Requirements of Software Visualization Tools: A Literature Survey

Holger M. Kienle and Hausi A. Müller
University of Victoria, Canada
hkienle@acm.org and hausi@cs.uvic.ca

Abstract

Objective:[1] *Our objective is to identify requirements (i.e., quality attributes and functional requirements) for software visualization tools. We especially focus on requirements for research tools that target the domains of visualization for software maintenance, reengineering, and reverse engineering.*

Method: *The requirements are identified with a comprehensive literature survey based on relevant publications in journals, conference proceedings, and theses.*

Results: *The literature survey has identified seven quality attributes (i.e., rendering scalability, information scalability, interoperability, customizability, interactivity, usability, and adoptability) and seven functional requirements (i.e., views, abstraction, search, filters, code proximity, automatic layouts, and undo/history).*

Conclusions: *The identified requirements are useful for researchers in the software visualization field to build and evaluate tools, and to reason about the domain of software visualization.*

1. Introduction

The identification of requirements—explicitly documented or not—is an important part of each software development project. This is also true for the construction of software visualization tools, regardless whether the objective is building of an industrial-strength tool or a research prototype.

Software systems built for a certain domain often have to meet established "base-line" requirements of that domain. These base-line requirements emerge over time and are agreed upon by the domain's experts. The goal of this paper is to identify base-line requirements for the domain of software visualization by means of a literature survey. We assume that requirements reported (independently) by several expert sources are a good indication that tools should meet them in order to be useful and fulfill the expectations of users. The survey's requirements are objective in the sense that they are based on a comprehensive literature review that covers the opinions stated by many experts of the domain. For the literature survey, we have mostly focused on software visualization for software maintenance, reengineering, and reverse engineering.

[1]The formatting of the abstract follows the suggestion of evidence-based software engineering [30].

Even though we chose the term requirements, it is not meant to be interpreted in an overly restrictive sense. As argued by Carney and Wallnau, the terms "preference" or "desire" might be more suitable [12]. Furthermore, builders of software visualization tools cannot expect to meet all requirements equally well. In the words of Bass et al., "no quality can be maximized in a system without sacrificing some other quality or qualities" [5, p. 75]. Following this line of thought, a software visualization tool that is missing some of the requirements identified can still be satisfactory.

The requirements exposed by our survey can contribute to advance software visualization in several ways. Firstly, builders of software visualization tools can use the survey's requirements as a starting point for requirements elicitation of their tools. Specifically, the survey can be used as a checklist to make sure that no important requirements have been missed, thus increasing the confidence of the developers. Secondly, the survey can be used to pick suitable requirements for measuring and comparing tools. An evaluation can pick and (formally) assess a set of accepted requirements to convince fellow researchers of a tool's adequacy or superiority compared to another tool. Lastly, (idealized) requirements can drive tool research. Storey et al. believe that "surveys in any research discipline play an important role as they help researchers synthesize key research results and expose new areas for research" [59]. This survey reflects the current state of tool requirements and thus can serve to identify open and new requirements for future research directions.

The paper is organized as follows. Sec. 2 describes the research approach that we followed when conducting the literature survey. Sections 3 and 4 discuss the requirements uncovered by the survey, separating them into quality attributes and functional requirements. Sec. 5 draws conclusions and points out future work.

2. Followed Research Approach

Often surveys and reviews are conducted, or frameworks are proposed without explicitly stating the process of their creation. This makes it difficult for other researchers to judge their scope, applicability, and validity. The requirements in this survey have been identified following a process proposed by Kitchenham et al. called evidence-based software engineering (EBSE) [30].

Step	EBSE [30]	Our survey
1	Converting the need for information into an answerable question.	What requirements should software visualization tools satisfy?
2	Tracking down the best evidence with which to answer that question.	Literature search of selected publications (with authors' domain knowledge).
3	Critically appraising that evidence for its validity, impact, and applicability.	Review and classification of requirements.
4	Integrating the critical appraisal with our software engineering expertise and with our stakeholders' values and circumstances.	Application of the survey to assess several reverse engineering tools [29].
5	Evaluating our effectiveness and efficiency in executing Steps 1–4 and seeking ways to improve them both for next time.	Assessment of survey coverage; feedback from other researchers.

Table 1. The five steps of EBSE and how they are implemented by this paper's survey

The five steps needed to practice EBSE are summarized in Table 1. Whereas the goal of EBSE is quite broad, its process is applicable to guide the survey. In the following, we briefly discuss each of EBSE's steps and how it was implemented by our survey. A more detailed description can be found in the first author's dissertation [29].

