoftVis) tools for software maintenance has been advocated for in a variety of studies [11]

In the previous chapter, we evaluated a number of desirable features that users require from SoftVis tools in general, by collecting feedback from software engineers who used such tools for the general task of program understanding. The focus was on the so-called early adoption phase, when users want to quickly decide whether to further study a given tool or reject it as unfit from a usability perspective.

In this chapter, we refine and enlarge the set of considered features and focus on SoftVis tools supporting CM. In this respect, we focus mainly on software developers, testers, and system integrators from the software industry as our target user group. In this sense, the proposed classification model for SoftVis tools for CM presented here is actually a refinement of the general URC presented in Chapter 5 for SoftVis tools in general.

In practice, it usually takes a considerable amount of time for users to try out the many available SoftVis tools before they can asses their suitability and effectiveness in a given context. Although there exist several tool surveys on the internet, it is still very hard to answer the questions "does this tool fit my user profile?" (for tool users) and "what do the users most (dis)like in this tool?" (for tool developers). The lack of such surveys which reflect the degree of support of concrete desirable properties makes the tool acceptance even harder.

Our aim is to provide an evaluation that guides users in the selection of SoftVis tools useful in corrective maintenance, and also exposes the desirable features for such tools, as perceived by their users. Finally, we aim to test and refine the unified requirements classification (URC) model introduced in Chapter 5 on the more focused and more specific area of SoftVis tools for corrective maintenance. This will serve as a test of the suitability of the proposed URC in capturing the desirable features of a more specific set of SoftVis tools, as opposed to its usage for general SoftVis tools which was introduced in Chapter 5.

In this respect, our contribution is an aid for pre-selecting SoftVis tools, either from the set discussed here or from the larger set of tools available in general, based on the identification of a number of desirable features. Secondly, and as a new element in comparison to the work presented in Chapter 5, we are interested to discover possible correlations between the perceived tool acceptance levels and the usage of certain visual techniques, to further determine what makes a tool accepted (or not).

The rest of the chapter is organized as follows. Section 6.2 presents our model for desirable features for SoftVis tools for corrective maintenance. Section 6.3 discusses how this classification extends and refines the URC introduced in Chapter 5. Section 6.4 then introduces the evaluated tools. The evaluation procedure that was used is elaborated in Section 6.5. Section 6.6 presents and discusses the evaluation results, and correlates our findings with a study on program comprehension during corrective maintenance. Finally, Section 6.7 concludes the chapter.

6.2 Classification Model

In this section, our proposed tool classification model is detailed. We use four main categories to classify the software visualization tools: Effectiveness, Tasks supported, Techniques used and Availability. These categories extend our previous analysis, introduced in Chapter 5 in which we empirically studied how users graded a number of general-purpose SoftVis tools along desirable requirements from the perspective of early adoption.

After additional user interviewing aiming at understanding how users choose and grade a SoftVis tool, we observed that, after early adoption issues are settled (positively), users consider different types of requirements to further determine adoption or rejection. In contrast to the early adoption requirements, which appear to capture the immediate usability impression a user has on a given tool, further adoption decisions focus on whether a tool supports the user's tasks effectively while at the same time it is efficiently usable. The URC model introduced in Chapter 5 modeled the early adoption phase. The classification model described in this phase describes the second adoption phase, when users test a tool's usefulness for corrective maintenance.

We structured this insight into four types of requirements. Under each of these feature categories, sub-categories exist. Within each sub-category, we model the degree up to which a tool fulfills the respective requirements on numerical or categorical scales. In contrast, the early adoption URC model did not use such scales, since at that level we were concerned with modeling simple acceptance/rejection decisions.

In the following, the four types of requirements for SoftVis tools for corrective maintenance are described.

6.2.1 Category A: Effectiveness

This category groups non-functional requirements related to the effectiveness of the tools. We call a tool *effective* when it can arguably help users to solve the problems that it was designed to assist with. Effectiveness is strongly task-specific, hence it is hard to measure in general. Yet, there are a number of properties we have identified which SoftVis tools should have to be effective, namely Scalability, Integration and Query support. We focus mainly on non-functional properties here. The functional properties are covered under the category Tasks supported (Sec. 6.2.2).

A1: Scalability

Scalability refers to the extent to which a tool can support large-scale industrial code. By large-scale, we consider systems of millions of lines-of-code and/or thousands of classes and/or source files. We measure scalability on a scale of *low* – *high*. A tool is given a scalability of low if it was created mainly for educational purposes, as a proof-of-concept or as a research prototype. A grading of *medium* is given to tools that go beyond the proof-of-concept level but, at the same time, have limitations with large-scale software. For example, many graph layout techniques have typically medium scalability: They can quickly produce useful structural diagrams of systems having hundreds of elements, but often have problems in visualizing systems having (tens of) thousands of elements and relations, either in speed or result quality. A level of *high* is used to represent tools that comfortably support large-scale code. For example, treemaps and line-oriented pixel visualizations scale well to tens of thousands of data elements, with little or no user intervention. Scalability is determined by the units that each tool uses as data input. For example, if a tool works at file level, then the more files it can handle the higher its scalability [150].

A2: Integration

A tool is considered to be well *integrated* if it can plug into an IDE or similar set-up where it combines seamlessly with other tools that support debugging, software comprehension and other tasks which are part of corrective maintenance. We measure integration by considering the level at which different tools can easily be switched and communicate in completing a given task. We are not interested to assess only the ability of tools to co-exist in the same environment, but also the ability of tools to interact by exchanging and observing exchanged data. This translates into the ability to use an output from one tool as input for another tool and the ability for a tool to discover and present changes to a shared data model performed by another tool.

On a scale of *low – high*, a tool with all the qualities mentioned above is considered to have *high* integration. Tools that are, for the purpose of start-up, shutdown, and user interface, integrated within an IDE but do not actively communicate data and/or change events with peer tools, are given a *medium.* Tools that are not designed to be integrated in an IDE so as to cooperate with other tools, as well as standalone tools, are graded *low.*

A3: Query support

A corrective maintenance tool requires query facilities in order to reduce the time needed to identify specific areas within the code or to answer targeted questions on the code at hand.

We grade the query category on a scale of *low – high.* Tools that do not have this provision, *e.g.,* work in batch or pipeline mode, are considered to have *low* query facilities. Tools that support limited searching (*e.g.,* purely lexical) are given *medium.* There are other tools however that have sophisticated searching facilities. These may include correlating a search in the code base with the visualization displayed (search highlighting) or the use of complex search expressions (*e.g.,* syntax-aware or type-based semantic searches). In this case, a movement from the code to the visualization and the other way round is supported as well as more complex queries. Such tools are graded with *high.*

6.2.2 Category B: Tasks supported

This section details the particular tasks that we aim to support with the considered tools, and the grading of the support level. We mainly focused on the tasks of detecting code smells, code refactoring, trace analysis, and support for debugging activities. While these tasks do not fully cover all corrective maintenance activities, they are typical for a large range of use cases. Moreover, the set of tasks considered here was well understood by our interviewed software engineers, which was a fundamental initial condition for the evaluation.

B1: Detecting code smells / Refactoring

Code smells, related to code quality metrics, generally refer to symptoms related to bad programming and/or design in code. Refactoring is the process carried out to restructure the code and improve one or several quality metrics, *e.g.,* remove the code smells found. For this category a level of *low – high* is applied. Tools that support neither refactoring nor detection of any type of code smells, such as debuggers, will be graded with a *low*. Tools are given *medium* if they have the ability to detect code smells but are unable to suggest ways of correcting them. This is the case of many static analysis tools. A grading of *high* is used for tools that not only detect the smells but can suggest ways to correct them as well.

B2: Trace analysis

Traces are defined as data gathered during the execution of a program [113]. On a level of *low – high*, tools that are graded as *low* are those that can neither generate traces on their own nor examine and visualize traces generated by other programs. Tools that can generate and save traces but can not visualize them, *e.g.,* profilers, will be given *medium*. The same will be applied for tools which examine traces that are generated by other programs, *e.g.,* loosely coupled visualizations of software behavior [23, 90]. A grading of *high* will be for tools that are able to generate traces on their own and create visualizations out of them as well.

B3: Support for debugging activities

Support for debugging activities is measured on a scale of *low – high*. A *low* will be for tools that support only basic debugging *e.g.,* setting breakpoints, stepping through code, and offering basic code watches *e.g.,* for individual data variables.

There are other tools however that focus on higher-level bug-related information. For example, these show the level of debugging activity over a long period of time *e.g.,* by displaying the users that were involved in each activity [149]. Other tools may allow the viewing of many different files within a project with emphasis on those that have failed test cases and those that do not. These type of tools will be given *medium*.

A *high* grade will be allocated for tools that support presenting both low-level and high-level bug-related data and a combination of the tasks explained above. Tools that are able to relate the bugs in a project to the response carried out to remove the bugs while supporting all the other tasks will be also given *high*.

6.2.3 Category C: Availability

There exist other tool aspects that were not specifically covered in the previous sections and which are important in the use of the tools. Some of these include: programming languages supported by the tool, whether the tool is free or commercial, and the platform on which the tool is supported. We gather all these aspects under the description *availability*. We did not grade these aspects separately, since they all had to be met for one to be able to use (or evaluate) the tool. However, we provide information on these aspects as textual descriptions.

6.2.4 Category D: Techniques used

We now analyze the particular visualization techniques that are used by the considered tools. For quantifying the support level, we shall use a simple binary scale *yes or no* for those techniques which are either present or not; and a scale *low – high* for techniques where a more fine-grained level of support can be measured. Assessing the types of techniques used and their support levels is interesting as it can shed more light on factors such as usability and ease of learning, as we shall see later. Since there are too many techniques than one could evaluate, we selected our subset of interest based on previous SoftVis taxonomies [79] as well as our own insight in what the software engineers involved in the study considered to be the most important ones.

D1: Two-dimensional displays

The ability for a tool to show data using just two dimensions is measured on a *yes-no* scale. 2D visualizations are sometimes seen as less expressive than 3D ones, but may be easier to learn and use.

D2: Three-dimensional displays

The usage of a third dimension in order to increase the amount of information being shown by a tool is graded on a *yes-no* scale. As mentioned above, 3D visualizations offer additional data display space, but may come with additional costs.

D3: Animation

Tools are considered to support animation if motion is used when displaying the scrutinized data. Animation can be used to show time-dependent data such as traces or program code evolution. Tools that support animation will be graded with *yes*. Those that do not support animation for these purposes, but support animation only for navigation purposes, *e.g.,* smooth zooming and panning [1, 47], will be given *no*.

D4: Color usage

Tools that do not use color at all, *e.g.,* editors without syntax highlighting or command-line debuggers, will be categorized with *low*. Tools that use just a few saturated colors only, *e.g.,* to display categorical values, will have *medium* for color usage. Tools that use a wide range of colors, *e.g.,* doing blending for multivariate data display or color mapping for showing continuous value ranges, will be given *high*.

D5: User interaction

Within the *user interaction* category, we quantify the extent to which a user can interact with the displayed information. Tools that show a view that the user cannot alter in any way are graded *low.* Tools that support a single main action via user interaction, *e.g.,* adjusting the layout of some software architecture view with the mouse by moving the nodes displayed in graph view are given *medium* [111]. If the tool supports two or more different actions (tasks) by combining two or more medium-graded user interaction facilities, *e.g.,* direct mouse manipulation and rubberbanding, it will be given *high.*

D6: Multiple views

Several tools are able to display the same (or related) bits of information in many different ways and perspectives. These are considered to have *multiple views.* We grade the multiple view support based on how these views interact. Tools that have only one view for all the tasks that they carry out are given *low.* Tools that have several not linked views, *i.e.,* interaction and/or selection in a view does not affect the other views, are given *medium.* Linking implies that whenever the user interacts with the data in a view, *e.g.,* selects or changes an object, then the other views respond accordingly, *e.g.,* show the same object selected or modified. Tools that have several linked views are graded *high.*

D7: Information density

The *information density* dimension quantifies the amount of information that a given tool can display using a given unit of screen space. Tools that are given *high* are those that can display the greatest amount of information per screen space unit, *i.e.,* achieve the highest information density.

It is hard to precisely define how to quantify information density, since different tools display different types of information. We refer to the type of information units being displayed as information granularity. We see several fixed levels of information granularity recurring in many SoftVis tools, inversely correlated with the information units' sizes. From high to low granularity, these are: code lines, syntactic components (*e.g.,* classes, functions, services, or components), and high-level organization units (*e.g.,* files, folders or packages). This scale is a natural element of the type of data involved in software: Whereas in the physical world (and thus in scientific visualization) most data is continuous, therefore naturally allows for meaningful interpolation and sampling at any desired scale, software data (and thus software visualization) is inherently discrete. There are no natural ways to *e.g.* interpolate between a line of code and a function, or subsample a set of code lines.

Tools can use the same given screen space to show source code in a file or class (*e.g.,* Coderush [27]), classes in a subsystem as UML diagrams or graphs (*e.g.,* JIVE [42]), or all files in a repository as dense pixel displays or using treemaps (*e.g.,* CVSgrab [149]). We compare information density by counting the number of displayed units per screen space unit, on a scale of *low, medium, high.* For example, a text editor has *low* information density, a UML browser *medium* density, and a dense-pixel treemap visualization *high* density.

D7: Navigation

Panning, scrolling, zooming and viewpoint change are all categorized under navigation options. We evaluate this level using *yes or no* depending on whether these options are supported or not.

D8: Dynamic visualization

This refers to the ability of a tool to display data generated on-the-fly while the program is running, *e.g.,* debugging or tracing data. Tools that do not have this ability will be given *no.* The tools that are able to do this will be given *yes.*

6.3 URC Refinement for Corrective Maintenance SoftVis Tools

We have described so far a Requirements Classification for SoftVis tools for Corrective Maintenance (further referred to as CMRC). In this section, we describe the way in which this classification relates to the Universal Requirements Classification (URC) model introduced in Chapter 5 (see also Figure 5.1).

First and foremost, the two classifications have different purposes. While the URC models requirements which are important during an early adoption phase of SoftVis tools, the CMRC models requirements which are relevant during a subsequent adoption phase, during which one looks at specific support for a given set of activities and specific features. As such, the two classifications can be seen as independent, serving two different purposes.

However, it is clear that there are also overlaps of the two classifications, since the usability and usefulness of a SoftVis tool has aspects which are relevant at any moment during its evaluation. In this sense, The CMRC is a refinement, but also an extension, of the URC for general-purpose SoftVis tools introduced in Chapter 5. Below we show how each of the four components of the CMRC (*Effectiveness, Tasks supported, Availability and Techniques used*) maps to the original URC.

First, the various sub-components of the *Effectiveness* CMRC requirement refine several URC requirements, as follows. The *Scalability* CMRC requirement refines the Scalability/Ease to use URC requirement. Indeed, for a tool to be easy to use, it must further be scalable and handle efficiently large code bases. The *Integration* CMRC requirement refines the requirement from the URC with the same name. The *Query support* CMRC requirement refines the *Search enabled* URC requirement.

Secondly, the *Tasks supported* CMRC requirement refines the *Realistic/Adress problems of real interest* URC requirement, by specifying a number of tasks (or problems) of interest in CM. For a SoftVis tool for CM to be realistically useful, it has to address specific CM tasks.

Thirdly, the *Availability* CMRC requirement refines the *Realistic/Uses realistic computer specifications* URC requirement. A SoftVis tool for CM will be realistically usable if it is indeed available, *i.e.*, runs on the target platform, comes with a suitable licensing model, and understands the source code for the CM task at hand.

The fourth and final CMRC category, *Techniques used*, does not however map to (or refine) one of the existing categories in the original URC model. This category is to be seen as an extension of the URC, *i.e.*, a new dimension of requirements. While the original URC categories (Search enabled, Meta-data display, Simplicity/Lightweight, Added advantage, Integrated, and Realistic) all address the general suitability of a SoftVis tool for a wide range of activities, the new *Techniques used* category is different: It addresses the way users see visualization techniques as useful or not. The introduction of this category was deemed necessary as there are several studies in SoftVis and InfoVis that compare visualization techniques from the perspective of user acceptance (see *e.g.,*[83, 47, 11]). In the terminology of Maletic *et al.*, the new *Techniques used* category covers the 'representation' dimension [79].

One may argue that introducing this new category of Techniques used is not part of an URC that should purely describe user requirements, and should be agnostic of the techniques employed to achieve those requirements. However, in practice, many researchers have argued that certain visualization techniques are easier accepted than others (we refer here again to [83, 47, 11]). This is further supported by our analysis of concrete SoftVis tools presented further in this and the next chapter, where 3D visualization was perceived by industrial users as less accepted than 2D visualization. In this case, having *Techniques used* as a CMRC category is useful, since it can help assessing the suitability or acceptance of a tool based on the type of techniques the tool employs.

6.4 Evaluated Tools

The tool selection procedure used here followed the same guidelines as for the selection of general SofVis tools for the study described in Chapter 4 (Sec. 4.2.1), with a narrower focus on tools that can, or claim to, support corrective maintenance activities.

We now present the tools considered for the evaluation, starting with a short description for each tool. We chose a mix of software visualization tools that support corrective maintenance from both commercial and research areas, all of them however freely available for evaluation purposes. Secondly, we tried to choose tools that support different types of tasks in order to cater for a wider audience and broader evaluation.

6.4.1 Allinea DDT

This tool is designed for debugging of scalar, multi-threaded large scale parallel applications written in C, C++ and Fortran [3]. It can interoperate with all major MPI implementations and batch queuing systems. Visualization is used mainly in the control of program execution as well as the display and manipulation of multidimensional data [114]. A snapshot of Allinea DDT is shown in Figure 6.1.

