International Symposium on Visual Analytics Science and Technology (2010)
J. Kohlhammer and D. Keim (Editors)

Visual Analytics in Software Maintenance: Challenges and Opportunities

A. Telea[1] and and O. Ersoy[1] and L. Voinea[2]

[1]Institute Johann Bernoulli, University of Groningen, the Netherlands
[2]SolidSource BV, Eindhoven, the Netherlands

Abstract

Visual analytics (VA) is an emerging science at the crossroads of data and information visualization, graphics, data mining, and knowledge representation, with many successful applications in engineering, business and finance, security, geosciences, and e-governance and health. Tools using visualization, data mining, and data analysis are also prominently present in a different field: software maintenance. However, an integrated VA is relatively new for this field. In this paper, we discuss the specific challenges and particularities of applying VA in software engineering, highlight the added value of a VA approach, as distilled by us from several large-scale software engineering industrial projects.

1. Introduction

The modern IT industry is confronted with large, complex software and increased pressure on delivery time and product quality. Studies over 15 years estimate that 80% of the software costs are maintenance, of which 40% goes into program *understanding* [Sta84, Cor99].

Visual analytics (VA) integrates techniques from graphics, visualization, interaction, data analysis, and data mining to support reasoning and sensemaking for complex problem solving in engineering, finances, security, and geosciences [WT04, TC05]. These fields share many similarities with software maintenance in terms of *data* (databases with millions of records, highly structured text, and graphs), *tasks* (making sense of data by hypothesis creation, refinement, and validation), and *tools* (combined analysis and visualization).

However, software visualization (SV) is still only marginally accepted in the industry. Explanations include the limited maturity, learning ease, and integration of SV tools [Kos03]. Prominent researchers have voiced concerns over this situation [Rei05, CTM03], which echoes similar concerns in the field of data visualization [Lor04].

In this paper, we discuss the reasons for limited penetration of SV tools in the IT industry. We argue that similarities in data, tasks, and tools between VA and SV advocate for a VA approach to software understanding in maintenance. We support this claim by analyzing several industrial maintenance projects involving SV tools we took part in, and use this insight to better understand the acceptance challenges of SV. In contrast to other studies on SV tool acceptance, we use an approach based on value and waste as perceived by stakeholders, inspired from the lean development philosophy [PP06].

This paper is structured as follows. Section 2 provides a background on software analysis and visualization. Section 3 introduces our value-based model for SV for three user groups: developers, management, and consultants. Section 4 discusses several industrial projects in which SV was used, and discusses relations between VA and software analysis and visualization. Section 5 discusses observed challenges to SV adoption and indicates possible ways forward. Section 6 concludes the paper.

2. Background

Two types of techniques are present in software understanding for maintenance. *Analysis* tools extract facts from software, *e.g.* syntax trees, dependency graphs, and execution information [BF03, TV08a, LHM03, BPM04]. Facts can be refined into quality metrics, *e.g.* code readability, complexity, cohesion, and coupling [LM06], or higher-level artifacts like design patterns or code smells [Kos03, TWSM94]. *Visualization* tools present these facts using techniques such as data-annotated graphs [TWSM94, Lan04, LKG07, TMR02], table lenses and treemaps [TV08a], and metric-annotated code [ESS92]. An overview of SV is given by [Die07].

Most studies on the challenges and difficulties of SV adoption for program understanding in the industry focus on specific SV tools or techniques [DD06, HDS02, ED06]. In this paper, we are interested in understanding the adoption challenges of SV *as an integral technique*, at equal level with other technologies such as software analysis or testing.

DOI: 10.2312/PE/EuroVAST/EuroVAST10/075-080

3. Value Model

In the past seven years, we participated in over 20 industry projects involving SV and analysis tools for software understanding in maintenance in projects of tens of thousands up to 17 million lines of code; teams of 10 up to 600 developers; different programming languages, platforms, and architectures; and development methods from agile and extreme programming to strictly standardized workflows. In nearly all cases, we observed moderate to strong skepticism on SV. Technical issues such as tool scalability, limited visual clutter, details on demand, customizability were not the main blockers [SOT08, SOT09]. To quote a senior project manager, the central issue was "what does a SV tool bring as *measurable* added value to me?" Precisely the same issue was recently raised for the adoption of software static analysis by a major tool vendor [BBC*10].