As the first step an answerable research question is defined. The question should be general enough so that a sufficient body of work applies. Our research question is: What (non-functional) requirements should software visualization tools satisfy? Step 2 involves identifying the relevant work that helps in answering the question posed in Step 1. The survey follows the ideas of *systematic review*, which defines an appropriate search method to ensure a sufficient coverage of the literature. To structure the survey, in step 3 the identified requirements of individual sources were grouped into suitable categories. At the top-level we distinguish between quality attributes and functional requirements. In step 4, we have so far applied the identified requirements to assess the quality of reverse engineering tools built from components [29]. Otherwise, the survey currently has no explicit stakeholders; we hope for other researchers to provide feedback and report on the usefulness of the survey for their own research.

It is difficult to evaluate literature surveys of this kind (step 5) because an exhaustive search is impossible. However, we believe that the survey's coverage of sources is representative of the domain of software visualization research. Whereas the survey certainly has missed sources, it succeeded to identify the critical tool requirements. We encourage other researchers to identify missed sources and to critique the survey.

3. Quality Attributes

This section describes the seven quality attributes identified by the survey. Due to space constraints, we only report here a subset of the sources that our literature search identified.

Rendering Scalability A survey among users of software visualization tools has found that there is equal weight on the visualization of small, medium, and large target systems [6]. Thus, in order to accommodate typical user needs, a visualizer's rendering speed should scale up to large amounts of data. This is especially the case for interactive systems that allow direct manipulation of graphs or diagrams. Storey et al., reporting on a user study with the SHriMP visualizer, found the following: "The single most important problem with SHriMP views was the slow response of the interface. Since SHriMP views are based on direct manipulation, users expecting immediacy were disturbed by the slow response" [62]. Armstrong and Trudeau state in their tool evaluation that "fast graph loading and drawing is essential to the usability of any visualization tool" [1]. Bellay and Gall use "speed of generation" of textual and graphical reports as an assessment criterion and experienced that "graphical views often need an unacceptable amount of time to be generated because of the layout algorithms" [7]. Bassil and Keller's survey on software visualization tools includes "tool performance (execution speed of the tool)" as a practical aspect [6, P5].

Information Scalability Besides rendering speed, there is another scalability issue: cognitive overload of the user caused by "overplotting." For example, hypertext navigation of reverse engineering information can suffer if the "information web" gets too large: "As the size of this web grows, the well-known 'lost in hyperspace' syndrome limits its navigational efficiency" [69, sec. 3.3.1]. The authors of the REforDI reengineering environment say, "very soon we recognized that due to the limited perceptivity of human beings it is very important to be able to reduce and extend the amount of visualized information on demand" [13]. Similarly, Reiss says that "practical program visualization must provide tools to select and display just the information of interest" [49]. Examples of techniques to accomplish these goals are abstraction and filters (cf. Sec. 4). Koschke explicitly identifies both rendering and information scalability problems as major research challenges: "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" [32].

Interoperability For tools to interoperate, they have to agree on a suitable interoperability mechanism in some form or another. As a consequence, research papers often directly address the question of *how* to achieve interoperability, without explicit stating it as a requirement first. Researchers that leverage software visualizations stress the importance of interoperability because it enables them to reuse existing functionality, for instance to (opportunistically) assemble a tool set that supports a particular reverse engineering task or process [9].

A lightweight interoperability mechanism used by many tools are (graph) exchange formats [28]. Exchange formats use simple files as a (temporary) repository for information transfer. The coupling between tools that exchange information is loose; an information producer need not to know its information consumers. Examples of popular formats to represent graphs for visualizing software structures are RSF, TA, and GXL.

Researchers have proposed different approaches to make tools more interoperable. For example, Panas et al. have implemented an object-oriented tools framework in Java [44]. Tools have to implement certain interfaces to plug into the infrastructure. The SHriMP tool uses a JavaBeans architecture [8]. This approach greatly reduced the effort to integrate it with the Eclipse framework and the Protégé knowledge-management tool. Furthermore, new exchange formats can be supported by implementing a new instance of SHriMP's Persistent Storage Bean. The Common Object-based Reengineering Unified Model (CORUM) [77] and CORUM II [27] [76] proposals strive to provide a common framework, or middleware architecture, for reverse engineering tools, encompassing standard APIs and schemas for ASTs, CFGs, call graphs, symbol tables, metrics information, execution traces, and so on. However, the more sophisticated the interoperability mechanism, the more standardization and agreement among tools is necessary.

Customizability Typically, tools enable customization of their functionalities via configuration files, built-in scripting support, or programmable interfaces [36]. Customizability enables to meet needs that cannot be foreseen by tool developers, for instance, if a tool is applied in a new context.

