

A gentle overview of software visualisation

Marian Petre ⁽¹⁾ and Ed de Quincey ⁽²⁾

(1) Centre for Research in Computing, The Open University, Walton Hall, Milton Keynes, UK

(2) School of Computing and Mathematics, Keele University, UK.

1. Introduction

Software design is a realm of messy or “wicked” problems that are often too big, too ill-defined, and too complex for easy comprehension and solution (DeGrace and Stahl, 1998). Software itself is created, complex, abstract, and difficult to observe. Software is different from created physical artefacts, because it lacks their tangibility and visibility (e.g., What does a compiler look like? What is the size, weight and shape of an operating system?). Code may be manifest, but how code *works* must be discovered and understood. Nevertheless, software interacts with objects in the physical world, often in complex and sophisticated ways. Software is dynamic, and software developers must reason not just about its properties, but about its behaviour – potentially complex behaviour – in time. Moreover, software is created and maintained in a social and organisational context which itself changes over time. Software ages as this social and organisational context evolves: teams change, knowledge decays, documentation falls out of date, intentions and rationale are forgotten over time (Ball & Eick, 1996).

Software visualisation uses visual representations to make software more visible. Visualisation concerns the graphical (or semi-graphical) representation of information in order to assist human comprehension of and reasoning about that information. There are a number of loosely distinguished themes within visualisation, among them information visualisation (representation of large data sets), software visualisation (visualisation for software engineering), and program or algorithm visualisation (representation of the structure and behaviour of algorithms for educational purposes). Despite their different foci, they have much in common, in terms of issues, techniques, and vocabulary. For example, visualization for software engineering must also include representation of the structure and behaviour of algorithms, but typically within the context of large-scale software development and maintenance, rather than the educational context of first understanding the algorithm. This paper focuses on software visualisation, the application of graphical techniques to represent different aspects of software – such as the source code, the software structure, runtime behaviour, component interaction, or software evolution – in order to reveal patterns and behaviours that inform software comprehension through all stages of software development.

Software visualisation can range from simple ‘pretty printers’, which use typographic enhancements such as indentation and colour coding (e.g., Baecker and Marcus, 1990), to 3D ‘landscapes’ representing the structure of large software systems, to multiple, linked, dynamic visualisations of the interaction of system components at runtime. Price, Baecker and Small (1998) defined software visualisation as “...the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software”. Visualisations may be based on static structures (views of the source code), or generated from dynamic data generated at runtime (views of execution, whether data flow or control flow). Software visualisation has its roots in earliest software development practice, when programmers watched the lights on the computer’s control panel and listened to the sounds of disk access to try to understand what the program was doing in the absence of other perceivable cues.

This paper presents a brief review of visualisation of software for software professionals. It is by no means comprehensive, but rather is intended as a light overview, to provide entry points into the literature, indicate the current state of the art, and discuss some of the persistent issues. It does not describe systems in detail, rather it considers factors that relate software visualisations to the tasks involved in software development, and it signals some (but certainly not all) systems that address particular perspectives.

PPIG Newsletter – September 2006

2. Software development tasks

Software development is a complex undertaking, involving a number of related tasks: specification, design, implementation, testing, debugging, maintenance, modification, re-engineering. Each of those tasks involves a number of cognitive demands, such as search, comprehension, analysis, problem solving, representation. It's no wonder that software development is so often described as complex. The complexity derives from the nature of the problems addressed, the diversity of the tasks involved, the nature of the artefacts produced, the changing social and organisational environment in which it is conducted, and the impact of time on environment, needs, teams, and artefacts.

Fundamentally, software visualisation is concerned with software comprehension, because comprehension underpins all stages and tasks of software development: design, debugging, maintenance and modification all require sufficient understanding of the software. In an ideal world, software is well-structured, testing is designed alongside code, documentation reflects appropriate models and rationale, and changes in requirements and modification of software are reflected in the documentation (Charters, Thomas and Munroe, 2002). However, as teams change and documentation falls out of date, code becomes the only guide to system structure and behaviour (Ball and Eick, 1966), and maintenance and evolution tasks are hindered by the inability of developers to comprehend system components and their interactions.

