Remixing Visualization to Support Collaboration in Software Maintenance

Margaret-Anne Storey Chris Bennett R. Ian Bull Daniel M. German
Department of Computer Science, University of Victoria
{mstorey, cbennet, irbull, dmg}@uvic.ca

Abstract

We propose that collaborative software visualization can improve team software maintenance. We first review how visualization can support software maintenance from the perspectives of system understanding, process understanding and software evolution. From this, we conclude that visualization tools are rarely designed to provide explicit support for collaborative authoring and sharing of views. We then provide an overview of research from a Computer Supported Cooperative Work perspective, and propose that this research should be applied to software visualization. We explore the opportunities and challenges this research focus presents and conclude that more attention paid to the social aspects of software visualization should improve both individual and team processes in software maintenance.

1. Introduction

Software maintenance is a cognitively challenging task that benefits from effective tool support for activities such as program understanding, debugging and testing. Software maintenance tools can use visualization to reveal information that is not obvious from directly examining the system and related artifacts. Despite the many novel visualization techniques developed by researchers, uptake by industry has been relatively slow. We speculate that one reason may be a lack of attention to the social aspects of software maintenance.

Software maintenance is inherently a social activity - large systems typically involve teams of developers and the participation of many stakeholders throughout the software lifecycle [1]. Computer supported collaborative work (CSCW) explores how tools can more effectively support work practices within socio-technical systems, such as software maintenance. To date, there has been limited research on how tools support collaborative software maintenance (some exceptions being work by Ko *et al.* [2] and Whitehead [3]). Even less research has explored how visualizations can support collaborative software maintenance.

In this paper, we explore the intersection of CSCW, information visualization, and software maintenance (Figure 1). We suggest that software visualization tool designers borrow theories and tools from the disciplines of social computing and CSCW. We explore the opportunities and highlight the challenges that this line of research introduces.

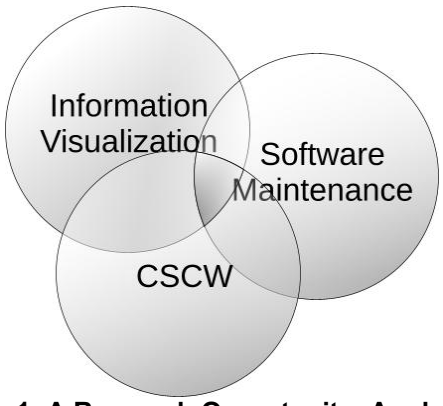

Figure 1: A Research Opportunity: Applying CSCW and information visualization to software maintenance

This paper is structured as follows. In Section 2, we briefly review how visualization has been used to support software maintenance. Next, we provide an overview of the concepts, theories and tools from CSCW research and examine how these have been applied to software maintenance and information visualization (Section 3). In Section 4, we consider the role of CSCW research in how visualization supports collaborative maintenance activities and explore some opportunities and challenges that this presents. We conclude the paper by proposing that research that focuses on the social aspects of software visualization will improve how teams and individuals carry out software maintenance (Section 5).

978-1-4244-2655-3/08/$25.00 © 2008 IEEE

2. Visualization in software maintenance

Software maintenance requires an understanding of both software systems and the processes by which they are engineered. Since systems change over time, it is also important to understand the evolution of both systems and processes. Visualization has been applied to software maintenance to communicate information through images, diagrams, and animations. In this section, we review selected visualization tools and techniques that support understanding of the system, process, and evolution.

2.1 Understanding the system

Understanding a system is a precursor to many software maintenance activities, consuming more than 50% of reverse engineering effort [4]. Understanding software is a cognitively challenging task that has benefited from software visualization techniques. Graphs (e.g. call graphs, class diagrams, and reverse engineered sequence charts) are often used to visually represent concepts and relationships. Some uses of visualization include support for understanding static structure, runtime behaviour, architecture, code metrics, patterns, and re-design.

Understanding static structure is supported by tools such as SHriMP [5] and Creole [6], which use nested graph visualizations to explore objects and relationships within a software system.