In Bassil and Keller's survey, 45% of the respondents rated the "possibility to customize visualization" as "absolutely essential" [6, F29]. The developers of the Sextant software comprehension tool say, "we conclude that software exploration tools should be extensible to accommodate for domain-specific navigation elements and relationships as needed" [17]. Even though customization seems important, tools are lacking in this respect [59]. Wang et al. conclude that "existing visualization tools typically do not allow easy extension by new visualization techniques" [72]. They have developed the EVolve software visualization framework, which "is extensible in the sense that it is very easy to integrate new data sources and new kinds of visualizations." A new visualization is realized by extending the EVolve Java framework, which already provides abstractions for bar charts, tables, dot-plots, and so on. Similarly, Reiss has analyzed why software understanding tools are often not used and concludes that "the primary reason was that they failed to address the actual issues that arise in software understanding. In particular, they provided fixed views of a fixed set of information and were not flexible enough" [50]. He also states that "since we cannot anticipate all visualizations initially, it should be easy to add new graphical objects to the system" [48]. Among the requirements that Favre states for his G^{SEE} software exploration environment is the need for "customizable exploration" [18]. Tilley states that "users need to be able to impose their own personal taste on the user interface ... The goal of environmental customizability includes modification of the system's interface components such as buttons, dialogs, menus, [and] scrollbars" [67, sec. 4.3.3.3].

Interactivity Many applications of software visualization are interactive and explorative in nature. For example, Marshall et al. state that tools that render graphs must allow the user "to manipulate it, in addition to navigate it" [38]. Wong states that tools supporting reverse engineering activities should "provide interactive... views, with the user in control" [74, Req. 15]. Koschke says 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" [32]. However, tool interactivity has to be balanced with suitable automation, otherwise significant reverse engineering tasks might suffer from the large number of required user interactions [13].

Among the tool features that Storey recommends for program comprehension tools are various forms of "browsing support" [57]. In her cognitive design framework, she also requires tools to "provide arbitrary navigation"[58, p. 149]. In the context of graph visualization, Herman et al. state that "navigation and interaction facilities are essential" [24].

Ducasse et al. say that "the exploratory nature of reverse engineering and reengineering demands that a reengineering environment does not impose rigid sequences of activities" [66, p. 52]. They give examples of how this requirement can be met, for instance, with the rapid presentation of source code in both textual and graphical views, allowing actions on views such as switching between different abstraction levels, and easy querying of entities. Brown gives several design consideration for program understanding tools, among them "flexibility of end-user navigation" [10]. His CodeNavigator tool "has a free-form navigation style that allows the user to control the scope and flow of investigations." Similarly, Froehlich and Dourish say about their tool, "rather than encoding specific workflows, we provide a visual tool that allows developers to explore views of their system" [20].

Usability Usability is a highly desirable requirement, but unfortunately very difficult to meet. In Bassil and Keller's survey, the requirement "ease of using the tool (e.g., no cumbersome functionality)" was selected as the second-most important practical aspect that software visualization tools should meet, which 72% rated as "very important" [6, P8]. The survey's researchers report that "unfortunately, we found a disturbing gap between the high importance attached to the two aspects ease of use and quality of the user interface, and the ratings of these qualities in the software visualization tools in practical use" [6].

The evaluation of the usability of reverse engineering tools is often focused on the user interface. In Bassil and Keller's survey, 69% believe that "quality of user interface (intuitive widgets)" is a very important requirement [6, P12]. Reiss, who has implemented many software engineering tools, believes that "having a good user interface is essential to providing a usable tool" [52]. He provides a rationalization for the user interface choices of the CLIME tool, but does not support his decisions with background literature. For the BLOOM software visualizer, Reiss emphasizes that both usefulness and usability are important: "While it is important to provide a wide range of different analyses and to permit these to be combined in order to do realistic software understanding, it is equally important to offer users an intuitive and easily used interface that lets them select and specify what information is relevant to their particular problem" [50].

Researchers have also suggested design heuristics to make a tool more usable. Storey's cognitive framework emphasizes usefulness aspects for comprehension tools, but also has a design element that requires to "reduce UI cognitive overhead" [58, p. 149]. Storey et al. further elaborate on this design element, stating that "poorly designed interfaces will of course induce extra overhead. Available functionality should be visible and relevant and should not impede the more cognitively challenging task of understanding a program" [60]. According to Reiss, a tool's usage should "have low overhead and be unintrusive" [51]. Especially, it should not interfere with existing tools or work practices. In order to keep reverse engineers motivated, tools should be enjoyable to use. Especially, "do not automate-away enjoyable activities and leave only boring ones" [35].

