JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2003; **15**:87–109 (DOI: 10.1002/smr.270)

Research

Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey

Rainer Koschke*,†

Institut für Softwaretechnologie, Universität Stuttgart, Breitwiesenstrasse 20–22, 70565 Stuttgart, Germany

SUMMARY

Software visualization is concerned with the static visualization as well as the animation of software artifacts, such as source code, executable programs, and the data they manipulate, and their attributes, such as size, complexity, or dependencies. Software visualization techniques are widely used in the areas of software maintenance, reverse engineering, and re-engineering, where typically large amounts of complex data need to be understood and a high degree of interaction between software engineers and automatic analyses is required. This paper reports the results of a survey on the perspectives of 82 researchers in software maintenance, reverse engineering, and re-engineering on software visualization. It describes to which degree the researchers are involved in software visualization themselves, what is visualized and how, whether animation is frequently used, whether the researchers believe animation is useful at all, which automatic graph layouts are used if at all, whether the layout algorithms have deficiencies, and—last but not least—where the medium-term and long-term research in software visualization should be directed. The results of this survey help to ascertain the current role of software visualization in software engineering from the perspective of researchers in these domains and give hints on future research avenues. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: software visualization; software maintenance; reverse engineering; re-engineering

1. INTRODUCTION

Extending Roman and Cox's [1] definition of program visualization to other software artifacts; software visualization can be defined as the mapping from software artifacts—including programs—to graphical representations. Software visualization is needed because software is invisible. In the simplest case, we may visualize artifacts textually, which is considered the most primitive kind of visualization (strictly

*Correspondence to: Rainer Koschke, Institut für Softwaretechnologie, Universität Stuttgart, Breitwiesenstrasse 20–22, 70565 Stuttgart, Germany.
†E-mail: koschke@informatik.uni-stuttgart.de

Received 22 March 2002
Revised 19 October 2002

speaking, given the above definition of software visualization, presenting plain text is not software visualization if one does not consider a purely textual representation as a graphical representation). More advanced graphical visualization techniques promise to improve the understanding of software, usually in combination with techniques that raise the level of abstraction, reduce the amount of information to what is needed to perform the task at hand, or to ease browsing the large information space. However, whether graphical representation is really superior to textual representation is rarely proven empirically. Seeking for a general proof is unrealistic as the benefit of graphical representations depends upon the method of representation, the domain and range of the mapping from software to graphical representation, and the addressed viewer and his or her problem. There are empirical studies that show evidence that specific ways of graphical visualization work better than textual visualization for certain tasks [2]. In other cases, a textual presentation is likely to be the most appropriate [3,4]. Nevertheless, many researchers believe in the value of software visualization. In particular, in the domains of software maintenance, reverse engineering, and re-engineering—where typically large amounts of complex data need to be understood—software visualization may play an important role. We know from empirical studies that maintenance programmers spend 50% of their time simply trying to understand the software to be changed [5] and it is plausible that the method of visualization has a substantial effect on the time needed to comprehend large programs—be it positive or negative.

A recent Dagstuhl seminar on software visualization [6] brought together researchers from different domains of software visualization. The topics covered by the seminar included visual languages, algorithm animations, metaphorical visualization, techniques to preserve the mental maps while changing from one view to another, visualization of metrics, automatic graph drawing, and techniques to recognize repeated patterns in the data to be visualized in order to reduce their visual complexity. These aspects are more oriented toward specific visualization techniques. However, there are also people from software engineering domains who are driven by a particular problem and are seeking suitable visualization techniques in order to solve that problem. However, the problem-driven group was in the minority and there were significantly few people from the domains of software maintenance, reverse engineering, and re-engineering—even though the visualization problem is considered a central problem in these domains. Why did so few people of these domains attend the seminar? Was it because the seminar coincided with the *International Workshop on Program Comprehension* and the *International Conference on Software Engineering 2001*, important conferences in these fields? Or was it because researchers in these domains think software visualization is actually less relevant, or even already solved? So, what are the perspectives of researchers in software maintenance, reverse engineering, and re-engineering on software visualization? To which degree are they involved in software visualization themselves? What is visualized and how? Is animation frequently used? Do researchers think animation is useful at all? Are automatic graph layouts used? Which ones? Do they have deficiencies? Last but not least, where should the medium-term and long-term research in software visualization be directed? In order to find answers to these open questions, a survey was conducted among researchers in software maintenance, reverse engineering, and re-engineering. The results of this survey are presented here and help to ascertain the current role of software visualization in software engineering from the perspective of researchers in these domains and to give hints on future research avenues.

The remainder of this paper is organized as follows. Section 2 summarizes related research in software visualization. Section 3 gives background information on the survey and Section 4 lists the original questions. The answers to these questions are summarized in Section 5. Conclusions for future research are drawn in Section 6.

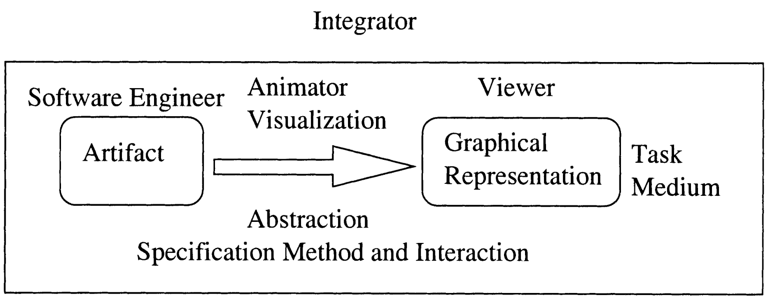


Figure 1. Extended model of visualization.

2. RELATED RESEARCH

This section introduces the conceptual model for software visualization by Roman and Cox and describes the related surveys on software visualization.

Roman and Cox [1] point out the following relevant roles in, and aspects of, software visualization (I use more general terms, like ‘software engineer’ and ‘artifact’ instead of their original terms ‘programmer’ and ‘program’; see Figure 1).

Software engineer: the producer of the artifact to be visualized.

Animator: the person who creates the mapping onto graphical representations.

Viewer (also referred to as the end-user): the person who views the graphical representation.

Scope: what artifact and which of its aspects are visualized?

Abstraction: what kind of information is conveyed by the visualization?

Specification method (and interaction): how is the visualization constructed? The specification method can range from ‘hard-wired’, where the viewer has no influence on the representation, to arbitrary redefinition.

Graphical representation (technique, originally): how is the graphical representation used to convey the information? This aspect considers issues having to do with effective visual communication, including the visual vocabulary, the use of the specific visual elements to convey particular kinds of information, the organization of visual information, and the order in which material is represented to the viewer.

