

Visual Tools for Software Architecture Understanding: A Stakeholder Perspective

Alexandru C. Telea, *University of Groningen*

Lucian Voinea, *SolidSource BV*

Hans Sassenburg, *SeCure GmbH*

Framing visual-tool adoption in a lean development setting establishes a model for choosing the right tool for a task based on its value-added versus waste.

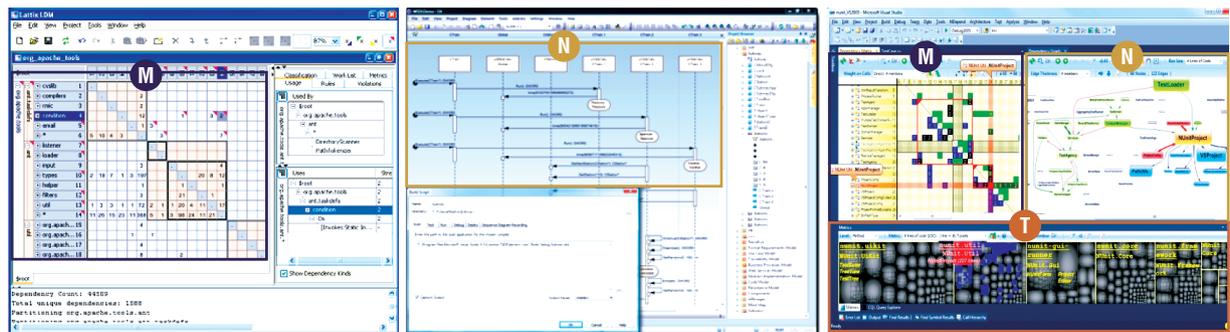
The IEEE 1471 standard defines a software architecture as “the fundamental organization of a system embodied in its components, their relationships to each other, and the principles guiding its design and evolution.” Visual tools have long been available to support an architecture’s roles to describe both how a software system should be and how the system actually is.¹ We can classify visual tools correspondingly as either design tools for a new architecture (for example, UML modeling tools) or tools for visual understanding of an existing architecture.

Visual understanding tools aim to support several tasks, such as comparing desired and actual architectures, identifying architecture violations, highlighting architectural patterns or layers extracted from code bases, assessing architecture quality, and discovering evolutionary patterns such as architectural erosion. These tasks require tools that are tightly integrated with static analyzers and code checkers to obtain basic program structure and dependency information for generating the architectural views.

Visual design tools, such as IBM’s Rational Suite, Poseidon, and plug-ins for Visual Studio and Eclipse, are well established and widely used in industry. In contrast, tools for architecture understanding are much less widespread. This isn’t for lack of need. Software maintenance costs about 80

percent of a software product’s total life-cycle costs, and 40 percent of that cost is software understanding.^{2,3} Nor is it for want of candidates. Researchers have spent considerable effort and care to develop hundreds of visualization tools. Yet, most of them remain confined to academia or to niche projects in industry.

Why is this so? To clarify how industry can tap the potential of architecture visualization tools (AVTs), we survey the state of the art and characterize the added value for specific industry user groups as well as the development challenges still remaining. Based on our experience using AVTs for understanding software in industrial contexts, we analyze the success and failure factors we noticed in our work and derive guidelines to help industrial practitioners achieve a best match between their requirements and the capabilities in state-of-the-art visualization tools and techniques.

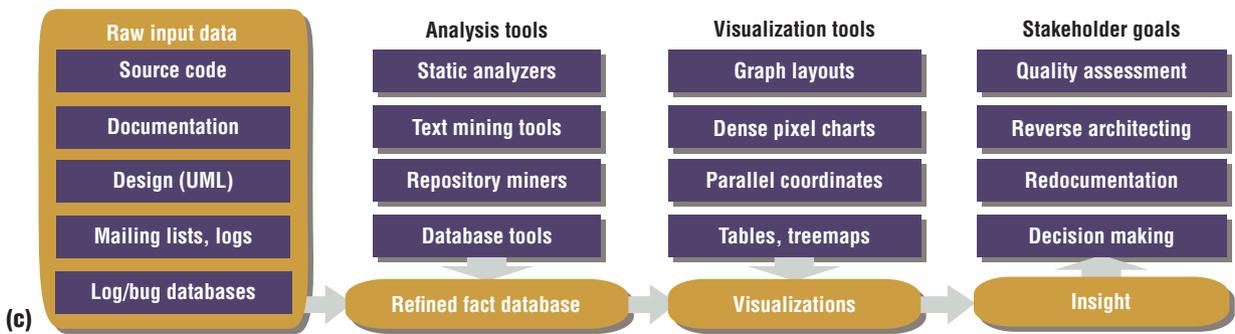


(a) Lattix LDV (www.lattix.com) Enterprise Architect (www.sparxsystems.com) NDepend (www.ndepend.com)