We try to answer this question by a different approach from typical tool evaluations. We formulate SV adoption as a *lean development* problem [PP06]: To be accepted, SV must ultimately yield *value* and/or diminish *waste* as perceived by its users.

Obviously, different user groups have different definitions of value. In our work, we have observed three such groups:

1. *Technical users* focus mainly on creating a software *product*, and include developers, designers, testers, and architects;
2. *Managers* focus on the integral execution of projects over long periods of time;
3. *Consultants* work over relatively short periods of time and assist in integral strategic decision making.

We argue that a VA approach is highly beneficial to increasing value and decreasing waste for all these user groups, but in different ways. This point is detailed next.

4. Case Studies

To refine our understanding of the challenges and opportunities of VA in software maintenance, we have gathered insight from three types of studies over a period of several years. Each study type focuses on a user type (Sec. 3), tries to elicit perceived value drivers and translates these to tool and process requirements. These studies are presented next.

4.1. User group 1: Technical stakeholders

SV tools in corrective maintenance: Four different SV tools were considered: CodePro Analytix [Ins09], Ispace [I. 09], SonarJ [Hel09], and SolidSX [Sol10] (Fig. 1). These tools integrate with IDEs to support corrective maintenance (debugging) by hierarchy-and-dependency visualizations linked with code views. The tools were pre-selected to meet features deemed desirable by developers, as identified by earlier studies, *e.g.* scalability, ease of use, IDE integration, quick learning, and robustness [SOT08, SOT09], and also to have a similar look and feel. 29 professional developers used the tools to debug a known issue in a Java Mobile application of 10000 lines.

From the collected quantitative and qualitative feedback [SOT10], as well as silent user observation, we gathered the following points:

- all users (except one) ranked the degree of IDE integration as the most important tool effectiveness aspect. Tools with stronger integration, *e.g.* easy search/selection-based navigation across the dependency and code views, scored better;
- all users required 'what-if' scenario support, *e.g.* have the tool suggest code areas affected by a certain modification;
- all users required multiple views to correlate code text, structure, and execution;

In particular, a seamless integration between analysis (*e.g.* debugging and static source code analysis) and visualization was found absolutely crucial. Similar statements are made by other researchers in SV, *e.g.* [TH02, Kos03, Sto98, CTM03, SM05]. Yet, most existing SV tools lack such integration, which we believe to be a major blocker for their wider adoption.

Program structure comprehension: Ten developers used two SV tools (Tulip [Aub09] and SolidSX [Sol10]) for program structure-and-dependency visualization to answer modularity-related questions on several large C/C++ systems (*bison*, Mozilla Firefox, and the C++ parser from [HERT09]). Both SV tools score strongly on scalability, speed, robustness, ease of use, interactive navigation, and query facilities. Data extraction was done by a separate static analysis tool [HERT09]. Although the tasks and data were different, user feedback matched insight from the previous study. Multiple views and easy cross-view navigation were highly appreciated. The *lack* of integration between the static analysis and visualization was named as the most important drawback, which seriously reduced the perceived added value of the visualization.

4.2. User group 2: Project leading and management

Build process optimization: A major hardware company has an embedded C system of over 17 million lines. In maintenance, even small changes to some headers can cause huge build (compilation) times. The system is developed by 600 programmers worldwide, so build bottlenecks significantly delay testing and ultimately product releases. The project managers needed to answer the following questions [TV08b]:

- what is the exact model of the build impact, *i.e.* how can one predict the build cost (time) given a certain code change? This is a typical 'what if' question (what if I modify this file?)
- how is the build cost spread over the entire system? Which are the main build bottlenecks, now and in the future?