Even comprehension is complex, influenced by which aspect of the software is in focus: structure, errors, bottlenecks in execution. Each requires a different understanding – and a different view – of the software. Hence, it is difficult to consider software visualisation without also considering the task it is meant to support, and it is unlikely that any single software visualization tool can address all software development tasks simultaneously (Maletic, Marcus and Collard, 2002). The challenge is to identify the most appropriate visualisation for a given task. Even simple tools can improve software comprehension, if they're the right ones.

3. What do people visualise?

Typical tasks which software visualisation (SV) serves include design, bug detection, comprehension of legacy systems, and maintenance. Gómez Henríquez (2001) summarised: “Generally speaking, we can say SV systems are mainly used for the following purposes: 1) Program behaviour exhibition, normally for pedagogical purposes; 2) Logical debugging; and 3) Performance debugging.” (p. 7) His list is a fair reflection of the focus of the majority of existing software visualization systems, however, the literature reflects a more detailed set of focal tasks, and the list might reasonably be extended to include things like software behaviour exhibition for comprehension of existing large-scale systems. For example, Maletic, Marcus and Collard (2002), in their task-oriented discussion of software visualization, list: development activities, debugging, testing, maintenance and fault-detection, re-engineering, reverse engineering, software process management, and marketing. During early development of new software systems, visualisation of software systems can be useful in capturing requirements and aiding collaboration between developers. Software visualisation allows developers to discover quality defects automatically during the software development life cycle and is particularly applied to legacy code maintenance where maintainers may not have previously seen the system.

Petre (2002), in reporting an industry-based study of expert use of software visualization in software generation (rather than maintenance), notes that expert software developers themselves talk about software visualisation with respect to three major activities: comprehension (particularly comprehension of inherited code), debugging, and design reasoning. They build different visualizations for “low-level aspect visualization” (for comprehension of software behaviour, debugging and tuning) and for “conceptual visualization” (for design reasoning). The two categories of tool differ in how they are used. “The low-level aspect visualisations tend to be used to debug the artefact. They pre-suppose that the expert’s understanding of the artefact is correct, and they examine the artefact in order to investigate its behaviour. The conceptual visualisations tend to be used to debug the concept or process – to reason about the design.”

PPIG Newsletter – September 2006

The point to note (argued as well by Maletic, Marcus and Collard, 2002) is that different tasks require different information. Consider some examples:

- **design:**
How should the problem be interpreted (which problem should I solve)? What are the requirements? What is the structure of the solution? Does the conceptual design meet the specification? Design requires an understanding of problem and context, and ultimately of how those are interpreted and how they addressed the solution. It requires concept-oriented representations, rather than code-oriented ones. (e.g., Petre, 2002, describes conceptual visualisations used by experts)
- **comprehension of large software systems:**
How does the code *work* (e.g., Jinsight – DePauw, Kimelman and Vlissides, 1998, and De Pauw et al., 2001; PV from IBM – Kimelmann, Rosenburg and Roth, 1998)? Where does the complexity lie and how can it be made visible? In the context of parallel systems: how is activity distributed (e.g., Zhou, Summers and Caudell, 2003; Blochinger, Kaufmann and Siebenhaller, 2005)? In the context of distributed systems: how do components / agents / processes interact? What services are available, and how are they deployed? (e.g., Frishman and Tal, 2005)
- **performance tuning:**
Given that the software does what it's meant to, does it do it as efficiently as possible? What are the activity 'hot spots'? Where are the bottlenecks? Are there 'fossils' (unused code)? In the context of distributed systems: how is failure identified and accommodated? Complexity analysis and tuning are particularly important in the largest, most complex systems, e.g., in considering distribution of work in parallel systems. (e.g., ParaGraph – Heath and Etheridge, 1991; PV from IBM – Kimelmann, Rosenburg and Roth, 1998; Mu et al., 2003, Paradyn – Miller et al., 1995)
- **debugging:**
Does the implementation behave as expected? Does the implementation do what's specified? Debugging requires an overall understanding of the software, plus detailed understanding of specific components. It requires direct links to the source code as well as execution information. (e.g., Jacobs and Musial, 2003, enhance UML with dynamic program execution state information and high level abstractions)