(b) CodeCrawler (www.inf.usi.ch/faculty/lanza/codecrawler) Rigi (www.csc.rigi.uvic.ca) EvoSpaces (www.inf.usi.ch/projects/evospaces)

(b) **M** Matrix plots **N** Node-link layouts **T** Treemaps **P** Pixel charts



(c)

AVT State of the Art

Many vendors have developed commercial AVTs. Figure 1a shows three well-known tools: Lattix, Enterprise Architect, and NDepend; Klocwork Architect, IBM Rational Architect, and Bauhaus are among the many others.

The research community has also developed numerous tools. Figure 1b shows three: CodeCrawler, Rigi, and EvoSpaces. Unlike commercial tools, which are black-box products, research tools are typically open source, allowing users to tweak and customize them.

In general, software visualization tools operate as a pipeline, as shown in Figure 1c. They mine data

from various sources: code bases, software repositories, text documents (such as requirements documents, email, and notes), design documents (such as UML diagrams based on XML Metadata Interchange [XMI]), and test logs. Next, they analyze the data using techniques such as static analyzers, text miners, repository access clients, and database clients. A refined-fact database stores the analysis results, using an entity-relationship (ER) model: *entities* model software artifacts, such as files, classes, functions, requirements, diagrams, and requirements; *relationships* describe entity interactions, such as calls, uses, inheritance, ownership, and logical dependencies. Key-value attributes store entity

Figure 1. Visual tools for software architecture understanding: (a) commercial examples, (b) research examples, and (c) functional pipeline. The legend defines some of the visualization tools available to help different stakeholders obtain insight to software architectures.

What measurable added value does a new visualization tool bring? And at what cost?

and relationship properties, such as quality metrics, change and test statistics, and code ownership.

Visualizations, or views, depict user-selected ER elements from the refined-fact database. Some visualization techniques are well-known, such as node-link layouts and matrix plots. Less well-known techniques have also proven their effectiveness in helping to understand complex data in many fields. These include 3D views, treemaps, parallel coordinates, bundled diagram layouts, parallel coordinates, and pixel charts (see Figure 1).

Most tools support searches of both the fact database and the views using simple textual and value-based queries as well as complex queries written in SQL-like languages. They also correlate views so that changes in input data or search hits in one view are automatically highlighted in other views.

We recommend two books for good overviews of software visualization and analysis: Stephan Diehl's *Software Visualization*³ and Michele Lanza and Radu Marinescu's *Object-Oriented Metrics in Practice*.² Bredemeyer Consulting offers an excellent practical overview of recent advances in architecture visualization on its website (www.bredemeyer.com/ArchitectingProcess/ArchitectureVisualization.htm), and Visualcomplexity.com presents good examples of the power of data visualizations (www.visualcomplexity.com). Finally, the proceedings of the ACM Symposium on Software Visualization (SoftVis), IEEE Workshop on Visualizing Software for Understanding and Analysis (Vissoft), and IEEE Working Conference on Mining Software Repositories (MSR) provide a wealth of implementation details and additional tool examples.

What Makes a Good AVT: The Theory

The efforts of researchers and practitioners to understand what makes a good AVT include requirement-elicitation studies, tool-usage case studies, and side-by-side tool comparisons. Table 1 summarizes the general requirements identified from this work.^{1,3,4} A good AVT should support numerous data types and provide interactive ways to compare, correlate, and search both views and data. It should integrate well with existing toolchains and be flexibly priced, easy to deploy and customize, and scalable.

What Makes a Good AVT: The Practice

Over the past seven years, we've used AVTs in over 25 industry projects comprising tens of thou-

sands to millions of lines of code; teams of 10 to 600 developers; different programming languages, platforms, and architectures; and development cultures from agile and extreme programming to strict workflows.