We approached these questions following a VA approach. First, we measured actual build time upon changing each header. Analyzing this data showed that 80% of the headers have small impacts, so build bottlenecks indeed exist (Fig. 2). Next, we designed a build cost and build impact model. We first hypothesized that a header's build impact equals the *number* of sources which use it directly or indirectly, and computed this impact using a file dependency graph extracted with static analysis tools [Spi09, TV08a]. Comparison with actual build times showed that this model is close, but not exact, to measured build times (see outliers in Fig. 3. We next refined our hypothesis: a header's impact is the sum of the build *costs* of all sources using it directly or not. This compound model matched measured costs including outliers. Further measurements revealed that the build time of a

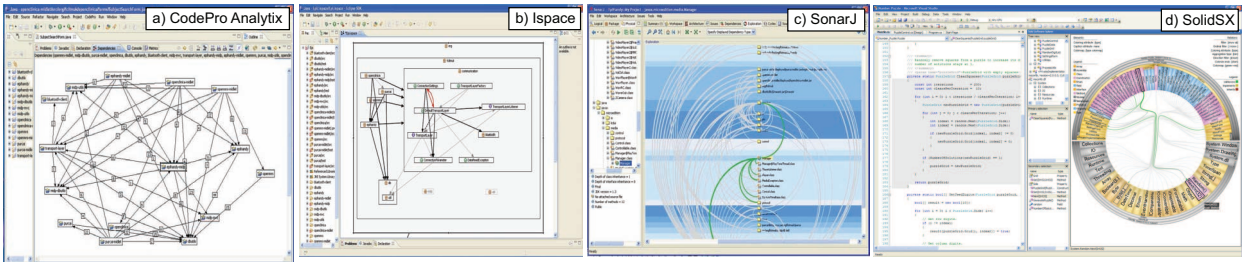

Figure 1: *Visual tools for corrective maintenance (Sec. 4.1). From left to right: CodePro Analytix [Ins09], Ispace [I. 09], SonarJ [Hel09], and SolidSX [Sol10]*

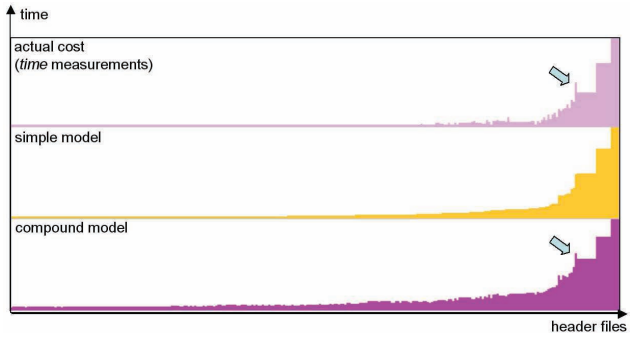

Figure 2: *Build impact: actual measurements (top), simple model (middle), compound model (bottom). Headers (x axis entries) are sorted on increasing build impact in the simple model. The y axis shows build time*

source is dominated by pure file access, and not file sizes, which supports our compound model.