Adoptability Researchers have reported on many diverse factors—both technical and non-technical—that influence the chances of a tool to get adopted. Storey says, "although there are many software exploration tools in existence, few of them are successful in industry. This is because many of these tools do not support the right tasks" [58, p. 154]. Bull et al. state, "in any field, ease of use and adaptability to the tasks at hand are what causes a tool to be adopted" [11]. Wong believes that "lightweight tools that are specialized or adaptable to do a few things very well may be needed for easier technology insertion" [73]. He also says, "by making tools programmable, they can more easily be incorporated into other toolsets, thus easing an adoption issue of tool compatibility" [74, p. 94].

Zayour and Lethbridge point out that reverse engineering tools face an additional adoption hurdle because "their adoption is generally 'optional' " [78] in the sense that they are not a necessity to complete, say, a maintenance task. They propose a method based on cognitive analysis to improve tool adoption, assuming that a tool that reduces the cognitive load will positively influences the users' perception of ease of use and usefulness and thus improve tool adoption. The ACSE project (`www.acse.cs.uvic.ca`) identifies a number of factors that affect the adoption of reverse engineering tools, which are equally applicable to software visualizers [4]. Tilley et al. suggest that research tools might be more adoptable if they were more understandable, robust, and complete [68]. Reiss says that developers will adopt tools only if "the cost of learning that tool does not exceed its expected rewards and the tool has been and can easily shown to provide real benefits" [53].

Singer lists a number of desirable qualities for tools to improve their use; tools should be unobtrusive and lightweight; have subtle feedback and small real estate; and require no setup and training [56]. In contrast to most research tools that require some effort and expertise for installation, popular tools are easy installed or even already pre-installed [39]. Devanbu stresses that tools have to integrate smoothly into an existing process: "Software tools work best if their operation is well-tuned to existing processes. Thus, a tool that generates paper output may not be helpful in an email-based culture" [14]. Similarly, Wong says "software understanding techniques and tools need to be packaged effectively and made compatible with existing processes, users, and tools" [74, p. 93].

4. Functional Requirements

Researchers have also identified functional requirements for visualization tools. We summarize them briefly by grouping them into the following seven requirements.

Views Software visualizers typically provide different views of the target software system (e.g., to satisfy the need of different stakeholders and to emphasize different dimensions of the data such as the time dimension or level of abstraction) [31] [55]. An example of integrated views is provided by Kazman's SAAMtool, where "any node within any view can have links to nodes in one or more of the other views" [26].

Wong requires visualizers to provide views that are consistent and integrated [74, Req. 15]. He further requires to "integrate graphical and textual software views, where effective and appropriate" [74, Req. 20]. The developers of the Rigi system advocate *dynamic views* [75]. In contrast to static views, which only provide information that has been obtained at a certain point in time, dynamic views synchronize and recompute their information when the underlying data source changes. Blaine et al.'s software visualization taxonomy asks, "to what degrees can the system provide multiple synchronized views of different parts of the software being visualized?" [47, C.4]. Storey says, "programming environments should provide different ways of visualizing programs. . . . orthogonal views, if easily accessible, can facilitate comprehension, especially when combined" [57]. Similarly, Meyers states that "there is a crucial need for programming environments that facilitate the creation of multiple views of systems" [40].

Koschke has conducted a survey about software visualization; he summarizes the responses as follows: "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" [32]. Bassil and Keller's survey of software visualization tools consider "synchronized graphical and textual browsing," which rated 62% of the respondent as "useful, but not essential" [6, F7]. Hainaut et al. state the requirement that views should address different levels of granularity: "Multiple textual and graphical views, summary and fine-grained presentations must be available" [22].

Abstraction Most visualizations are based on an underlying graph model. Since the resulting graphs can become too complex to effectively visualize (cf. Sec. 3), even for small software systems, abstraction mechanisms are needed. Von Mayrhauser and Vans conclude from their studies that "maintenance programmers work at all levels of abstraction" [70] and consequently stress the "importance of the availability of information about the program at different levels of abstraction" [71]. Hierarchical graphs are a popular abstraction mechanism [25]. They allow grouping of nodes to create higher levels of abstractions, resulting in a layered view [7]. Ducasse and Tichelaar believe that "grouping several entities together is an important technique in reengineering, primarily to build higher-level abstractions from low-level program elements or to categorize elements with a certain property" [16]. According to Kazman, "a tool for software architecture should be able to aggregate architectural elements recursively" [26]. Rigi is an example of a tool that fulfills this requirement. It has "support for abstraction activities (grouping/collapsing) with an easy to use GUI" [54]. Similarly, the SHriMP tool fulfills the requirement to "provide abstraction mechanism" by providing "subsystem nodes and composite arcs in the nested graph" [60, E3]. There are many more examples of tools that provide similar abstraction mechanisms (e.g., EDGE [43] [45], daVinci [21], and Balmas' work [2] [3]). In Bassil and Keller's survey about software visualization, "hierarchical representations (of subsystems, classes, etc.)" was the third-most useful functional aspect, rated by 57% of the respondents as "absolutely essential" [6, F18]. Lastly, abstractions are also possible in the presentation of dynamic information. For instance, tools such as ISVis allow the reverse engineer to "view the trace content at different levels of abstraction (e.g., object interactions, class interaction)" [23].