In nearly all cases, we initially met with moderate to strong skepticism regarding innovative AVTs. This was true even when the tools conformed well with the theoretical requirements listed in Table 1. The preeminent issue was soon apparent. To quote several senior project managers, "What measurable added value does a new visualization tool bring? And at what cost?" We got identical signals from consultants specializing in software product and process assessment, fellow researchers involved in creating new software-visualization tools, and participants in the tool demonstration sessions at the SoftVis, Vissoft, and MSR conferences (for example, see Stuart Charters and colleagues⁵ and Steven P. Reiss⁶). A major tool vendor recently raised similar questions about adopting static analysis.⁷

However, we also observed significantly reduced understanding for time and cost and improved results quality when projects that had used no visualizations adopted AVTs. The same was true for projects that replaced an existing tool with a better one. So what makes the difference between AVT success and failure in industry?

To answer this question, we consider tool adoption from a lean development perspective.⁸ A useful software visualization tool must be perceived by domain stakeholders to add value or diminish waste. This sounds obvious, but matching a stakeholder's value and waste concepts to a tool's provisions and requirements is not easy. Stakeholders don't have time to try out every new tool to see if one (or any) will suit their context, and tool developers can't create an ultimate product that satisfies all possible needs. Understanding the added value of new, cutting-edge tools is especially difficult when the tools are marketed in technical visualization terms, such as slice-and-dice treemaps, bundled-edge layouts, icicle plots, and multivariate charts. These terms often mean little to IT project professionals.

We observed that effective adoption of new AVTs strongly correlates with three different stakeholder types and their perceptions of value and waste. Technical users—developers, designers, testers, and architects—focus on creating a software product. Managers focus on integral project execution over long periods—often years. Finally, consultants work over short periods to assist in strategic decision-making.

Table 1**General requirements of software architecture visualization tools**

Requirement	Description
Data representation: Types of data the visualization tool should support	
Static	Generic ER models supporting multiple relation types (containment, association, and dependency)
	Multiple key-value attributes per entity and/or relationship (numerical, text, and ordinal value types)
Dynamic	Time-series attribute values per entity and/or relationship
Evolution	Multiple versions of ER models, annotated with change information (authors and commit logs)
Operations: Types of operations the visualization tool should support	
General	Interactive navigation and annotation in multiple correlated views
Comparison	Find and display differences between two or more views
Searching	Entity, relationship, and pattern searches according to custom, user-specified rules
Integration: How the visualization tool should integrate with data mining	
Data mining	Static analysis of different programming languages and dialects (C, C++, Java, C#/NET, and Cobol)
	Reconstruction of UML diagrams from source code
	Fact extraction from software repositories (Subversion, ClearCase, CVS [Concurrent Versions Systems], CM Synergy, and Visual Studio Team System)
	Robustness of data mining with respect to incomplete or incorrect data sources
Toolchain embedding	Interoperability with existing tools (IDEs, build systems, batch systems, software configuration management tools, testing and documentation tools, and bug reporting tools); data interchange with widely accepted formats (UML/XMI, Datrix, GXL [Graphics Exchange Language], SQL)
Effectiveness: Nonfunctional requirements of a useful and usable visualization tool	
Cost	Flexible pricing, open source model
	Short learning curve, quick deployment, and high automation of routine tasks
Benefits	Effective support of custom viewpoints (data types, analyses, queries, and views)
Scalability	Efficient support of datasets of thousands of diagrams, millions of lines of code, thousands of revisions for all operations (data mining, analysis, searching, and visualization)

Lean Visualization-Tool Adoption

The information we collected on AVT adoption and use came from several industrial projects. We structure our insights from the perspective of the stakeholder types. Our aggregated findings focus on perceived value drivers, tool requirements, and potential adoption pitfalls.

Technical Users in Product Development

Technical users require visualization to navigate and search software structure, dependencies, and attributes such as quality metrics. In one of our studies at the University of Groningen,⁹ 23 professional developers used four visualization tools (CodeProAnalytix, Ispace, SonarJ, and SolidSX) as an understanding aid to debug and refactor a Java

Mobile application of 10,000 LOC. We selected these tools because all four met the adoption criteria we've described for scalability, tight IDE integration, ease of use and deployment, robustness, search and navigation, and automatic data extraction from code. The tools use different layouts: classical node-link diagrams (CodePro and Ispace), grid layouts with curved edges (SonarJ), and radial layouts with edges bundled by system hierarchy (SolidSX). Figure 2 shows the same dataset displayed for the four tools. The study collected quantitative and qualitative feedback on what the developers valued (and missed) most in using each tool.