Dynamic analysis tools, such as SEAT [7], SCED [8], and Jinsight [9], visualize application behaviour in the form of a sequence diagram or call tree. TraceCrawler [10] provides an interactive 3D visualization of feature execution in the context of static structure. Dynamic behaviour can also be visualized during debugging by linking program execution state to UML diagrams [11]. TPTP [12] visualizes profiling and runtime performance.

Architecture and design recovery tools, such as Rigi [13] and the Portable Bookshelf (PBS) [14], provide a conceptual understanding at a higher level of abstraction. Re-documentation tools, such as Reef [15] and Doxygen [16], capture the results of architecture and design recovery.

Code metrics are frequently used as a quality assurance tool. Bieman *et al.* describes the use of box plots and class diagrams to highlight change-prone classes and their interdependencies [17]. Systa *et al.* use graph visualization to show complexity coupling, and inheritance metrics [18].

Pattern recovery is a form of design reconstruction that is particularly relevant to OO systems [19]. Trese and Tilley suggest that a visualization that documents the pattern is preferable to scattered code comments [19].

Design, or at least re-design, is arguably part of software maintenance. Tools, such as Rational Rose XDE [20], support visual design activity using UML visualizations. Pounamu [21] extends this to support distributed collaborative design activities.

2.2 Understanding the process

An understanding of the software engineering process supports project management, quality assurance, and day to day maintenance tasks. Visualization contributes to this through, for example, a graphical summary of test coverage, team assignments, and defects.

Panas *et al.* propose a 3D visual approach to depict software cost-related information in support of software maintenance [22]. The Lagrein tool [23] helps managers and developers visualize user requirements and development efforts. The Xia tool [24] visualizes code ownership of system artifacts.

Process management can also benefit from visualization. Project plans are often visualized as Gantt or Pert charts to show task dependencies and task progress. Change requests, change traffic, and configuration management are candidates for visualization. For example, Palantir [25] provides visualizations of configuration management activities. Jazz [26] displays charts of the maintenance process, e.g. work-items completed and health of the build.

Visualization has also been used in testing. Jones *et al.* describe a program named TARANTULA that provides a SeeSoft [27] style visualization of test coverage, mapping test state to source code [28].

2.3 Understanding evolution

Understanding the evolution of a system and the processes by which it was engineered is a precursor to many software maintenance activities. Some software maintenance tools provide engineers with visualizations of how systems evolve over time and how development teams work and collaborate.

Tools such as SeeSoft [27] and Beagle [29] mine source control repositories to visualize source code evolution. CodeCrawler [30] visualizes software evolution as a matrix of class attributes changing over time. SoftChange [31] uses histograms and graphs to show evolution statistics and relationships between files and authors. Ogawa [32] uses small multiples to provide a summary view of social groups that interact over the lifetime of a project. For more information on visualization of human activities in software development, see Storey *et al.* [33].

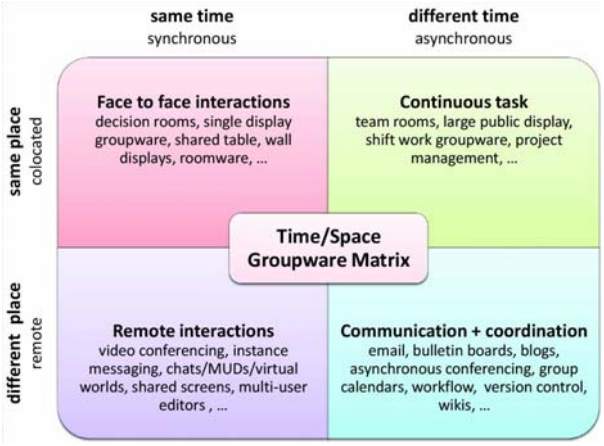

Figure 2: Time / Space Groupware Matrix (from Wikipedia, in the public domain)

2.4 Adoption

To better understand the adoption of software visualization tools, Basil and Keller [34] surveyed commercial and research users of such tools. They determined that these tools are particularly useful in software maintenance, rather than development, and noted that code comprehension was the primary use of these tools. Other researchers have found that visualizations are appropriate for some tasks but text-based solutions may be preferred for others [35], an important consideration for tool designers.

