Program Visualization Support for Highly Iterative Development Environments

Michele Lanza
lanza@iam.unibe.ch
Software Composition Group
University of Bern, Switzerland

Abstract

Software Visualization is, despite the many publications and advances in this research field, still not being considered by mainstream software industry: currently very few integrated development environments offer (if at all) only limited visualization support, and in general it can be said that software visualization is being ignored at a professional level by the average software developer. Moreover, even relatively successful software visualization tools (such as Rigi, Shrimp, JInsight, etc.) are seldom being used except by their developers themselves. In this position paper, based on our own experience and an analysis of the current state and possible future trends of integrated development environments, we put up a non-exhaustive list of features that software visualization tools should possess in the future to have more consideration by mainstream development.

1 Introduction

Software visualization is a fairly recent research field dating back to the 1960's, and started to become an established research field in the 1980's. The main benefit that software visualization (as a specialization of the more general field of information visualization) brings, is that it "provides an ability to comprehend huge amounts of data" and "allows the perception of emergent properties [of the data] that were not anticipated" [28]. Despite these and other benefits of software visualization, the contributions that this field has made to mainstream software industry are barely noticeable and largely ignored.

In this position paper we want to analyze this research field from different points of view, investigate and discuss some of the reasons that make software visualization still a "secondary" research domain, and put up a non-exhaustive list of features that need to be implemented by the developers of software visualization tools, should they want to propagate their research into an industrial context.

2 Software Visualization

> "Software visualization is [..] the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software." [20]

Software visualization is a specialization of *information visualization*, whose goal is to visualize any kind of data, while in software visualization the sole focus lies on visualizing software. Information visualization is defined as "the use of computer-supported, interactive, visual representations of abstract data to amplify cognition." [3]. It derives from several communities. Starting with Playfair (1786), the classical methods of plotting data were developed. In 1967, Jacques Bertin, a French cartographer, published his theory in *the semiology of graphics* [2]. This theory identifies the basic elements of diagrams and describes a framework for their design. A few decades later Edward Tufte published a theory of data graphics that emphasized maximizing the density of useful information and minimized recurrent errors in data visualization [25, 26, 27]. Both Bertin's and Tufte's theories have influenced the various communities that led to the development of information visualization.

The goal of information visualization is to *visualize any kind of data*. Note that the above definition by Card *et al.* of information visualization does not necessarily imply the use of vision for perception: visualizing does not only involve *visual* approaches, but any kind of *perceptive* approach. Data can be perceived by a person by using the senses at his/her disposition, *e.g.*, apart from seeing the data, a person can also hear it (information auralization) and/or touch it (by using virtual reality technology). However, most information visualization systems currently use computer graphics which render the data using 2D- and/or 3D-views of the data. Applications in information visualization are so frequent and common, that most people do not notice them:

examples include metereology (weather maps), geography (street maps), geology, medicine (computer-aided displays to show the inner of the human body), transportation (train tables and metro maps), etc.

In short, information visualization is about visualizing any kind of data, while software visualization is about visualizing software.

According to Stasko *et al.* the field of software visualization can be divided into two separate areas [20]:

1. *Program visualization* is the visualization of actual program code or data structures in either static or dynamic form. Most of the present approaches deal with *static code visualization*, because the source source code is visualized by using only information which can be *statically* extracted without the need to actually run the system.

2. *Algorithm visualization* is the visualization of higher-level abstractions which describe software. A good example is *algorithm animation*, which is the dynamic visualization of an algorithm and its execution. It is used to explain the inner working of algorithms like sort-algorithms. In the meantime this discipline has lost importance, mainly because the advancement in computer hardware and the possibility to use standard libraries containing such algorithms have shifted the focus away from the implementation of such algorithms.

In this paper we concentrate ourselves on program visualization, because most software visualization tools belong to this category, and because algorithm visualization has greatly lost importance in the past two decades, except for educational contexts (*e.g.*, teaching algorithms to students).

3 The Mission

The overall mission of program visualization is to visualize the static structure or the dynamic behavior of a software system.