All developers found correlated views of code, metrics, structure, and dependencies to be indispensable. All (save one) said that strong IDE

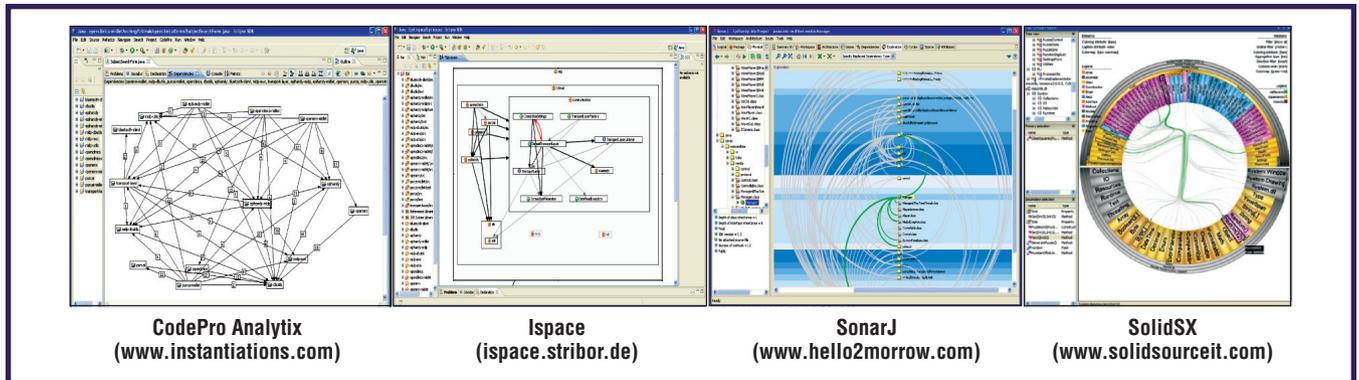


Figure 2. Visualization tools for structure, dependencies, and metrics. All four tools are displaying the same dataset for a Java mobile application of 10,000 LOC.

integration is the most important tool-effectiveness factor. All required the ability to easily define custom viewpoints for specific questions. Search, selection, and sorting operations support viewpoint definition to some extent, but generating readable node-link layouts required considerable manual tweaking. The developers saw this as a waste. They wanted to focus on their task, not fine-tune visualizations. In general, we know of no tool, commercial or research, that can automatically generate readable node-link layouts for more than about 1,000 entities and relationships. This lack is a key adoption blocker for this user group.

However, there are solutions. The treemap technique (used by NDepend) and the hierarchically bundled edges technique (used by SolidSX) produce readable, clutter-free layouts of thousands of entities and relationships with zero user intervention. The bundled edges technique can also show structure, dependencies, and metrics in a single view. These visualizations look very different from classical UML-like node-link diagrams, but no user in our study found them unintuitive. Quality open source implementations are available for these techniques (for example, see www.cs.umd.edu/hcil/treemap-history).

Project Managers in Large-Scale Maintenance

Project managers require tools to monitor the evolution of large-scale projects over years. As opposed to technical users, the focus is time, not code.

Managers must anticipate trends, such as architectural erosion, rule violation, and quality decay. Accordingly, their visualization needs differ from those of technical users. Data volumes extracted from repositories maintained by source control management (SCM) systems such as CVS and Subversion are huge. A project like KDE Office and Mozilla Firefox contains thousands of versions, each containing more than 1 million LOC. Modification authors, bug reports (stored in systems like

Trac or Bugzilla), and change requests yield two orders of magnitude more data than technical users work with.

Instead of showing individual code lines or call relations, AVTs for project management use coarser detail levels: file, class, and user (author). Sorting and ranking is essential. Project managers find the famous 20 percent of items that cause 80 percent of the problems by looking at distributions, not individual artifacts. Visual techniques as parallel coordinates, pixel charts, and timelines are the tools of the trade here (see www.bredemeyer.com/ArchitectingProcess/ArchitectingVisualization.htm for examples).

Consider the case of a major embedded software producer with a system of 17.5 million lines of embedded C, developed by 600 programmers worldwide over a decade. A system build took over nine hours. Because code was modified round the clock, there was no downtime to execute a build, so testing was hardly possible. Even small code changes could cause massive recompilation delays. The main questions were to understand what causes the build delay, how to prevent or postpone code changes that cause bottlenecks, and, ultimately, how to refactor the system to decrease such delays in the future.