Search Evidence of the importance of searching in software visualization is provided by a user study conducted by Storey et al., in which the authors observed that "in Rigi and SHriMP, the lack of a searching tool to find text strings in the source code definitely hindered the users" [63]. In Bassil and Keller's survey, "search tools for graphical and/or textual elements" was rated as the most useful functional aspect of software visualization tools; 74% of the respondents classified it as "absolutely essential" [6, F14].

However, visualization tools often have rather limited searching capabilities that go no further than searching for the labels of graph entities.[2] Storey et al. have surveyed a number of visualization tools and come to the conclusion that there is need for improved querying support [59].

Filters Filtering of information in visualization tools can be seen as a rudimentary form of (structural) querying [59]. It is an effective approach of graph visualizations to prune nodes or arcs (based on their types, or other criteria), and is supported by tools such as Rigi and SHriMP [63] [64]. According to Storey's research, filtering helps to reduce disorientation effects by users [60, E14]. According to Riva's experience, an important feature in Rigi is the "possibility to filer/unfilter information. This allows us to reduce the amount of visualized data and to limit our analysis" [54]. Maletic et al. describe requirements of visualization tools; one of these is to "filter out uninteresting items" [37]. Other

[2] Storey et al. report that whereas this "search on node labels was very useful" for the users of Rigi and SHriMP, there was also the problem that "some users mistakenly thought they were searching for strings in the code rather than searching for node labels in the graph" [63, sec. 6.3].

examples of filtering are pruning of traces or call trees based on specific categories [23] [70].

Code Proximity Code proximity means the ability of the visualizer to provide easy and fast access to the original, underlying source code [33]. One of the criteria in Bellay and Gall's reverse engineering tool evaluation addresses code proximity as follows: "Point and click movement from reports to source code allows to access the source code representation of a specific entity in a report. The browsing of the source code can be seen as switching the abstraction level and is done often during code comprehension" [7]. Ducasse et al. state, "to minimize the distance between the representation of an entity and the actual entity in the source code, an environment should provide every entity with a direct linkage to its source code" [15].

Most reverse engineering and reengineering tools fulfill this requirement, but differ in their capabilities. For instance, Rigi supports the navigation from a node in the graph to a source file and line number, while SHriMP can visualize the source directly within the node. The TkSee search tool provides search results of different kinds (e.g., files, functions, variables, and lines in a file). For each kind of search result, the tool user has "instant access to the source code" [34, p. 78]. Code proximity is just a special case of a certain abstraction mapping. There are other mappings (e.g., between the application and implementation domains), which are also important but less well explored by researchers [41].

Automatic Layouts Bassil and Keller's survey asked for the importance of "automatic layout capabilities" as one of the functional aspects of visualization tools. About 40% of the survey's respondents rated this as absolutely essential [6, F10]. In fact, many visualization are less effective or practically impossible to construct without automatic layouts. Even if automatic layouts cannot be perfect, they provide a good starting point for further, manual refinement. Layouts often strive to optimize certain properties of the graph [42] [65]. Bellay and Gall say, "special layout algorithms can make a graph more readable; for example, minimizing the crossings of the arcs" [7]. The SHriMP layout adjustment algorithms allows to apply a layout without adversely affecting the layout of the original graph [58]. Different layout approaches can represent the same data quite differently (and influence their interpretation [46]).[3] For instance, hierarchical graphs can be visualized in a single window as a nested view (e.g., SHriMP), as a tree view (e.g., Rigi), or in separate views showing a single nesting only (e.g., Rigi). There are also 3D layouts such as Reiss' Plum framework [49].

[3]Riva states about the Rigi tool that it has "support for automatic layout of graphs according to different algorithms. This allows us to look at the data from different perspectives" [54].

In Koschke's survey, 78% of the users who visualize graphs are using automated layouts. This seems to support the claim that they constitute an important requirement. However, respondents complained about deficiencies such as "lack of incremental, semantic, and scalable layouts, bad space consumption, and lack of user control" [32]. Koschke also raises the issue that layout algorithms should take semantics of nodes and edges into account (e.g., in a UML class model one would expect that all inheritance links point in the same direction) [31]. Koschke recommends that the reverse engineering community should collaborate more actively with graph drawing researchers [32].