In that sense software visualization researchers are trying to visualize an immaterial construct (software has no physical limits, no notion of proximity or distance) like software *the way it is*, although this is (by definition) not feasible: there is no unique and correct way of visualizing software. Taking an ambitious stance, we claim that the ultimate goal of a good visualization is to become the preferred way of developers of looking at software. We do not claim that software visualization could replace the most important and still most used way of perceiving software: code reading. We rather suggest that software visualization should have a symbiotic relationship with the practice of code reading by pointing the viewer to the location in the system where he should read and/or modify the code. According to the program cognition model vocabulary proposed by Littman *et al.* [15] we propagate an approach of software understanding that is *opportunistic* in the sense that it is not based on a *systematic* line-by-line understanding but *as needed*.

Moreover, software visualization has become relevant in the reverse engineering research community. Software reverse engineering is defined by Chikosfky and Cross as "the process of analyzing a subject system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction" [4]. The goal is thus to *construct a mental model* of a software system. Storey *et al.* have highlighted in various papers that a good software visualization is a powerful asset in the building of such a mental model [23, 21, 22].

Although software visualization is at least in a reverse engineering context of great importance (as shown by the number of publications on software visualization in the reverse engineering community), this could lead to a detrimental distinction between a forward and a reverse engineering phase. This view is not up-to-date anymore: an evolutionary view of software is taking its place, putting forward a notion of continuous iterative development including tasks such as code editing, refactoring, reverse engineering, and (in a larger context) reengineering.

Of course software visualization tools cannot ignore the current evolutionary/highly iterative view of software, even less so because they could be the key to propagate this view by combining and compressing large amounts of information into simple, yet expressive, visualizations.

Based on the assumption that such an evolutionary view of software will be predominant in the next years or decades, we want to briefly highlight some characteristics and features that software visualization tools of the future should possess to propagate such a view:

Symbiotic relationship with the development environment. The best way to propagate software visualization is to infiltrate existing development environments and complement the existing functionalities. We do not think that standalone software visualization tools would be used extensively, mainly because working people dislike changing their habits: a separate visualization tool introduces a disruptive latency between what one is seeing and what one is editing and/or manipulating. The market, represented by a few million programmers on this planet, will only adapt itself if there are evident technical, cognitive, and ultimately financial benefits provided by the software visualization facilities.

Refactoring support. Code refactoring, originally intro-

duced by Opdyke at the beginning of the 1990s [16], has become an issue in software development since its first mainstream appearance in the book of Fowler *et al.* which comes up with a list of dozens of ways to manipulate object-oriented software [9], most of which can preserve the behavioral semantics of the manipulated software, *i.e.*, it is certain that the system will still work after the manipulations. Such manipulations include renaming a class/method/attribute, pushing up and down methods/attributes from/to a subclass/superclass, transforming temporary variables into instance variables, etc. A few years ago, elegant and powerful implementations of software refactoring engines have made their way into existing development environments such as the Visualworks Smalltalk Refactoring Browser [19, 18] and the refactoring engine plugin of the IDE developed by the OpenSource Eclipse project[1]. It is clear that if a software visualization tool is to become a preferred way of looking at software, the manipulations must be possible as part of the software visualization tool and should be rendered visually, if only by updating the view given by the visualization tool. However, a technical problem is given if the source code of the development environment or the refactoring engine cannot be obtained and understood by the visualization tool makers.

Multi-user support. Complex software systems are being designed and developed by many people concurrently. To support a cooperative view as can be done with collaborative/versioning tools (such as the concurrent versions system CVS[2], Microsoft SourceSafe[3], and VisualWorks Store[4]), a good visualization tool would certainly need to visually render the current point of interest (*i.e.*, the position) of the developers and their most recently changed software artifacts.

Evolution analysis support. Software systems are constantly being evolved (at whatever pace) to cope with new requirements and to integrate bug fixes. The study of the past, present, and future of software systems is the research focus of the expanding and increasingly interesting field of software evolution research [14]. It would be useful for the developers to be able to replay the past lifetime of a class or a group of classes. This could provide an important source of information for decision making. Moreover, it could also help to identify costly parts (*e.g.*, if a class is changed over and over again, it is costing more than other parts in the system) or obsolete parts (*e.g.*, if a class is never changed, it is either dead code or good code). A thorough knowledge of the history of a system represents important information about that system.