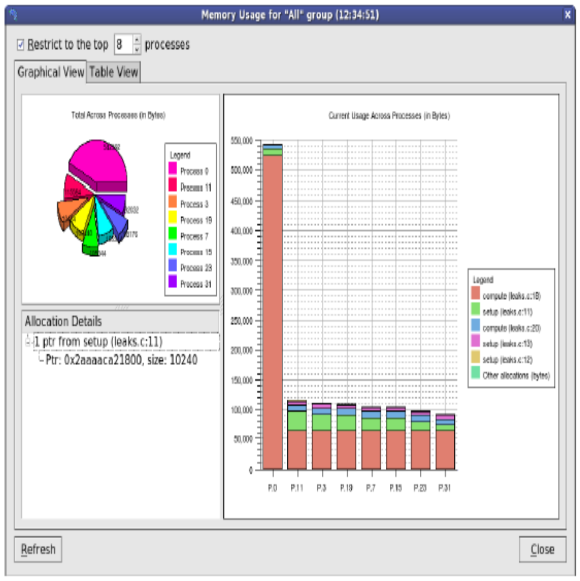
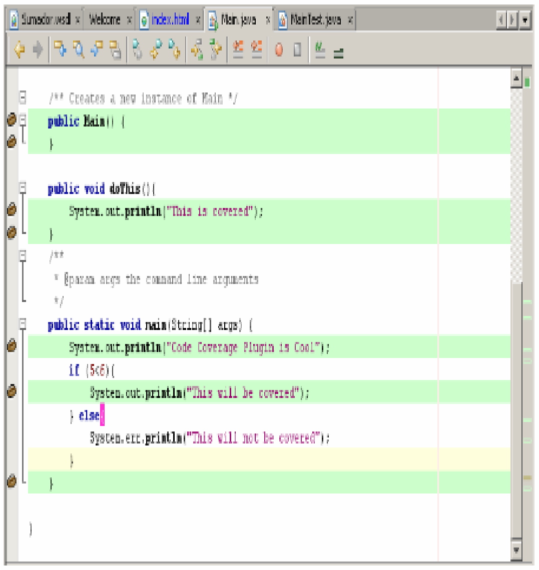

Memory Usage for "All" group (12:34:51)
Restrict to the top 8 processes
Graphical View
Table View
Total Across Processes (in Bytes)
Current Usage Across Processes (in Bytes)
Allocation Details
1 ptr from setup (leaks.c:11)
Ptr: 0x2aaaaca21800, size: 10240
Refresh
Close

Figure 6.1: Allinea DDT [3]



```
    /** Creates a new instance of Main */
    public Main() {
    }

    public void doThis(){
        System.out.println("This is covered");
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Code Coverage Plugin is Cool");
        if (5<6){
            System.out.println("This will be covered");
        } else{
            System.err.println("This will not be covered");
        }
    }

}
```

Figure 6.2: Code Coverage [93]

6.4.2 CodeRush with RefactorPro

In collaboration with Visual Studio, this tool enables the refactoring of code by providing change options with hints as well. Visualization clues are continuously presented within the code so as to prompt action when needed [27].

6.4.3 CodePro AnalytiX

This is an Eclipse plugin that enables code coverage analysis, dependency analysis as well as report generation with the aid of visualization [53]. CodePro AnalytiX also integrates into the Rational and WebSphere development environments.

6.4.4 Code Coverage Plugin NetBeans

Using coloring of source code, this plugin helps in visually identifying code portions that have low coverage according to a given set of test cases [93]. It supports Java and works with the

NetBeans IDE. A Code Coverage is shown in Figure 6.2.

6.4.5 CVSgrab

CVSgrab visualizes data from the CVS and Subversion software configuration management systems. CVSgrab supports corrective maintenance by displaying the correlation of project activity and bug data mined from a Bugzilla database, using dense pixel displays. Sort and cluster mechanisms enable users to correlate various aspects of interest in a given project evolution [149]. A snapshot of CVSgrab is shown in Figure 6.3.

6.4.6 Gammatella

This visualization tool supports the remote monitoring of Java programs [98]. It is able to gather the execution data from remote machines and visualize it locally using a combination of execution metric-colored treemaps and SeeSoft-like code displays.

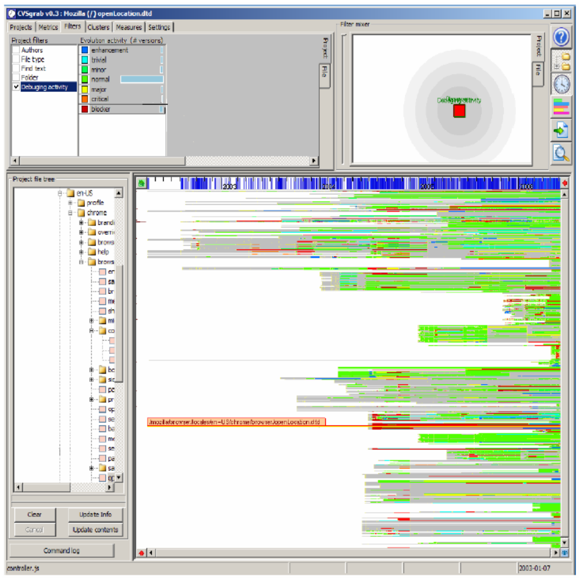


Figure 6.3: CVSgrab [149]

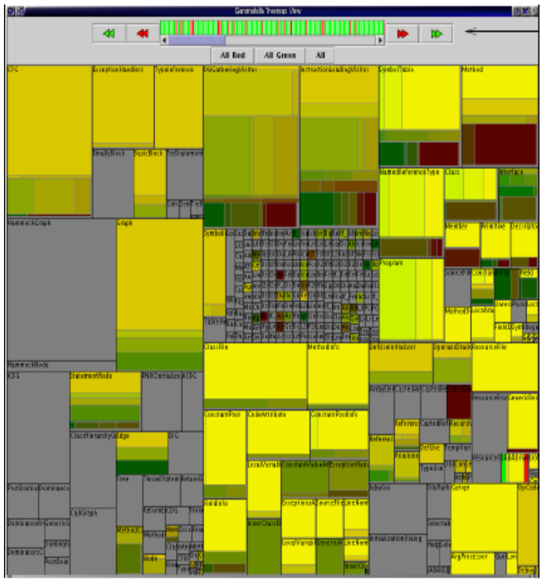


Figure 6.4: Gammatella's main view [98]

6.4.7 JBIXBE

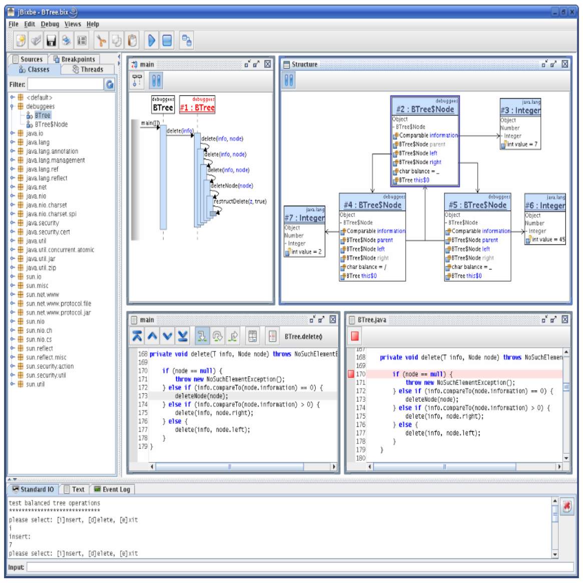
This is a standalone Java debugger enhanced with visualization features [30]. It is customized for Java programs and simplifies debugging of multi-threaded applications. It combines classical debugger interaction (breakpoints, step-mode execution, watches) with execution flowcharts and UML-like class diagrams of the debugged code. A snapshot of JBIXBE is shown in Figure 6.5.

6.4.8 JIVE

JIVE supports the analysis of Java programs during runtime. Using forward and backward stepping through program execution, JIVE facilitates debugging. JIVE visualizes execution history by means of interactive UML sequence diagrams [42]. A snapshot of JIVE is shown in Figure 6.6.

6.4.9 JSwat

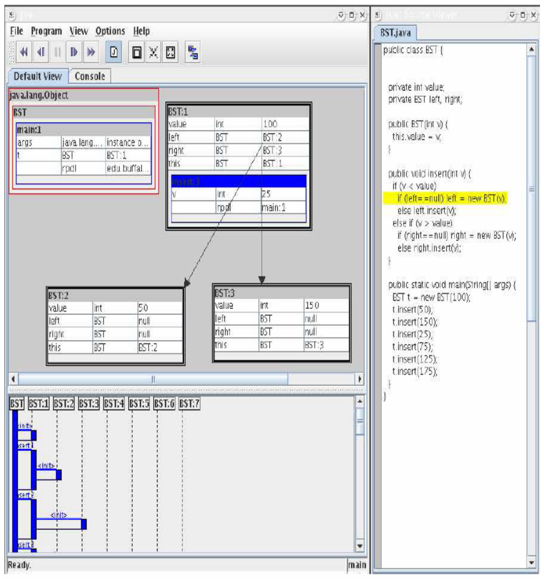
Jswat is a graphical debugger that uses visualization combined with debugging of Java source code [37]. The provided visualizations include standard IDE-like breakpoints and watches. A snapshot of JSwat is shown in Figure 6.7.



```
private void delete(T info, Node node) throws NoSuchElementE
    if (node == null) {
        throw new NoSuchElementException();
    } else if (info.compareTo(node.information) == 0) {
        deleteNode(node);
    } else if (info.compareTo(node.information) > 0) {
        delete(info, node.right);
    } else {
        delete(info, node.left);
    }
}
```

```
test balanced tree operations
******************************
please select: [i]nsert, [d]elete, [e]xit
i
insert:
7
please select: [i]nsert, [d]elete, [e]xit
```

Figure 6.5: JBIXBE [30]



```
public class BST {

  private int value;
  private BST left, right;

  public BST(int v) {
    this.value = v;
  }

  public void insert(int v) {
    if (v < value)
      if (left==null) left = new BST(v);
      else left.insert(v);
    else if (v > value)
      if (right==null) right = new BST(v);
      else right.insert(v);
  }

  public static void main(String[] args) {
    BST t = new BST(100);
    t.insert(50);
    t.insert(150);
    t.insert(25);
    t.insert(75);
    t.insert(125);
    t.insert(175);
  }
}
```

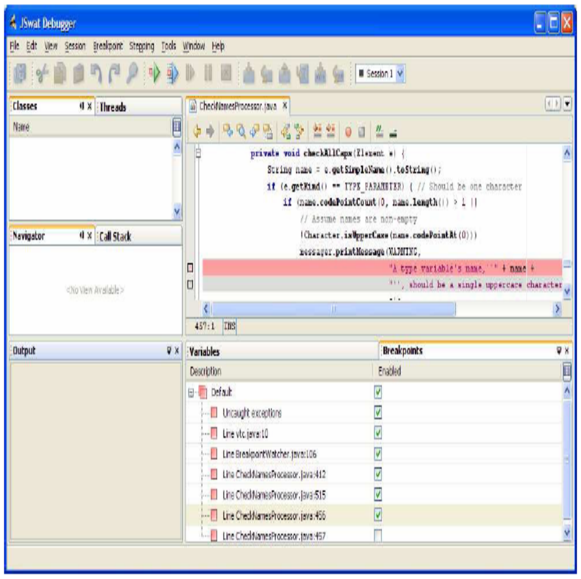
Figure 6.6: JIVE [42]

6.4.10 Paraver

Paraver is a tool that analyzes and visualizes program traces. It is capable of concurrent visualization of separate trace files as well as showing multiple views of the same trace file [104]. It uses 2D colored dense pixel displays very similar to the ones used by CVSgrab for software evolution visualization [149]. A snapshot of three views of Paraver is shown in Figure 6.9.

6.4.11 Project Analyzer

Project Analyzer is a standalone tool that aids in detecting errors of Visual Basic code with the aid of visualization. Project Analyzer helps in checking for error proneness in the tool as well as guiding on ways in which to improve the code [4]. It comes as an IDE using syntax highlighting and multiple linked views of the code.


JSwat Debugger
File Edit View Session Breakpoint Stepping Tools Window Help
Session 1
Classes
Threads
Name
Navigator
Call Stack
<No View Available>
CheckNamesProcessor.java
457:1 INS
Output
Variables
Breakpoints
Description
Enabled
Default
Uncaught exceptions
Line vtc.java:10
Line BreakpointWatcher.java:106
Line CheckNamesProcessor.java:412
Line CheckNamesProcessor.java:515
Line CheckNamesProcessor.java:456
Line CheckNamesProcessor.java:457

```
    private void checkAllCaps(Element e) {
        String name = e.getSimpleName().toString();
        if (e.getKind() == TYPE_PARAMETER) { // Should be one character
            if (name.codePointCount(0, name.length()) > 1 ||
                // Assume names are non-empty
                !Character.isUpperCase(name.codePointAt(0)))
                messager.printMessage(WARNING,
                                      "A type variable's name,``" + name +
                                      "'', should be a single uppercase character
```

Figure 6.7: Jswat [37]

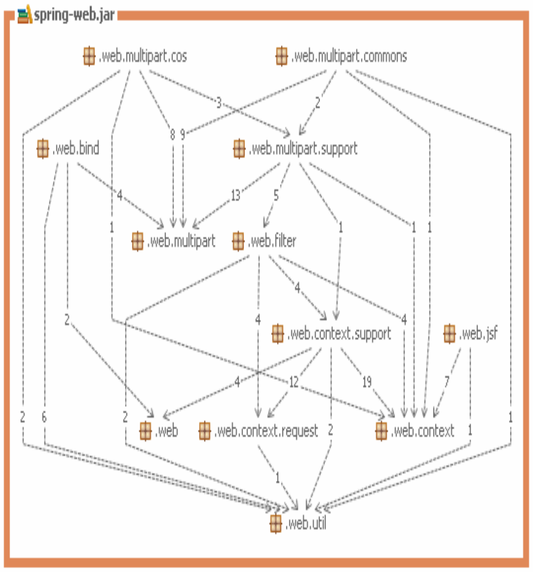


Figure 6.8: STAN [96]

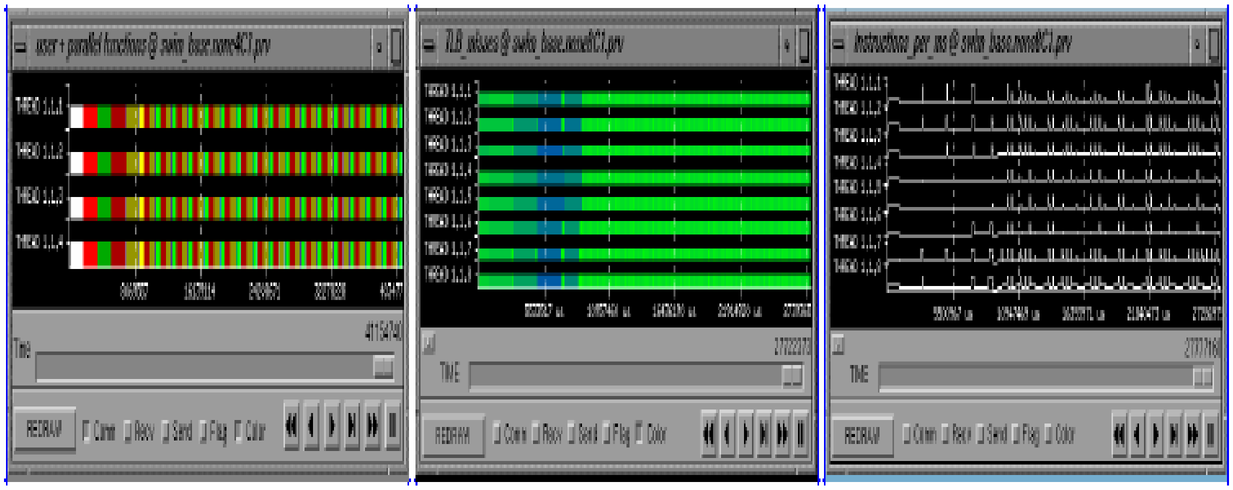


Figure 6.9: Three different views of Paraver [104]

6.4.12 Source Navigator

Source Navigator is an open source tool that loads information extracted from source code into a project database, *e.g.,* file names, program symbols, and relationships like declaration-to-definition [126]. Source Navigator provides different browsers for this database, both code-based using syntax highlighting and graph-based for showing code relationships. The tool supports C, C++, Fortran, COBOL and Java and uses the `grep` tool to search within the source code.

6.4.13 STAN

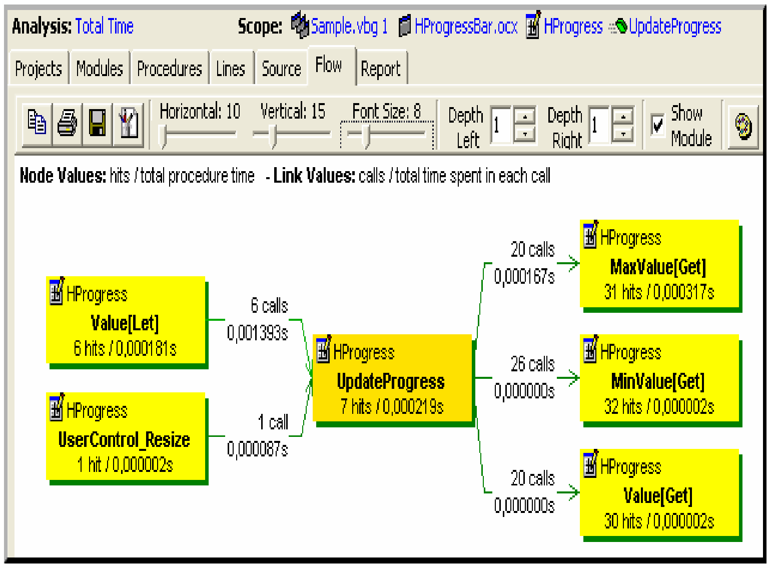
Structure Analysis for Java (STAN) is an Eclipse plugin that visualizes Java code with the aim of understanding and detecting design flaws [96]. Visualization is done using several linked graph views, metric-colored treemaps, and metric histograms. A snapshot of STAN is shown in Figure 6.8.

6.4.14 Tarantula

This is a system that helps users find faults or problems with their systems using visualization. Color is used to differentiate between statements that have passed the test (green) and those that have failed (red) [57]. Yellow represents cases that have both passed and failed bits within them. This system extends the dense pixel visualization technique introduced by Seesoft [34]. A snapshot of Tarantula is shown in Figure 5.6 in Chapter 5.

6.4.15 VB Watch v2

This is a toolset of three tools: VB Watch Profiler, Protector and Debugger [5]. It enables the testing and debugging of Visual Basic 6 code. Visualizations include code-level multiple views, metrics histograms, and metric-annotated call graphs. A snapshot of VB Watch is shown in Figure 6.10.



Analysis: Total Time
Scope: Sample.vbg 1 HProgressBar.ocx HProgress UpdateProgress
Projects
Modules
Procedures
Lines
Source
Flow
Report
Horizontal: 10
Vertical: 15
Font Size: 8
Depth Left 1
Depth Right 1
Show Module
Node Values: hits / total procedure time - Link Values: calls / total time spent in each call
HProgress Value[Let] 6 hits / 0,000181s
HProgress UserControl_Resize 1 hit / 0,000002s
6 calls 0,001393s
1 call 0,000087s
HProgress UpdateProgress 7 hits / 0,000219s
20 calls 0,000167s
26 calls 0,000000s
20 calls 0,000000s
HProgress MaxValue[Get] 31 hits / 0,000317s
HProgress MinValue[Get] 32 hits / 0,000002s
HProgress Value[Get] 30 hits / 0,000002s

Figure 6.10: VB Watch [5]

6.5 Tools Evaluation

In this section, we describe our tool evaluation procedure.