While visualization has played a key role in many software maintenance activities, industrial adoption of many visual techniques is still sparse. Adoption can be slow when the tools proposed are not industrial strength nor integrated with commonly used environments. Few tools are designed with an adequate consideration of the social aspects of maintenance, such as communication, awareness, and collaboration. CSCW is a field of research dedicated to exploring how technology can facilitate collaborative work. The application of lessons learned from CSCW holds great potential for addressing social barriers to the adoption of visualization tools in software maintenance.

3. CSCW

Research on improving human computer interaction has shifted from considering human factors to enhancing systems for human actors [36]. In this section, we provide a brief overview of the concepts, tools and evaluation approaches within CSCW. We then examine the role that CSCW research plays in information visualization and software maintenance.

3.1 Concepts and tools

Key concepts in CSCW include *communication*, *location*, and *synchronization*. *Communication* refers to how humans communicate, e.g. face-to-face or using audio, video or text. *Location* describes whether the collaborating individuals are in the same place or remotely located from one another. *Synchronization* describes whether participants are collaborating in a *synchronous* fashion (at the same time) or *asynchronously* (different times). Figure 2 displays a time/space matrix showing variations on collaboration and types of tools that can support collaborative work [37].

Additional concepts in CSCW include *awareness* and *coordination*. *Awareness* refers to "an understanding of the activities of others, which provides a context for your own activity" [38]. We may take awareness for granted when we work in the same location where gesturing is easy and one can look over a co-worker's shoulder. However, many systems lack support for awareness, a problem when people try to use these systems for distributed collaboration. Distributed collaboration relies on explicit cues to create awareness, e.g. indicating who is currently working with a shared artifact and using changing colour as a *feedthrough* mechanism to alert users when an artifact has been changed.

Coordination refers to what people have to do in order to work together on a task. *Articulation work* [38] is additional work beyond the defined formal work task (e.g. allocation of tasks, distribution of resources, and scheduling of tasks). Although, much of the CSCW literature seems to imply that people willingly or intentionally cooperate, this is often not the case and collaboration may occur unintentionally and even within a competitive environment [39].

Another aspect that CSCW tool designers need to consider is the modes of work that groups engage in, and their motivation for the tasks they do. McGrath suggests the following modes: inception, execution, problem-solving and conflict resolution [40]. From his studies of groups *in the wild*, he identifies three functions of group work: the intended production work, work that improves group well-being, and work that provides member-support. The last two should also be considered when designing tools.

CSCW research has produced a variety of tools that use visualization to support the concepts described above. *Groupware* refers to software that supports group interactions [41]. Telepointers, avatars and video images are examples of how a user can be embodied within the groupware system. Visualization techniques are also often used to show feedthrough on shared artifacts, and multiple views are particularly important

for showing different aspects of the shared information and how collaboration is occurring.

In the next section we review how CSCW has been applied to both software maintenance and to information visualization.

3.2 Applying CSCW research

The notion of collaboration in software maintenance is not new. Parnas noted that assigning programmers to modules with low coupling would decrease communication [42]. Brooks talks about the challenges in collaborative work [43], and Olson *et al.* discovered that distance matters in collaborative environments and does slow down development work [44]. Ye's socio-technical framework on programmers emphasizes that tools should be used to reduce interruptions (e.g. through showing awareness) and should help in finding experts (potentially by showing experts visually) [45]. Ye also talks about the importance of tools fitting into the existing environment. In addition to coding, there are many activities in software maintenance that are entirely, or to a large degree, collaborative, such as code review, redesign and testing.