Undo/History Since users perform interactive manipulations that change the visualization, there should be an undo mechanism that allows them to revert to previous states. Ducasse et al. name a history mechanism, which should contain all previously performed steps of graph operations [15]. In the LSEdit graph editor, "any edit that can be done can be also undone. Any edit that has been undone can also be re-done" [64]. Maletic et al. say visualizers should "keep a history of actions to support undo, replay, and progressive refinement" [37]. Bassil and Keller's survey of software visualization tools consider both the "possibility to record the step of an arbitrary navigation" and the "possibility to return back to preceding navigation steps" [6, F22/23]. Based on their research in program comprehension, Mayrhauser and Vans believe that a tool should have a "history of browsed locations" [70].

Miscellaneous Other requirements mentioned by researchers are the use of colors [47] [6, F32], the ability to place annotations on graphical entities [59] [7] [60] or code [70], and to manipulate graphs, specifically zooming [6, F11] [15] [59] [37], panning [61] [24], deleting/editing of view entities [15] [7], and saving [6, F24] [37] [60].

5. Conclusions and Future Work

This paper has identified quality attributes and functional requirements for software visualization tools by means of a comprehensive literature survey. To our knowledge, this is the first attempt of a literature survey with the goal to arrive at a comprehensive list of requirements for this particular domain. Furthermore, the reported requirements are objectively derived in the sense that they represent the current consensus among researchers as reflected in the literature of the field.

This paper represents a snapshot of our work towards a comprehensive and complete set of requirements that is agreed on by domain experts. We plan to evolve our list of requirements by soliciting feedback from experts. This can be done, for example, with the help of an (online) questionnaire, which could point out requirements that have been missed or that are not universally agreed upon.

Furthermore, we belive it would be useful to expand this work towards construction of a domain-specific quality model for software visualization tools. Such a quality model would take our elicited requirements as a starting point and augment them with metrics and benchmarks so that they can be quantified [19]. The resulting model can then be used for tool assessments and comparisons.

References

[1] M. N. Armstrong and C. Trudeau. Evaluating architectural extractors. *5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 30–39, Oct. 1998.

[2] F. Balmas. Displaying dependence graphs: a hierarchical approach. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 261–270, Oct. 2001.

[3] F. Balmas. Displaying dependence graphs: a hierarchical approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(3):151–185, 2004.

[4] R. Balzer, J.-H. Jahnke, M. Litoiu, H. A. Müller, D. B. Smith, M. Storey, S. R. Tilley, K. Wong, and A. Weber. 3rd international workshop on adoption-centric software engineering. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 1–2, May 2003.

[5] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 1998.

[6] S. Bassil and R. K. Keller. Software visualization tools: Survey and analysis. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 7–17, May 2001.

[7] B. Bellay and H. Gall. An evaluation of reverse engineering tool capabilities. *Journal of Software Maintenance: Research and Practice*, 10(5):305–331, Sept./Oct. 1998.

[8] C. Best. Designing a component-based framework for a domain independnet visualization tool. Master's thesis, Department of Computer Science, University of Victoria, 2002.

[9] I. T. Bowman, M. W. Godfrey, and R. C. Holt. Connecting software architecture recovery frameworks. *1st International Symposium on Constructing Software Engineering Tools (CoSET'99)*, May 1999.

[10] P. Brown. Integrated hypertext and program understanding tools. *IBM Systems Journal*, 30(3):363–392, Mar. 1991.

[11] R. I. Bull, A. Trevors, A. J. Malton, and M. W. Godfrey. Semantic grep: Regular expressions + relational abstraction. *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pages 267–276, Oct. 2002.

[12] D. J. Carney and K. C. Wallnau. A basis for evaluation of commercial software. *Information and Software Technology*, 40(14):851–860, Dec. 1998.

[13] K. Cremer, A. Marburger, and B. Westfechtel. Graph-based tools for re-engineering. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(4):257–292, July/Aug. 2002.

[14] P. T. Devanbu. Re-targetability in software tools. *ACM SIGAPP Applied Computing Review*, 7(3):19–26, Fall 1999.

[15] S. Ducasse, M. Lanza, and S. Tichelaar. The Moose reengineering environment. *Smalltalk Chronicles*, 3(2), Aug. 2001.

[16] S. Ducasse and S. Tichelaar. Dimensions of reengineering environment infrastructures. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(5):345–373, Sept./Oct. 2003.