Other relevant roles and aspects that are not considered by Roman and Cox are (partly pointed out by respondents of the survey):

Integrator: a person who integrates the visualization tool in a more general tool suite supporting, for instance, software maintenance tasks.

Task: what does the viewer try to achieve using the visualization?

Medium: where to represent? Examples for different media include computer screens, paper, or virtual-reality environments.

Bassil and Keller conducted a similar survey on software visualization [7]. The focus of Bassil and Keller's survey is on several functional, practical (like costs, portability, quality of documentation, etc.), and cognitive aspects of existing software visualization tools. One of the distinguishing factors is that Bassil and Keller's survey is primarily aimed at the use of existing software visualization tools in the software industry (two-thirds of the respondents are actually from industry and only one-third stems from the academic world; the majority of evaluated tools are commercial) whereas the survey reported in this paper targeted researchers that develop or integrate visualization techniques and, hence, is more oriented toward future research. In order to distinguish this survey from the one of Bassil and Keller in the following, this survey will be referred to as the **research survey**. Moreover, Bassil and Keller let participants rate certain functional aspects selected by themselves as either 'absolutely essential', 'useful, but not essential', or 'not at all useful'. If participants were unsure, they could also check 'don't know/not applicable'. In this research survey, more open questions were asked. In Section 5, the results of Bassil and Keller's survey for the functional aspects are compared to the result of the research survey.

Whitley has conducted a literature survey on experimental evaluation of visual languages [8] and found evidence for and against these languages. Whitley agrees to Gilmore and Green's match-mismatch hypothesis [9], which says that problem-solving performance depends on whether the structure of a problem is matched by the structure of a notation. This hypothesis is stronger than the basic principle that the benefits of a notation are relative to a particular task. The match-mismatch hypothesis states that every notation highlights some kinds of information, while obscuring others. Thus, it also predicts that every notation will perform relatively poorly for certain tasks. Results for visual languages used to specify programs quite likely also apply to graphical representations that are derived from existing software artifacts.

In a literature survey by Blackwell, the metacognitive theories of authors that make a general statement about the nature of programming, the nature of thought or problem-solving, or the nature of vision are summarized [10]. Metacognitive theories are the beliefs that we have about the way we carry out mental tasks. The survey states unproven beliefs of researchers in visual languages and has found evidence for a remarkably consistent range of metacognitive theories held by researchers in the field. Identifying this knowledge is important, mainly because it determines the ways in which visual environments are designed to improve cognitive ergonomics, but also because it forms the foundation for the scientific paradigm underlying research in visual programming and software visualization. According to Blackwell, most of the metacognitive beliefs found in his survey have a logical basis in introspection, cognitive theory, or folk psychology. In some cases, they are well-founded, but in others there is either logical or empirical evidence making them inappropriate.

3. SURVEY META DATA

This section gives background information on the survey. The questions asked in this survey are listed in the following section.

This research survey was conducted by way of electronic mails sent to researchers in software maintenance, reverse engineering, and re-engineering. The list was compiled by merging lists of program committee members and conference attendees of diverse scientific conferences and workshops in the following domains, the *International Conference on Software Maintenance*, the *Working Conference on Reverse Engineering*, the *International Workshop on Program Comprehension*, the *Conference on Software Maintenance and Re-engineering*, the *International Conference on Re-Technologies for Information Systems*, and the *Dagstuhl Seminar on Interoperability of Re-engineering Tools 2001*. I intentionally did not use the list of attendees of the *Dagstuhl Seminar on Software Visualization* because I wanted answers from the more problem-driven researchers of software engineering domains who primarily try to integrate and exploit existing visualization techniques rather than from people that are active researchers in software visualization specifically. The goal of this survey is to generate ideas and requirements for the latter research community.

Because the addressed people belong to the domains of software maintenance, reverse engineering, and re-engineering, this survey is neither a survey on the use of software visualization in software engineering in general (but rather in these related subdomains only), nor is it a survey on the use of software visualization as perceived by end-users of maintenance, reverse engineering, or re-engineering tools. It also goes without saying that it is not a substitute for real empirical studies. Yet, it takes a snapshot of the current perspectives of researchers on software visualization. In the absence of empirical studies, these opinions will have a strong influence on future research. The research survey also reveals shortcomings of current techniques and identifies areas in visualization that are particularly important.

This research survey is neither a survey on specific visualization tools or more general tools for software maintenance, reverse engineering, and re-engineering. Surveys on such tools have been published elsewhere [11–13].

The questionnaire was sent out on June 25 2001 to about 580 different e-mail addresses (the estimation of aliased addresses, i.e. addresses that refer to the same person, in this list is less than 2%) and was answered by 121 people. The last response was received on July 24 2001. The first question required confirmation that one develops or uses maintenance, reverse engineering, or re-engineering tools. If someone answered 'no' to this question and hence does not belong to the targeted domain, he or she could skip all remaining questions. 83 people confirmed that they are developing or using maintenance, reverse engineering, or re-engineering tools. For the statistical summary, only 82 positive answers were actually considered because one person added only one aspect concerning deficiencies of automatic layouts and otherwise referred to a colleague's answer to which he agreed.

4. QUESTIONNAIRE

The original questions of this survey are given in this section. The goal of this survey was to gather opinions and perspectives. It primarily targeted open questions for which a predefined set of answers was not known, and consequently free text was expected for most answers. In particular, questions 4a and 5e were formulated in a way that required a sentence to answer them unambiguously, in order to provoke more elaborated answers. The answers to question 4a and 5e were all unambiguous.