Visualization systems oriented to different tasks may use similar techniques (or widely differing ones), but a given technique is interesting primarily in terms of its appropriateness to the task.

A number of taxonomies have been proposed. Two of the most interesting are those by Price, Baecker and Small (1998) and Maletic, Marcus and Collard (2002). The now classic taxonomy by Price, Baecker and Small (1998) identified six major categories of attributes: scope (the range of programs the system can take as input), content (the subset of information about the software that is visualised by the system), form (the characteristics of the output of the visualization), method (how the visualization is specified), interaction (how the user interacts with and controls the visualization), and effectiveness (how well the system communicates information to the user). Each of these categories in turn has sub-categories. Maletic, Marcus and Collard (2002) orient their framework on tasks, rather than functionality *per se*, in order: "to emphasize the general tasks of understanding and analysis during development and maintenance of large-scale software systems". Within this programming-in-the-large focus, they identify five "dimensions" of software visualization: tasks (*why* the visualization is needed), audience (*who* will use the visualization), target (*what* aspects of the software are to be represented), representation (*how* it will be represented), and medium (*where* it will be represented). The emphasis on who, what, where, when, how and why makes the framework memorably accessible.

4. What makes a ‘good’ software visualization?

Young and Munro (2003) propose a list of desirable properties for a given visualization:

- *Simple navigation with minimum disorientation*: the visualisation should be structured and should include features to aid the user in navigating the visualisation, for example using techniques such as landmarks to reduce the user’s chance of becoming ‘lost’.
- *High information content*: “Visualisations should present as much information as possible without overwhelming the user.”
- *Low visualisation complexity, well structured*: Well structured information should result in easier navigation. Low complexity trades off with high information content.
- *Varying levels of detail*: Granularity, abstraction, information content and type of information should vary to accommodate users’ interests.
- *Resilience to change*: Small changes of content or shifts in attention should not cause major differences in the visualisation (cf. ‘viscosity’, Green and Petre, 1996)
- *Good use of visual metaphors*: Metaphors provide cues to understanding. Mackinlay (1986) articulated two criteria for evaluating the mapping of data to a visual metaphor: expressiveness and effectiveness. “Expressiveness criteria determine whether a graphical language can express the desired information. Effectiveness criteria determine whether a graphical language exploits the capabilities of the output medium and the human visual system.” These are particularly pertinent and challenging in this context.
- *Approachable user interface*: The user interface should be flexible and intuitive, and should avoid unnecessary overheads. Shneiderman (1996) suggests that: “A useful starting point for designing advanced graphical user interfaces is the Visual Information-Seeking Mantra: overview first, zoom and filter, then details on demand.” He further enumerates seven tasks that should be supported: overview, zoom, filter, details-on-demand, relate, history, and extracts.
- *Integration with other information sources*: It is desirable to be able to link between the visualisation and the original information it represents (the source code). (cf. Charters et al.’s comments on “round trip visualizations”, 2003)
- *Good use of interaction*: Interaction provides mechanisms for gaining more information and maintaining attention.
- *Suitability for automation*: “A good level of automation is required in order to make the visualisations of any practical worth.”