6.5.1 Participants

Fifteen professional software developers, 11 male and 4 female, participated in the study. The participants did not include ourselves, and were all working in non-research-related fields of the software industry. Among the software developers that participated in the study, only two had ever used a SoftVis tool before in their work. Their details are given in Table 6.1.

A pre-study questionnaire was used for the participants in order to ensure that the right skill level was present for the evaluation. The questionnaire sought information about the programming languages that the developers had working knowledge of as well as the visualization tools that they had used before. We specifically selected people having a minimal number of years in software maintenance (over 4), and who were familiar with at least two, but preferably three, mainstream development languages. These constraints were necessary to ensure that the evaluation would not get biased by a lack of experience and/or exposure to the problems of maintenance, and also that the users would be familiar with visualization tools in general, and willing to experiment with new tools. We also specifically assigned users to evaluate tools they never used before so as to avoid prior knowledge bias.

In order to ensure efficiency, tools were grouped according to similarity and allocated to the developers as shown by Table 6.2. Next, we assigned tools to developers, trying to match the tool's constraints (programming language and running platform) with the ones best known by the respective developer.

6.5.2 Source Code

Three source code bases were provided to the participants in order to perform the study. For tools that use Java, the ArgoUML system [9] source code was used. Tools that targeted C++ were evaluated on the source code of the network simulator *ns 2* [45]. Tools that targeted Visual Basic were evaluated on the proprietary code of a vehicle registry system written in VB6. Finally, CVSgrab was evaluated on both the ArgoUML and Visualization Toolkit [66] open-source repositories. The programming language is here not important, as the type of information CVSgrab visualizes is language independent.

6.5.3 Evaluation Procedure

The developers were let to use the assigned tool(s) to perform several analysis tasks on the assigned systems, as described further. The 15 tools were compared against the four categories of desirable features described in Section 6.2.

User	Languages known	Years of experience	Years in software maintenance	Group of tools evaluated
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 6.1: Evaluation participants

Each developer had a 15 minutes introduction to each of the tools that they had to grade. After the introduction, the developers were asked to grade the tools against our four categories based on the pre-defined criteria of *Effectiveness*, *Tasks supported*, *Availability* and *Techniques used*. We used six generic tasks to aid the developers in assessing the tools' capabilities for the sub-categories of (a) *scalability*, (b) *Integration*, (c) *Query support*, (d) *Detecting code smells*, (e) *Trace analysis*, and (f) *Support for debugging activities.*

These tasks were as follows:

a Analyze the provided code for errors or error proneness incrementally, from classes to files to packages and finally the whole project;

b Using the tool, make a change in the source code provided and analyze the effect of your change using the complementary tools;

c Query the code base for all classes that have errors and visualize them to determine how they interact with the rest of the code;

d Use the tool in order to find two code smells and use the refactoring hints provided by the tool to improve it;

e Use the tool to generate traces from the code and analyze the generated traces in order to know the codes behavior at run time;

f Use the tool to set breakpoints, step through the code and display the users that have been involved in debugging activities over the last 2 years;

Comparing these tasks with those outlined for the tool evaluations in Chapters 4 and 5, we

Tools		
Tool groups		**Group rationale**
1	- VBWatch - CodeRush	-Use Windows platform - Support Visual Basic
2	- JIVE - JBIXE - Source Navigator	- All support java
3	- Allinea DDT - Paraver - NetBeans code coverage plugin	- Use Linux platform
4	- Project Analyzer - CodePro Analytix	- Use Windows platform
5	- Tarantula - Gammatella	-Research tools
6	- CVSgrab	- Tool's input is history of industry-size projects
7	- JSwat - STAN	- All support Java - Use Windows platform

Table 6.2: Tools grouping by common features

immediately see that they are more complex and fine-grained. This is an expected point, as we are now interested in an in-depth evaluation of the desirable features of SoftVis tools for a specific domain (corrective maintenance) as opposed to the quick type of evaluation described in the previous chapters for the early tool adoption process.

After carrying out those tasks, the participants were in a position to know the techniques supported by the tools and the level to which they are supported. Next, the participants graded the presence of these techniques, and finalized the evaluation by writing down any other remarks they had, after the grading was completed.

In total, each tool was evaluated by at least two developers. Tables 6.3 and 6.4 show the results of the tools' evaluation against the desirable requirements and set of supported techniques, obtained by averaging the grades produced by the two or more developers who evaluated that tool.

We are fully aware that a set of specific, uniform corrective maintenance tasks would have been ideal in measuring the selected tools against the categories identified. However, given the high heterogeneity of the tools and analyzed systems, we could not easily design such a set. Moreover, we were interested in extracting user feedback at a broader level, so it can be easier extrapolated to other tools. This therefore made it difficult to evaluate tools that support two different tasks (*e.g., detection of code smells* and *debugging of parallel applications*) using the same specific predefined tasks.

6.6 Evaluation Results

Tables 6.3 and 6.4 show the results of the evaluation of the considered tools against the four categories of desirable features (Sec. 6.2). We discuss these results below.

TOOLS	EFFECTIVENESS			TASKS SUPPORTED			AVAILABILITY		
	Scalability	Integration	Query support	Detecting Code Smells	Trace Analysis	Support for Debugging Activities	Language Support	Licensing	Platform
Allinea DDT	High	Low	High	Medium	Medium	Medium	C, C++ Fotran	Commercial	Linux
CodeRush	Medium	High	Medium	High	Low	Low	VB, C#	Commercial	Windows
Code Pro Analytix	High	Medium	High	High	Low	Low	Java	Commercial	Linux Windows
Code Coverage Plugin Netbeans	Medium	High	Medium	Medium	Low	Medium	Java	Non Commercial	Linux Windows
CVSgrab	High	High	Medium	Medium	Low	Medium		Non Commercial	Linux Windows
Gammatella	Low	Low	High	Medium	High	Medium	Java	Non Commercial	Linux Windows
JBIXBE	Medium	Low	Medium	Low	Low	Medium	Java	Commercial	Linux Windows
JIVE	Medium	Medium	High	Medium	High	Medium	Java	Non Commercial	Linux Windows
JSwat	Medium	High	Medium	Medium	Low	Low	Java	Non Commercial	Linux Windows
Paraver	High	Low	High	Medium	High	High	Java	Non Commercial	Linux
Project Analyzer	Medium	Low	Medium	Medium	Low	Low	VB	Commercial	Windows
Source Navigator	High	High	Medium	Medium	Low	Low	Java, C++ C,Fotran	Non Commercial	Windows
STAN	High	Medium	High	Medium	Medium	Low	Java	Commercial	Windows Linux
Tarantula	Low	Low	Low	Medium	High	Medium	C	Non Commercial	Linux
VB Watch	Medium	Low	Medium	Medium	Medium	Medium	VB 6	Commercial	Windows

Table 6.3: Tools against desirable features

	TECHNIQUES USED								
TOOLS	2D	3D	Animation	Color usage	User Interaction	Multi Views	Information Density	Navigation	Dynamic Visualization
Allinea DDT	Yes	Yes	No	Medium	Low	Medium	Low	Yes	No
CodeRush with RefactorPro	Yes	No	Yes	Medium	Medium	High	Low	Yes	No
CodePro Analytix	Yes	No	No	Medium	Medium	High	Medium	Yes	No
Code Coverage plugin for NetBeans	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 6.4: Tools against techniques used

As already mentioned, only two developers had used SoftVis tools before in their work, although all were aware of the existence of several such tools. When asked why, many of the participants noted that their choice of a tool to use for corrective maintenance tasks at work was influenced by what their competitors (or companies perceived as larger than theirs) use. The developers also noted that their busy schedules did not give them enough time to evaluate the different tools available in sufficient detail in order to form their own opinions. This may be an indicator of the fact that the so-called early adoption phase did not essentially succeed on these tools, as more time was needed to form a good impression of the tools' usability.

This aspect presents a challenge for new tool developers, as the chances of small-to-medium size companies using their SoftVis tools might be tied to the tool acceptance by larger companies. The lack of time for the developers to validate the various tools also highlights the need for tool selection guides in the different areas of SoftVis, on the one hand, and SoftVis tools that are specifically designed (and presented) in ways that favor a quick early adoption process.

In the few cases where the developers had used SoftVis tools before, preference was given to those that would not show too much detail or interaction features. Even for this study, we noticed that tool acceptance is heavily influenced by the amount of user interaction provided. When interaction is hardly supported, tools tend to become unusable without a considerable amount of learning time. Yet, too much interaction becomes cumbersome for the user to learn, thus affecting usability as well, *e.g.,* the complex 2D widgets and dense visualizations offered by CVSgrab [150]. An average amount of user interaction would therefore be suitable for users.

This finding correlates well with our earlier study [117]. During that study, software developers also showed discontent with a tool which required a lot of user input before displaying any output. Nevertheless, tools that did not enable the user to contribute to or tune at all the final visualization (layout, colors, annotations) were also not desired.

Another observation, noted from more than half the participants, is the frequent use of the Eclipse IDE. Many developers felt that the features provided by Eclipse were sufficient for their corrective maintenance tasks and thus never felt the need for additional SoftVis tools for corrective maintenance. A few even raised questions as to whether the Eclipse IDE could not be categorized as a SoftVis tool.

This observation suggests that new SoftVis tool developers focusing on corrective maintenance would need to show the added advantage that their tool provides in comparison with a plain IDE, a point already mentioned in the URC presented in Chapter 5. An evaluation such as the one described here is relevant in proving the level to which a tool does what it claims to do. We believe that tool developers are also interested in knowing that the users of their SoftVis tools are actually performing faster or better than the ones without the tools. Failure to have comparative studies may create skepticism on the users' side.

Tool developers can use the developers' IDE attachment to their benefit. First, they can create tools that plug into the IDE's. Second, they can use the IDE metaphor in organizing their visualization tools. For example, the developers that evaluated the STAN tool found it easy and natural to use as its visualizations utilized the basic Eclipse layouts. On the other hand, five of the users taking part in our study felt that education had a major role to play in tool adoption in the industry. They said that, had they been exposed to several SoftVis tools during their undergraduate training, they would have been more inclined and motivated to use such tools in their work.

6.6.1 Observations from the study

We have observed a variety of tool trends from the evaluation results. These trends are detailed below.

Among the considered tools, there is more support for *static* visualization compared to *dynamic* visualization. Users that need tools that use either technique would, however, be able to find them. Based on this trend, tool developers may choose to increase on the tools that support dynamic visualization as those for static visualization seem to be sufficiently represented.

Animation and *3D*, on the other hand, are not frequently utilized by the tools. Also, we noticed that 2D visualizations are much better accepted by virtually all users. This trend also correlates to previous observations of users favoring 3D and animation less, emphe.g. [150].

The preciseness of answering *queries* while carrying out corrective maintenance was ranked as very important. The popularity of Rational Purify Plus within the industry, albeit its poor presentation capabilities, supports this point [51]. Most of the tools considered here, however, focus on visualizing the data, but do not offer analyses helping the actual problem correction. A tighter integration between analysis and presentation was deemed as extremely desirable by all users.

Along the same line, tool *integration* was mentioned by all users to be greatly desired for SoftVis tools in program comprehension. 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, which were appreciated by the users, support 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* and *multiple views* are well supported and accepted, and most tools make use of them. Availability of multiple views, enabling user interaction and facilitating navigation in SoftVis tools, are three features extremely well received by most users. These three features enable developers to switch to the preferred visual representation of the data under observation as well as customize the view, emphe.g., rearrange objects in layouts, to decrease ambiguous or hard-to-interpret displays.

Within the programming *languages*, Java is well supported by most of the tools. Java programmers therefore have the most SoftVis tool support for corrective maintenance. The ease with which Java engines enable data extraction and analysis seems to be responsible for this trend. This is both true for static analysis and dynamic analysis. Hence, future tool developers can choose to support Java based on the ease explained above or may choose instead to capitalize on the much lower support of the other languages *e.g.* C++ so as to develop tools for them, as recently shown in another study [139].

The visual *scalability* of the tools, or information density, is not high in most of the tools. This can be a hindrance to user acceptance of the tools. The higher the information density, the more one can monitor at any given time. This is especially important when monitoring high-volume dynamic data such as traces. Low density means that users takes more time to analyze code as each session represents only a small bit of the total project. However, a too high density, such as in dense pixel visualizations, *e.g.,* [150] and similar, was also seen as quite hard to learn and understand, and may prove an initial acceptance blocker. Overall, we believe tool developers should incorporate *multiscale methods* for flexibly selecting the amount of information desired by the user to be shown at a given time.

6.6.2 Results comparison with program comprehension studies

We now compare our evaluation results with the results of von Mayrhauser and Vans who looked at program understanding behavior during corrective maintenance of large-scale software by observing developers carry out corrective maintenance tasks on industry-size projects [148].

Actions and process

The actions of the developers carrying out corrective maintenance involved chunking and hypothesis generation at all the different levels of abstraction [148]. Based on this, SoftVis tools that support corrective maintenance should enable a user to view diverse abstraction levels while enabling the generation of the hypothesis. This can only be done if query facilities are supported within the tool as they form a basis from which a hypothesis can be proved or disproved accordingly. Inability to search large code also makes it impossible for the developer to carry out chunking. Additional factors crucial in the presentation of different abstraction levels are the multiple view support and, partly, and the IDE integration to coordinate inter-view navigation.

The categorization presented in this paper also highlights the importance of *high* query support. This level enables a query within the code to be related to the visualization displayed thus enabling faster chunking. Six of the 15 tools evaluated had *high* query support with 8 tools having *medium* support. The majority of the tools are therefore tending towards *high* query support which is the prefered mode for corrective maintenance [117, 148].

Information needs

Software maintenance professionals have varying information needs depending on their expertise [148]. A tendency was, however, outlined for the more experienced programmers to search for very specific types of information. This further emphasizes the need for refined, *high* query abilities in the SoftVis tools for corrective maintenance. These facilities can enable the maintainer to relate the information got to the tasks to be carried out in order to perform corrective maintenance.

Finally, we should note that, as program comprehension is an essential ingredient of (corrective) maintenance, a further comparison of existing SoftVis tools for this area with studies of non-visual program comprehension tools and techniques would bring additional valuable information.

6.7 Conclusions

In this chapter, the general URC model for SoftVis tools from Chapter 5 has been refined to a more specific one for SoftVis and CM (CMRC). The usefulness of this refined model has been demonstrated by doing a user study on actual tools and CM tasks. The model was instrumental in structuring the comparison and discussion of the tools. While the URC model covers the general desirable features involved in the early acceptance process of a new tool, the refined CMRC model covers the more fine-grained features involved in supporting the more specific corrective maintenance activities.

An evaluation of fifteen software visualization tools that support corrective maintenance was carried out. The evaluation was not intended to choose one tool as superior over others. The aim was to provide a pattern to evaluate existing visualization tools in corrective maintenance, so as to guide tool users and developers in their choice of tool to use or techniques to support, and to gain supportive evidence for the proposed CMRC model.

Furthermore, in this evaluation, the users assigned actual scores to the degree up to which a given SoftVis tool matches a desirable requirement. In comparison, the users of the more general evaluation of SoftVis tools against requirements presented in Chapter 5 did not quantify their observations, but delivered them only qualitatively. Several observations can be made here. Constructing a quantitative evaluation of a broad range of tools, against a relatively high-level set of desirable features, and comparing such quantitative results obtained from different users, is a delicate process. For this reason, we restricted the degree of detail of the quantification scales to a few levels (Low, Medium, High). However, although imperfect, we believe that this approach is more useful, and more scalable, for drawing actual conclusions about the suitability of a given SoftVis tool for a given sub-discipline (like corrective maintenance) than having a purely qualitative evaluation of tools (like the one presented in Chapter 5). This is in line with the different intentions of the two models: Whereas the URC model describes the early adoption process, which has typically a pass/fail outcome, the CMRC model describes the more subtle evaluation process of a SoftVis tool for CM, which has more gradations.

From the techniques perspective, we see several tool features such as IDE integration, scalability, visual multiscales and multiple views, and query support being strongly required by users and provided by increasingly many tools, whereas some other features like 3D or animation are less present in tools.

The study presented here still has 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 professional work?

c When developing SoftVis tools for corrective maintenance, at what point should the views of practicing software maintainers be *mainly* involved? Should it be at the point of conceiving the idea; during the development of the tool; or after the tool is developed and is fully functional? Involvement at all points, although desirable, may not be practically feasible.

In the next chapter, three tools that score differently against the CMRC model will be evaluated with concrete corrective maintenance tasks. This will provide a more quantitative view of the tools' abilities to address their tasks as well as give feedback related to the predictive powers of the CMRC.

Chapter 7

Validating the classification model

In Chapter 6, a classification model for desirable features of SoftVis tools used in corrective maintenance was proposed. The model gradually emerged out of three studies carried out over a period of two years. In the first study, expert programmers were exposed to three SoftVis tools and asked to carry out several simple comprehension and maintenance tasks (Chapter 4). Their feedback was collected and structured in a model of desirable features. In a second study, the model was compared against ten SoftVis tools to validate and refine the initially elicited desirable features in the context of early tool adoption (Chapter 5). The result of the second study is a Unified Requirements Classification (URC) model that is applicable to SoftVis tools in general.

In a third study, our URC model of desirable features was particularized for corrective maintenance (CM) tasks, by exposing 16 programmers to 15 SoftVis tools that target CM tasks, and having them both emphasize which features they consider desirable for assisting with a number of submitted CM tasks, and also grade the tools according to the degree to which they implement these desirable features (Chapter 6). The tasks included analyzing source code for errors, debugging activities, finding classes and class interactions involved in a given error, and generating and analyzing traces. The result, shown in Figure 7.1, classifies desirable features in four main categories, each having sub-categories.

In terms of the case study design discussed by Yin [162], the above-mentioned three studies led to the formation of a hypothesis, or theory. Specifically, let us further consider the refined URC for SoftVis tools for CM (CMRC). The main hypothesis of the CMRC presented is that a SoftVis tool that scores high on most of the model's desirable features, will be indeed effective in addressing (a subset of) the targeted CM tasks. However, this hypothesis needs to be tested from various angles:

- *predictive power:* is a SoftVis tool, that scores high on the model's desirable features, perceived as useful by actual users in a concrete CM task during their actual work?
- *completeness:* would other desirable features, beyond those covered by the model, emerge when actually using such tools?