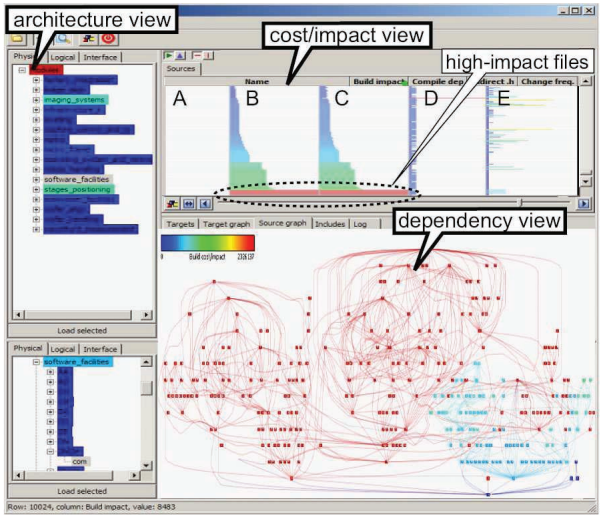

Figure 3: *Build analysis visual tool (Sec. 4.2)*

To answer the posed questions, we designed an interactive tool [TV08b] that integrates our build cost model with several visualizations (Fig. 3). The archictecture view shows the project hierarchy with subsystems colored by total build time. For a selected subsystem, the cost/impact view uses a table lens to show metrics for all files in that subsystem: the Þle name (A), impact (B), simple cost (C), and compound cost (D). Sorting this table quickly helps project managers to quickly locate high impact files to *e.g.* plan changes and selectively grant change access to such files. Sorting the table on build impact times change frequency (E) further allows finding if high impact files are potential build bottlenecks in the future. System headers, for instance, have very high impact, but zero change frequency, so are thus eliminated from true build bottlenecks.

4.3. User group 3: Consultants

Product and process assessment: A major automotive company developed an embedded software stack of 3.5 million lines of code in 15 releases over 6 years with three developer teams in Western Europe, Eastern Europe, and Asia. Towards the end, it was seen that the project could not be finished on schedule and that new features were hard to introduce. The management was not sure what went wrong. The main questions were: was the failure caused by bad architecture, coding, or management; and how to follow up - start from scratch or redesign the existing code. An external consultant team performed a post-mortem analysis. This team had only *one week* to deliver its findings and only the code repository as information source [VT09a].

The approach involved several steps of data acquisition, hypothesis creation, refinement, and (in)validation, and result aggregation and presentation (see Fig. 4). First, we mined change requests (CRs), commit authors, static quality metrics, and call and dependency graphs from the code repository into a SQL fact database (1). Next, we examined the distribution of CRs over project structure. Several folders were with many of open CRs emerged (red treemap cells in Fig. 4 (2)). These correlate quite well with the team structure: the 'red' team owns most CRs (3). To further see if this is a problem, we looked at the CR distribution over files over time. In Fig. 4 (4), files are shown as gray lines vertically stacked on age (oldest at bottom), and CRs are red dots (the same layout is used *e.g.* in [VT09b]). The gray area's shape shows almost no project size increase in the second project half, but many red dots over *all* files in this phase. These are CRs involving old files that were never closed. When seeing these images, the managers recalled that the 'red' team (located in Asia) had communication problems with the European teams, and ac-

knowledged that it was a mistake to assign so many CRs to this team.

We next analyzed the evolution of various quality metrics: fan-in, fan-out, number of functions and function calls, and average and total McCabe complexity. The graphs in (5) show that these metrics have a slow or no increase in the second project half. Hence, the missed deadlines were not caused by code size or complexity explosion. Yet, the average complexity per function is high, which implies difficult testing. This was further confirmed by the project leader.

Finally, to identify possible refactoring problems, we analyzed the project structure. Fig. 4 (6) shows disallowed dependencies, *i.e.* modules that interact bypassing interfaces. Fig. 4 (7) shows modules related by mutual calls, which violate the product's desired strict architectural layering. These two views suggest difficult step-by-step refactoring and also difficult unit testing. Again, these findings were confirmed by the project leaders.

5. Discussion

During the above studies, we gathered insight both implicitly (our own observations) and also by explicitly asking stakeholders about what they liked (or not) in the proposed SV tools. Our overall observations are discussed below (see also Table 1).

5.1. Reasoning and sensemaking patterns

We observed several differences in the way of using software visualizations to understand software for our three stakeholder types.