This is clearly an ambitious ‘wish list’, aspirational rather than pragmatic. It begs many of the questions of cognition and insight that were articulated in Petre, Blackwell and Green (1998). A focus on a given purpose and context might assist designers in negotiating the tradeoffs inherent in the list, drawing on the increasing repertoire of analysis visualization techniques, but means and mechanisms remain largely a matter of ‘art’.

5. Advances in recent years

Card, Mackinlay and Shneiderman (1999) “propose six major ways in which visualization can amplify cognition...: (1) by increasing the memory and processing resources available to the users, (2) by reducing the search for information, (3) by using visual representations to enhance the detection of patterns, (4) by enabling perceptual inference operations, (5) by using perceptual attention mechanisms for monitoring, and (6) by encoding information in a manipulable medium.” (p. 16) Do we really understand enough about the design of software visualizations to realise these gains? Petre, Green and Blackwell (1998) raised a number of cognitive issues facing software visualisation, many of them identifying the gaps between the sorts of potential gains enumerated above, and the specific insights and techniques required to provide sufficiently selective and well-designed visualisations which focus appropriately on human cognitive activity. Many of the advances in the past 6-10 years in software visualization can be viewed as progress toward the potential gains, through increasingly powerful and

PPIG Newsletter – September 2006

flexible tools and an increasing attunement to the human processes of software development.

Scale

One of the biggest issues in software visualization is scale: systems are increasingly large and complex. Software developers observe that “simply repackaging massive textual information into a massive graphical representation is not helpful”, and that they need means of reasoning about artefacts too enormous to encompass fully in one view (Petre, 2002). Selection and abstraction are crucial in addressing scale; the challenge is to find appropriate and meaningful selections and abstractions in order to provide a useful focused view. There have been a variety of approaches to filtering, zooming and selection, particularly techniques that afford user control via attributes or direct manipulation. (e.g., Marcus, Feng and Maletic 2003)

Linked, multiple representations

Software visualizations usually present two or more representations of the same information, e.g., source code and a graphical representation of execution ‘hot spots’. Petre, Blackwell and Green (1998) discuss the cognitive overheads of multiple representations – as well as how they might be used. Ideally, using multiple representations effectively makes more information, different selections, and different perspectives available and hence structures and expands the exploration space. There are technical challenges in linking representations, and especially in linking visualizations of execution back to source code, which Gómez Henríquez (2001) calls the “missing link” (e.g., Lieberman and Fry, 1998; Imagix 4D). Increasingly, attention is being given to linked representation among multiple levels of abstraction and between static and dynamic visualizations (e.g., Gestwicki and Jayaraman, 2005; Knight and Munro, 2001; Storey, Best and Michaud, 2001). Petre (2002) describes dynamically-linked representations as a common feature of the visualizations software developers build for themselves for debugging activities. Among others, Reiss (Reiss, 2003; Reiss and Renieris, 2005) is trying to offer the linkage between source and execution visualization into real-time with JOVE: “We have developed a dynamic Java visualizer that provides a view of a program in action with low enough overhead so that it can be used almost all the time by programmers to understand what their program is doing while it is doing it.” (Reiss, 2003).

Attention to software as evolving

Although configuration management has been a long-term issue (e.g., Gulla, 1992), increasing attention has been given to software as an evolving artefact for which temporal views are important (e.g., Burch, Diehl and Weissgerber, 2005; Collberg et al. 2003; Gall, Jazayeri and Riva, 1999; Pinzger et al., 2005; Voinea, Telea and van Wijk, 2005; Fischer et al., 2005). Tracking software evolution is necessary to understanding the impact of accumulated changes on the architecture of a software system, as well as for identifying ‘drift’ in the conceptual design, for detecting problems emergent from the changes, and for detecting historical trends in software evolution. There is also recognition that issues with software may only become manifest in the field (Orso, Jones and Harrold, 2003).