Unification of information sources. There is a great spectrum of different sources of information about a software system. Apart from the primary one, the source code itself, one can also take into account documentation, bug reports, comments in the source code, UML diagrams, CRC Cards, user stories, unit tests, etc. Software visualization is an ideal vehicle to unify all these sources into one data pool which can then be visualized. Of course most of these data sources come without a formal definition and must be formalized before they can be integrated into any visualization. A simple example are comments in the source code: in the Java programming language the tool JavaDoc parses the declarations and documentation comments in a set of source files and produces a group of cross-linked HTML pages describing the software artifacts. An attempt to formalize these comments to use them for reverse engineering purposes has for example been proposed by Torchiano [24]. A visualization tool could display the comments for example as tool tips when the point of attention of the viewer is moving around. The benefits of these formalizations is that several of these informal sources of information could enrich the already present visualizations, thus augmenting the amount of information transmitted by them. The technical problems involved with such a unification are not to be underestimated, and even a standardization of the information sources (*e.g.*, with XML [7, 8]) will only solve part of the problems. An example of an open problem is keeping the various sources synchronized.

Spectrum of views. Various software visualization tools visualize software in different ways. Some of the tools propose views residing at different/complementary levels of granularity and visualize also different kinds of information (classes, applications, collaboration, subsystems, etc.). A good software visualization tool should not only propose good/complementary views, but also keep the views synchronized between them and allow the user to easily define new views. This is important in a reverse engineering context where a software visualization tool must be specialized to take case study-specific aspects into account.

Real-world validation. A crucial test that software visualization tools must undergo is certainly the industrial validation: the real world has many challenges, such as scaling up, being able to even parse the system or extract whatever information which can be fed into

[1] See http://www.eclipse.org/ for more information.

[2] See http://www.cvshome.org/ for more information.

[3] See http://msdn.microsoft.com/ssafe/ for more information.

[4] See http://www.cincomsmalltalk.com:8080/CincomSmalltalkWiki/ for more information.

the visualization tool. Through repeated confrontation with real case studies, one will also remark where the tool still needs general improvements or where there is need for specialization. A simple example of such a specialization are the acronyms often present within class names, which convey hidden semantic information about which subsystem or subarea of the system the class belongs to from the developer's point of view. After repeatedly encountering this kind of information one quickly wants to get an elegant way of modeling and handling it, for example by encoding it into nominal colors.

4 An Example

In this section we want to give a simple example of how some of the previous features can be achieved without a huge effort, although we do not want to minimize the technical difficulties that some of these points involve and which we still did not solve yet.

In Figure 1 we see a screenshot of the VisualWorks Refactoring Browser which we extended to accomodate visualizations provided by our software visualization tool CodeCrawler [11, 12]. In the figure we see a Class Blueprint view [13] of the currently selected class. Moreover we see that all methods selected in the browser (actually a complete method protocol 'private' has been selected) are also selected in the visualization. Furthermore a rename refactoring is being performed on one method of this class. Note that the visualization occupies the space normally used for displaying the method body. However, since during the browsing of the class (*e.g.,* looking for a certain method, method protocol, or attribute) the method body panel remains often empty anyway, this is not such a severe problem. Our implementation (re)uses the refactoring engine of the Refactoring Browser and thus allows us to perform even more complex refactorings like push-up and pull-down of methods or attributes. After the software has changed, our tool gets notified and automatically redisplays an updated view of the software.

The main drawback of our current integration is that between selecting a class or a group of classes and their visualization in the browser takes a few seconds[5]: this latency introduces wait times which disturb the viewer. We have already taken some countermeasures by implementing a cache which yields (the cached visualizations) in less than one second, but our personal experience shows that even small latencies disturb the viewer.

[5] from 2 to 20 seconds, depending on the number of classes and contained methods/attributes, measured on a PPC 500 MHz Apple G4.

5 The Goal?

The future of software visualization can hardly be predicted, we think however to be able to predict a possible and desirable goal that at least some program visualization tools will inevitably try to achieve: *quasi-real-time static software visualization*. Versioning systems like CVS have introduced a novel way of handling source code: they allow us to retrieve any version of any source file ever written by any person. If software visualization is to become the preferred way of developers of looking at source code, we cannot ignore the issue of this quasi-real-time: there must be a dependency mechanism between the versioning tool, the integrated development environment, and the software visualization tool: *e.g.,* as soon as someone changes a part of the system several people are editing and manipulating at once, they must be notified by the change at once. This is also a place where software visualization can fully exploit its potential: notifying the developers of system changes by means of text boxes, dialogs, log files, etc. are all clumsy approaches compared to literally *seeing* the changes happen and the system grow/shrink/change its shape in quasi-real-time.