Technical users reason mainly about technical software artifacts, *e.g.* bugs, test failures, calls, interfaces, and dependencies. Most SV and software analysis tools are built around such artifacts. However, this does not mean that SV is easily adopted by technical users. The main adoption blocker we found is, as mentioned in Sec. 4.1, integration of tools across in workflows. Although analysis tools are increasingly integrated with development tools, *e.g.* the highly successful testing, optimization, and quality measurement plug-ins in Visual Studio, KDevelop, and Eclipse, visualization tools lag behind. Unfortunately, this means that highly scalable and successful techniques such as treemaps, hierarchical edge bundles, pixel charts, and parallel coordinates are ultimately not valued to their full potential by users. We also noticed that many software analysis tools are designed to work in batch (black-box) mode. This does not support what-if, sensemaking, scenarios. We strongly advocate a finer grained interaction where users can easily change the queries submitted to such tools so as to directly support their questions [TV08a]. This poses several challenges. Our discussions with several static analysis tool builders indicated resistance to 'opening up' the internals of their engines, as this is seen as disclosing valuable commercial assets (for commercial tool makers) or involving too much effort (for OSS tool makers).

Management and project leaders reason about a mix of technical (product) and non-technical (process) artifacts. Key to their sensemaking loop is mining information from a wide variety of sources, *e.g.* team activity, project structure, change requests, and architecture quality metrics. Secondly, in all our studies, management stressed the importance of SV and analysis tools to handle large data amounts over long time periods. SV tools that support *evolution* analysis are essential for this role. A third key requirement was the ability to quickly and easily change viewpoints, *e.g.* select which variables are to be tested for correlation.

Several challenges for current SV tools exist here. First and foremost, software evolution visualization tools, albeit well known in research, still have to become scalable, customizable, and robust products as demanded by the industry. Data mining from source control management (SCM) systems, *e.g.* CVS, SVN, ClearCase, JIt, CM/Synergy, and SourceSafe, is challenging. Such systems were designed to perform file check-ins and check-outs, not massive scale data querying, so they lack the uniform access protocols, speed, and robustness needed for data mining. In earlier research [VT09b] and the studies in Secs. 4.2 and 4.3, we noticed that roughly 30% of the repository-wide data mining requests were aborted by CVS, Subversion, and CM/Synergy servers. A second problem regards static analysis of code in repositories. This is very hard to automate, as analysis tools need to be configured for specific build processes, makefile types, and languages. Yet, unless this aim is reached, most of the key quality metrics and dependencies in Secs. 4.2 and 4.3 cannot be obtained without manual effort, so the perceived value of such analyses decreases. Attempts have been done to alleviate such problems by adding analysis tools at the SCM server side, *e.g.* in the SoftFab testing framework [SGL*06]. Discussions with the SoftFab developers outlined that this effort was quite high (over 2 years) and had to be redone for new projects requiring different analysis tools, which diminished the perceived value and amplified the feeling of waste.

Consultants reason about the widest, and most heterogeneous, set of artifacts: technical, product, process, risk, cost, and business strategy. Given their high hourly tariffs, data mining, analysis, and presentation must be done in very short timeframes, *i.e.* days or even hours [VT09a]. In contrast to managers and technical users, consultants often deal with non-technical stakeholders (upper management) so they favor simple, widely familiar visualizations such as business graphics. Also, consultants use visualizations to convey a message to other stakeholders, whereas developers and project managers are both visualization users and stakeholders (Fig. 4 (8)). As such, we noticed that visualization *usability* factors are much less important for SV acceptance by the final stakeholders for consultants.

5.2. General findings

Apart from the above differences in working patterns with SV tools of our three user groups, we note several common aspects, also typical for VA applications. These findings were distilled from the three types of studies presented in Sec. 4.

Integration: All studies showed the need of integration of analysis, knowledge representation, and visualization in coherent solutions (tools). It cannot be stressed often enough that the lack of integration is a main cause of the limited impact of SV tools in the software industry.