User selection / customisation of visualisations

As the field has matured, attention has shifted somewhat from the development perspective (what can be built) to the user perspective (what makes a difference). There is increasing empirical attention to users and use (e.g., Douglas, Hundhausen and McKeown, 1995). There is also increasing attention to the potential to exploit user insight in shaping visualizations to purpose, by engaging users in tuning and customizing visualization tools (e.g., Reiss, 2002, Panas, Lincke and Lowe, 2005). As the number of techniques multiply, attention is being given to extensible visualization environments (e.g., Wang et al., 2003)

Use of 3D

The introduction of 3D is largely a technical advance – driven by what’s possible, in advance of a full understanding of what’s useful or appropriate. The hope is that 3D will overcome some of the limitations of 2D in terms of the number of attributes and types of relationships that can be visualized. It is an aspiration to find new metaphors to exploit the additional representational richness. (e.g., sv3D – Marcus, Feng and Maletic, 2003; elements of CodeCrawler – Lanza, 2004

PPIG Newsletter – September 2006

Awareness of human activities

Whereas most software visualization is oriented to the *artefact*, the software, visualization in other domains, such as computer-supported collaborative work (CSCW) and information visualization has attended more to the development *process* and the people conducting it. This use of visualization to promote awareness of human activities has begun to seep into software visualization (e.g., O'Reilly, Bustard and Morrow, 2005), sometimes subtly (e.g., as a consequence of attention to software change). Storey, Čubranić and German (2005) offer a perspective on this.

6. Continuing issues

Gómez Henriquez (2001) identifies 4 “difficulties” in software visualization: program instrumentation, visualization definition, scalability, and the “missing link” (as discussed above). These are largely technical issues. It is our view that the big issues that face software visualization are to do with matching visualizations to human needs. The big technical challenges to the analysis and selection techniques needed to tailor visualizations to support human cognition.

What's being visualised

What can we and can't we visualize? Are we visualising the right things? Currently, it is still arguable that what is visualized is what can be visualized, not necessarily what needs to be visualized. Tools that simply re-present available information (e.g., simplistic diagram generation from program text) don't provide insight. Software developers seek facilities that contribute to insight, e.g., useful abstractions, ready juxtapositions, information about otherwise obscure transformations, informed selection of key information, etc. Visualization developers are still struggling with what Gómez Henriquez (2001) calls the ‘probe effect’: ensuring that the visualisation is reliable enough to ensure that what the user sees is what is really happening – whether or not it's what the user needs to see. Underlying the challenge to identify the most appropriate visualisation for a given task is the need to reach a well-founded understanding of what makes visualizations appropriate for given tasks. Most of the visualizations available focus on large-scale system comprehension – on helping developers understand how software works. Within that, different systems focus on different comprehension contexts, such as parallel or distributed systems.

Visualizing concepts and intentions vs. re-presenting implementations

Is visualization reaching beyond mere re-presentation of the code? Are we visualising design, implementation, or code? Usually the latter two, only rarely the former. There are few visualisations yet to support conceptual design. There is a need to provide conceptual visualizations, rather than just performance or data flow. This highlights the need to make available information that is not typically contained in the source code: information about the originators' intentions and models of the software. According to expert software developers: “Most tools are generic and hence are too low-level. Tools that work from the code, or from the code and some data it operates on, are unlikely to provide selection, abstraction, or insight useful at a design level, because the information most crucial to the programmer – what the program represents, rather than the computer representation of it – is not in the code. At best, the programmer's intentions might be captured in the comments. As the level of abstraction rises, the tools needed are more specific, they must contain more knowledge of the application domain. Experts want to see software visualised in context – not just what the code does, but what it means.” (Petre, 2002)

Intelligent selection:

The utility of visualization lies not in mere re-presentation of data, but in an appropriate and meaningful distillation and abstraction of the data in order to provide access to desired information about the software. That is, it's no good translating massive source code into an equally massive visualization; what's required is views on the artefact that disclose significant patterns within it. Tudoreanu (2003) discusses software visualization in terms of “cognitive economy”: minimising cognitive load by reducing the amount of information handled by the user and maximising the information pertinent to the user problem (which is different from reducing complexity related to visual displays). Cognitive economy requires that the visualization be customised for the problem at hand,