[17] M. Eichberg, M. Haupt, M. Mezini, and T. Schäfer. Comprehensive software understanding with Sextant. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 315–324, Sept. 2005.

[18] J. Favre. G^{SEE}: a generic software exploration environment. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 233–244, May 2001.

[19] X. Franch and J. P. Carvallo. Using quality models in software package selection. *IEEE Software*, 20(1):34–41, Jan./Feb. 2003.

[20] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development. *26th ACM/IEEE International Conference on Software Engineering (ICSE'04)*, pages 387–396, May 2004.

[21] M. Fröhlich and M. Werner. The graph visualization system daVinci—a user interface for applications. Technical Report 5/94, Department of Computer Science, University of Bremen, 1994. `ftp://ftp.tzi.de/tzi/biss/daVinci/papers/techrep0594.ps.gz`.

[22] J. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. Requirements for information system reverse engineering support. *2nd IEEE Working Conference on Reverse Engineering (WCRE'95)*, pages 136–145, July 1995.

[23] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools. *Conference of The Centre for Advanced Studies On Collaborative Research (CASCON'04)*, pages 42–55, Oct. 2004.

[24] I. Herman, G. Melancon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, Jan.–Mar. 2000.

[25] J. H. Jahnke, H. A. Müller, A. Walenstein, N. Mansurov, and K. Wong. Fused data-centric visualizations for software evolution environments. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 187–196, June 2002.

[26] R. Kazman. Tool support for architecture analysis and design. *2nd International Software Architecture Workshop (ISAW-2)*, pages 94–97, Oct. 1996.

[27] R. Kazman, S. G. Woods, and S. J. Carriere. Requirements for integrating software architecture and reengineering models: CORUM II. *5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 154–163, Oct. 1998.

[28] H. M. Kienle. Exchange format bibliography. *ACM SIGSOFT Software Engineering Notes*, 26(1):56–60, Jan. 2001.

[29] H. M. Kienle. *Building Reverse Engineering Tools with Software Components*. PhD thesis, Department of Computer Science, University of Victoria, Nov. 2006. `https://dspace.library.uvic.ca:8443/handle/1828/115`.

[30] B. A. Kitchenham, T. Dyba, and M. Jorgensen. Evidence-based software engineering. *26th ACM/IEEE International Conference on Software Engineering (ICSE'04)*, pages 273–281, May 2004.

[31] R. Koschke. Software visualization for reverse engineering. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 138–150. Springer-Verlag, 2002.

[32] R. Koschke. Software visualization in software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, Mar. 2003.

[33] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.

[34] T. C. Lethbridge and F. Herrera. Assessing the usefulness of the tksee software exploration tool. In H. Erdogmus and O. Tanir, editors, *Advances in Software Engineering: Topics in Comprehension, Evolution, and Evaluation*, chapter 11, pages 73–93. Springer-Verlag, Dec. 2001.

[35] T. C. Lethbridge and J. Singer. Strategies for studying maintenance. *2nd Workshop on Empirical Studies of Software Maintenance (WESS'96)*, pages 79–83, Nov. 1996.

[36] J. Ma, H. M. Kienle, P. Kaminski, A. Weber, and M. Litoiu. Customizing Lotus Notes to build software engineering tools. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'03)*, pages 276–287, Oct. 2003.

[37] J. I. Maletic, A. Marcus, and M. L. Collard. A task oriented view of software visualization. *1st IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*, pages 32–42, June 2002.

[38] M. S. Marshall, I. Herman, and G. Melancon. An object-oriented design for graph visualization. *Software—Practice and Experience*, 31(8):739–756, July 2001.

[39] J. Martin. Tool adoption: A software developer's perspective. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 7–9, May 2003.

[40] S. Meyers. Difficulties in integrating multiple development systems. *IEEE Software*, 8(1):49–57, Jan. 1991.

[41] H. Müller, J. Jahnke, D. Smith, M. Storey, S. Tilley, and K. Wong. Reverse engineering: A roadmap. *Conference on The Future of Software Engineering*, pages 49–60, June 2000.

[42] P. Mutzel and P. Eades. Graphs in software visualization: Introduction. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 285–294. Springer-Verlag, 2002.

[43] F. J. Newbery. An interface description language for graph editors. *IEEE Symposium on Visual Languages (VL'88)*, pages 144–149, Oct. 1988.

[44] T. Panas, J. Lundberg, and W. Löwe. Reuse in reverse engineering. *12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 52–61, June 2004.

[45] F. N. Paulish and W. F. Tichy. Edge: An extensible graph editor. *Software—Practice and Experience*, 20(S1):S1/63–S1/88, June 1990.

[46] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.