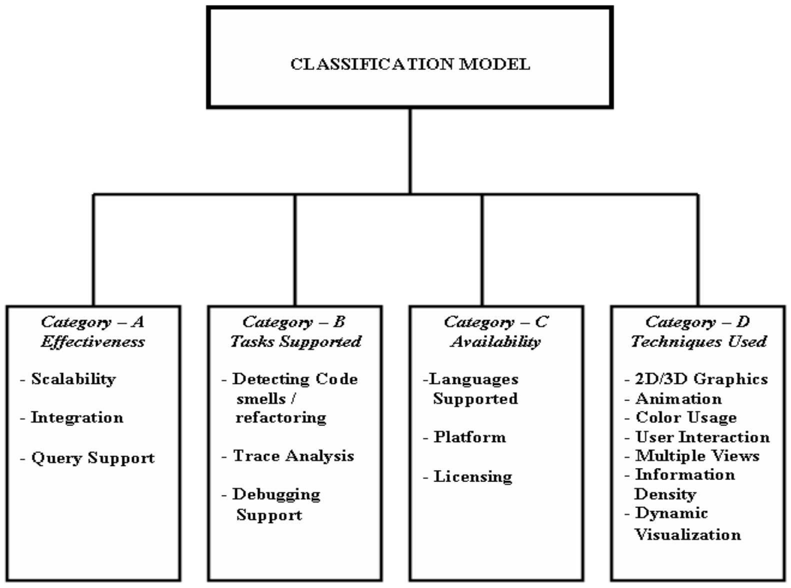


Figure 7.1: Requirements Classification model for SoftVis tools for Corrective Maintenance (CMRC)

In this chapter, we aim to analyze the model's predictive power and completeness, defined as explained above. For this, we proceed as follows:

1. use the model to select three SoftVis tools for CM which fit the model's requirements for a good tool to different degrees
2. select a specific CM task and user group
3. ask the users to address the CM task using the selected SoftVis tools
4. gather quantitative and qualitative data on the effectiveness of the tools to help in solving the task
5. interpret the data to understand the model's completeness aspects and predictive power

For the predictive power analysis, in step 4 above, we measure different variables to assess the tools' usefulness, including the ability to complete the tasks successfully, the duration for task completion, and the actual feedback from the developers concerning the tools' usefulness (Section 7.3.2). Next, we check the model's predictive power by testing how these measurements correlate with the model predictions. 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); and if the tool features they made use of are in line with those claimed to be desirable by the model.

For the completeness analysis, in step 4 above, we gather qualitative data from written and oral feedback given by the study participants after completing the given CM tasks (Section 7.4). The details of the questionnaire used to gather the written feedback are given in Section 7.3.3.

7.1 Study Structure

The structure of our study follows the *explanatory case study* design, as described by Yin [163, 162]. Case studies are the method of choice when the phenomenon under study (in our case: the effectiveness of a SoftVis tool 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: the way developers use SoftVis tools for CM); and when the object of study is a contemporary phenomenon in a real-life context.

Our case study is of the explanatory (causal) type, as we try to analyze the predictive power of our SoftVis tool classification model, or in other words test the hypothesis "if a SoftVis tool fits the model's requirements, then it is highly likely to be a good SoftVis tool". The used methods involve both quantitative elements, *e.g.,* measured variables such as developer experience, ability to complete task, task duration, accuracy, and correctness; and qualitative elements, *e.g.,* the various questions addressed to the users in the post-study phase (Section 7.3.3).

Following the terminology of Yin [162], our study is of a *single-case* design, the case itself 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 (see further Section 7.3). The *theoretical proposition* is that a SoftVis tool which fits our classification model is highly likely to be effective in supporting 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 validates our proposition (the model's predictive power). The *criteria for interpreting the findings* uses the qualitative textual and oral feedback from the users to interpret the hard measurements (time and accuracy of task completion), and thereby elicit the reasons of *why* a certain SoftVis tool, or tool aspect, was good (or not) for addressing a certain CM task.

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

7.2 Tools

We describe now the tools used for the evaluation. The selection process was restricted to SoftVis tools usable for CM which comply with the following:

- score medium to high on most features deemed as desirable by our classification model. These are also among the most important tool usability and effectiveness factors mentioned in the literature;
- are widely available, well known, and mature. This was done so as to remove any potential bias present in some innovative, but not fully operational or mainstream, research-grade

tools;

- target the same programming language, run on the same platform, and integrate with the same IDE (Eclipse). This was done so as to diminish further bias caused by differences in such factors which are not relevant for our hypothesis testing.

Overall, the selection procedure follows the same procedure as described in Chapter 4, Section 4.2.1, and also used in the study in Chapter 6, with the addition that we now only select tools that score relatively high on our CMRC. This is needed as we aim to check the predictive power of the CMRC.

The availability of professional programmers familiar with the chosen target language (Java) and willing to invest time in the study, narrowed the alternatives to three actual tools. The actual scores that the three tools obtained, based on the proposed classification model, are shown in Table 7.1. The scores were given by the authors of this paper, independently on, and in no interaction with, the actual developers who further tested the tools in the given CM task.

	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

	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

Table 7.1: Tools against desirable features of the classification model

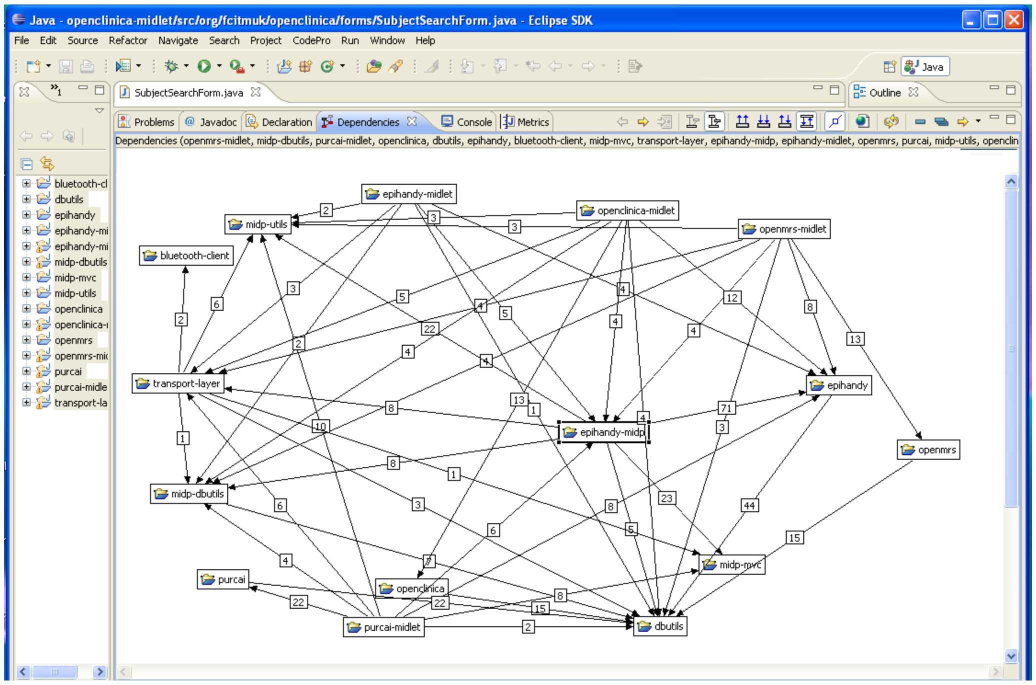

Java - openclinica-midlet/src/org/fcitmuk/openclinica/forms/SubjectSearchForm.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project CodePro Run Window Help
SubjectSearchForm.java
Outline
Problems Javadoc Declaration Dependencies Console Metrics
Dependencies (openmrs-midlet, midp-dbutils, purcai-midlet, openclinica, dbutils, epihandy, bluetooth-client, midp-mvc, transport-layer, epihandy-midp, epihandy-midlet, openmrs, purcai, midp-utils, openclin
bluetooth-cl
dbutils
epihandy
epihandy-mi
epihandy-mi
midp-dbutils
midp-mvc
midp-utils
openclinica
openclinica-
openmrs
openmrs-mi
purcai
purcai-midle
transport-la
epihandy-midlet
openclinica-midlet
openmrs-midlet
midp-utils
bluetooth-client
transport-layer
epihandy
epihandy-midp
openmrs
midp-dbutils
midp-mvc
purcai
openclinica
purcai-midlet
dbutils

Figure 7.2: CodePro Analytix visualization plugin (actual screen snapshot from the tested code base)

The first tool, CodePro Analytix, was selected based on the results of an earlier study presented in Chapter 6. For the other two tools, Ispace and SonarJ, we actually evaluated the tools with respect to the model by using them on several small-scale, example-like, programs. In all cases, the tool classification was *only* performed following the definition of the desirable features as given in Chapter 6, and *not* based on the actual tool suitability for the current CM task (described further in Section 7.3.2). CodePro and SonarJ are commercial tools. Ispace is an open-source tool, but nevertheless shows the same level of maturity, documentation support, ease of installation, and overall ease of use as the other two tools. All three tools are widely known in the software engineering community. All three tools were used on comparable PC machines running Windows XP and Eclipse, to remove further bias.

The three tools are detailed next.

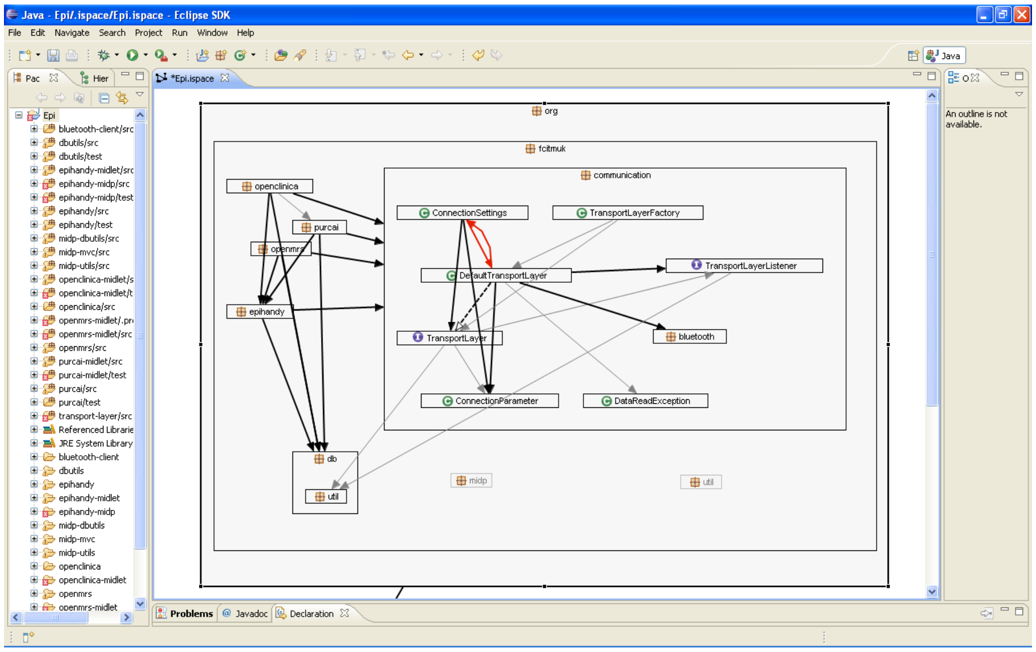


Figure 7.3: Ispace visualization plugin (actual screen snapshot from the tested code base)

7.2.1 CodePro Analytix

This tool plugs into the Eclipse environment and adds several visual functionalities to the Java IDE that can be used during CM [53]. 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. Visualization options include various types of force-directed and hierarchical graph layouts for extracted containment and dependency relations, possibly annotated by metrics, such as the example shown in Figure 7.2.

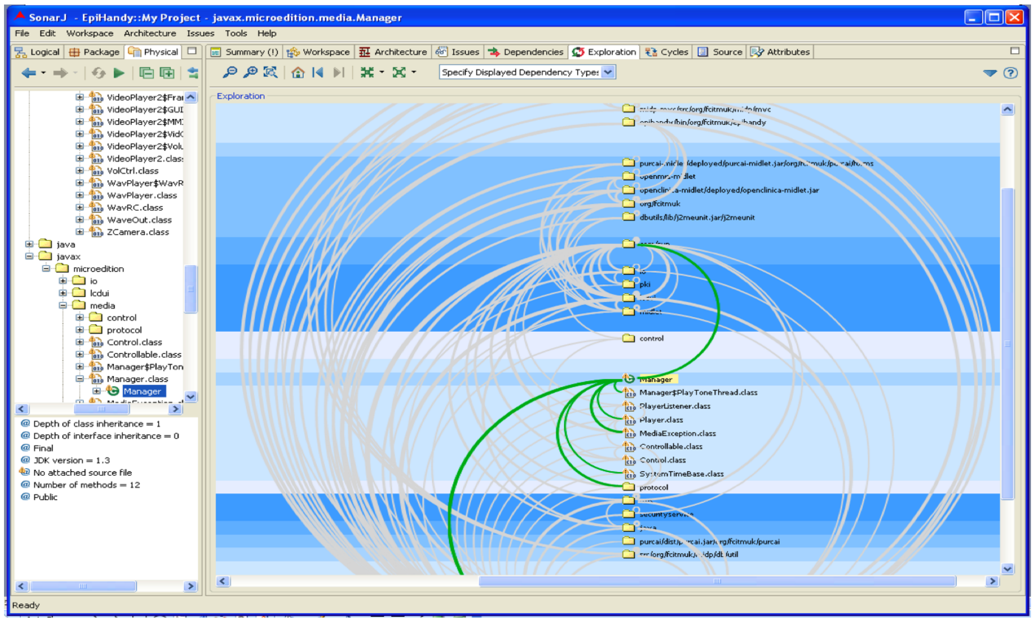

SonarJ - EpiHandy::My Project - javax.microedition.media.Manager
File Edit Workspace Architecture Issues Tools Help
Logical Package Physical
Summary (!) Workspace Architecture Issues Dependencies Exploration Cycles Source Attributes
Specify Displayed Dependency Types
Exploration
Depth of class inheritance = 1
Depth of interface inheritance = 0
Final
JDK version = 1.3
No attached source file
Number of methods = 12
Public
Ready

Figure 7.4: SonarJ visualization plugin (actual screen snapshot from the tested code base)

7.2.2 Ispace

Ispace is also an Eclipse plugin targeted at visualizing Java code [50]. 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 7.3 shows a snapshot of an Ispace view showing containment and dependency relations between classes.

7.2.3 SonarJ

SonarJ is a standalone system that provides functions for navigating the structure, dependencies, and source code of Java programs [44]. 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, but also features an operation mode where it functions integrated with the Eclipse IDE. Figure 7.4 shows a snapshot of SonarJ depicting the visualization of dependencies between the packages of a Java system, with dependencies highlighted based on a user query.

7.2.4 Source Code

The code base used as target for the CM activities of this study was EpiHandyMobile, an extension of EpiHandy, which is a mature Windows Mobile data collection tool [61]. 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).

7.3 Evaluation procedure

7.3.1 Participants

In total, seventeen programmers participated in the SoftVis tools evaluation, distributed in several roles, as follows. First, two developers who had advanced knowledge of the code base (Section 7.2.4) were assigned, one for CodePro, the second for Ispace. 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, fifteen professional developers were assigned to the evaluation, six for CodePro (CP), six for Ispace (IS), and three for SonarJ (SJ), as shown in Table 7.2, explained next. From the six developers in each of the first two groups, one also had prior exposure to the considered code, but was not aware of the existing problem that was to be solved during this experiment (see Section 7.3.2). We included this participant in order to see how experienced developers, with good information about the code, use SoftVis tools in the process of CM.

All participants were sought from software development companies. A pre-study questionnaire was used to ensure that they had all needed skills, *i.e.,* 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 an earlier tool evaluation presented in Chapter 6, participants were exposed to each tool that they had to evaluate for 15 minutes. Over half of those participants, however, felt that a developer needs 3 to 7 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 understood well, we had the subjects use the SoftVis tools on their own code bases during this training phase. Before starting the actual CM task, we confirmed with the developers that they did not have unanswered questions on the SoftVis tools that they had to use.

Furthermore, the participants were clearly told that the authors of this paper were not related in

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# Visual Basic .NET	5 . . . 10	< 5	Eclipse Visual Studio	No	Yes
$CP2$	M	Java PHP Visual Basic	5 . . . 10	< 5	Netbeans	Yes	No
$CP3$	M	Java Python HTML	5 . . . 10	5 . . . 10	none (command line tools)	No	No
$CP4$	M	Java C# Visual Basic	5 . . . 10	< 5	Visual Studio	Yes	No
$CP5$	F	Java	< 5	< 5	Eclipse	Yes	No
$CP6$	M	Java C++ Visual Basic	5 . . . 10	5 . . . 10	Eclipse Visual Studio	No	No
$IS1$	M	Java Visual Basic .NET C#	< 5	< 5	Eclipse	No	Yes
$IS2$	M	Python Java C	5 . . . 10	5 . . . 10	Visual Studio Eclipse	No	No
$IS3$	M	C++ Java C#	5 . . . 10	5 . . . 10	Visual Studio Eclipse	No	No
$IS4$	M	C Java HTML scripting	5 . . . 10	5 . . . 10	Eclipse	No	No
$IS5$	M	Java Visual Basic	< 5	< 5	NetBeans	No	No
$IS6$	M	Java C++	5 . . . 10	5 . . . 10	Eclipse	No	No
$SJ1$	M	Java C, C++ Python	> 10	> 10	Eclipse Visual Studio	Yes	No
$SJ2$	M	Java C++	5 . . . 10	5 . . . 10	Eclipse Visual Studio	Yes	No
$SJ3$	M	Java C++ Python	< 5	< 5	Eclipse Visual Studio	Yes	No

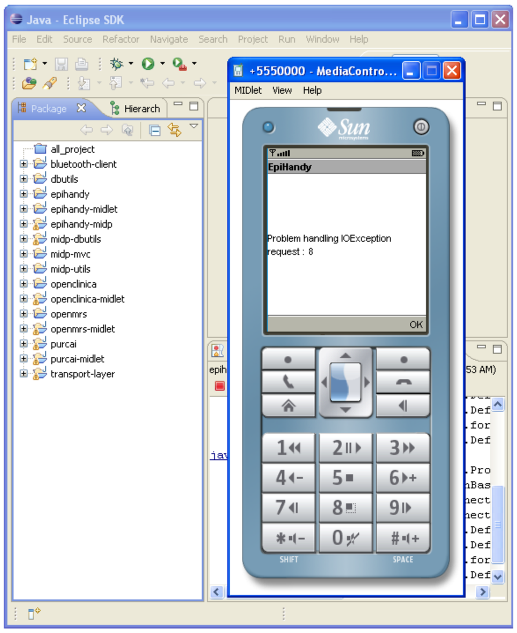
Table 7.2: Programmers evaluating the CodePro Analytix (CP), Ispace (IS) and SonarJ (SJ) visualization tools

any way to the evaluated tools, in order to remove any possible positive bias. Finally, following the recommendations in [122] on improving experiments that involve participants from industry, the participants were paid nominal fees in order to increase both their dedication and lower the risk of drop-out.