PPIG Newsletter – September 2006

tailored to the user's task and goals. This, in turn, requires some knowledge of the domain. According to Chi, "...good visualisations are coupled with good analysis algorithms. We can get the most power out of visualization if we use a sophisticated analysis computation that distils the data further from the raw data." (Chi, 2000)

7. Entering the software visualisation literature

Various visualisation tools are available on the web, on individual websites, and in compilations such as 'SCG Smallwiki CodeCrawler: A non-exhaustive list of software visualisation tools'. The literature on software visualisation is substantial, yet reasonably localised, making it relatively easy to find an entry point. Four anthologies give good overviews: *Software Visualization* (Eades and Zhang, 1996); *Software Visualization: Programming as a Multimedia Experience* (Stasko et al., 1998), which includes an authoritative 'Early History' chapter by Baecker and Price; *Software Visualization* (Diehl, 2002), which resulted from a Software Visualization Dagstuhl (i.e., day symposium) in 2001; and *Software Visualization: Theory into Practice* (Zhang, 2003), a collection of extended versions of refereed papers from a special issue of the cancelled *Annals of Software Engineering*. The first three encompass both software visualisation and program visualisation, the fourth focuses on software visualisation. In addition, there are several key conferences, including: ACM Symposium on Software Visualization (SoftVis) 2003, 2005, 2006; IEEE International Workshop on Visualizing Software for Understanding and Analysis; ICSE Workshop on Software Visualization, 2001; OOPSLA Workshop on Software Visualisation, 2001. A few other conferences also have relatively high concentrations of software visualisation papers, including: International Workshop on Program Comprehension (IWPC) and Visual Languages/Human-Centric Computing (VL/HCC), which has taken various names over the years. In addition, there are conferences on program visualisation, such as the biennial Program Visualization Workshops, 2000, 2002, 2004, and Visualisation of Software, reported in an April 2001 special issue of *Informatik*.

References

- Baecker, R., and Marcus, A. (1990) *Human Factors and Typography for More Readable Programs*. ACM Press.
- Ball, T., and Eick, S.G. (1996) Software visualization in the large. *IEEE Computer*, April 1996. 33-43.
- Blochinger, W., Kaufmann, M., and Siebenhaller, M. (2005) Visualising structural properties of irregular parallel computations. *ACM Symposium on Software Visualization*. ACM Press. 125-134, 212.
- Burch, M., Diehl, S., and Weissgerber, P. (2005) Visual data mining in software archives. *ACM Symposium on Software Visualization*. ACM Press. 37-46.
- Charters, S.M., Thomas, N., and Munro, M. (2003) The end of the line for software visualisation? *IEEE Second Workshop on Visualizing Software for Analysis and Understanding (VISSOFT)*. IEEE Computer Society Press.
- Chi, E.H. (2000) The Future of Software: Visualization+Computation. *Tools* (Future of Software special issue). Fawcette Technical Publishing.
- Collberg, C., Kobourov, S., Nagra, J., Pitts, J., and Wampler, K (2003) A system for graph-based visualization of the evolution of software. *ACM Symposium on Software Visualization*. ACM Press 77-86, 212.
- DeGrace, P., Stahl, L.H. (1998) *Wicked Problems, Righteous Solutions: A Catalog of Modern Engineering Paradigms*. Prentice Hall.

PPIG Newsletter – September 2006

DePauw, W., Kimelman, D., and Vlissides, J. (1998) Visualizing object-oriented software execution. . In: Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.), *Software Visualization: Programming as a Multimedia Experience*. MIT Press. 329-346.