1a Are you using or developing tools for software maintenance, reverse engineering, re-engineering, or metrics?
Answer: yes/no

1b If so, what is the domain in which you are using them?
Answer: maintenance /reverse engineering / re-engineering / metrics

1c If so, what kind of tools are these? Please describe them with a few words.
Answer: free text

2 What kind of artifacts, if any, are visualized with these tools? Examples of artifacts include: module dependencies, object models, call graphs, source text, etc. If no artifacts are visualized, you may continue with question 6.
Answer: free text

3a How are these artifacts visualized?
Answer: free text

3b Do you think the visualization is appropriate?
Answer: free text

4a Are these visualizations static or are these visualizations in some way animated? Do they allow you, for instance, to fly through a three-dimensional space? Or are graphs moved somehow?
Answer: free text

4b Do you think animated visualizations are/would be helpful?
Answer: free text

5a If the visualization is based on some kind of graph (nodes and edges), are automatic layout algorithms used for these visualizations?
Answer: yes/no

5b If so, do you use a graph layout tool/library not developed by your group?
Answer: free text

5c If so, what is the name of this layout tool/library?
Answer: free text

5d What kinds of automatic layout algorithms are you using? Examples are spring embedder, Sugiyama's algorithm, planar graph algorithms, tree layouts, etc.
Answer: free text

5e Do the automatic layout algorithms serve your needs or do they have deficiencies?
Answer: free text

5f If they do, what are these deficiencies?
Answer: free text

6 How important is software visualization to software maintenance, reverse engineering, re-engineering, and metrics in general? Please check one of the following answers.
Answer: absolutely necessary / important but not critical / relevant / can do without but is nice to have / not an issue at all

7 If you are developing software maintenance, reverse engineering, re-engineering, or metric tools, are your main research interests related to software visualization?
Please check one of the following answers (or add any free text if the following alternatives are

not appropriate).
Answer: primary research interest / substantial part / every now and then / just using or integrating / not at all

8 What do you consider the specific needs and challenges for software visualization in the context of software maintenance, reverse engineering, re-engineering, and metrics tools?
Answer: free text

9a In your wildest dreams, what kind of software visualization would you like to have?
Answer: free text
9b Being more modest, what kind of software visualization would you want to be developed in the next, say, 2 or 3 years?
Answer: free text

The next section presents the answers to these questions.

5. ANSWERS

This section summarizes the questionnaire. The detailed answers to the survey can be found on-line [14].

In the following, the answers to this research survey are not presented in the original order but in the order of questions related to the respondents and their beliefs (questions 1, 6, and 7), scope (question 2), representation issues of software visualization (questions 3, 4, and 5), and challenges and future research avenues (questions 8 and 9).

At the end of the following subsections, I present selected individual comments of the respondents, our own thoughts, and then compare the results of this survey with the results of Bassil and Keller's survey.

5.1. Background of respondents

Questions 1, 6, and 7 were targeted at the background of the respondents and their general opinion about software visualization. The answers to these questions are summarized in this section.

Figure 2 shows the domains to which the respondents of the research survey belong (question 1b). Multiple answers were allowed. Some respondents added other domains, namely, code generation, software development, architectural visualization and verification, and program understanding, yet are also part of at least one of the targeted domains.

As Figure 3 shows, the vast majority of researchers believe that software visualization is absolutely necessary or at least important to their domain. Only 1%, i.e. one person, believes software visualization is unimportant.

On the other hand, the research focus of most researchers of this survey is not on software visualization (see Figure 4). Most researchers deal with software visualization only now and then or prefer just to use or integrate existing visualization techniques.

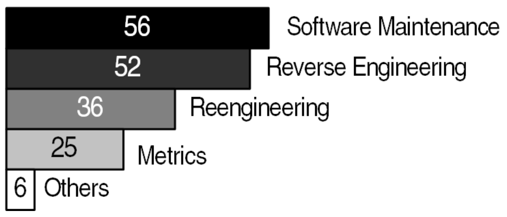


Figure 2. Responses to question 1b: the background of participants.

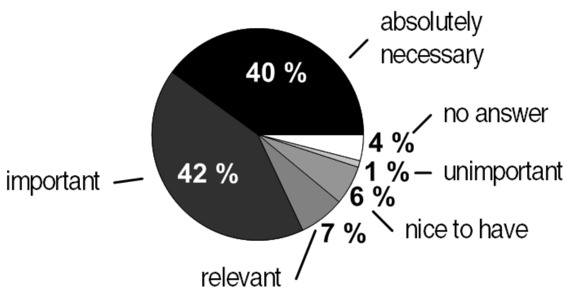


Figure 3. Responses to question 6: the importance of software visualization.

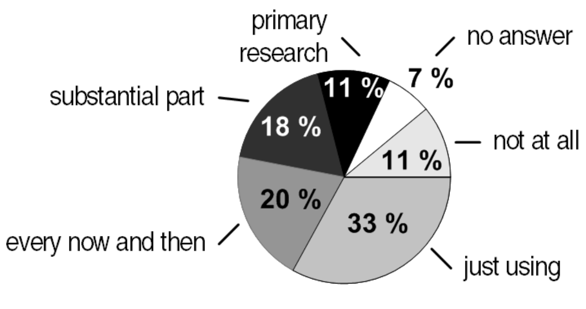


Figure 4. Responses to question 7: the research interests in visualization.

Comments

The divergence between the researchers' acknowledgment of the importance of software visualization and their own involvement in developing suitable visualization is not surprising because—as one researcher pointed out—software visualization tools are a major investment in infrastructure. Furthermore, the expertise of most researchers lies in other fields. However, the drawback of relying on external tools and knowledge is that the means of visualization are restricted to what is available. Moreover, if the gap between own expertise and expertise in software visualization is too wide, there is also the problem—as another researcher noted—that software visualization becomes just an afterthought and, thus, loses some if its potential. Conversely, one researcher noted that research on software visualization is often isolated from a thorough analysis of its dedicated applications, e.g. the activities during the software development life cycle. Software visualization could gain more impact if relevant scenarios for usage were identified and the visualization were adapted for the identified tasks. Obviously, both sides—integrators and primary software visualization researchers—could benefit from a closer collaboration.

5.2. Scope

This section summarizes the wide spectrum of software artifacts that are visualized according to the answers to question 2. Artifacts from all phases of the software life cycle are mentioned, including ontologies, knowledge data, requirements and risks, software architectures, architectural styles, design models and patterns, subsystems, modules, source code, test case tables, and evolutionary aspects. Static and dynamic data, high-, mid- and low-level information, data and control flow, structural and other dependencies and attributes are visualized.

The structural decomposition of the system into subsystems, directory structures, modules, interfaces, and even into single statements and expressions in terms of abstract syntax trees is visualized (where we may assume that abstract syntax trees are visualized for internal debugging purposes only, as the end-user likely prefers to see the source code at this level). Control flow information is visualized by way of call graphs and intraprocedural control flow graphs. With respect to data flow, variable references, side effects, and def-use dependencies are visualized. Three people write that they use program dependency graphs, which integrate def-use and control dependencies and which form a basis for program slicing [15]. Program dependency graphs are often visualized only for internal debugging and the end-user rather sees source code as a result of program slicing. However, one researcher writes that the program dependency graph is actually presented to the end-user.