7.3.2 Tasks

All three 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 throws an exception. A screenshot of this situation is shown in Figure 7.5.

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 the additional SoftVis tool (Section 7.2). The set-up of this CM



Problem handling IOException
request : 8

Figure 7.5: Application use-case showing a user provoked error

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 was actually locating the code component 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 section of the code 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 Section 7.3.1, the entire process could take a considerable amount of time.

7.3.3 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, as described below. The questionnaire consisted of multiple choice and free-text answers. 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 (select one 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 purchase this or a similar visualization tool that aids software debugging?
Answer (multiple choices possible):

- the tool has 2D graphics
- the tool has 3D graphics
- the tool provides animation
- the tool has a high use of color
- the tool allows users to interact with the visualization
- the tool supports several 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
- 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 discussion in explaining their answers in more detail, and providing any additional feedback, and noted down the comments made.

7.4 Results

The results presented below are compiled from the two questionnaires filled in by the participants, *i.e.,* pre-study (Section 7.3.1) and post-study (Section 7.3.3), as well as the results and timings of the CM tasks, which are presented in Table 7.3.

7.4.1 Background

Results from the pre-study questionnaire showed that most participants had never used a SoftVis tool before despite the fact that several 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. Figure 7.6 shows a pie chart summarizing the users' responses to questions about SoftVis tool usage. These results re-emphasize 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.

7.4.2 Task completion and duration

All users of CodePro and Ispace successfully completed the assigned task, as measured by the two expert developers in each group (see Section 7.3.1). All these users but two explicitly stated that the SoftVis tool was helpful (Table 7.3). Ispace users had lower completion times on the average (Figure 7.8), and Ispace was also perceived as more useful than CodePro (Figure 7.7). Since both user groups did not know any of the tools prior to the study learn phase (Section 7.3.1), we may conclude that Ispace was easier to use than CodePro. This point is detailed further in Sec. 7.4.3 below.

In contrast, none of the three users of SonarJ was able to complete the CM task successfully. The reasons for this outcome, as inferred from the users, are detailed separately in Section 7.4.6. The times reported in Table 7.3 and Figure 7.8 for these users indicate the time until they gave up the task.

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 7.8). Note that completion times include a combination of using the IDE and visualization plug-in for both searching for and correcting the error. Although 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, a qualitative observation of the users' behavior suggests that roughly a third up to half of the total time was related to 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 7.2 vs Figure 7.8): 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	Task completion duration (mins)	Task completion status	Prior code exposure	SoftVis tool was useful
$CP1$	49	Correct	Yes	No
$CP2$	55	Correct	No	Yes
$CP3$	43	Correct	No	No
$CP4$	65	Correct	No	Yes
$CP5$	70	Correct	No	Yes
$CP6$	54	Correct	No	Yes
$IS1$	78	Correct	Yes	Yes
$IS2$	42	Correct	No	Yes
$IS3$	44	Correct	No	Yes
$IS4$	47	Correct	No	Yes
$IS5$	50	Correct	No	Yes
$IS6$	45	Correct	No	Yes
$SJ1$	40	Failed	No	No
$SJ2$	45	Failed	No	No
$SJ3$	60	Failed	No	No

Table 7.3: Post-study questionnaire results for the CodePro Analytix (CP), Ispace (IS) and SonarJ (SJ) tools

7.4.3 Used tool 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.

We noticed that all users of CodePro, Ispace and SonarJ mainly used dependency visualizations to support their CM task. However, the three tools use quite different visualizations for showing dependencies, as follows. Ispace uses the drill-down method which enables both detail and overview. CodePro does show the overall picture but, when requesting details, looses the overview. Furthermore, the actual nested layout implemented by Ispace was perceived as delivering a higher information density than CodePro. However, while both CodePro and Ispace enable users to customize the dependency layout, SonarJ uses a predefined, fixed layout based on a simple grid-based or row-based ordering of the entities, with relations always drawn as arcs (Figure 7.4 exemplifies this grid-based layout). While this layout is reasonably effective in navigating hierarchically layered systems [94], it is less effective, produces more visual clutter, and has a lower visual scalability, than the force-directed layouts used by CodePro and Ispace.

Most of the other visual elements present in the three tools look quite similar. They all provide a folder explorer to navigate the system hierarchy (see Figures 7.2 - 7.4), linked views, including source code views, and very similar graphical icons to show the elements in the visualizations. However, since visualizing dependencies was heavily used by all users, the above-mentioned technical differences in the dependency visualizations contribute in explaining the difference in user performance and completion success (Section 7.4.2). This is an instance of the well-known information visualization principle of combining overviews with zooming and details-on-demand [118]. As one CodePro participant put it

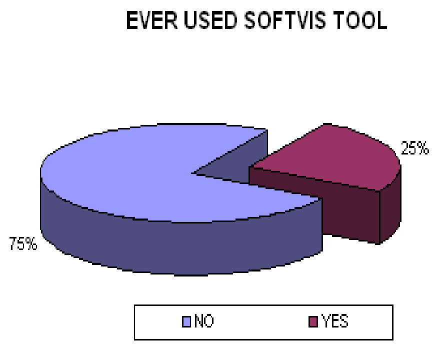


Figure 7.6: Prior exposure to SoftVis tools

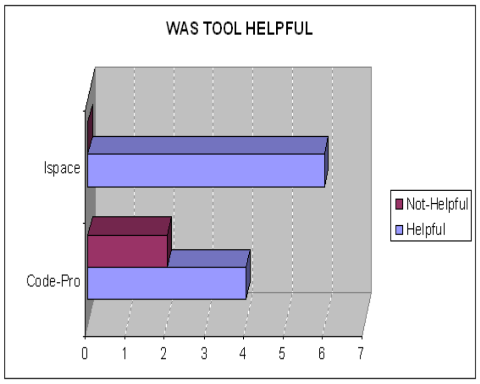


Figure 7.7: Response to question 7.3.3a

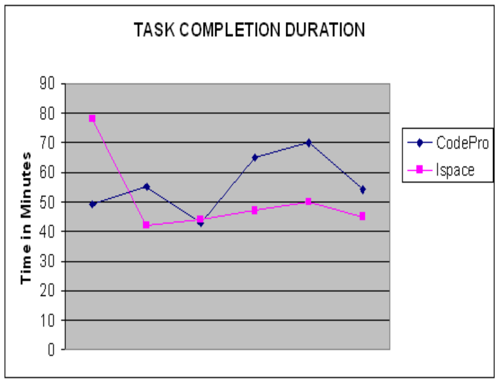


Figure 7.8: Comparison of task completion durations. Users are sorted on cumulative programming and CM experience

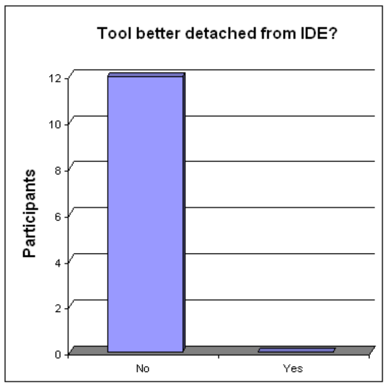


Figure 7.9: Response to question 7.3.3b

"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 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, queriability). 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 (see Chapter 6). 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.

Interestingly, Ispace, which was found to be more useful by its users, is an open-source tool, while CodePro and SonarJ 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 tool immaturity.

7.4.4 Query facilities

Figure 7.10 shows responses on how important the query facility (of the SoftVis tool) was to the participants. While most users found this facility to have been helpful, preciseness was still 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 7.3.2). 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 both the URC and CMRC classification models, that extensive query mechanisms are crucial to SoftVis tools in CM. Yet, one 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 Section 7.5.2 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.

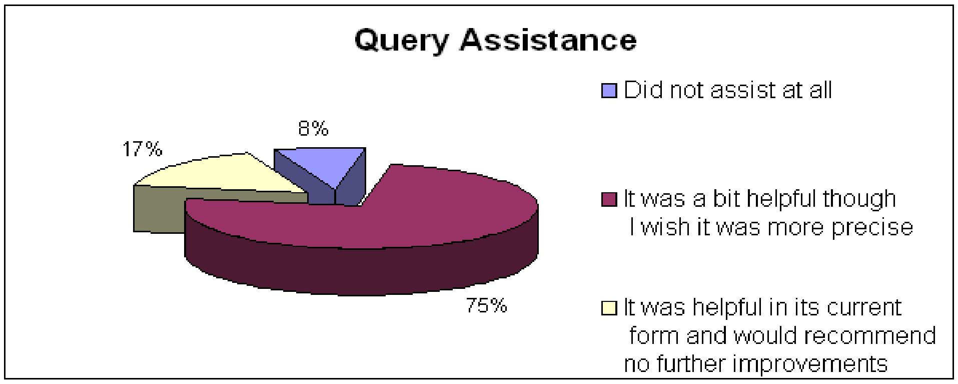


Figure 7.10: Response to question 7.3.3c

7.4.5 IDE integration

All participants strongly agreed that the SoftVis tools would work better attached to the IDE than standalone (Figure 7.9). 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, by far, as having the weakest IDE integration of the three studied tools. Although the tool 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 [141, 69, 129, 20, 116, 138]. 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 [138, 137, 149, 47], 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.

7.4.6 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 interesting point to note here is that the above facts are in line with the classification of SonarJ, performed independently on the actual CM task, user group, and code base (Section 7.2, Table 7.1). That is, the classification model ranks SonarJ as 'weak' on the same desirable features which are mentioned by its actual users.

7.4.7 Purchase factors

Figure 7.11 shows the results to the question on which factors would positively influence a purchase decision for a SoftVis tool for CM. We asked this question separately in order to determine, first of all, whether the CMRC classification model dimensions are perceived, indeed, as relevant by the users (note the similarity between question 7.3.3 and the model's dimensions, detailed in Chapter 6). The following elements were observed:

- *Dynamic visualization:* Over half of the participants felt that dynamic visualization tools would be very helpful, which is also argued for in earlier studies [11, 129, 69]. Note that none of the evaluated tools provided such features (Table 7.1).
- *Show suggestions:* A similar number of participants also wanted tools to be able to dynamically 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 may be of interest to developers of SoftVis tools for CM.
- *3D rendering and animation:* None of the participants felt that 3D rendering or animation would motivate purchase. Again, note that none of the evaluated tools provided such features. We will not insist on this issue, as there is a certain split in the research community between advocates and skeptics of the effectiveness of 3D software visualizations (see *e.g.,* [80, 159, 69, 140]). However, given this aspect, in line with our previous observations (Chapter 6), we suggest adapting the proposed classification model by removing 3D visualization and animation as 'desirable features', as these did not appear, in this study or our previous ones, as being mentioned as highly desirable by users. Animation needs a few clarifications. By animation, we mean in our study the use of tools that animate given *algorithms* at a high-level of abstraction, such as described *e.g.,* in [11]. This is in contrast to dynamic visualizations of running code (mentioned earlier), where live data is extracted from the *runtime* object and visualized. The distinction is potentially important for tool success.

- *Commercial aspect:* The participants were roughly equally divided between being willing to pay for the tool and requesting a free tool. Although this response may be influenced by the fact that our participants were all from the industry, thus more likely to accept fee-based tools than *e.g.,* academic users, this aspect is important as it does not outline the commercial status of a tool as being a major acceptance hurdle.

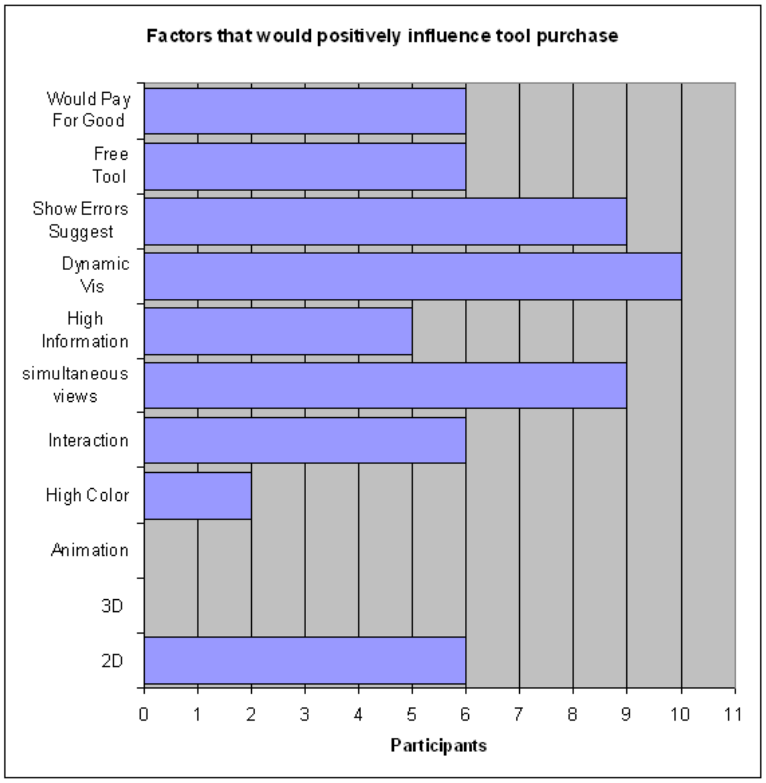


Figure 7.11: Response to question 7.3.3d

7.4.8 Model validation and refinement

We aimed to check the predictive abilities and completeness of the CMRC model proposed in Chapter 6. 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 7.3.3a and Figure 7.7). All participants using these tools completed the given task successfully. 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 7.11 vs the model's *dynamic visualization* and *debugging* dimensions (Table 7.1). The participants using SonarJ, which scored lower on all three effectiveness dimensions and three of the visual techniques used (Table 7.1) was not effective in supporting the given CM task. Furthermore, the textual and verbal comments on SonarJ indicated limitations mainly along these dimensions (Section 7.4.6).

Although not exhaustive, the information above suggests that the dimensions of the proposed CMRC classification model are, indeed, relevant for selecting effective SoftVis tools for CM. In other words, from the answers to questions 7.3.3b and 7.3.3d, as well as to the free-text comments (7.3.3e and 7.3.3f, 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 model 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 [107, 64, 134, 133, 82].

7.4.9 Threats to validity

We acknowledge several threats to validity concerning our study. Following *e.g.,* Huberman and Miles [48] and Trochim and Donnely [146], we can classify these as follows:

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 7.3.3a, 7.3.3c and 7.3.3f). Moreover, in the five cases when the tool was not helpful, this was explicitly reported and justified by the users (see also Table 7.3). 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 (Section 7.3.1). 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 7.2).

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 Section 7.4.8. 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 (Section 7.3.1). Reactive effects to experimental arrangements were minimized by letting the users work individually in their own familiar environment.

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. 7.4.8, 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 7.3.3e and 7.3.3f.

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 [82, 129, 141, 69, 133, 134]. 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 is limited. We tried to reduce, though not eliminate, this by designing questions that measure ‘usefulness’ in different ways, *e.g.,* question 7.3.3a (direct one), 7.3.3d (willing to buy a tool is another usefulness measure), 7.3.3e (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 workplaces at their own companies), 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. Concerning the second point, the programmers involved in this study should arguably not be strongly confused by the presence of a silent observer, given that they are are used to pair programming techniques.

Questionnaire design

A separate potential threat to validity is related to the design of the post-study questionnaire (Sec. 7.3.3). For instance, following the funneling method described by Oppenheim [97], the order of the questions should start with broader questions and then narrow down to more specific questions, one of which is actually question 7.3.3a. 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 7.3.3b towards answering it negatively.

Following Yin [163, 162], 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 very 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 7.5).

7.5 Additional points

In this section, we discuss additional points related to the organization and outcome of the presented user study. For a detailed description of the conditions affecting the set-up of case studies involving SoftVis tools, we refer to Chapter 8.

7.5.1 Motivating participants

Although the participants were motivated using the fee payment, it was still 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 present the results in the questionnaire. This was clearly a non-trivial task, which required from the participants as much attention as they would pay 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. Among the mechanisms used to increase the study's success were allowing negotiations for higher fees than initially planned and accepting to carry out the study at their workplace at a time that was convenient for them. Had there been no payments, it would have been almost impossible to get any one of this specific developer group to dedicate enough time and attention to do the study. This point on compensations may be relevant when planning other studies of tools in industrial contexts, as also mentioned in [122].

7.5.2 Adoption by experts

As mentioned earlier, there was a code expert in each of the CodePro and Ispace groups (Section 7.3.1). Interestingly, the expert in the Ispace group found the tool helpful in carrying out the task, while the expert in the CodePro group, who had the highest code exposure, did not find the tool helpful and eventually fixed the bug using only the source code and the IDE's text editing options. 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 Section 7.4.2. Despite the fact that a programmer is familiar with the code at hand, 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 task-specific features into such SoftVis tools. Any wrong step with this group would lead to major adoption hurdles.

7.5.3 Evaluation constraints

As a general observation, the quality and generality of the results of such studies on tool effectiveness and acceptance 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 [122]. Concerning this essential point, it is noted that the role of human subjects in empirical evaluations is sometimes taken too lightly [144, 129]. Similar observations are made by Koschke when distinguishing research and end-user contexts when evaluating future directions of software visualization [69]. 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 [15, 28, 144]. 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.

7.6 Conclusion

In this chapter, we have presented an evaluation of three 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 the classification model proposed in Chapter 6 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 desirable features identified by the model seem strongly 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, and that there are no other hidden context variables which affect the results.

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 and customizable dependency visualizations. These are in line with the main factors of SoftVis tool acceptance in the industry, independently identified by different types of studies [107, 13]. Additional desirable features 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" [69]. This, together with the high costs associated to user studies in the industry, which limit the insight we have into what this user group finds really useful in SoftVis tools, may be one of the primary causes of the gap between the SoftVis research and its industrial acceptance.

As a last comment, there is still a section 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 need to debug.

In the next chapter, we will summarize the lessons learned over the period of evaluating SoftVis tools.

Chapter 8

Lessons Learned

In Chapters 3-7, we presented several user studies involving software visualization tools. One common aim of all of these studies has been to elicit and analyze features of such tools that are perceived as useful for software understanding tasks. This insight has been used to build and refine a classification model of such features (Chapters 4 - 6) and, further, test the completeness and predictive power of this model for SoftVis tools in corrective maintenance (Chapter 7).

During these studies, we have made several observations related to the set-up and execution of user studies involving software visualization tools. This chapter discusses the lessons learned during this process, and outlines recommendations for the execution of such further studies. Here, we focus on the methodology of such a study, and not (the interpretation of) its results. The latter have been discussed separately in the previous chapters.

8.1 Introduction

The set-up and execution of user studies involving software visualization tools is an arduous process. A successful study is influenced by two types of factors. First, there are specific constraints that pertain to the specific nature of a given study, *e.g.,* the way to measure task completeness or accuracy for a study that involves a given SoftVis tool for the execution of a precise, given, software engineering tasks. We shall not discuss such factors here, as they are specific to the concrete context of the task and tool under study. A second type of factors, however, is arguably applicable to most user studies involving SoftVis tools. In the following, we shall discuss the lessons learned during our studies presented earlier in this thesis that pertain to such general factors.