DePauw, W., Mitchell, N., Robillard, M., Sevitsky, G., and Srinivasan, H. (2001) Drive-by analysis of running programs. *ICSE Workshop on Software Visualization*.

Diehl, S. (ed) (2002) *Software Visualization*. Springer.

Douglas, S.A., Hundhausen, C.D., and McKeown, D. (1995): Toward Empirically-Based Software Visualization Languages. *IEEE Symposium on Visual Languages* . 342-350.

Eades and Zhang, K. (eds) (1996) *Software Visualization*. World Scientific Publications.

Eick, S.G., Graves, T.L., Karr, A.F., Mockus, A., and Schuster, P. (2002) Visualizing software changes. *IEEE Transactions on Software Engineering*. **28** (4), 396-412.

Fischer, M., Oberleitner, J., Gall, H., and Gschwind, T. (2005) System Evolution Tracking through Execution Trace Analysis. *International Workshop on Program Comprehension*. 237-246.

Frishman, Y., and Tal, A. (2005) Visualization of mobile object environments. *ACM Symposium on Software Visualization*. ACM Press.. 145-154, 213

Gall, H., Jazayeri, M., and Riva, C. (1999) Visualizing software release histories: the use of color and third dimension. *The International Conference on Software Maintenance*. IEEE Computer Society Press. 99-108.

Gestwicki, P., and Jayaraman, B. (2005) Methodology and architecture of JIVE. *ACM Symposium on software Visualization*. ACM Press. 95-104.

Gómez Henríquez, L.M. (2001) Software visualization: an overview. *Informatik*, No. 2., 4 -7.

Green, T.R.G., and Petre, M. (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing* (special issue on HCI in visual programming), **7** (2), pp. 131 - 174.

Gulla, B. (1992) Improved maintenance support by multi-version visualizations. *The International Conference on Software Maintenance*. IEEE Computer Society Press. 376-383.

Heath, M., and Etheridge, J. (1991) Visualizing the performance of parallel programs. *IEEE Software*. September.

Imagix 4D, Imagix Corp. <http://www.imagix.com/products/products.html> [Accessed June 2006].

Jacobs, T., and Musial, B. (2003) Interactive visual debugging with UML. *ACM Symposium on Software Visualization*. ACM Press. 115-122.

Kimelman, D., Rosenburg, B., and Roth, T. (1998) visualization of dynamics in real world software systems. . In: Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.), *Software Visualization: Programming as a Multimedia Experience*. MIT Press. 293-314.

Knight, C., and Munro, M. (2001) Mediating diverse visualisations for comprehension. *Ninth International Workshop on Program Comprehension*. 18-25.