Value reflection: A SV tool must reflect as directly as possi-

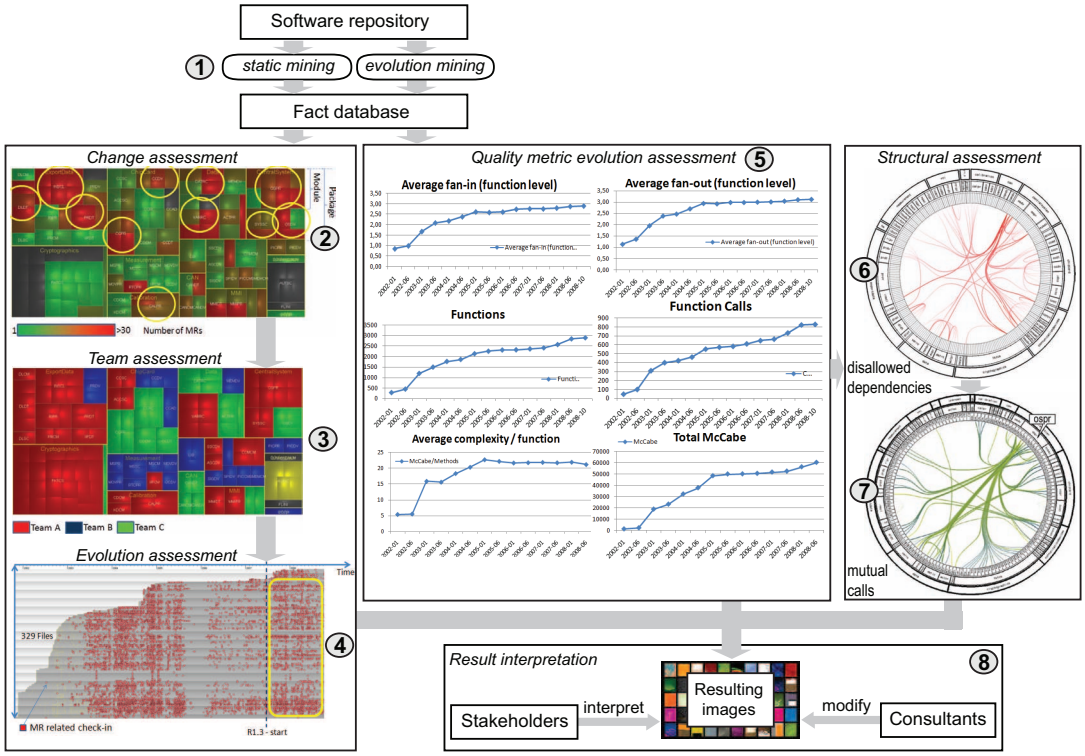

Figure 4: *Data collection, hypothesis forming, and result interpretation for product and process assessment (Sec. 4.3). Arrows and numbers indicate the order of the performed steps*

Stakeholder	Technical	Management	Consultant
artifacts	code, design, use cases (mainly product)	plans, schedules, quality models and metrics, impact dependencies (product and process)	blockers, risks, cost, schedule, standards, business rules (mainly process)
input heterogeneity	relatively low	medium	very high
perceived value	quickly design, debug, test, optimize, learn new code	quickly check product plan conformance discover hidden risks upfront	deliver answers to integral problems communicate them to stakeholders quickly and clearly
value measures	task execution time (how quickly I can do my work)	analysis speed and quality (how quickly and precisely I can analyze a large, long, project)	integral solution time (how quickly can I give a global answer to customers)
visualizations	detailed code text, structure, dependencies runtime metric tables (performance, bugs)	evolution plots of aggregated product and process metrics vs product structure	simple business graphics (charts, scattered plots, parallel coordinates)
key tool requirements	seamless integration with existing workflow and toolchain; details on demand; quick for precise tasks	ability to handle large repositories freely navigate across levels of details correlate many aspects easily define custom quality models support what-if scenarios	simple visualizations, one aspect per view highly adaptable to new data sources very fast customization
adoption resistance points	limited scalability, robustness, speed hard to integrate, learn, use	limited scalability and customizability limited repository support hard to customize data analyses	complex visualizations or interaction hard to customize on-the-fly expensive or restrictive licenses

Table 1: *Relevant characteristics of SV tools for different stakeholder types in software maintenance*

ble the artifacts underlying the value system of its intended user group, *e.g.* code-level artifacts for developers, project metrics for managers, and combined process and product metrics for consultants. Failing to do so will significantly decrease a visualization's perceived value, making it a 'nice to have' item, or even a waste of time.