CSCW has also been applied in the field of information visualization. Collaborative visualization refers to a subset of CSCW applications in which control over parameters or products of the scientific or information visualization process is shared [46]. When exploring collaborative visualizations, we need to consider who authors and uses these views. Visualizations are not collaborative if they are created by an individual for their own use. However, individually authored visualizations can be shared. Collaboration increases if they are created and edited in a collaborative manner. Interestingly, in Bly's studies on how people use diagrams, she noticed that the act of creating a drawing was often more important than the final diagram itself [47].

Collaborative visualization can also be discussed with respect to the dimensions in the time/space matrix (Figure 2). Brodlie [48] indicates that many visualizations can be shared by having a group of users sit around a single workstation, with one user 'driving' the visualization and other participants observing and commenting (i.e. synchronous and co-located). To support distributed collaboration, users may use application or desktop sharing tools so that the distributed collaborators can view the same image. These systems are typically limited to one controlling user. More sophisticated media for supporting distributed collaboration include Access Grid [1], Skype, WebEx, and Google Docs.

Some example application areas for collaborative visualization include multiplayer online games, multi-user enabling of single user applications, collaborative problem solving tools, and virtual reality environments [5]. Collaborative visualization has also shown promise in the scientific visualization community [3] where scientists collaborate around scientific data in an distributed / asynchronous manner. ManyEyes [49] is a successful example of a tool for collaborative creation of shared information visualizations. Working over the web, ManyEyes allows a user to upload and share a data source with related visualizations. It supports a discussion board for users to comment on views and allows an existing visualization to be used as the starting point for a new one. Other tools that support sharing of visualizations include Swivel [50], Sense.us [51], and DecisionSite [52].

In the next section we explore how ideas from collaborative visualization can be applied to software maintenance.

4. Collaborative visualization for software maintenance

In this section, we speculate on how collaborative visualization can play a larger role in software maintenance. While many software visualization tools are suited for individuals, and some offer support to export or share views, very few tools explicitly support the collaborative authoring of visualizations for the purpose of understanding in software maintenance. We first review how visualization is used in a collaborative manner in software maintenance today. We then suggest how its use may be broadened and mention challenges that researchers are likely to face.

4.1 Research emphasis to date

The task of understanding a large software system is often distributed among diverse stakeholders that may or may not be collaborating at the same time or place. To exchange and share information, developers may use diagrams or other artifacts to collaborate, e.g. whiteboard sketches, remote application sharing, or projecting a UML design during a meeting. However, in contrast to software engineering tools that support distributed asynchronous maintenance (e.g. version control, bug tracking systems, notification tools, and software planning applications), most software visualization tools are designed for the individual maintainer with little explicit support for collaborative creation or use.

In Table 1, we categorize the tools mentioned in Section 2 with respect to their ability to collaboratively create and share visualizations. The tools are placed within a matrix that cross-references their level of

Table 1: Categorization of Software Maintenance Tools

	Software Systems	Processes	Software System Evolution	Process Evolution
Individual creation and use	Jinsight, TPTP, Rigi, PBS, SCED, Box Plot Metrics, CodeCrawler, TraceCrawler, …	Lagrein, Tarantula, ...	SeeSoft, Beagle, SoftChange, …	Lagrein, SoftChange, …
Individually creation and shared use	PBS, Creole, Doxygen, …	Palantir Jazz …	Xia …	Xia, CodeSwarms, …
Shared creation and use	Reef, Pounamu	None identified	None identified	None identified

collaboration (i.e. individual creation and use, individual creation with shared use, and shared creation and use) to what is being visualized (systems, processes, system evolution, or process evolution). Note that this table is not meant to provide an exhaustive survey of tools, but rather to provoke discussion on how current tools rarely emphasize collaborative authoring and use. We also noted from our literature review that the collaborative aspect is rarely mentioned in studies that evaluated software visualization tools.

In Table 1, levels of collaboration are shown as rows. Since each level of collaboration subsumes lower levels of collaboration (e.g. a tool that supports the shared use of visualizations also supports individual use), a tool appears in at most one row. While one could argue that any visualization can be shared by emailing a screen capture, only when a tool explicitly supports sharing of visualizations do we include it in rows 2 or 3. Areas of understanding are shown in four columns and a tool may appear in multiple columns (e.g. if a tool supports both process and system understanding).