The studies performed in this thesis involved, in increasing level of specificity, a general set of SoftVis tools (Chapters 4 and 5), and SoftVis tools focusing on corrective maintenance (Chapters 6 and 7). The users were professional programmers involved in software engineering in general, and in corrective maintenance in particular. All these studies were related to eliciting, refining, and measuring the so-called desirable features that a SoftVis tool offers to a developer involved in a given set of software maintenance activities.

As such, the generally applicable lessons learned in these studies address factors that can influence, positively or negatively, the execution of a study involving SoftVis tools and users from the computer industry, with a focus on maintenance. We group these factors into ten classes: Tool selection, participants, tool exposure, task selection, experiment duration, experiment location, experiment input, participants motivation, relation of evaluators with the tools, and analysis of results. A schematic overview of the typical experimental workflow, including the above steps, is given in Figure 8.1.

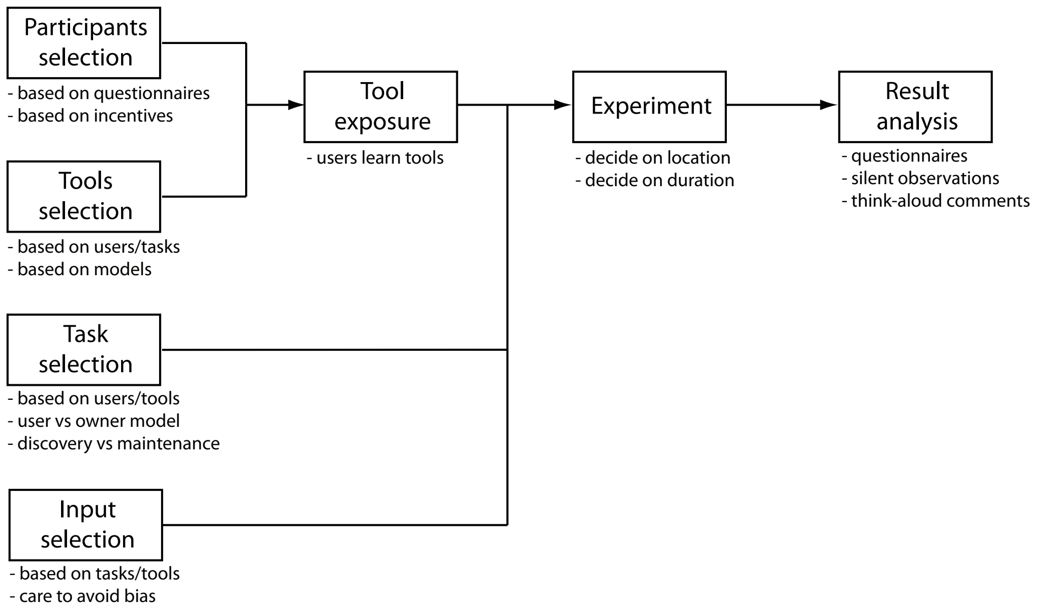


Figure 8.1: Workflow of setting up and executing user experiments involving SoftVis tools

The lessons learned that involve each of the above-mentioned class of factors during execution of SoftVis user studies are described and discussed next.

8.2 Tool selection

When carrying out evaluations, the first step is usually deciding which SoftVis tools to evaluate. It is important that the tool selection is done with a clearly predefined motive, as it may not be possible to evaluate all the SoftVis tools that exist. Different types of motives will determine different styles, and set-ups, of the evaluation process, as follows.

Evaluating techniques: If the motive is to evaluate techniques, like in *e.g.* [136, 47], then tools that use each of the techniques under evaluation may need to be chosen and later measured against each other. The data collection should also focus on the techniques themselves rather than the overall effectiveness of the tool, *e.g.* by means of questions specifically targeting the former. Indeed, a tool may use an otherwise perfect technique, but in such a way that the task aimed to be solved is not addressed effectively. Conversely, a tool may be very effective in addressing a task, but not because a certain technique is used.

Evaluating effectiveness: If the motive is to evaluate the effectiveness of a SoftVis tool for a software engineering task, strictly looking at the visualization tool is not enough in general. The effectiveness of such a tool is strongly influenced by the amount of integration thereof in an entire workflow, which implies evaluating the communication with other established tools [69]. In this case, we noticed that it only makes sense to evaluate tools that have comparable amounts of integration within the same workflow and toolset, as developers experience using weakly-integrated tools as highly disruptive (see Chapters 4 and 7).

Comparative tool evaluation Finally, when comparing functionally identical tools, with the aim of selecting a 'winner', the focus should be on the tool itself and less on the integration aspects. While this is feasible, with some effort, the insights gained from such an evaluation are naturally limited to the set of tools being compared. Ideally, comparative evaluations should use the same subjects in evaluating different tools. However, if the task to be done has an important program comprehension element, prior learning effects are hard to avoid in such a case.

Here, we also noticed the important impact the tool audience has. It is risky to compare tools whose target audience differs, *e.g.,* tools targeted towards novice users such as Jeliot [60] against tools aimed at professional developers such as CodeProAnalytix [53]. From the reactions of the involved participants in our four studies described in the previous four chapters, we noticed that this, first and foremost, causes confusion for the users themselves, as they have trouble positioning themselves in a clear way against the tool (as novices, or professionals, respectively).

8.3 Participants selection

While many people may volunteer for a tool evaluation, it is essential to get a pre-study screening procedure that is in line with the objectives of the study. This ensures that participants that will be indeed helpful to the evaluation at hand are selected. For a counter-example, in a related study on visualizing annotations on UML diagrams, a wide range of users were involved [18]. These included fresh graduates, PhD students, seasoned researchers, and professional developers with over 10 years of experience. After that study, results had to be post-filtered based on the developer experience, as several results delivered by the inexperienced developers were suboptimal (due to limited knowledge) and would have biased the study's results. In chapter 7, studying SoftVis tool integration with IDEs such as Visual Studio was important, so accepting participants with limited knowledge of that IDE would greatly affect the results of the study. Precious time would be spent trying to bring the participants knowledge to an acceptable level instead of carrying out the core experiment.

Pre-selection can be done with the aid of a questionnaire that asks about the knowledge of the willing participants as was done in Chapters 6 and 7. We found this method better than pre-selection based on the professional level (years of experience in the field). People with identical amounts of professional training years can have widely different skills, as it was evident in a related study on software evolution [136].

Finally, it is very important to use tool evaluators that are similar, or identical, to the final target group of the tool. It may not be appropriate to use students for a tool that is targeting industrial practitioners, as they may not have the background to provide useful feedback [35, 18]. For this reason, our evaluations described in the earlier chapters involved only industrial participants. However, in other cases, the tasks to be completed are comparatively simpler, and thus easier to understand even by users with limited experience, such as the earlier cited software evolution study [136]. If developers with a wide experience-level spread are involved, a post-study classification, or weighting, of the results based on the experience, is a good correction factor [18].

8.4 Tool exposure

To be able to get the most out of the experiment, the participants should be allowed sufficient time to study and understand the SoftVis tools that they are going to use. In our experiment carried out in chapter 6 the participants were given 15 minutes to study the tools before carrying out their tasks. More than half of the evaluators later complained that the tool exposure duration was too short and advised longer exposure durations days before the actual evaluation, a point also noted by Plaisant *et al.* in a different context [105]. The learning phase does not need to be contiguous, but has to be of sufficient duration. A learning phase of at least a few hours, spread over maximally a week, seemed to be sufficient for the types of experiments carried out in this work. Spreading the learning phase is also helpful for highly experienced IT personnel that have very busy schedules. They require flexibility in order to learn the tool at their own time without unnecessary pressure.

Novices may even need a longer tool learning phase. This was also earlier observed by Marcus *et al.* [82] who had students perform poorly during the evaluations due to their low tool exposure duration. The advantage with students, however, is that they are more willing to ask when they do not understand and are more comfortable with slightly longer training durations. In [136], the learning phase for the involved over 45 students was of roughly four full days, spread over a period of 6..8 working days. Overall, it is not advisable to expect the participants to learn the tool just minutes before the experiment. This is very hard and does not also reflect the real-life scenarios.

However, there are exception to the need of extensive learning periods. First, there are cases when the involved tasks are simple and thus involve short durations for learning the tools, like in [18, 136], where the learning phase was approximately 20-30 minutes, due. Secondly, if the evaluation aims precisely at understanding *early adoption* issues, *i.e.* the way in which users quickly decide to further study (or discard) a new SoftVis tool after a quick scan, then extensive learning periods should actually be avoided, since we are interested in understanding the users' first impressions. This was the approach taken in Chapter 5.

Overall, we strongly encourage on reporting the learning phase duration in publications involving user studies, as an added measure of the confidence in the evaluation's results and a way to compare different evaluations in meta-studies.

8.5 Task selection

Tasks are an essential part of tool evaluations. There are many ways in which these tasks can be selected. Regardless of the method used, however, the tasks should be reflective of the scenario being simulated in order for the evaluation to be helpful. An example would be a case where a tool is being measured for its ability to answer software maintenance questions. We see several axes along which task selection can be carried out, thereby influencing the purpose of evaluation, as follows.

Task author: users vs owners The tasks can be generated by the tool user or tool owner. By the owner, we mean here the person that is interested in the evaluation results, be it the tool builder or a third party like a researcher interested in studying tools usability. If tasks are generated by the users, *e.g.,* software maintainers, then these will naturally include questions that they are usually faced with during their work. These tasks/questions can be given to additional tool participants along with the tools and then observed to see if they can indeed answer them, as long as the participants share the same working environment and goals as the original task author. This was the scenario taken in [136], where we pre-selected the tasks from earlier discussions with KDE developers [151]. The chief advantage of this method is that a positive evaluation is a very strong signal in terms of tool usability and/or effectiveness.

Alternatively, tasks can be generated by the tool owners. When this method is used, it is advisable for either these tasks or their solutions to be validated by the domain experts, as done *e.g.,* in [151]. Failure to do this may pose the risk of generating tasks that are either irrelevant, too simple or too difficult for the target participants, or biased to reflect the tool under evaluation. This may in turn affect the evaluation by reducing its ability to reflect real life scenarios. This second type of task generation seems predominant in many research papers where authors aim to gather a-posteriori evidence for the design choices of their proposed tools.

Task type: discovery vs maintenance There are numerous types of tasks in software engineering where SoftVis tools can help, and thereby many ways to classify SoftVis tools [29], Within our tool evaluations, we have found a marked difference between tasks involving program discovery and program maintenance. By program discovery, we denote the subset of comprehension activities that enable one to get familiar with an unknown code base. We have noticed at several occasions that programmers that maintain their own code, usually for long periods of time, have much less need for tools that support generic discovery activities such as *e.g.* showing the overall structure of the software or presenting evolution trends. The needs here go towards detailed support for precise, fine-grained activities like debugging, refactoring or optimization.

In contrast, tools that support discovery activities have a somewhat different aim: enable the user to get familiar with a wide range of aspects of a given system. Setting up evaluations for tools in the two classes (discovery and maintenance) is also different. For discovery, it is harder to quantify the effectiveness of a tool (how can one measure if a tool is effective in discovering if it is not known what one is looking for?) For maintenance, it is easier to measure a tool's effectiveness, as the tasks are more precise, so one can quantify *e.g.,* the duration or precision of a task's completion using a given tool. We also noticed that the above two task selection dimensions seem to correlate with two types of participants: professionals are interested in defining the tasks to be supported and focus more on maintenance and less on discovery; tools supporting discovery and tasks generated by tool owners are mostly evaluated in academic and research environments.

We believe, although we do not have hard evidence, that there is a direct correlation between the task selection (when defining tool evaluations) and the perceived value of the tool evaluation. In terms of the lean development philosophy, the *value* of a SoftVis tool evaluation would be different for industrial users and academic groups [106]. For industrial users, the value is in obtaining measurable improvement in supporting an ongoing software engineering activity, *e.g.* reducing costs, increasing quality. For academic groups, the value is often in obtaining early evidence supporting a novel tool design. The two notions of value should meet (a tool is valuable when it measurably supports a valuable task).

8.6 Experiment duration

It has been advised in the past that tools be studied over long periods of time, *e.g.,* months, in order to fully assess their capabilities [119]. From our studies, however, we have noticed that there are certain durations beyond which most tool participants become reluctant to continue as the benefit of carrying on with the evaluation becomes less obvious for them. This is manifested during the process of recruiting evaluators, with many asking upfront about the duration of the experiment. We have noticed, for example, that expert programmers or industrial participants are not comfortable with very long durations [18]. For our studies, 2-3 hour experiments, excluding the time taken to learn the tool, were generally felt as appropriate by this group. Researchers in the past who have worked with students have managed to use longer experiment durations. For example, Lattu *et al.* were able to train a first group of students for 52 hours and a second group for 12 hours as part of a tool evaluation [74]. This training was done in form of introductory programming courses both at university and high school. In [136], the total study duration for the targeted evolution visualization tool was 2-3 weeks per participant. Similar results are shown by Haaster *et al.* [147]. Overall, experiments involving only students can be set up in line with the students' course contents, so longer durations are possible. This is harder, or even unachievable, for industrial participants.

8.7 Experiment location

One of the factors that can affect a tool evaluation is the location of the experiment. In our previous studies presented in Chapters 4 and 5, we have had both lab-based experiments as well as 'mobile' evaluations that could be taken to the evaluators' locations. We have observed that, when working with experts, preference is given to perform studies in their workplace. The inconvenience on the participant is reduced as they can continue with their usual work immediately after the experiment. In order to entice industrial users to participate in rigid experiment set-ups located in labs, the incentives given to them may need to be higher than for the in-place set-ups.

In contrast, previous researchers who have worked with students have used university premises [75, 147, 74, 136]. This is mainly due to the nature of the experiment setups. Finally, in mixed academic-research studies such as [18], using the project meetings to schedule the experiments proved very convenient. All in all, we hypothesize that locations that provide the least bother, and least time consumption, for the participants are the ideal ones for such tool evaluations.

8.8 Experiment input

Depending on the motive of the evaluation, there are many ways in which the actual experiment can be carried out. These may include measuring a tool's ability to solve a program comprehension task or comparing several SoftVis tools. In all our experiments, the input was a given system's source code. When source code is analyzed, this code should be reflective of the tool's targets in order for realistic results to be achieved. As Hundhausen *et al.* noted, however, there are tools developed whose authors are not sure of the targeted group, a phenomenon termed as system roulette [49]. When evaluating such systems, it can be challenging to decide the type of code to be used for the input. Regardless of this, care should be taken to ensure that the code selected does not bias the experiment in any way. Examples of this bias include using code that some participants have prior knowledge of, experimenting with very simple code for a tool that should be targeting large scale code, or conversely.

Our studies described in Chapters 4 and 5 all targeted small-to-middle size code bases (under 10 KLOC), while the repository evolution study in [151] targeted large repositories of millions of LOC. As such, issues such as optimization, speed of processing, and stability were mentioned as the most important usability factors by the subjects involved in the repository study, whereas these were less prominently mentioned by the users involved in the other studies we performed, given the much smaller size of the input datasets.

8.9 Participants motivation

Motivation is an important element of SoftVis tools evaluations and should thus be planned and taken into account for when organizing such a study. Depending on the group that one is working with, different forms of motivation may be used.

When working with professionals, one should keep in mind that this group already holds jobs that are paying them on an hourly or monthly basis. As such, motivation may need to be equated to tangible benefits at the study's outcome. These may be in terms of learning to use a tool that will provide measurable benefits in the regular work activities after the study is completed, or possibly also financial incentives. In our studies, we tried both types of motivation, and we cannot say that one motivation type is definitely more successful than the other one.

Within industry professional, there exists a slightly different sub-group. This includes PhD holders who are in industry, as well as academic staff that also double as consultants or developers. Some of our industrial participants in [18] were in this category. This group is at times willing to take part in studies for the sake of gaining knowledge and may require less or no additional motivation. However, for all industrial users, whether having a research affinity or not, we noticed that a clear added value for the participants must be present in the study set-up to motivate them to take part.

Motivation for students differs considerably. In previous experiments, students were motivated using extra credits for their course project [82]. Experiments which were structured in form of course work did not have major motivation hurdles, as the students had to complete the course as part of their programs [136]. However, from our experience, we noticed that this 'implicit' motivation of students does also usually imply a less critical attitude towards the tools involved in the study, as they do not identify themselves strongly with future tool users.

Regarding motivation, an essential point in doing evaluations of software visualization tools, or other software engineering tools for that matter, regards correlating the tool's provisions with the users' needs. It may sound obvious that any tool evaluation will be validated in measuring how well a tool actually satisfied a concrete need of a concrete user. However, to be able to quantify that, the users should have some concrete stakeholding in the analyzed software. This correlates with the above user categories: Professionals would rarely give a truly positive evaluation of a tool unless that solves problems on their own software. One may think that to be less true for students. In previous experiments where both categories of participants were involved, different results were observed. Professionals used the evaluated visualization tools on software they were actively working on, and expressed clearly that tool usability has to be proved on that software, not a third-party one (see Chapter 5). In a SoftVis tool study related in aim and structure to the work in this thesis, a mixed population was used: Some of the participants were active KDE developers (so knew the software under study), whereas the others were not familiar with the visualized systems [151]. The participants who were actively involved in development were, overall, more critical with respect to the studied tool's usefulness than the unfamiliar participants, who focused most on general usability features of the tool.

Overall, we believe that users familiar with a code base will be significantly more strict when evaluating a visualization tool than users unfamiliar with the code. The former ones have very specific questions and wishes *and* also already have prior knowledge of the code, so they find less value in tools that produce general-level facts. This opens the question: Is it, then, meaningful to let users evaluate a SoftVis tool on input code in which they have no explicit interest? From our experience, the answer is that this is possible, but mainly when the SoftVis tool addresses general program comprehension tasks. In that case, it makes sense to evaluate the tool on 'unknown' code bases. However, if the tool's claims are of different nature, *e.g.,* support maintenance, refactoring, or assessing code quality, for example, then it is much better for the tool to be evaluated by users holding some type of stake in, thus familiar with, the tool's input.

8.10 Evaluators relationship with the tools

In many cases, the tool developers are in a better position to evaluate their tools since they are very familiar with it, as it was the case in *e.g.,* [151]. The learning curve is practically zero in such cases, and there is high confidence in the quality of the results obtained. However, there are some dangers with this route. Apart from the problem of bias from the evaluators side, if participants know that the evaluator developed the tool, they may tend to be generous with compliments while minimizing criticism. We noticed this effect relatively strongly in [151]. This, in turn, can create false positives about either the technique or the tool being looked at thus reducing its chances of being improved.