Value vs cost: Visualizations must demonstrably bring value within the cost range acceptable to their user groups. Developers appear to be the most willing to try and tweak new techniques, but also swiftly discard (visual) tools when these create effort without quickly visible benefits. Consultants have even higher demands to see quick returns for their invested effort and are less willing to spend time to tweak tools. Project managers appear to be the most willing to invest more effort and time into SV tools until obtaining returns, as they have much longer time frames over which value can be returned.

6. Conclusions

In this paper, we have discussed the challenges and opportunities of using visual analytics techniques for software process and product understanding in software maintenance. Such techniques, *i.e.* raw data collection, hypothesis creation, refinement,

and (in)validation, map perfectly to the problems and challenges of software understanding. We have presented several industrial case studies in which we observed, or followed, a 'VA way of working', as well as several challenges to current software visualizations. Overall, these challenges mix technical issues with less-than-optimal matches of features with the value drivers of their target user groups. We do not claim universality for our observations. Still, the size and variety of our sample set makes us to consider these findings as very relevant.

Wider adoption of VA principles in this industry has huge potentials. IT professionals are well aware of the high cost of program understanding [Cor99]. Yet, for increased adoption, software visualization designers should focus more on visualization-analysis integration and designing *simple* visual metaphors that convey precisely and directly the value drivers and way of working of specific user groups. If such aspects are considered, we are convinced that VA will make a significant impact to the software industry.

References

[Aub09] AUBER D.: Tulip visualization system. tulip.labri.fr.

[BBC*10] BESSEY A., BLOCK K., CHELF B., CHOU A., FULTON B., HALLEM S., GROS C. H., CAMSKY A., MCPEAK S., ENGLER D.: A few billion of lones of code later: Using static analysis to find bugs in the real world. *Comm. of the ACM 53*, 2 (2010), 66–75.

[BF03] BALANYI Z., FERENC R.: Mining design patterns from C++ source code. In *Proc. ICSM* (2003), IEEE, pp. 305–314.

[BPM04] BAXTER I., PIDGEON C., MEHLICH M.: DMS: Program transformations for practical scalable software evolution. In *Proc. ICSE* (2004), IEEE, pp. 625–634.

[Cor99] CORBI T.: Program understanding: Challenge for the 1990s. *IBM Systems Journal 28*, 2 (1999), 294–306.

[CTM03] CHARTERS S., THOMAS N., MUNRO M.: The end of the line for Software Visualisation? In *Proc. Vissoft* (2003), pp. 27–35.

[DD06] DILUCCA G., DIPENTA M.: Experimental settings in program comprehension: Challenges and open issues. In *Proc. ICPC* (2006), pp. 229–234.

[Die07] DIEHL S.: *Software Visualization Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[ED06] ELLIS G., DIX A.: An explorative analysis of user evaluation studies in information visualisation. In *Proc. AVI Workshop on Beyond Time and Errors: Novel Evaluation methods for information visualization* (2006).

[ESS92] EICK S., STEFFEN S., SUMNER E.: Seesoft-a tool for visualizing line oriented software statistics. *IEEE TSE 18*, 11 (1992), 957–968.

[HDS02] HUNDHAUSEN C., DOUGLAS S., STASKO J.: A meta-study of software visualization effectiveness. *J. Vis. Lang. Comput.* (2002), 259–290.

[Hel09] HELLO2MORROW, INC.: SonarJ. www.hello2morrow.com.

[HERT09] HOOGENDORP H., ERSOY O., RENIERS D., TELEA A.: Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In *Proc. ACM Vissoft* (2009), pp. 137–145.

[I. 09] I. ARACIC: Ispace. website, 2009. ispace.stribor.de.

[Ins09] INSTANTIATIONS, INC.: CodePro Analytix. website, 2009. www.instantiations.com.

[Kos03] KOSCHKE R.: Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *J. of Software Maintenance and Evolution*, 15 (2003), 87–109.