For manager-level AVTs, we propose three key ingredients: repository data mining, static analysis, and presentation. We summarize them here briefly; a more detailed description is available elsewhere.¹⁰

Automating repository mining sounds trivial, but it's not. Implementations exist for some systems, such as CVSScan¹¹ for CVS and Moose (www.moosetechnology.org) for Subversion. Developers can customize these implementations with limited effort. However, other SCM systems—CM Synergy, in the embedded software case—can take months to implement.

Manager-level tools must include static analysis to query program concepts—for example, to find coincident function or class-level changes, interface splits or merges, or architectural viola-

tions. Automating static analysis is even harder than automating repository mining because SCM systems were designed to check code in and out, not to query or analyze it. Some easy-to-use solutions are available: Reflector (www.red-gate.com/products/reflector) for .NET/C# and (Recoder, <http://recoder.sourceforge.net>) for Java. However, no solution works for C and C++ without significant manual effort.

Finally, in presenting data for this user group, we found that visualizations capable of displaying several attributes simultaneously, such as treemaps and dense pixel charts, work best and are seen as extremely valuable. Implementations for these visualizations are readily available (for example, see the treemap resource website (www.cs.umd.edu/hcil/treemap-history) and Prefuse (<http://prefuse.org>).

Consultants in Product and Process Assessment

Consultants reason about the broadest, most heterogeneous artifacts: technical, product, process, risk, cost, and business strategy. Given the high fees consultants charge, they need tools that address data mining, analysis, and presentation requirements over time intervals of days or even hours. Consultants often deal with nontechnical stakeholders, especially upper management, so they need highly simplified visualizations. Moreover, they seldom know up front which visualization is best because they don't know the context up front. This makes fast customizability crucial. Overall, consultants impose the highest demands on AVTs to work out of the box, quickly and flexibly, and to produce simple images.

Current AVTs aren't sufficient for such users. All the consultants we talked to used either hand-crafted PowerPoint presentations (created with significant manual effort) or tools such as NDepend or Lattix, which are easy to use but limited in data acquisition, scalability, evolution support, and customizability.

However, pairing consultants with visualization experts can solve the problem. Consultants provide the customer interface, while visualization experts generate the required images from available data on demand. For example, in a post-mortem assessment, we were able to generate key insight in a 1.5 million LOC product developed over six years in a matter of days by using eight different visualizations and three different static analysis tools. The consultants involved in assessing the product found this approach highly cost-effective.¹²

Prefuse and Tableau (www.tableausoftware.com)

are visual tools for quickly generating attractive business graphics from complex multivariate data. However, quickly extracting relevant facts from large, unknown software repositories is still an open challenge and serious adoption blocker for consultants.

Lessons Learned

We see strong possibilities, but also challenges, for AVTs. The main key to success is correctly identifying the stakeholder type. Table 2 summarizes our adoption criteria for each type.

Current tools come closest to satisfying technical users' needs. We foresee significant short-term advances in this field, such as integration of treemaps, pixel charts, and bundled-edge layouts in mainstream IDEs such as Visual Studio and Eclipse. Tools for project managers exist but lack automated data mining from large repositories, which is the largest blocker to widespread adoption. For C/C++ code bases and several repository types, this problem could take several years to automate. Current tools fall shortest in meeting consultants' presentation needs, but mixed consultant-visualization expert teams offer a way to deliver high-quality results in short time frames.

Major technical limitations to architecture analysis include the lack of reliable, easily reusable solutions for automated architecture extraction, architecture comparison (visual or not), and architecture pattern detection, despite sustained ongoing research. For instance, we still have no effective way to run a diff function between two (or more) architectural views or viewpoints. Experts foresee this remaining a fundamental problem for several years.^{1,3}

Tool Strengths and Weaknesses

The strongest and weakest aspects of state-of-the-art AVTs fall along five dimensions.

Techniques. Tools that implement techniques such as treemaps, pixel charts, and bundled-edge layouts are clearly more scalable and simpler to use than traditional node-link layouts. They should be preferred whenever available.

Customizability. Research software is usually open source and thus fully customizable. For project managers and consultants, this one-time investment can provide large benefits compared to the typical costs of commercial closed-source visualization tools—from hundreds to thousands of euros or dollars per year. Yet, a research tool is never free. Overall, we observed a constant 80/20 percentage split

The main key to AVT success is correctly identifying the stakeholder type.