Class and object models are visualized by several tools, including many different relationships between classes and objects, respectively, such as inheritance, association, and calls. Similarly, modules (comparable to classes in the procedural world) and their dependencies are another important visualized artifact. The dependencies comprise explicit import and export as well as aggregated dependencies of entities that are declared in these modules, such as call, variable reference, and type relationships. Another respondent mentions typed capsules and their logical dependencies in Real-Time Object-Oriented Modeling (ROOM) models [16].

Behavior is visualized through state charts and collaboration diagrams, processes and messages or events, and dynamic execution traces (at the level of basic blocks, routine calls, or interprocess messages). Other visualized attributes are statement level complexity and regions of (different degrees of) homogeneity. One researcher also considers evolutionary data, i.e. occurrence and evolution of an attribute under study.

Visualized artifacts beyond source code and artifacts derived from it are logical and conceptual database models and static and dynamic Web elements.

The most often visualized artifacts are those at the mid-level, i.e. artifacts between the conceptual architecture and source code, namely, call graphs, module dependencies, global variables, routines, and user-defined types and their interrelationships, and class and object models.

Comments

A large variety of software artifacts needs to be visualized. The variety of different artifacts, different levels of abstractions of these artifacts, and their intended users raises the question of how we can find suitable ways of visualization for all these artifacts as well as a consistent visualization of the connections between these artifacts.

An important aspect of the artifacts beyond their diversity is the size and dimensionality of the information space spanned by the artifacts. Often it is believed that visualization is particularly suited to give an overview of a large information space. However, several researchers in this survey state that visualization is only appropriate for small-to-medium-sized systems. One researcher, in particular, says that, for large systems or systems with an overwhelming number of dependencies, visualization does not help and it is better to use queries. The researcher's experience has been that programmers do not like visualization techniques. They prefer to get answer lists to specific questions. He writes that the academic community overrates the value of visualization. One could argue against this objection that visualization is more effective in domains where a precise answer cannot be pre-computed. However, this counter argument is still speculation and we clearly need to address the question of which characteristics of the artifacts and the information space suggest specific kinds of visualization techniques. In particular, when are advanced visualization techniques more appropriate than simple textual representations?

5.3. Representation

This section describes the answers on representation issues of software visualization (questions 3, 4, and 5).

5.3.1. Visualized artifacts

Question 3 aimed at graphical representations used to convey the information. As shown by Figure 5, graphs are the dominant way to represent information. UML diagrams are also often used, where one could view most of the UML class diagram as graphs with predefined semantics and rendering characteristics. Some researchers list different UML diagrams, seven researchers only mention UML but do not name the types of diagrams explicitly. Because of that and because different graph structures (like hierarchical graphs) were neither distinguished in Figure 5, the number of nominations of UML diagrams in Figure 5 counts uses of several different UML diagrams only once per participant. The detailed nominations are as follows: 9 class, 3 state charts, 3 collaboration, 1 sequence, 1 interaction (not further distinguished in collaboration or sequence diagram), and 2 activity diagrams.

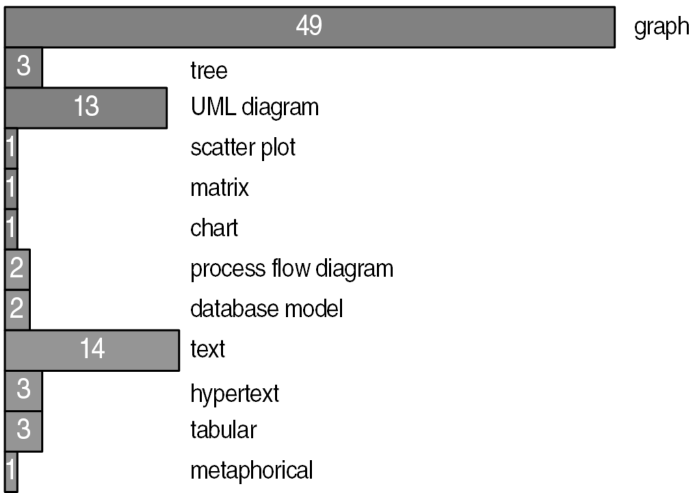


Figure 5. Responses to question 3a: how are artifacts visualized?

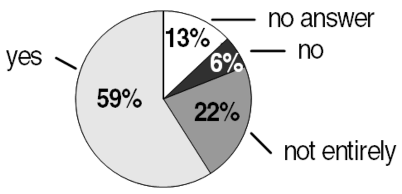


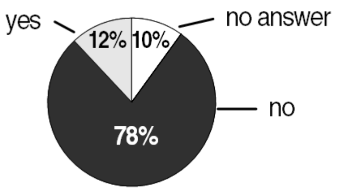
Figure 6. Responses to question 3b: is the visualization appropriate?

Comments

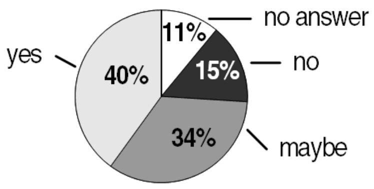
Text and hypertext are also often mentioned but we may assume that the actual number is even higher. Many people do not perceive text as a kind of visualization. In particular, the survey by Bassil and Keller has shown that visualization of source code is essential to end-users.

5.3.2. Appropriateness of visualization

Question 3b asked whether researchers believe the visualization they chose is appropriate. In fact, most researchers take the view that their kind of visualization is appropriate (see Figure 6).



(a)



(b)

Figure 7. Responses to question 4: (a) animated visualization and (b) animation helpful?

Comments

One researcher writes that their graphical syntax was carefully designed in interaction with software engineers. Another one writes that the visualization tool is enthusiastically used by users. Only one researcher notes that he has empirical evidence for this view through controlled experiments. The lack of empirical evidence for the superiority of visual programming languages through controlled experiments has already been pointed out by Whitley [8]. The same holds for visualization in the area of software maintenance, reverse engineering, and re-engineering. Experiments as those by Hendrix *et al.* [2] and Storey *et al.* [17] are rare exceptions in these areas.

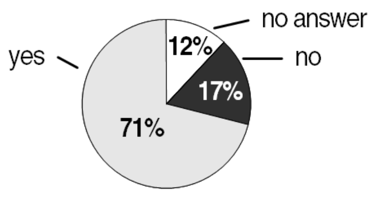
5.3.3. *Use of animation*

Figure 7 shows that animation is currently rarely used (question 4a) but that researchers still believe animation is useful (question 4b) as shown by Figure 7. Several researchers gave visualization of dynamic information as an example of the usefulness of animation.