In order to get the most objective results, its advisable for the evaluator to be as detached as possible from the tool being evaluated. This can be done by letting a different person supervise the evaluation of the tool or not informing the participants who developed the technique or tool under evaluation. This was the route taken in [18]. In that study, we noticed no difference in the qualitative output of the participants between participants who knew the (non-disclosed) developers of the studied technique and those who had no relation whatsoever with the developers. In other studies involving student populations [136], the tool under evaluation was originally developed by the main course lecturer. To eliminate bias, a slightly different version of the tool was used, and presented under a different name and provided it from a third party (web site). Although it was possible for the participants to trace the connection of the tool and the course lecturer, and thus generate positive bias, this did not happen. All 45 student reports, with no single exception, contained clearly critical observations on the ineffectiveness of the evaluated tool with respect to certain tasks.

Another important element in student evaluations was found to be the decoupling of the evaluation itself from the success of completing the task. From previous tool evaluations done with student populations, it was found that students are either positively or negatively biased when the assignments goal is the completion of the task, depending on the student's success in completing, or failing to complete, that task. In [136], it was found that this bias can be eliminated by structuring the assignment in terms of describing the results of a number of actions done with the tool, and asking the users to comment on their findings (whatever those are), rather than stressing on obtaining some results with the tool, and asking the students to describe those results.

8.11 Analysis of results

As a final point learned during the tool evaluation studies we performed, there is a lot of data that a study can generate, and that the evaluator has to interpret. In order for this evaluation to be beneficial to a potential adopter of the evaluated tools, or to the developers of the tools, the results should be analyzed in direct relation to the objective of the study (the task). If visualization *techniques* are the ones being analyzed, the results should clearly indicate which of the analyzed techniques is better than the other and also offer potential reasons why, see *e.g.* [47]. This is very important as many practitioners are only concerned about the results as opposed to the procedure itself.

If, however, the evaluation's aims include understanding whether (and how) a tool is effective in supporting a given software engineering *task*, other issues become prominent. The relation of a visualization tool and software engineering task is rarely a direct one, the tool being effective for that task only as part of a tightly integrated toolset and workflow, as outlined already in Section 8.2. In that case, we see no other reliable and generic solution than to spend the added effort to achieve a high level of integration prior to the evaluation. This is the road that was taken, for example, in [136], where the repository visualization tool is tightly integrated with the software configuration management (SCM) system used (CVS or Subversion) and also with several software quality metric tools. A similar route was taken in [17, 18], where the targeted UML visualization tool can be used as a drop-in replacement for other similar UML tools such as Poseidon.

8.12 Limitations

We do not, in any way, suggest that the evaluations carried out in this thesis were perfect. As outlined at several instances our evaluations have meaningful, extrapolable, results only within specific conditions. All in all, were evaluated just 20 SoftVis tools, and involved only around 30 users. However, in presenting the lessons learned in this chapter, we limit ourselves to the common denominator over which strong consensus from nearly all participants and evaluations existed. As such, we believe these points to be important, and valid, for a wide range of evaluations of SoftVis tools in general.

In this work, the only tool evaluations that we could study in detail, were the ones in which we were directly involved as evaluators. This is, on the one hand, hard to avoid, as it is very difficult to be aware of all the preconditions and details of a tool evaluation process done by a third party, if these are not all explicitly reported in the respective publications. Also, the number of SoftVis tool evaluations which are comparable in the sense mentioned in share comparable aims, tasks, types of users, learning curves, and experiment durations, is relatively small. To strengthen the lessons learned mentioned in this chapter, it would be important to further search for such studies in the literature, and enhance (or possibly invalidate) the conclusions drawn here based on such additional evidence.

8.13 Summary

In this chapter we presented a number of lessons learned that target the context of organizing evaluations of software visualization (SoftVis) tools. We distilled several dimensions which are important to consider both when organizing a tool evaluation, and also when interpreting the results of such a study. These dimensions cover areas ranging from tool and task selection, choosing and training of participants, and analyzing the results from the evaluation.

Future research may include showing a different perspective of the evaluations in order to present the respective lessons learned. This can include areas such as lessons learned in evaluating SoftVis tools for software evolution, software maintenance as well as education in software engineering. By analyzing tool evaluations for more specific, narrower, areas, more specific criteria that influence such evaluations can be elicited, thereby helping the organization and comparison of such tool evaluations in the future.

Chapter 9

Conclusions

The main aim of this thesis was to develop a framework that shows the correlation between desired features of SoftVis tools in relation to software maintenance and the existing facilities provided by such the tools. The purpose of such a framework is twofold. First, it can serve to catalogue and classify existing SoftVis tools with respect to features perceived as desirable by their targeted users so it helps users in choosing a SoftVis tool for a concrete context. Secondly, it can serve in motivating tool developers in building increasingly suitable tools for a given target audience. Our target user group involves professional developers working on software maintenance in the industry.

We have approached the above aim of developing this framework in a top-down fashion. First, a literature study was performed in order to extract several types of requirements (or desirable features) perceived as being important for SoftVis tools in general (Chapter 2), as well as different types of methodologies involved in user studies of SoftVis tools. In this respect, numerous studies exist, ranging from taxonomies of SoftVis tools in general or according to a specific field up to evaluations of concrete SoftVis tools for concrete tasks. From our literature study, the general conclusion that emerges is that there does not yet exist a comprehensive framework that captures and classifies software visualization tools in a way that makes it easy to link, in a straightforward way, the features offered by these tools to the requirements deemed to be important by their users. More importantly for the context of this thesis, there does not exist a general framework that captures and classifies features of SoftVis tools perceived as desirable by their main target users, the software engineers.

The second step of our work involved the creation of a unified requirements framework (URC) that describes the SoftVis tool features perceived as important, or desirable, by typical industrial users, during the so-called *early adoption* phase, *i.e.* the short amount of time during which users explore a new SoftVis tool and decide to further learn it in more depth, or to reject it (Chapters 4 and 5). The creation of this framework has involved two studies in which three, respectively ten, SoftVis tools that target various aspects of software maintenance were involved. The first study was exploratory in nature, aiming at understanding which are the challenges involved in early adoption evaluation, and which are typical desirable features that users can identify at this stage. This insight was refined in the second study, after which the URC model emerged. The model captures six main classes of desirable features: search functions, meta-data display, simplicity, integration, perceived added advantage, and realism.

The third step of our work involved studying desirable features of SoftVis tools for a more specific area of software engineering: corrective maintenance (Chapter 6). This corresponds to the features that users identify as desirable after a given tool has passed the early adoption stage, and is considered for actual use in support of a concrete task. In this context, we presented a Corrective Maintenance Requirements Classification (CMRC) for SoftVis tools. The CMRC refines and extends the URC for the early adoption phase with subsequent features that relate to specific tasks in corrective maintenance. The CMRC is structured around four main categories of desirable features, targeting effectiveness, tasks supported, availability, and techniques used. As expected, these categories are more specific than the ones presented by the URC for the early adoption phase.

The choice for corrective maintenance as an area of interest was motivated by the predominant amount of effort that is spent in this activity in the software industry, the large amount of persons involved in corrective maintenance, as opposed *e.g.,* to persons involved in design or other forms of maintenance, and the relatively small number of studies in the literature that discuss SoftVis tools for corrective maintenance, as opposed *e.g.,* to studies that discuss SoftVis tools for program understanding in general.

The fourth and last step of our work involved the testing of the completeness and predictive powers of our refined CMRC model for desirable features of SoftVis tools for corrective maintenance (Chapter 7). This step aimed to validate the initial claims of usefulness of such a model, namely that it can be used to select the suitability of a tool for a given context based on the tool's scores with respect of the URC model features. For this, we carried out a case study in which a concrete corrective maintenance problem had to be solved on a given code base with three different tools. Two of these tools have high scores for the CMRC model requirements, whereas the third tool has lower scores. All three tools have similar scores from the perspective of desirable features identifiable at an early adoption stage, so they could all be potentially selected by developers. The study performed outlined that the effectiveness of the three tools is in line with their scores on the CMRC, and that the perceived limitations of the tools for the task at hand are correlated with their limitations in covering desirable aspects of the CMRC.

Throughout this work, several user studies were carried out, involving over 30 software professionals and over 20 SoftVis tools. The lessons learned with respect to methodological aspects in carrying our such user studies are compiled and summarized in Chapter 8. From these studies, several general observations were made. First and foremost, there seems to exist a large gap between the desirable features needed by software professionals involved in the corrective maintenance industry and the features provided by many of the widely available SoftVis tools. This gap involves features such as toolset integration, ease of use, query support, and the support of dynamic visualizations. On the other hand, SoftVis tools excel in providing a large number of visual metaphors, or representations, for visualizing the data at hand. Although unsupported by further evidence, we believe that this situation is typical for the research stage in which many of the SoftVis tools currently exist: Many provide novel visualization paradigms, but relatively few take the further implementation-level steps of becoming fully operational products in an industrial context.

The above observations relate to the future of software visualization, and specifically the steps needed for SoftVis tools to make the transition to widely accepted (and used) instruments in the software industry. Several conclusions can be drawn here. First, a refinement of the work in this thesis in the direction of bringing additional evidence that emphasizes and details the differences between desirable features and offered features of SoftVis tools would be of important added value. Such a refinement could support our claims that more attention is needed for features such as integration, query support, ease of use, and dynamic visualizations. A second future work direction would be in refining and extending the CMRC model proposed here for other fields beyond corrective maintenance, such as perfective and preventive maintenance. A third direction of future work involves the execution of a large-scale broad survey of the usage of SoftVis tools in the software industry during the past years, with the aim of eliciting evidence for the factors which favor, or block, the wide acceptance of such tools in daily development activities.

Bibliography

[1] J. Abello and F. van Ham. Matrix Zoom: A visual interface to semi-external graphs. In *Proc. InfoVis*, pages 183–190. IEEE, 2004.

[2] AccessExcellence. Writing Hypotheses, 2009. `http://www.accessexcellence.org/LC/TL/filson/writhypo`.

[3] Aertia. Allinea ddt, 2009. `http://www.aertia.com/en/productos.asp?pid=158`.

[4] Aivosto. Project analyzer v8.1, 2008. `http://www.aivosto.com/project/project.html`.

[5] Aivosto. Vb watch, 2008. `http://www.aivosto.com/vbwatch.html`.

[6] Apache Software Foundation. Apache Beehive Project, 2007. `http://beehive.apache.org`.

[7] Apache Software Foundation. Apache Tomcat, 2007. `http://tomcat.apache.org`.

[8] Apache Software Foundation. Lucene, 2007. `http://lucene.apache.org`.

[9] ArgoUML. ArgoUML diagram visualization tool. `http://argouml.tigris.org`, 2008.

[10] Aristotle Research Group. The Tarantula visualization tool. `http://www.cc.gatech.edu/aristotle/Tools/tarantula`, 2008.

[11] R. Baecker, C. DiGiano, and A. Marcus. Software visualization for debugging. *Comm. of the ACM*, 40(4):44–54, 1997.

[12] V. R. Basili. Is there a future for empirical software engineering? In *Proc. Intl. Symp. on Empirical Software Engineering (ISESE)*, pages 1–1. ACM, 2006. keynote presentation.

[13] S. Bassil and R. K. Keller. Software visualization tools: Survey and analysis. In *Proc. Intl. Workshop on Program Comprehension (IWPC)*, pages 7–17. IEEE, 2001.

[14] S. P. Bassil and R. K. Keller. A qualitative and quantitative evaluation of software visualization tools. In *Proc. Intl. Workshop on Software Visualization (Vissoft)*, pages 33–37. IEEE, 2003.

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

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

[17] H. Byelas and A. Telea. Visualization of areas of interest in software architecture diagrams. In *Proc. Intl. Symp. on Software Visualization (SOFTVIS)*, pages 105–114. ACM, 2006.

[18] H. Byelas and A. Telea. Towards realism in drawing areas of interest on architecture diagrams. In *J. of Visual Languages and Computing*, pages 110–128. Elsevier, 2009.

[19] P. Charland, D. Dessureault, M. Lizotte, D. Ouellet, and C. Nécaille. Using software analysis tools to understand military applications: A qualitative study. In *Tech. Memorandum TM-425, R&D Canada-Valcartier, Valcartier Québec, Canada*, volume 425, 2005.

[20] S. M. Charters, N. Thomas, and M. Munro. The end of the line for software visualisation? In *Proc. of the* 2nd *Workshop on Visualizing Software for Analysis and Understanding (VISSOFT)*, pages 27–35. IEEE, 2003.

[21] CHISEL. The CHISEL software analysis tool. `http://www.thechiselgroup.org`, 2006.

[22] J. Cordy. Comprehending reality practical barriers to industrial adoption of software maintenance automation. In *Proc. of the* 11th *Intl. Workshop on Program Comprehension (IWPC)*, pages 196–204. IEEE, 2003.

[23] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proc. Intl. Conf. on Program Comprehension (ICPC)*, pages 49–58. IEEE, 2007.

[24] L. Coulmann. General requirements for a program visualization tool to be used in engineering of 4GL programs. In *Proc. Symposium on Visual Languages (VL)*, pages 37–41. IEEE, 1993.

[25] D. Crocker. Safe object-oriented software: The verified design-by-contract paradigm. In *Proc. of the* 12th *Symp. on Safety-Critical Systems*, pages 19–41, 2004.

[26] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering approach combining metrics and program visualization. In *Proc. Intl. Working Conf. on Reverse Engineering (WCRE)*, pages 175–186. IEEE, 1999.

[27] Devexpress Inc. Coderush with Refactor Pro. `http://www.devexpress.com/Products/NET/IDETools/CodeRush`, 2007.

[28] M. di Penta, R. E. K. Stirewalt, and E. Kraemer. Designing your next empirical study on program comprehension. *Proc. of the* 15th *IEEE Intl. Conf. on Program Comprehension (ICPC)*, pages 281–285, 2007.

[29] S. Diehl. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[30] Ds-emedia. jbixbe, 2008. `http://www.jbixbe.com`.

[31] S. Ducasse and S. Demeyer. The FAMOOS object-oriented reengineering handbook, 1999. `scg.unibe.ch/archive/famoos/handbook`.

[32] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proc. of the* 2^{nd} *Intl. Symposium on Constructing Software Engineering Tools (CoSET)*, pages 13–25, 2000.

[33] Eclipse Team. Eclipse java development tools. `http://www.eclipse.org/jdt`, 2009.

[34] S. Eick, J. Steffen, and E. Sumner. Seesoft: A tool for visualizing line oriented software statistics. *Transactions on Software Engineering*, 18(11):957–968, 1992.

[35] G. Ellis and A. Dix. An explorative analysis of user evaluation studies in information visualisation. In *Proc. VI workshop on Beyond Time and Errors: Novel Evaluation Methods for Information Visualization*. ACM, 2006.

[36] FAMOOS. The Goose visualization tool. `http://esche.fzi.de/PROSTextern/software/goose`, 2008.

[37] N. Fiedler. Jswat java debugger. `http://jswat.sourceforge.net`, 2008.

[38] B. Flyvbjerg. Five misunderstandings about case-study research. *Qualitative Inquiry*, 12(2):219, 2006.

[39] Apache Software Foundation. Apache web server. `http://www.apache.org`, 2008.

[40] FrontEndART. The columbus C++ static analyzer tool. `www.frontendart.com`, 2008.

[41] H. Fujita and P. Johannesson. *New Trends in Software Methodologies, Tools and Techniques (Proc. of Lyee-W'02)*. IOS Press, 2002.

[42] P. V. Gestwicki and B. Jayaraman. JIVE: Java interactive visualization environment. In *Proc. OOPSLA Companion*, pages 226–228. ACM, 2004.

[43] D. J. Gilmore, R. L. Winder, and F. Détienne. *User-centred Requirements for Software Engineering Environments*. Springer, 1994.

[44] hello2morrow. SonarJ Visualization Plugin, 2008. `http://www.hello2morrow.com/products/sonarj`.

[45] T. Henderson. ns-allinone-2.33 release, 2008. `http://sourceforge.net/`.

[46] F. Herrera. *A Usability Study of the Tksee Software Exploration Tool*. PhD thesis, Dept. of Computer Science, Carleton University, USA, 1999. PhD thesis.

[47] H. Hoogendorp, O. Ersoy, D. Reniers, and A. Telea. Extraction and Visualization of Call Dependencies for Large C/C++ Code Bases: A Comparative Study. In *Proc. Intl. Workshop on Software Visualization (Vissoft)*. IEEE, 2009.

[48] A. Huberman and B. Miles. *The Qualitative Researcher's Handbook*. Sage Publications Ltd., 2002.

[49] C. Hundhausen, S. Douglas, and J. Stasko. A meta-study of software visualization effectiveness. *J. of Visual Languages and Computing*, pages 259–290, 2002.

[50] I. Aracic. Ispace Visualization Plugin, 2008. `http://ispace.stribor.de`.

[51] IBM. Rational PurifyPlus. `http://www-306.ibm.com/software/awdtools/purifyplus`, 2007.

[52] IEEE. IEEE Software Maintenance Standard I2 19-1992, 1992.

[53] Instantiations, Inc. CodePro Analytix, 2008. `http://www.instantiations.com/codepro`.

[54] K. Jambor-Sadeghi, M. A. Ketabchi, J. Chue, and M. Ghiassi. A systematic approach to corrective maintenance. *The Computer Journal*, 37(9):764–778, 1994.

[55] T. D. Jick. Mixing qualitative and quantitative methods: Triangulation in action. *Administrative science quarterly*, pages 602–611, 1979.

[56] C. Johnson. Top scientific visualization research problems. *Computer Graphics & Applications*, 24(4):13–17, 2004.

[57] J. Jones, M. Harrold, and J. Stasko. Visualization for fault localization. In *Proc. Intl. Workshop on Software Visualization (Vissoft)*, pages 71–75. IEEE, 2001.

[58] H. Kagdi, S. Yusuf, and J. I. Maletic. On using eye tracking in empirical assessment of software visualizations. In *Proc. of the* 1^{st} *ACM Intl. Workshop on Empirical Assessment of Software Engineering Languages and Technologies, held with the* 22^{nd} *Intl. Conf. on Automated Software Engineering (ASE)*, pages 21–22. ACM, 2007.

[59] M. Kajko-Mattsson. The state of documentation practice within corrective maintenance. In *Proc. Intl. Conf. on Software Maintenance (ICSM)*, pages 354–363. IEEE, 2001.

[60] O. Kannusmaki and A. Moreno. What a novice wants: Students using program visualization in distance programming courses. In *Proc.* 3^{rd} *Program Visualization Workshop*, pages 126–133. Department of Computer Science, University of Warwick, UK, 2004.