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

[48] S. P. Reiss. A framework for abstract 3D visualization. *IEEE Symposium on Visual Languages (VL'93)*, pages 108–115, Aug. 1993.

[49] S. P. Reiss. An engine for the 3D visualization of program information. *Journal of Visual Languages and Computing*, 6(3):299–323, Sept. 1995.

[50] S. P. Reiss. An overview of BLOOM. *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 38–45, June 2001.

[51] S. P. Reiss. Constraining software evolution. *18th IEEE International Conference on Software Maintenance (ICSM'02)*, pages 162–171, Oct. 2002.

[52] S. P. Reiss. Incremental maintenance of software artifacts. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 113–122, Sept. 2005.

[53] S. P. Reiss. The paradox of software visualization. *3rd International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'05)*, pages 59–63, Sept. 2005.

[54] C. Riva. Reverse architecting: an industrial experience report. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 42–50, Nov. 2000.

[55] P. Romero, R. Cox, B. du Boulay, and R. Lutz. A survey of external representations employed in object-oriented programming environments. *Journal of Visual Languages and Computing*, 14(5):387–419, Oct. 2003.

[56] J. Singer. Creating software engineering tools that are useable, useful, and actually used. Talk given at the University of Victoria, Dec. 2004.

[57] M. Storey. Theories, methods and tools in program comprehension: Past, present and future. *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 181–191, May 2005.

[58] M.-A. D. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, Dec. 1998.

[59] M. D. Storey, D. Cubranic, and D. M. German. On the use of visualization to support awareness of human activities in software development: A survey and a framework. *ACM Symposium on Software visualization (SoftVis'05)*, pages 193–202, May 2005.

[60] M. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, Jan. 1999.

[61] M. D. Storey, F. D. Fracchia, and H. A. Müller. Customizing a fisheye view algorithm to preserve the mental map. *Journal of Visual Languages and Computing*, 10(3):245–267, June 1999.

[62] M. D. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. A. Müller. On designing an experiment to evaluate a reverse engineering tool. *3rd IEEE Working Conference on Reverse Engineering (WCRE'96)*, pages 31–40, Nov. 1996.

[63] M. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs. *4th IEEE Working Conference on Reverse Engineering (WCRE'97)*, pages 12–21, Oct. 1997.

[64] N. Synytskyy, R. C. Holt, and I. Davis. Browsing software architectures with LSEdit. *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 176–178, May 2005.

[65] R. Tamassia, G. D. Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):61–79, Jan./Feb. 1988.

[66] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, Universität Bern, Dec. 2001.

[67] S. Tilley. A reverse-engineering environment framework. Technical Report CMU/SEI-98-TR-005, Software Engineering Institute, Carnegie Mellon University, Apr. 1998. `http://www.sei.cmu.edu/pub/documents/98.reports/pdf/98tr005.pdf`.

[68] S. Tilley, S. Huang, and T. Payne. On the challenges of adopting ROTS software. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 3–6, May 2003.

[69] S. R. Tilley and D. B. Smith. Coming attractions in program understanding. Technical Report CMU/SEI-96-TR-019, Software Engineering Institute, Carnegie Mellon University, Dec. 1996. `http://www.sei.cmu.edu/pub/documents/96.reports/pdf/96.tr.019.pdf`.

[70] A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements for an industrial environment. *2nd IEEE Workshop on Program Comprehension (WPC'93)*, pages 78–86, July 1993.

[71] A. von Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437, June 1995.

[72] Q. Wang, W. Wang, R. Brown, K. Driesen, and B. Dufour. EVolve: An open extensible software visualization framework. *ACM Symposium on Software Visualization (SoftVis'03)*, pages 37–38, June 2003.

[73] K. Wong. On inserting program understanding technology into the software change process. *4th IEEE Workshop on Program Comprehension (WPC'96)*, pages 90–99, Mar. 1996.

[74] K. Wong. *The Reverse Engineering Notebook*. PhD thesis, Department of Computer Science, University of Victoria, 1999.

[75] K. Wong, S. R. Tilley, H. A. Müller, and M. D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, Jan. 1995.

[76] S. Woods, S. J. Carriere, and R. Kazman. A semantic foundation for architectural reengineering and interchange. *15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 391–398, Aug. 1999.

[77] S. Woods, L. O'Brien, T. Lin, K. Gallagher, and A. Quilici. An architecture for interoperable program understanding tools. *6th IEEE International Workshop on Program Comprehension (IWPC'98)*, pages 54–63, June 1998.

[78] I. Zayour and T. C. Lethbridge. Adoption of reverse engineering tools: a cognitive perspective and methodology. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 245–255, May 2001.