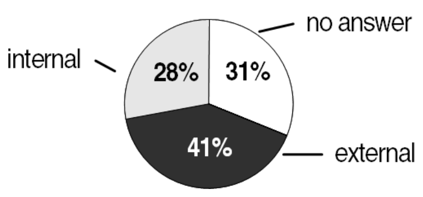
Comments

There is currently no clear agreement among researchers on whether animated visualization is useful, although there are substantially more people believing that it is. Also, the divergence of the hope in and actual use of animated visualization is interesting. This divergence explains part of the uncertainty about animated visualization. Only a few researchers have experiences with animated visualization.

Interestingly enough, in the ranking of usefulness of functional aspects in Bassil and Keller's survey, the use of animation is one of the least required aspects. Only about 10% of the participants qualified animation as essential, about 35% as useful but not essential, and 38% as not at all useful. 17% did not answer the question. Interestingly enough, Bassil and Keller noticed that the researchers among the participants rated animation as more important than people from industrial settings (suggested by significant statistical evidence). Bassil and Keller note that the divergence of academics and practitioners could be due to the fact that animation (as well as 3D visualization) is an uncommon aspect in current software visualization tools (most tools of their survey are commercial) and industrial



(a)



(b)

Figure 8. Responses to question 5: (a) automatic graph layouts and (b) own development?

end-users are not familiar with animation (which may also explain that a relatively large number of participants did not answer the question).

Consequently, further case studies and experiments should be conducted before more research in specific animation techniques is invested. Domains need to be identified where animation could actually be helpful. Visualization of dynamic aspects is proposed by many respondents.

5.3.4. Graphs

As shown by Figure 5, graphs and UML diagrams are frequently used to convey information. As opposed to forward engineering, where graphs and UML diagrams are used for specification and, hence, are generally drawn manually, graphs in software maintenance, reverse engineering, and re-engineering are usually generated by automatic analyses. Since these resulting graphs are often quite large, the question is raised whether automatic layout algorithms are used to draw graphs (question 5a). As a matter of fact, Figure 8 shows that the majority of researchers use automatic layouts, but also that a surprisingly high number of people implemented their own graph layout tool even though many non-commercial and commercial layout packages are readily available (see Table I). The most often used external graph layout tools are listed in Figure 9.

While the tools listed in Table I are generic packages to draw and lay-out graphs, graph drawing is often embedded in more specific tools. For instance, in order to draw UML diagrams, several commercial UML tools are used by the respondents of this research survey. *Together* is used by five researchers, *Rational Rose* by three, and one more is not further specified. Similarly, the graph editor, *Rigi*, developed at the University of Victoria was explicitly designed to support program comprehension [18]. Beyond its graphical representations and automatic layouts, it also offers links to the source code from the graph nodes (but not *vice versa*). *Rigi* has its own built-in layout algorithms for trees and grids but uses *Graphlet's* predecessor, *GraphEd*, as an integrated layout tool for the more advanced layouts (Sugiyama layout for directed acyclic graphs and a spring embedder for general graphs). Thus, whoever uses *Rigi* also uses *GraphEd*. Consequently, there is actually only one researcher in this survey who is using *GraphEd* in its stand-alone variant.

The graph drawing library *AGD* was not mentioned by any of the respondents. *AGD* specializes in graph planarization and—as Table I suggests—planarization layouts are rarely used altogether. Neither was *GDToolkit* mentioned in this research survey.

Table I. List of graph layout packages.

GraphViz (dot, dotty) (www.research.att.com/sw/tools/graphviz)	*AT&T*	Open Source
VCG (rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html)	*Univ. of Saarbrücken*	Gnu Public License
Graphlet (www.graphlet.de)	*Univ. of Passau / BrainSys*	free executable for non-commercial use; commercial license by Brainsys
GraphEd (www.infosun.fmi.uni-passau.de/GraphEd)	*Univ. of Passau*	free executable for non-commercial use;
Tom Sawyer Graph Layout Toolkit (www.tomsawyer.com)	*Tom Sawyer Software*	commercial
AGD (www.mpi-sb.mpg.de/AGD)	*Max-Planck Institute Saarbrücken / Univ. of Cologne / Univ. of Halle / Algorithmic Solutions*	free executable for non-commercial use; commercial license by Algorithmic Solutions
GDToolkit (www.dia.uniroma3.it/∼gdt)	*Universita' Degli Studi Roma Tre*	free executable for non-commercial use; commercial license by INTEGRA Sistemi S.r.l. (Italy)

Figure 10 contains the most widely used types of layouts internally implemented by respondents of this research survey (the layouts offered by the tools listed in Figure 9 are not considered in Figure 10). Tree layouts are surprisingly frequently used. As previously shown in Figure 5, researchers state that they use more general graph structures to represent the information. The relatively high frequency of tree layouts may suggest that many graphs used to represent information also have an inherent tree structure as, for instance, a strictly hierarchical structure (actually, four researchers explicitly or implicitly stated using hierarchical graphs). If this is true (and some more hints could actually be found in the answers to questions 8 and 9), layout algorithms should be able to distinguish edges. Some of the edges might form a tree as the principal structure of the data, but there may be additional secondary edges that do not obey the tree structure. For instance, in UML class diagrams, the inheritance relationship typically dominates the layout and association relationships (which might lead to cycles) are drawn after the classes have been laid out as a tree.

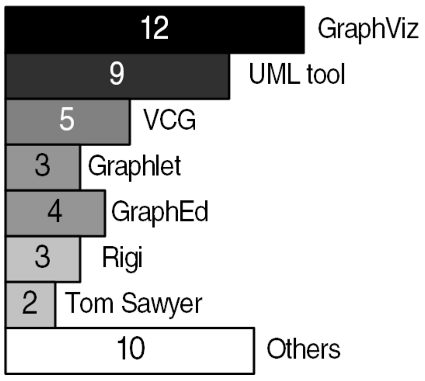


Figure 9. Responses to question 5c: graph layout tools used.

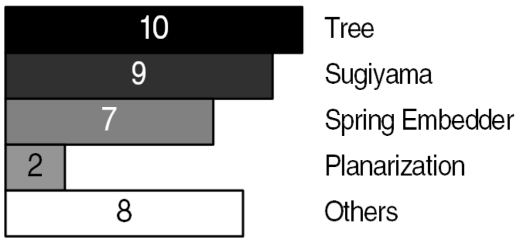


Figure 10. Responses to question 5d: the type of graph layout used.

Figure 10 shows that simpler layouts are more frequently implemented by the respondents than more challenging ones, like planarization layouts, which might suggest that the required expertise or manpower to develop more demanding layout algorithms was not available. Alternatively, one could conclude that the researchers believe more advanced layouts are not necessary. At any rate, planarization layouts are currently rarely available to end-users of recent research prototypes.