[61] D. Kayiwa, C. Tumwebaze, and F. Nkuyahaga. Epihandy mobile code base, 2008. `http://code.google.com/p/epihandymobile`.

[62] KBRE. The tksee visualization tool. `http://www.site.uottawa.ca/\~tcl/kbre`.

[63] H. Kienle. *Building Reverse Engineering Tools with Software Components*. PhD thesis, Dept. of Computer Science, University of Victoria, Canada, 2006.

[64] H. M. Kienle and H. A. Müller. Requirements of software visualization tools: A literature survey. In *Proc. of the* 4^{th} *IEEE Intl. Workshop on Visualizing Software for Understanding and Analysis (Vissoft)*, pages 2–9, 2007.

[65] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, 1995.

[66] Kitware, Inc. The visualization toolkit, version 4. `www.kitware.org`, 2007.

[67] C. Knight and M. Munro. Mediating diverse visualisations for comprehension. In *Proc. 9th Intl. Workshop on Program Comprehension (IWPC)*, pages 18–25. IEEE, 2001.

[68] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Proc. of the 27th Intl. Conf. on Software Engineering (ICSE)*, pages 126–135. IEEE, 2005.

[69] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey. *J. of Software Maintenance and Evolution*, 15:87–109, 2003.

[70] R. Koschke, A. Telea, and C. Hundhausen. Proc. 4th acm symposium on software visualization (softvis). ACM, 2008.

[71] C.F.J. Lange and M.R.V. Chaudron. Effects of defects in UML models - an experimental investigation. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 401–411. IEEE, 2006.

[72] M. Lanza. CodeCrawler - a lightweight software visualization tool. In *Proc. 3rd Intl. Workshop on Visualizing Software for Understanding and Analysis (Vissoft)*, pages 56–58. IEEE, 2003.

[73] M. Lanza. CodeCrawler: Lessons learned in building a software visualization tool. In *Proc. 7th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 409–418. IEEE, 2003.

[74] M. Lattu and J. Tarhio. How a visualization tool can be used: Evaluating a tool in a research and development project. In *Proc. 12th Workshop of the Psychology of Programming Interest Group (PPIG)*, 2000.

[75] A. Lawrence, A. Badre, and J. Stasko. Empirically evaluating the use of animations to teach algorithms. In *Proc. Intl. Symp. on Visual Languages (VL)*, pages 48–54. IEEE, 1994.

[76] F. Lemieux and M. Salois. Visualization techniques for program comprehension. In *New Trends in Software Methodologies, Tools and Techniques (eds. H. Fujita and M. Mejri)*, pages 22–47. IOS Press, 2006.

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

[78] W. Lowe, M. Ericsson, J. Lundberg, T. Panas, and N. Petersson. VizzAnalyzer: A software comprehension framework. In *Proc. 3rd Conf. on Software Engineering Research and Practice in Sweden.* Dept. of Computer Science, Lund University, Sweden, 2003.

[79] J. Maletic, A. Marcus, and M. Collard. A task oriented view of software visualization. In *Proc. 2nd Intl. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 32–39. IEEE, 2002.

[80] J. Maletic, A. Marcus, and L. Feng. sv3D: A framework for software visualization. In *Proc. of the* 25th *Intl. Conf. on Software Engineering (ICSE)*, pages 812–818. IEEE, 2003.

[81] J. I. Maletic and H. Kagdi. Expressiveness and effectiveness of program comprehension: Thoughts on future research directions. In *Proc. Intl. Conf. on Frontiers of Software Maintenance (FoSM)*, pages 31–37. IEEE, 2008.

[82] A. Marcus, D. Comorski, and A. Sergeyev. Supporting the evolution of a software visualization tool through usability studies. In *Proc.* 13th *Intl. Workshop on Program Comprehension (IWPC)*, pages 307–316. IEEE, 2005.

[83] A. Marcus, L. Feng, and J. I. Maletic. 3D representations for software visualization. In *Proc.* 2nd *Intl. Symp. on Software Visualization (SOFTVIS)*, pages 27–35. ACM, 2003.

[84] A. Marcus, L. Feng, and J. I. Maletic. Comprehension of software analysis data using 3D visualization. In *Proc.* 11th *Intl. Workshop on Program Comprehension (IWPC)*, pages 105–114. IEEE, 2003.

[85] A. V. Matveev. The advantages of employing quantitative and qualitative methods in intercultural research: practical implications from the study of the perceptions of intercultural communication competence by american and russian managers. *Bulletin of Russian Communication Association - Theory of Communication and Applied Communication*, 1:59–67, 2002.

[86] J. McNiff and J. Whitehead. *All you need to know about action research.* Sage Publications Ltd., 2006.

[87] L. S. Meyers, A. Guarino, and G. Gamst. *Applied Multivariate Research: Design and Interpretation.* Sage Publications Ltd., 2005.

[88] R. Michaud, M. A. Storey, and X. Wu. Plugging-in visualization: Experiences integrating a visualization tool with eclipse. In *Proc. Intl. Symp. on Software Visualization (SOFTVIS)*, pages 47–55. ACM, 2003.

[89] D. Moody. Validation of a method for representing large entity relationship models : An action research study. In *Proc.* 10th *European Conf. on Information Systems, Information Systems and the Future of the Digital Economy (ECIS)*, pages 237–245, 2002.

[90] S. Moreta and A. Telea. Multiscale visualization of dynamic software logs. In *Proc. EG/IEEE Symp. on Data Visualization (EuroVis)*, pages 11–18. IEEE, 2007.

[91] P. Mulholland. Using a fine-grained comparative evaluation technique to understand and design software visualization tools. In *Proc.* 7th *Workshop on Empirical Studies of Programmers*, pages 91–108. ACM, 1997.

[92] H. A. Müller, J. H. Jahnke, D. B. Smith, M. A. Storey, S. R. Tilley, and K. Wong. Reverse engineering: A roadmap. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 47–60. IEEE, 2000.

[93] Netbeans. Code coverage tool, 2008. `http://codecoverage.netbeans.org`.

[94] P. Neumann, S. Schlechtweg, and M. Carpendale. Arctrees: Visualizing relations in hierarchical data. In *Proc. EG/IEEE Symp. on Data Visualization (EuroVis)*, pages 53–60. IEEE, 2007.

[95] M. O'Brien. Software comprehension - a review and research direction. Technical Report UL-CSIS-03-3, 2003.

[96] Odysseus Software. Structure analysis for java (STAN). `http://stan4j.com`, 2008.

[97] A. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement.* Continuum, 1998.

[98] A. Orso, J. A. Jones, M. J. Harrold, and J. Stasko. Gammmatella: Visualization of program-execution data for deployed software. In *Proc. of the* 26th *Intl. Conf. on Software Engineering (ICSE).* IEEE, 2004.

[99] P. O'Shea and C. Exton. The application of content analysis to programmer mailing lists as a requirements method for a software visualisation tool. In *Proc. of the* 12th *Intl. Workshop on Software Technology and Engineering Practice (STEP)*, pages 30–39. IEEE, 2004.

[100] M. Pacione, M. Roper, and M. Wood. A comparative evaluation of dynamic visualization tools. In *Proc. of the* 10th *Working Conf. on Reverse Engineering (WCRE)*, pages 80–89. IEEE, 2003.

[101] M. J. Pacione. VANESSA: Visualisation abstraction network for software systems analysis. In *Proc.* 21st *Intl. Conf. on Software Maintenance (ICSM)*, pages 85–88. IEEE, 2005.

[102] D. E. Perry, A. A. Porter, and L. G. Votta. Empirical studies of software engineering: a roadmap. In *Proc. Intl. Conf. on the Future of Software Engineering (FoSE)*, pages 345–355. ACM, 2000.

[103] M. Petre, A. F. Blackwell, and T. R. G. Green. Cognitive questions in software visualization. In *Software Visualization: Programming as a Multimedia Experience (eds. J. Stasko, J. Domingue, J. Brown, B. Price)*, pages 453–480. MIT Press, 1998.

[104] V. Piller and J. Lebarta. Paraver: A tool to visualize and analyze parallel code. In *Proc. the* 18th *World Occam and Transputer User Group - Transputer and Occam Developments (WoTUG-18).* IOS Press, 1995.

[105] C. Plaisant. The challenge of information visualization evaluation. In *Proc. Working Conf. on Advanced Visual Interfaces (AVI)*, pages 109–116. ACM, 2004.

[106] M. Poppendieck and T. Poppendieck. *Implementing Lean Software Development: From Concept to Cash.* Addison-Wesley, 2006.

[107] B. A. Price, R. Baecker, and I. Small. A principled taxonomy of software visualization. *J. of Visual Languages and Computing*, 4(3):211–266, 1993.

[108] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proc. Intl. Workshop on Program Comprehension (IWPC)*, pages 271–278. IEEE, 2002.

[109] A. Raza, G. Vogel, and E. Plodereder. Bauhaus - a tool suite for program analysis and reverse engineering. Reliable Software Technologies, (Proc. Ada Europe), 2006.

[110] P. Reason and H. Bradbury. *The SAGE handbook of action research: Participative inquiry and practice.* Sage Publications Ltd., 2007.

[111] J. Rech and W. Schafer. Visual support of software engineers during development and maintenance. *SIGSOFT Softw. Eng. Notes*, 32(2):1–3, 2007.

[112] S. P. Reiss. The paradox of software visualization. In *Proc Intl. Workshop on Visualizing Software for Understanding and Analysis (Vissoft)*, pages 59–63. IEEE, 2005.

[113] M. Renieris and S. P. Reiss. Almost: Exploring program traces. In *Proc. Workshop on New Paradigms in Information Visualization and Manipulation*, pages 70–77, 1999.

[114] M. Rudgyard. Novel techniques for debugging and optimizing parallel applications. In *Proc. ACM Conf. on Supercomputing (SC)*, pages 281–290. ACM, 2006.

[115] W. Scacchi. Is open source software development faster, better, and cheaper than software engineering? In *Proc. 2^{nd} Workshop on Open Source Software Engineering.* IEEE, 2002.

[116] T. Schafer and M. Menzini. Towards more flexibility in software visualization tools. In *Proc. Intl. Workshop on Visualizing Software for Understanding and Analysis (Vissoft)*, pages 20–26. IEEE, 2005.

[117] M. Sensalire and P. Ogao. 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, 2007.

[118] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. Intl. Symp. on Visual Languages (VL)*, pages 336–343. IEEE, 1996.

[119] B. Shneiderman and C. Plaisant. Strategies for evaluating information visualization tools: Multi-dimensional in-depth long-term case studies. In *Proc. Conf. on Advanced Visual Interfaces (AVI)*, pages 1–7. ACM, 2006.

[120] S. E. Sim. *Supporting Multiple Program Comprehension Strategies During Software Maintenance.* PhD thesis, Dept. of Computer Science, University of Toronto, Canada, 1998.

[121] J. Singer. Practices of software maintenance. In *Proc. Intl. Conf. on Software Maintenance (ICSM)*, pages 139–145. IEEE, 1998.

[122] D. Sjöberg, T. Dybäa, and M. Jörgensen. The future of empirical methods in software engineering research. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 358–378. IEEE, 2007.

[123] M. J. Smith. *Contemporary communication research methods.* Wadsworth Pub. Co., 1988.

[124] H. Sneed. Bridging the gap between research and business in software maintenance. In *Proc. of the* 21st *Intl. Conf. on Software Maintenance (ICSM)*, pages 3–6. IEEE, 2005.

[125] Software Composition Group (SCG). The MOOSE program analysis environment. `http://www.iam.unibe.ch/scg/Research/Moose`, 2006.

[126] Source Navigator. The source navigator code analysis tool, 2008. `http://sourcenav.sourceforge.net`.

[127] M. J. Sousa. A survey on the software maintenance process. In *Proc. Intl. Conf. on Software Maintenance (ICSM)*, pages 265–273. IEEE, 1998.

[128] M. Storey, K. Wong, E. Fracchiat, and H. Miillert. On integrating visualization techniques for effective software exploration. In *Proc. Intl. Symp. on Information Visualization (InfoVis)*, pages 38–46. IEEE, 1997.

[129] M. A. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools.* PhD thesis, Simon Fraser University, Canada, 1998.

[130] M. A. Storey. Designing a software exploration tool using a cognitive framework. In *Software Visualization - From Theory to Practice (ed. K. Zhang)*, pages 113–145. Kluwer Academic, 2003.

[131] M. A. Storey and A. Frachia. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems & Software*, 44(3):171–185, 1999.

[132] M. A. Storey and D. M. German. On the use of visualization to support awareness of human activities in software development: A survey and a framework. In *Proc. Intl. Symp. on Software Visualization (SOFTVIS)*, pages 193–202. ACM, 2005.

[133] N. Subrahmaniyan, M. Burnett, and C. Bogart. Software visualization for end-user programmers: trial period obstacles. In *Proc. Intl. Symp. on Software Visualization (SOFTVIS)*, pages 135–144. ACM, 2008.

[134] D. Sun and K. Wong. On understanding software tool adoption using perceptual theories. In *Proc. Intl. Workshop on Adoption-Centric Software Engineering (ACSE)*, pages 51–55. IEEE, 2004.

[135] E. B. Swanson. The dimensions of maintenance. In *Proc. of the* 2nd *Intl. Conf. on Software Engineering (ICSE)*, pages 492–497. IEEE, 1976.

[136] A. Telea. Software evolution assessment study, 2008. `http://www.cs.rug.nl/alext/Assignment`, Univ. of Groningen, the Netherlands.

[137] A. Telea, A. Maccari, and C. Riva. An open toolkit for prototyping reverse engineering visualizations. In *Proc. Intl. Symp. on Data Visualization (VisSym)*, pages 241–249. ACM, 2002.

[138] A. Telea and L. Voinea. An integrated reverse engineering environment for large-scale C++ code. In *Proc. Symp. on Software Visualization (SOFTVIS)*, pages 67–76. ACM, 2008.

[139] A. Telea and L. Voinea. SolidFX: An integrated reverse-engineering environment for C++. In *Proc. Intl. Conf. on Software Maintenance and Reengineering (CSMR)*, pages 150–153, 2008.

[140] A. Teyseyre and R. M. Campo. An overview of 3d software visualization. *IEEE TVCG*, 15(1):87–105, 2009.

[141] S. Tilley and S. Huang. On selecting software visualization tools for program understanding in an industrial context. In *Proc. Intl. Workshop on Program Comprehension (IWPC)*, pages 285–288. IEEE, 2002.

[142] S. R. Tilley, K. Wong, M. A. Storey, and H. A. Müller. Programmable reverse engineering. *Intl. J. of Software Engineering and Knowledge Engineering*, 4:501–520, 1994.

[143] C. Tjortjis, N. Gold, P. Layzell, and K. Bennett. From system comprehension to program comprehension. In *Proc. of the 26th Annual Intl. Conf. on Computer Software and Applications*, 2002.

[144] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä. Empirical studies in reverse engineering: state of the art and future trends. *Empirical Software Engineering*, 12(5):551–571, 2007.

[145] M. Tory and T. Möller. Human factors in visualization research. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):72–84, 2004.

[146] W. Trochim and J. Donnelly. *The Research Methods Knowledge Base (3rd ed.)*. Atomic Dog Publ., 2007. `http://www.socialresearchmethods.net/kb`.

[147] K. van Haaster and D. Hagan. Teaching and learning with BlueJ: an evaluation of a pedagogical tool. *Issues in Informing Science & Information Technology*, 1:455–470, 2004.

[148] A. M. Vans, A. von Mayrhauser, and G. Somlo. Program understanding behavior during corrective maintenance of large-scale software. *Intl. Journal of Human-Computers Studies*, 51(1):31–70, 1999.

[149] L. Voinea and A. Telea. CVSgrab: Mining the History of Large Software Projects. In *Proc. EG/IEEE Symp. on Data Visualization (EuroVis)*, pages 187–194. IEEE, 2006.

[150] L. Voinea and A. Telea. Visual data mining and analysis of software repositories. *Computers & Graphics*, 31(3):410–428, 2007.

[151] L. Voinea and A. Telea. Visual querying and analysis of large software repositories. *Empirical Software Engineering*, 14(3):316–340, 2008.

[152] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: Visualization of code evolution. In *Proc. of the* 3rd *ACM Symposium on Software Visualization (SOFTVIS)*, pages 47–56. ACM, 2005.

[153] A. von Mayrhauser and A. Vans. Comprehension process during large scale maintenance. In *Proc.* 16th *Intl. Conf. on Software Engineering (ICSE)*, pages 39–48. IEEE, 1994.

[154] A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements for an industrial environment. In *Proc. of the* 2nd *Intl. Workshop on Program Comprehension (IWPC)*, pages 78–86. IEEE, 1993.

[155] A. Walenstein. Theory-based analysis of cognitive support in software comprehension tools. In *Proc. of the* 10th *Intl. Workshop on Program Comprehension (IWPC)*, pages 75–84. IEEE, 2002.

[156] A. E. Walenstein. Developing the designer's toolkit with software comprehension models. In *Proc. of the* 13th *Intl. Conf. on Automated Software Engineering (ASE)*, pages 13–16. IEEE, 1998.

[157] R. J. Walker, L. C. Briand, D. Notkin, C. B. Seaman, and W. F. Tichy. Panel: empirical validation: what, why, when, and how. In *Proc. of the* 25th *Intl. Conf. on Software Engineering (ICSE)*, pages 722–722. IEEE, 2003.

[158] Q. Wang, W. Wang, R. Brown, K. Driesen, B. Dufour, L. Hendren, and C. Verbrugge. EVolve: An open extensible software visualization framework. In *Proc. Symp. on Software Visualization (SOFTVIS)*, pages 37–45. ACM, 2003.

[159] R. Wettel and M. Lanza. Visually localizing design problems with disharmony maps. In *Proc. Intl. Symp. on Software Visualization (SOFTVIS)*, pages 155–164. ACM, 2008.

[160] R. Wood. Assisted software comprehension. In *MSc thesis*. Imperial College, UK, 2003. `http://www3.imperial.ac.uk/pls/portallive/docs/1/18619750.PDF`.

[161] R. Wuyts and S. Ducasse. Unanticipated integration of development tools using the classification model. *Computer Languages Systems & Structures*, 30:63–77, 2004.

[162] R. K. Yin. *Case study research: Design and methods*. Sage Publications Ltd., 1994.

[163] R. K. Yin. *Applications of Case Study Research (2nd ed.)*. Sage Publications Ltd., 2003.

[164] S. W. L. Yip and T. Lam. A software maintenance survey. In *Proc.* 1st *Asia-Pacific Software Engineering Conf.*, pages 70–79, 1994.

[165] I. Zayour and T. Lethbridge. Adoption of reverse engineering tools: A cognitive perspective and methodology. In *Proc. Intl. Workshop on Program Comprehension (IWPC)*, pages 245–258. IEEE, 2001.