Table 2**Tool and task adoption criteria for specific user groups**

Tool and task adoption criteria	User (stakeholder) types		
	Technical users (developers)	Project managers/lead architects	Consultants
Project duration	Several months	Months to years	Days to 2 weeks (maximum)
Project aims	Develop a software product (small-medium scale)	Develop and maintain a software product (large scale)	Develop and maintain a software product (large scale)
Team size	5–15 developers + 1 leader	50–200 developers + 5–10 leaders	30–300 developers
Typical toolchain	Compiler, debugger, IDE, and software visualization tool	Quality metric extractors; static analyzers; configuration management, project planning, bug-tracking tools	Integrated visualization + analysis tools and results-reporting tools (PowerPoint-like)
Types of views	Correlated structure, dependency, metrics	Multivariate plots of processes and product	Simple business graphics (bar charts, timelines, matrix plots, pie charts)
Key values for software visualization tool	Generate desired custom views with as few views/operations as possible	Discover software evolution problems; monitor project execution (team, quality, staying on course) with no tweaking	Easy to set up and run within hours; does not have to produce detailed, exact results
Key waste for software visualization tool	Tedious manual tuning of visual layouts; reading cluttered layouts takes too much effort	Insufficient insight in all key attributes; lengthy manual setup of data extraction	Lengthy set-up procedures, including visualization customization, requiring high technical expertise
Software visualization tool requirements	IDE integration; automatic/easy viewpoint generation; high visual scalability	Automatic fact extraction from repositories; automatic visualization generation; high scalability in number of data points	Highly automated: produces images out of the box; highly customizable: runs on widely different customer code bases

between tool customization and data acquisition efforts and tool utilization. The actual stakeholders must determine whether this is appropriate for a specific context. A cost-benefit analysis using the constraints defined in Tables 1 and 2 can quickly decide in this determination.

Integration. Out-of-the-box toolchain integration is rare. Data interchange formats like XMI, GXL, Datrix, and Famix exist, but most visualization tools rarely support them all. Yet, all the visualization tools we know use the simple ER dataset model we described earlier. This model is easy to support with minimal programming (days, rather than weeks).

Tool focus. In all our experience, we found that generic visualization tools are best for project managers and consultants and dedicated ones are best for developers. Customizability is essential for the first group, and rapid results are essential for the latter. We never found benefit in the effort needed to make a generic visualization tool valuable for a technical user or a dedicated tool valuable for a project manager or consultant. This can serve as a preliminary selection criterion for visualization tools based on user type.

Value of insight. Visualization is indispensable when there is no cheaper alternative to gaining insight into a software architecture. When you don't know up front which specific questions to ask or metrics to compute to solve a high-level problem, visualization tools can add value. When the data size makes manually checking all possibilities too costly or lengthy, visualizations can be cost-effective. It's a value versus waste proposition that must be evaluated separately in every single context.

Stakeholder Concerns

AVTs vary in the degree to which they address the concerns of the various architectural viewpoints outlined in IEEE 1471.

Multivariate and structure-dependency-metric visualizations are the most used (and best supported) tools for structural, behavioral, information, engineering, and computational viewpoints. System decomposition and requirements allocation viewpoints are much harder to support because these processes require active user participation. Although combining visual interaction with scalable structure-dependency-metric visualizations could meet this requirement, we're not aware of a tool that does this. The enterprise and technology view-

points don't pose specific visualization problems, but they do require data mining tools to extract concept such as purpose, scope, and policies. Such concept are highly abstract, context-dependent, and typically not saved in standardized formats, so automatic mining isn't yet possible.

Current visualization tools are sufficient for addressing performance, scalability, and reliability project concerns. Modern visualizations such as Prefuse and Tableau can navigate and visualize huge ER datasets. The software architecture community hasn't yet made widespread use of these techniques. However, it has acknowledged the limitations of node-link layouts, and newer diagram-layout techniques have already found their way into mainstream tools, such as Lattix and NDepend. From another perspective, if the data is readily available for a given architecture, current visualization tools have no problems in displaying it because it's just another variant of a multivariate relational dataset.

Finally, business concerns such as cost, schedule, and quality of service map well to multivariate visualizations, such as pixel charts and timelines, as the examples we've described here and elsewhere show.¹⁰⁻¹² The main challenge is obtaining the data, not presenting it.