PPIG Newsletter – September 2006

- Lanza, M. (2004) CodeCrawler – polymetric views in action. *19th IEEE International Conference on Automated Software Engineering*. 394-395.
- Lieberman, H., and Fry, C. (1998) ZStep95: a reversible, animated source code stepper. . In: Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.), *Software Visualization: Programming as a Multimedia Experience*. MIT Press. 277-292.
- Maletic, J.I., Marcus, A., and Collard, M.L. (2002) A task oriented view of software visualization . *IEEE First International workshop on Visualizing Software for Understanding and Analysis*. IEEE Computer Society Press. 32-40.
- Marcus, A., Feng, L., and Maletic, J.I. (2003) 3D representations for software visualization. *ACM Symposium on Software Visualization*. ACM Press. 27-36, 207.
- Miller, B.P., Cargille, J.M., Irvin, r.B., Kunchithap, K., Callaghan, M.D., Hollingsworth, J.K., Karavanic, K.L., and Newhall, T. (1995) The Paradyn parallel performance measurement tools. *IEEE Computer*. **11** (28). 37-46.
- Mu, T., Tao, J., Schulz, M., and McKee, S.A. (2003) Interactive locality optimisation on NUMA architectures. *ACM Symposium on Software Visualization*. ACM Press. 133-141, 214.
- O'Reilly, C., Bustard, D., and Morrow, P. (2005) The war room command console – shared visualizations for inclusive team coordination. *ACM Symposium on Software Visualization*. ACM Press. 57-65, 210.
- Orso, A., Jones, J., and Harrold, M.J. (2003) Visualization of program-execution data for deployed software. *ACM Symposium on Software Visualization*. 67-76, 211
- Panas, T., Lincke, R., and Lowe, W. (2005) Online-configuration of software visualizations with Vizz3D. *ACM Symposium on Software Visualization*. ACM Press. 173-182.
- Petre, M. (2002) Mental imagery, visualisation tools and team work. *Second Program Visualisation Workshop*.
- Petre, M., Blackwell, A., and Green, T.R.G. (1998) Cognitive questions in software visualisation. In: Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.), *Software Visualization: Programming as a Multimedia Experience*. MIT Press. pp. 453-480.
- Pinzger, M., Gall, H., Fischer, M., and Lanza, M. (2005) Visualizing multiple evolution metrics. *ACM Symposium on Software Visualization*. 67-75.
- Price, B.A., Baecker, R., and Small, I.S. (1998) A principled taxonomy of software visualisation. *Journal of Visual Languages and Computing*. **4**(3), pp. 211-266.
- Reiss, S.P. (2002) A visual query language for software visualisation. *IEEE Symposium on Human Centric Computing Languages and Environments*. 80.
- Reiss, S.P. (2003) Visualizing Java in action. *ACM Symposium on Software Visualization*. 57-65, 210.
- Reiss, S.P., and Reneiris, M. (2005) JOVE: Java as it happens. *ACM Symposium on Software Visualization*. 115-124, 211.
- SCG Smallwiki CodeCrawler: CodeCrawler: A non-exhaustive list of software visualisation tools' <http://smallwiki.unibe.ch/codecrawler/anon-exhaustivelistofsoftwarevisualisationtools/>. [Accessed

PPIG Newsletter – September 2006

June 2006].

Shneiderman, B. (1996). The eyes have it: a task by data type taxonomy for information visualizations. *IEEE Symposium on Visual Languages*. IEEE Computer Society Press. pp. 336-343.

Stasko, J., Domingue, J., Brown, M.H., and Price, B.A. (eds) (1998) *Software Visualization: Programming as a Multimedia Experience*. MIT Press.

Storey, M.-A., Best, C., and Michaud, J. (2001) SHriMP views: an interactive environment for exploring Java programs. *Proceedings of the Ninth International Workshop on Program Comprehension*. 111-112.

Storey, M.-A., Čubranić, D., and German, D. (2005) On the use of visualization to support awareness of human activities in software development: a survey and framework. *ACM Symposium on Software Visualization*. ACM Press. 193-202.

Tudoreanu, M.E. (2003) Designing effective program visualization tools for reducing user's cognitive effort. *ACM Symposium on Software Visualization*. ACM Press. 105-114, 213.

Voinea, L., Telea, A., and van Wijk, J.J. (2005) CVSscan: visualization of code evolution. *ACM Symposium on Software Visualization*. 47-56, 209.

Wang, Q., Wang, W., Brown, R., Driesen, K., Dufour, B., Hendren, L., and Verbrugge, C. (2003) EVolve: an open extensible software visualization framework. *ACM Symposium on Software Visualization*. ACM Press. 37-46, 208.

Young, P., and Munro, M. (2003) Visualising software in virtual reality. *IEEE First International Workshop on Visualizing Software for Understanding and Analysis*. IEEE Computer Society Press.

Zhang, K. (ed) (2003) *Software Visualization: From Theory to Practice*. Kluwer Academic Publishers.

Zhou, C., Summers, K.L., and Caudell, T.P. (2003) Graph visualization for the analysis of the structure and dynamics of extreme-scale supercomputers. *ACM Symposium on Software Visualization*. ACM Press. 143-149.