[Lan04] LANZA M.: *CodeCrawler* - polymetric views in action. In *Proc. ASE* (2004), pp. 394–395.

[LHM03] LIN Y., HOLT R. C., MALTON A. J.: Completeness of a fact extractor. In *Proc. WCRE* (2003), IEEE, pp. 196–204.

[LKG07] LIENHARDT A., KUHN A., GREEVY O.: Rapid prototyping of visualizations using Mondrian. In *Proc. IEEE Vissoft* (2007), pp. 67–70.

[LM06] LANZA M., MARINESCU R.: *Object-Oriented Metrics in Practice*. Springer, 2006.

[Lor04] LORENSEN B.: On the death of visualization: Can it survive without customers? In *Proc. of the NIH/NSF Fall Workshop on Visualization Research Challenges* (2004).

[PP06] POPPENDIECK M., POPPENDIECK T.: *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley, 2006.

[Rei05] REISS S.: The paradox of software visualization. In *Proc Vissoft* (2005), pp. 59–63.

[SGL*06] SPANJERS H., GRAAF B. T., LORMANS M., BENDAS D., SOLINGEN R. V.: Tool support for distributed software engineering export. In *Proc. ICGSE* (2006), pp. 187–198.

[SM05] SCHAFER T., MENZINI M.: Towards more flexibility in software visualization tools. In *Proc. Vissoft* (2005), pp. 20–26.

[Sol10] SOLIDSOURCE: SolidSX. www.solidsourceit.com.

[SOT08] SENSALIRE M., OGAO P., TELEA A.: Classifying desirable features of software visualization tools for corrective maintenance. In *Proc. ACM SOFTVIS* (2008), pp. 87–90.

[SOT09] SENSALIRE M., OGAO P., TELEA A.: Evaluation of software visualization tools: Lessons learned. In *Proc. Vissoft* (2009), pp. 156–164.

[SOT10] SENSALIRE M., OGAO P., TELEA A.: Analysis of desirable features for software visualization tools in corrective maintenance, 2010. www.cs.rug.nl/~alext/PAPERS/SeOgTe10.pdf.

[Spi09] SPINELLIS D.: Cscout, 2009. www.spinellis.gr.

[Sta84] STANDISH T. A.: An essay on software reuse. *IEEE Trans. on Software Engineering 10*, 5 (1984), 494–497.

[Sto98] STOREY M. A.: *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, Canada, 1998.

[TC05] THOMAS J. J., COOK K. A.: *Illuminating the Path: The Research and Development Agenda for Visual Analytics*. National Visualization and Analytics Center, 2005.

[TH02] TILLEY S., HUANG S.: On selecting software visualization tools for program understanding in an industrial context. In *Proc. IWPC* (2002), IEEE, pp. 285–288.

[TMR02] TELEA A., MACCARI A., RIVA C.: An open toolkit for prototyping reverse engineering visualizations. In *Proc. Data Visualization (IEEE VisSym)* (2002), IEEE.

[TV08a] TELEA A., VOINEA L.: An interactive reverse-engineering environment for large-scale C++ code. In *Proc. ACM SOFTVIS* (2008), pp. 67–76.

[TV08b] TELEA A., VOINEA L.: A tool for optimizing the build performance of large software code bases. In *Proc. CSMR* (2008), pp. 153–156.

[TWSM94] TILLEY S., WONG K., STOREY M., MÜLLER H.: Programmable reverse engineering. *Intl. J. Software Engineering and Knowledge Engineering 4*, 4 (1994), 501–520.

[VT09a] VOINEA L., TELEA A.: Case study: Visual analytics in software product assessments. In *Proc. VISSOFT* (2009), pp. 57–45.

[VT09b] VOINEA L., TELEA A.: Visual querying and analysis of large software repositories. *Empirical Software Engineering 14*, 3 (2009), 316–340.

[WT04] WONG P. C., THOMAS J. J.: Visual analytics. *IEEE Computer Graphics and Applications 24*, 5 (2004), 20–21.