Many visualization tools provide significant value in understanding large software architectures and supporting architectural maintenance and evolution, quality assessment, communication with stakeholders, and strategic product planning. However, deciding whether to use a tool (and which one) is still challenging and doesn't come free. We see visualization as an instrument that generates value and minimizes waste, and we've given several pointers to available, mature, open source resources for appropriate visualizations. Framing stakeholders' concerns is absolutely essential to selecting a good tool.

We see visualization techniques becoming increasingly integrated with standard development and maintenance pipelines, benefiting all types of users involved in software architecture. Until then, custom solutions can be valuable, but each case requires a careful analysis of costs versus benefits or value-added versus waste. ☞

References

- 1 K. Gallagher, A. Hatch, and M. Munro, "Software Architecture Visualization: An Evaluation Framework and Its Application," *IEEE Trans. Visualization and Computer Graphics*, vol. 34, no. 2, 2008, pp. 260-270.
- 2 M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice—Using Software Metrics to Characterize,*

About the Authors



Alexandru C. Telea is a professor in software visualization at the University of Groningen, the Netherlands. His interests cover data, information, and software visualization, static source code analysis, and C/C++ reverse engineering. Telea received a PhD in data visualization and software architectures at the University of Eindhoven, the Netherlands. He's a member of the ACM and served as general chair of ACM SoftVis 2010. Contact him at a.c.telea@rug.nl.

Lucian Voinea is a cofounder of SolidSource BV (www.solidsource.nl), a company specializing in tools and services for software product and process assessment. He also cofounded the Software Benchmarking Organization (www.sw-benchmarking.org). Voinea received a PhD in software evolution analysis and visualization from the University of Eindhoven, the Netherlands. Contact him at lucian.voinea@solidsource.nl.



Hans Sassenburg is a visiting scientist at the Carnegie Mellon's Software Engineering Institute and head of SeCure AG (www.se-cure.ch), which specializes in software product and process quality improvement. He also cofounded the Dutch Spider Organization for Software Process Improvement (www.st-spider.nl) and the Software Benchmarking Organization (www.sw-benchmarking.org). Sassenburg received a PhD in economics from the University of Groningen, the Netherlands, in the area of software release strategies. Contact him at hsassenburg@se-cure.ch.

Evaluate, and Improve the Design of Object-Oriented Systems, Springer, 2008.

- 3 S. Diehl, *Software Visualization—Visualizing the Structure, Behaviour, and Evolution of Software*, Springer, 2007.
- 4 K.D. Babu, P. Govindarajulu, and A.A. Kumari, "Development of a Conceptual Tool for Complete Software Architecture Visualization: DArch," *Int'l J. Computer Science and Network Security*, vol. 9, no. 4, 2009, pp. 277-286.
- 5 S. Charters, N. Thomas, and M. Munro, "The End of the Line for Software Visualization?" *Proc. 2nd IEEE Int'l Workshop Visualizing Software for Understanding and Analysis (Vissoft 03)*, IEEE CS Press, 2003, pp. 27-35.
- 6 S.P. Reiss, "The Paradox of Software Visualization," *Proc. 3rd IEEE Int'l Workshop Visualizing Software for Understanding and Analysis (Vissoft 05)*, IEEE CS Press, 2005, pp. 59-63.
- 7 A. Bessey et al., "A Few Billion of Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Comm. ACM*, vol. 53, no. 2, 2010, pp. 66-75.
- 8 M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2006.
- 9 M. Sensalire, P. Ogao, and A. Telea, *Model-Based Analysis of Adoption Factors for Software Visualization Tools in Corrective Maintenance*, tech. report SVCG-RUG-10-2010, Univ. of Groningen, the Netherlands, 2010; www.cs.rug.nl/~alex/PAPERS/Sen10.pdf.
- 10 A. Telea and L. Voinea, "A Tool for Optimizing the Build Performance of Large Software Code Bases," *Proc. IEEE Conf. Software Maintenance and Reengineering (CSMR 10)*, IEEE CS Press, 2010, pp. 323-325.
- 11 L. Voinea and A. Telea, "Visual Querying and Analysis of Large Software Repositories," *Empirical Software Eng.*, vol. 14, no. 3, 2009, pp. 316-340.
- 12 A. Telea and L. Voinea, "Case Study: Visual Analytics in Software Product Assessments," *Proc. 5th IEEE Int'l Workshop Visualizing Software for Understanding and Analysis (Vissoft 09)*, IEEE CS Press, 2009, pp. 65-72.