Many researchers complain of deficiencies of automatic layouts, as shown by Figure 11. Deficiencies are the lack of incremental, semantic, and scalable layouts, bad space consumption, and lack of user control. Several researchers stated that the automatic layouts are not perfect, but a good starting point for further human refinement.

Comments

Graphs have been identified in this survey as the most often used kind of visualization. Graphs are a simple, yet powerful concept to express (in general) binary relations and have a 'natural' visualization as boxes and arrows. Graphs can be considered the syntax of a symbology. The semantics of the

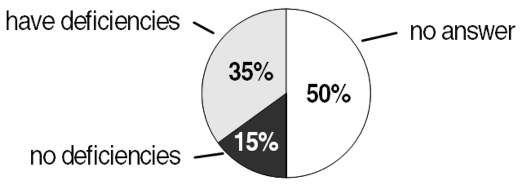


Figure 11. Responses to question 5e: is automatic layout sufficient?

symbology can then be defined in terms of the semantics of the nodes and edges. In this way, graphs represent a generic way to represent information, which is probably the reason why they are so popular.

Graphs are generally computed by automatic analyses in software maintenance, reverse engineering, and re-engineering and, hence, should also be drawn automatically if they are to be understood. As Petre showed for visual programming languages [19], graphical representations often depend upon a secondary notation, i.e. the use of layout and perceptual cues (elements such as adjacency, clustering, white space, labeling, and so on). Conversely, the layout has an important influence on the interpretation of visualized graphs. For instance, in the UML, inheritance relationships are usually drawn in the same direction and classes that are derived from each other are usually drawn closer to each other than classes that are only associated. Automatic graph drawing is therefore an important related research field.

Automatic graph drawing is an active research area in its own rights and with its own annual conference *Graph Drawing*. Good automatic graph layouts are difficult mathematical optimization problems that are typically NP-hard for all but trivial problems. The software engineering community should more often integrate graph drawing researchers and jointly experiment with more advanced techniques in order to investigate whether these techniques offer additional benefits to end-users. For instance, planarization techniques are rarely used. The collaboration with graph drawing researchers needs to be bidirectional as the aesthetic criteria used to lay-out a graph are often application-dependent and go beyond abstract criteria, like minimizing edge crossings. As a matter of fact, according to the survey by Bassil and Keller, the importance of graph visualization and automatic layout capabilities is also acknowledged by end-users [7]. These two aspects were still highly ranked by the respondents in particular for large systems to be analyzed.

The respondents of this research survey raise several issues specific to graph drawing, such as the need for scalable, incremental, and semantic layouts (i.e. layouts that take the semantics of nodes and edges into account). There is research in incremental layouts, and some systems, such as GDToolkit, do already offer incremental layouts. Technically, incremental layouts are not that difficult as long as one knows what should not change (Petra Mutzel, General Chair of Graph Drawing 2001, *Personal communication*, September 2001). Researchers in graph drawing also tackle the problem of large graphs [20]. Altogether one can note that many of the problems of graph drawing are currently recognized and investigated in the graph drawing community. However, there is as yet no tool that implements solutions to all problems.

5.4. Challenges and research directions

Questions 8 and 9 aimed at specific challenges and future research directions for software visualization in the context of software maintenance, reverse engineering, and re-engineering, where question 9a targeted long-term, and question 9b mid-term research. Naturally, challenges trigger suggestions for future research, and long-term research for some people is mid-term research for others. Consequently, there are many similar answers to questions 8, 9a, and 9b, and hence for the sake of brevity, the answers to all questions will be summarized in just one section and rather be separated by the aspects and roles in software visualization introduced in Section 2. However, each noticeable difference between the answers to questions 8, 9a, and 9b will be pointed out.

Altogether 70 people out of 82 answered question 8 on challenges for software visualization. For future research questions, 51 answered question 9a and 45 answered question 9b, which shows that many researchers do not have a clear vision for future research in software visualization. Among those who answered questions 9a and 9b, nine people explicitly referred to their previous answers to either question 8 or 9a in the sense of ‘something along these lines’.

General properties

Scalability and dealing with complexity are major challenges for software visualization for software maintenance, reverse engineering, and re-engineering. Scalability concerns both the space and time complexity of visualization techniques, as, for instance, automatic layouts for large graphs, as well as the need to avoid the information overload of the viewer. Software maintenance, reverse engineering, and re-engineering typically deals with large amounts of complex data.

Scope

Scope refers to the artifacts that are to be visualized. Aspects of scope are not aspects of visualization per se. Yet, the answers to questions 8, 9a, and 9b show the inevitable mixing of the issues of scope and visualization as such, as noted by a respondent as a feedback when he read the answers to the survey published on-line [14] (the survey was intentionally vague on how to separate them). For instance, some people criticized visualization research as being independent of a problem domain and argued for visualization features that are scope-related (e.g. displaying changes, displaying all software entity dependencies) rather than visualization-related (e.g. the use of familiar metaphor, stability, scalability, etc.). In 20 cases, respondents actually gave more scope-related issues, which indicates that the term ‘software visualization’ is often interpreted with a specific scope in mind.

Representation

Many answers concerning representation issues are related to the symbology, i.e. the graphical symbols, syntax, and semantics of the visual representation. The general question for a suitable symbology is: How does one draw an answer (likely generated by a tool) using a symbology meaningful to the viewer? On the one hand, respondents call for standard and general visual representations, recognized as universes of discourse. UML is mentioned as a possible candidate, but as one respondent notes, UML would need to be extended to model procedural systems, too. For instance, a simple Pascal procedure is difficult to model in UML: methods in UML are always associated with classes and UML parameter modes for methods are insufficient for Pascal's reference parameters. One could introduce UML stereotypes to model Pascal procedures; but doing so means creating a non-standard language using UML as a metalanguage. Another respondent proposes a visualization based on the general graph metaphor which should be easy to understand and powerful. On the other hand, one respondent notes that the symbology used should be familiar to the expected viewer and gave telecommunication software as an example, in which domain-specific graphical notations are widely used that differ from notations in other software domains.

Graph visualization and automatic layout are important issues according to Bassil and Keller's survey. Similarly, Figure 5 shows that graphs are the primary means of visualizations for the respondents of this research survey. The importance of graphs and their layout is also stressed by the answers to questions 8 and 9. Scalable layouts, incremental layouts, i.e. layouts that maintain as much of the layout as possible in the face of smaller changes of the graph, and layouts that take the semantics of node and edge types into account are often demanded by the respondents. In particular, some respondents explicitly requested better layouts for UML. Yet, it is difficult to exactly specify what 'better' means. Fortunately, there is a growing number of experimental studies attempting to identify aesthetic criteria empirically [21–23].