6 Discussion

Should the goal described above be achieved, the result would be a real-time visual collaborative environment in which software engineers develop a system together, and in which they all have a common view of the system, namely the one proposed by the software visualization tool. This of course makes the achievement of the goal heavily dependent on the quality and the success of such a software visualization tool: should the users (*i.e.,* the software engineers) dislike or disagree with the proposed visualization, they will not accept it as part of their mental model of the system. Therefore, before software visualization tool providers can give it a try at such a long-term vision, they must first cope with the following (and many other) issues:

Human-Computer-Interaction issues. The field of Human-Computer-Interaction (HCI) deals with how humans interact with computer and with how to increase the quality of the interactions. User Interface design [6, 17] plays an important part in this context, and is also an issue in context of software visualization: the better a human viewer can interact with the visualizations, the more (s)he will think that the visualizations are useful. These are largely unsolved issues which can only increase in the context of real-time- or 3D-visualization. Another obstacle are the current input devices we are using to communicate with a computer: a computer mouse is for example

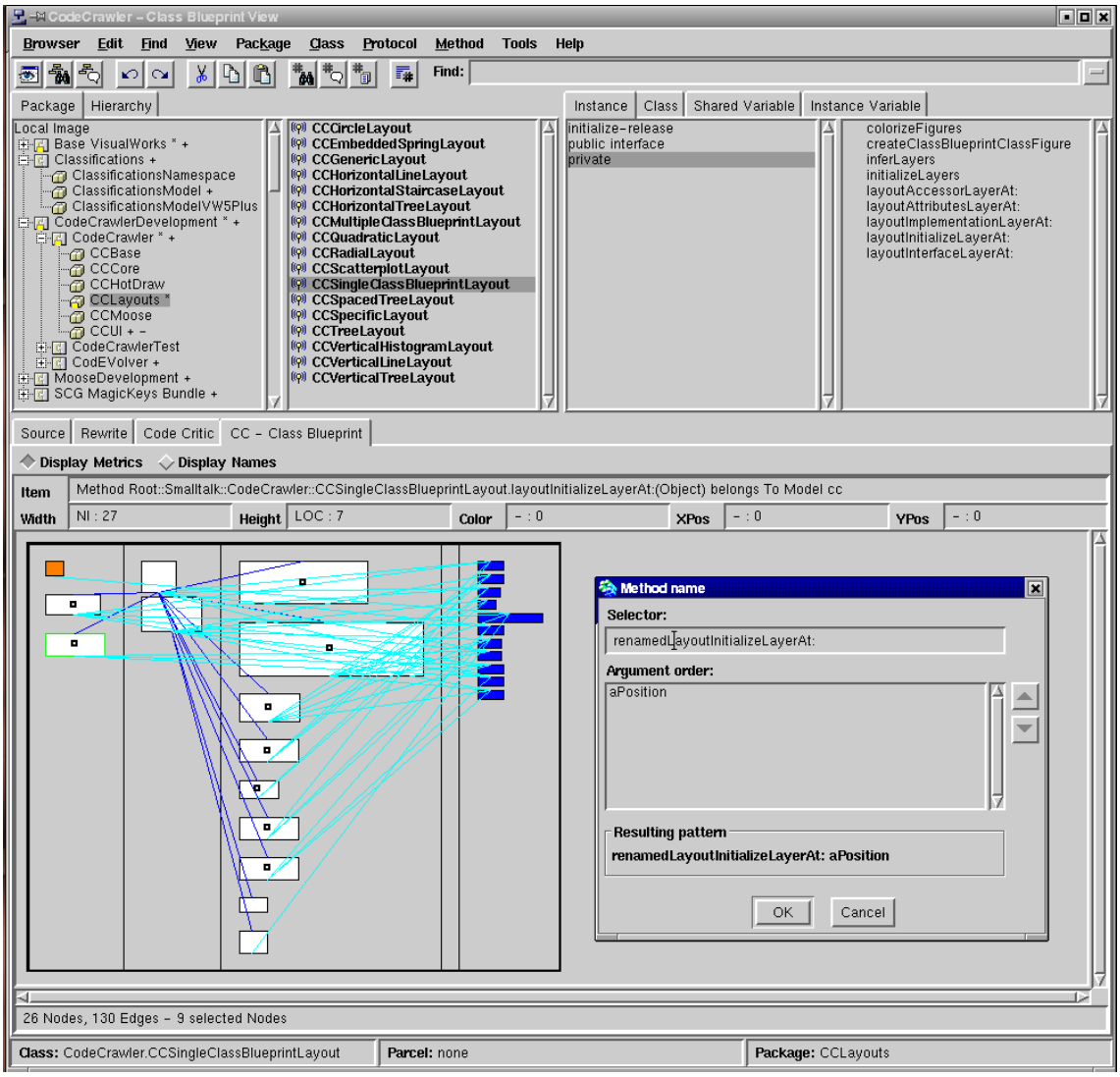

Figure 1. The integration of CodeCrawler with the VisualWorks Refactoring Browser.

not an ideal device to navigate three-dimensional information spaces, trackballs and other gadgets may be more appropriate, but are far less present at people's working place, and therefore not a recognized industry standard.

Scalability issues. Scalability has always been an issue in reverse engineering and reengineering, mainly because the examined subject systems are usually very large and complex. Although the ever-accelerating computer hardware can solve a part of this issue, our own experiments in the field of software evolution have shown that the massive amounts of data (hundreds of versions of systems which contain hundreds of classes and tens of thousands or software artifacts) put a heavy strain on even the fastest hardware. Therefore an important part of the scalability problem must be solved on the software side. Note also that a different kind of scalability problem, a purely visual one, came up during our experiments on software evolution: when a tool visualizes thousands of software artifacts the visualized items either take up too much space (generating navigation problems) or, in order to fit on a display, become too small to be interacted with.

7 Conclusion

New techniques like Design Patterns [10], Code Refactoring [9], and methodologies like eXtreme programming