One respondent calls for adaptable layout packages that recognize the type of input at hand and use the appropriate layout algorithm. Another one points out the importance of graphing literacy, that is, the ability to read, build, and manipulate large graphs. Another respondent complains that aesthetics are underrated in software visualization and he would hope for visualization tools that present good-looking views as opposed to ones with minimum edge crossings. For instance, he would be happier if he could read the names of all nodes in a graph rather than have a better layout according to some abstract criterion.

Interestingly enough, one respondent asks for end-user parameterization of automatic layouts, while another explicitly wants layouts without the need for parameterization.

Use of animation and 3D visualization are often requested in the answers to questions 8 and 9, which is in contrast to the ranking of functional aspects of visualization by end-users in Bassil and Keller's survey (see Section 5.3).

Specification and interaction

Most prominently, the need for multiple views in software maintenance, reverse engineering, and re-engineering is pointed out. Given multiple views, integration, synchronization, and navigation between these views while maintaining the mental map are important aspects. In particular, the connection between source code views and more abstract views derived from these needs to be maintained in both directions, which becomes difficult if the source code is changed. Capabilities like abstraction, filtering, composition, zooming, browsing, focusing, queries, and free user annotations are needed to make the information space accessible to the viewer. Beyond that, the user needs the ability to control the logic used to produce the visualization in order to speed the process of trying different combinations of techniques, as one respondent notes. In this sense, interaction means more than 'flying through one particular view of the data'; the user must be able to interactively change the view itself.

Bassil and Keller's survey likewise shows the importance of navigation between multiple views and the connection of graphical elements to source code. The ranking of functional aspects by the participants of Bassil and Keller's survey identifies the most important functional aspects (as those rated as absolutely essential by at least 50% of the respondents) as follows in decreasing order:

- search tools for graphical and/or textual elements;
- source code visualization (textual views);
- hierarchical representations;
- use of colors;
- source code browsing;
- navigation across hierarchies;
- easy access from the symbol list to the corresponding source code.

There are two interesting observations: first, many of the aspects are actually related to view source code, i.e. source code is a very important artifact for software visualization, and it confirms the importance of maintaining the connection between source code and graphical elements for more abstract artifacts. Second, only two aspects relate to visualization in a narrower sense, namely, hierarchical representation and the use of colors. The other aspects relate to interaction rather than visualization *per se*. Obviously, the term software visualization in our context is not to be interpreted in a static sense but comprises interaction with the viewer.

Medium

Because so much information needs to be presented, the output as well as the input medium are relevant factors. The screen real estate problem is mentioned as a specific challenge and whole-wall projection should be used for future visualization. The viewer should even escape the computer screen and be able to explore the information in 3D cyberspace and touch and directly manipulate the artifacts with a data-glove. In this scenario, the viewer should also be able to control the visualization through natural language (noted in question 9a as wild dreams).

Viewer

Cognitive models, real needs of viewers, and demand for experimentation are often stated as very important issues, in particular in question 8. In order to find suitable ways of visualization, we need to understand when and why certain kinds of visualization work dependent upon viewers and task at hand. How do viewers 'read' and understand visualizations? And what are their real needs? One respondent writes that he is frustrated by nice pictures about software seeking some interpretation, and, hence, the key questions he has with respect to visualization are: What mental models do developers/maintainers have about software? How can these models be best represented? What are simple mechanisms to bring them to paper or to a screen? What are solid bridges between different representations (be they visual or textual)? Experimentation and more practical experiences are demanded. Another respondent suggests integrating the experience of human computer interaction researchers into software visualization techniques.

Task

Similarly to the scope of visualization, several people give specific tasks for which they need visualization as opposed to primarily visualization-related issues. However, as noted for the aspect of the viewer, the task at hand is an important factor to be considered for a suitable visualization. A diverse range of tasks is mentioned, such as debugging, database migration, cost estimation, change impact analysis, and design recovery. As a more general goal for visualization, one respondent writes that a visualization tool should show him what he was not aware of. In other words, visualization needs to support a more fuzzy exploration process with a non-predefined or dynamically changing goal. If the question can be stated precisely and an answer be generated accurately, there is less need for visualization. Another respondent notes that the visualization should become the primary means of round-trip development, from which code is generated automatically and which reflects all necessary changes of the source code.

Integration

As shown by Figure 4, most researchers of this survey prefer to integrate only existing visualization tools, which is also reflected by the answers to questions 8 and 9 where a large number of respondents mention integration and interoperability issues as challenges and future research directions (both mid-term and long-term). Software visualization needs to be integrated with processes and tools for maintenance, reverse engineering, re-engineering, as well as forward engineering. The tools should be available to the research community, be customizable and composable, and should support standard exchange formats, such as GXL [24].

6. CONCLUSIONS

Viewers, integrators of visualization techniques, and visualization researchers are three different stakeholders in software visualization. Their concerns need to be addressed by further research.

Viewer concerns

In order to find suitable ways of visualization, we need to better understand when and why certain kinds of visualization work dependent upon viewer and task at hand. How do viewers ‘read’ and understand visualizations? And what are their real needs? All these questions call for a more thorough approach to software visualization, starting with the task and the needs of users, creating models of visual understanding, and experimenting with different kinds of visualization.

Experiments in this field, however, are difficult. They will have to take into account the viewer and the problem to be solved. Consequently, each experiment also evaluates whether the functionality offered by the visualization tool is appropriate to the problem, in other words, functionality and visualization go hand in hand.

Integrator concerns

Many researchers of this survey prefer to only integrate the existing visualization techniques, which calls for a means to achieve the required interoperability. Developing standard exchange formats and interoperability frameworks are difficult technical and, in particular, organizational problems. These problems have not been solved in other domains. Turning a research visualization prototype into a robust tool that can be integrated with other tools is a major investment and it is unclear how the investment pays off for a scientific tool builder. Overcoming the problem of high investments with unclear benefits for visualization researchers is the real challenge in integration and interoperability. Nevertheless, there are first achievements to report. The reverse engineering and re-engineering community, for instance, has recently adopted GXL as a standard interchange format [24]. The graph transformation community is part of this movement. There are ongoing discussions with the graph drawing community to further broaden the interchange format.

Concerns for visualization for maintenance, reverse engineering, and re-engineering

Some respondents call for standard representations of visualizations for visualization for maintenance, reverse engineering, and re-engineering, recognized as universes of discourse. Though standardization would offer many advantages, we can realistically (similarly to programming languages) hardly expect one graphical representation for all domains and purposes. One can simply recall the variety of artifacts that are to be visualized and the tasks that are to be supported by visualization, as shown by this survey. We might, however, at least find standards for subdomains. For instance, the promise of UML is to offer a standard notation for the design of object-oriented systems.

Because a standard graphical representation for all needs and purposes will hardly be found, we will need to cope with different representations. Suitability of certain representations for specific aspects, integration of different representations, and transformations between representations will then be interesting research challenges.

The most important challenge for constructing and interacting with the visualization are the many different views that arise in maintenance, reverse engineering, and re-engineering. Multiple views are not really specific to these domains, but can be found in information visualization in general (of which software visualization is a part). One more specific aspect of software visualization is the connection between source code views and more abstract views that needs to be maintained in both directions. Source code is suggested by this survey to be one of the most important artifacts.

This research survey has revealed a tendency to actually extend software visualization to what might be paraphrased as *software perception*. Beyond screen, paper, or whole-wall projections, researchers propose to make better use of other human senses through virtual reality technology. The term software visualization is extended here substantially. With our current technology many of these wishes could be realized today. The research challenges are then to find suitable metaphors for this virtual reality (as software does not have a real gestalt) and to experiment whether the augmented visualization is helpful at all. The respondents of Bassil and Keller's survey—at any rate—are sceptical. The use of virtual reality techniques is the least wanted functional aspect in their survey. In particular, the practitioners agree in this opinion; some researchers rate virtual reality higher.

I conclude with a quote from a respondent with whom the author agrees:

> '*I tend to think of visualization as the interface between the mind and the world to understand. [...]my dream visualization utilizes my mind as best as possible to understand the complexity of large software systems. To think that a visualization reduces this complexity is absurd, but it is reasonable to dream of making it maximally easy to comprehend using the cognitive resources available.*'

ACKNOWLEDGEMENTS

All participants of this research survey are thanked.

REFERENCES

1. Roman G-C, Cox KC. Program visualization: The art of mapping programs to pictures. *Proceedings of the International Conference on Software Engineering*. ACM Press: New York, 1992; 412–420.
2. Hendrix TD, Cross JH, Maghsoodloo S, McKinney ML. Do visualizations improve program comprehensibility experiments with control structure diagrams for Java? *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*. ACM Press: New York, 2000; 382–386.
3. Curtis B, Sheppard SB, Kruesi-Bailey E, Bailey J, Boehm-Davis DA. Experimental evaluation of software documentation formats. *Journal of Systems and Software* 1989; **9**(2):167–207.
4. Green TRG, Petre M. When visual programs are harder to read than textual programs. *Proceedings of the 6th European Conference on Cognitive Ergonomics*. Springer: Berlin, 1992; 167–180.
5. Fjeldstad RK, Hamlen WT. Application program maintenance study: Report to our respondents. *Tutorial on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1982; 13–30.
6. Diehl S (ed.). *Software Visualization. International Seminar, Dagstuhl Castle, Revised Papers*. Springer: Berlin, 2002.
7. Bassil S, Keller RK. Software visualization tools: Survey and analysis. *Proceedings International Workshop on Program Comprehension*. IEEE Computer Society Press: Los Alamitos CA, 2001; 7–17.
8. Whitley KN. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing* 1997; **8**(1):109–142.
9. Gilmore DJ, Green TRG. Comprehension and recall of miniature programs. *International Journal of Man–Machine Studies* 1984; **21**(1):31–48.
10. Blackwell AF. Metacognitive theories of visual programming: What do we think we are doing? *Proceedings IEEE Symposium on Visual Languages*. IEEE Computer Society Press: Los Alamitos CA, 1996; 240–246.
11. Brown MH. *Algorithm Animation*. MIT Press: Cambridge MA, 1988.
12. Knight C, Munro M. Mediating diverse visualizations for comprehensions. *Proceedings International Workshop on Program Comprehension*. IEEE Computer Society Press: Los Alamitos CA, 2001; 18–25.
13. Wiggins M. An overview of program visualization tools and systems. *Proceedings of the 36th Annual Conference on Southeast Regional Conference*. ACM Press: New York NY, 1998; 194–200.
14. Koschke R. Software visualization in software maintenance, reverse engineering, and reengineering—a research survey. http://www.bauhaus-stuttgart.de/softviz [23 July 2002].
15. Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 1990; **12**(1):26–60.
16. Selic B, Gullekson G, Ward PT. *Real-Time Object-Oriented Modeling*. Wiley: Chichester, 1994.
17. Storey M-A, Wong K, Müller H. How do program understanding tools affect how programmers understand programs? *Proceedings Working Conference on Reverse Engineering*. IEEE Computer Society Press: Los Alamitos CA, 1997; 12–21.
18. Müller H, Wong K, Tilley S. A reverse engineering environment spatial and visual software interconnection models. *ACM SIGSOFT Symposium Software Development Environments*. ACM Press: New York, 1992; 88–98.
19. Petre M. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM* 1995; **38**(6):33–44.
20. Quigley A, Eades P. Graph drawing, clustering, and visual abstraction. *Proceedings of the 8th International Symposium on Graph Drawing* (*Lecture Notes in Computer Science*, vol. 1984). Springer: Berlin, 2000; 197–210.
21. Ware C, Hui D, Franck G. Visualizing object-oriented software in three dimensions. *Proceedings of the IBM Centre for Advanced Studies Conference (CASCON)*. IBM: Toronto, Canada, 1993; 612–620.
22. Ware C, Franck G. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Transactions on Graphics* 1996; **15**(2):121–140.
23. Purchase H, Cohen RF, James MI. Validating graph drawing aesthetics. *Proceedings of the International Symposium on Graph Drawing* (*Lecture Notes in Computer Science*, vol. 1027). Springer: Berlin, 1995; 435–446.
24. Holt R, Winter A, Schür A. GXL: Toward a standard exchange format. *Proceedings 7th Working Conference on Reverse Engineering*. IEEE Computer Society Press: Los Alamitos CA, 2000; 162–171.

AUTHOR'S BIOGRAPHY

Rainer Koschke is a post-doctoral researcher in the Computer Science Department at the University of Stuttgart. His research interests are primarily in the fields of software engineering and program analyses. His current research includes architecture recovery, feature location, program analyses, and reverse engineering. He teaches re-engineering, compilers, and programming language concepts. He holds a doctoral degree in computer science from the University of Stuttgart, Germany.