[1] and Agile Development [5] have changed the way developers see software: we are more and more going towards an evolutionary view of a quasi-living software systems. This view is further amplified by the right tool support like refactoring engines, multi-way browsers, etc. We are convinced software visualization still does not exploit its full potential in such an evolutionary context, on the contrary it is rather being ignored so far. The future challenge of software visualization is thus to prove its value in the tough arena of mainstream (professional) software development.

Acknowledgments

We would like to thank Gabriela Arévalo and Stéphane Ducasse for commenting drafts of this paper.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[2] J. Bertin. *Graphische Semiologie*. Walter de Gruyter, 1974.

[3] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization - Using Vision to Think*. Morgan Kaufmann, 1999.

[4] E. J. Chikofsky and J. H. Cross, II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, Jan. 1990.

[5] A. Cockburn. *Agile Software Development*. Addison Wesley, 2001.

[6] A. Cooper. *About Face - The Essentials of User Interface Design*. Hungry Minds, 1995.

[7] S. DeRose, E. Maler, and D. Orchard. XML Linking Language (XLink) version 1.0 - w3c proposed recommendation 20 december 2000. Technical Report PR-xlink-20001220, World Wide Web Consortium, Dec. 2000.

[8] e. a. Didier Martin, Mark Birbeck. *Professional XML*. Wrox Press Ltd., 2000.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[11] M. Lanza. Codecrawler - lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409 – 418. IEEE Press, 2003.

[12] M. Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, may 2003.

[13] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.

[14] M. M. Lehman and L. Belady. *Program Evolution - Processes of Software Change*. London Academic Press, 1985.

[15] D. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In Soloway and Iyengar, editors, *Empirical Studies of Programmers, First Workshop*, pages 80–98, 1996.

[16] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[17] J. Raskin. *The Humane Interface*. Addison Wesley, 2000.

[18] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[19] D. Roberts, J. Brant, R. E. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.

[20] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization - Programming as a Multimedia Experience*. The MIT Press, 1998.

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

[22] M.-A. 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 Software Systems*, 44:171–185, 1999.

[23] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 12–21. IEEE Computer Society, 1997.

[24] M. Torchiano. Documenting pattern use in java programs. In *Proceedings of ICSM 2002 (International Conference on Software Maintenance)*, pages 230–233. IEEE Computer Society, IEEE Press, 2002.

[25] E. R. Tufte. *Envisioning Information*. Graphics Press, 1990.

[26] E. R. Tufte. *Visual Explanations*. Graphics Press, 1997.

[27] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.

[28] C. Ware. *Information Visualization*. Morgan Kaufmann, 2000.