Not surprisingly, the largest representation in this table is with tools that support individual system understanding. These include Rigi, TPTP, and SEAT. A number of tools also support individual visualization of processes and evolution (e.g. Lagrein, which supports both process and process evolution understanding, and SoftChange, which focuses on evolution understanding).

The second row in Table 1 shows only tools that have some form of built-in support for sharing visualizations, e.g. by exporting them to files, through a web interface that accesses a shared repository, or through email. Creole provides explicit support for sharing software views via email, through a filmstrip feature [53]. With Creole, the user can email snapshots to collaborators; the snapshot can be either viewed as a static image or reloaded in Creole as an interactive visualization. The Portable Bookshelf was designed to support dissemination of higher level design knowledge in the form of shared graphs accessible through a web interface. Doxygen produces sharable HTML-based visualizations of source code (e.g. call graphs). Code Swarms' animations are accessible via the web to illustrate the evolution of developer interactions within a software application. Palantir visualizes shared use of configuration management workspaces, ensuring that all users are simultaneously aware of potential conflicts. Xia supports the sharing of treemaps and other graphs to reveal system and process evolution information.

Our search for tools that support both collaborative authoring and use of visualizations turned up few examples. Pounamu supports the specification of collaborative visual language-oriented tools. The authors describe a collaborative UML design tool supporting both asynchronous and synchronous editing of designs. The Reef tool provides support for the automatic generation of documentation that can be distributed to maintainers for verification and update. However, this tool is currently a research prototype and has not yet been fully implemented or evaluated.

This categorization, although limited to a small selection of tools, highlights a lack of research attention to collaborative aspects of software maintenance. While this may be considered a shortcoming, it can also be viewed as a research opportunity filled with a number of exciting challenges.

4.2 Research challenges

There are several challenges that must be considered for collaborative visualization to offer enhanced

support within software maintenance. These challenges are organized by the areas introduced in Section 2.

4.2.1 Complexity in system understanding and tool use

Understanding a system is a complex cognitive activity, where the cognitive processes may be distributed across members of a team [45]. System understanding may also be distributed between internal structures (in the minds of the programmers) and external structures (captured by the tools). Sharing this distributed information requires sophisticated tool support that recognizes that some understanding is better held in the minds of the programmers.

Complex system interactions may lead to complex visualizations. These views can be difficult to interpret, especially for a developer who is not familiar with this type of visualization. Improved support for explaining how visualizations are created may increase their power to communicate complex facts across teams.

There is also a challenge for developers learning complex visualization environments. A visualization may only be accessible within the tool that was used to create it. Such tools require expertise, and if understanding a visualization requires more effort than understanding the system under study, it is unlikely that view will be adopted or shared.

Thus, collaborative environments need to be designed so that all members of the team can benefit from the visualizations created, otherwise, some members of a team may not embrace their use.

4.2.2 Visualization and the workflow of software maintainers

An important aspect of developing effective visualization tools is to understand how the tools will be embedded within the work practices and the workflow of the maintainers. Although there are several studies that document software maintenance work practices and workflow (Section 3.2), few have looked explicitly at the role that visualizations can play.

The CSCW research literature already gives some guidance with respect to tool support for awareness and coordination. Visualizations for showing awareness and assisting coordination during development have been proposed (e.g. Augur [54] and Tukan [55]), but there are other shared tasks that could benefit from collaborative techniques, e.g. exploring and discussing sequence diagrams and call graphs.

Another aspect to explore is how visualizations can be used to support communication across diverse stakeholders. For example, a software visualization could be used as a boundary object, where one stakeholder uses a visualization for one purpose, while another uses it for a different purpose (e.g. a developer may use a visualization for impact analysis, while a manager may use the same visualization for resource allocation) [56]. Lightweight tools, such as those proposed in Section 4.2.1, could further support how information visualizations are shared by a heterogeneous audience.

4.2.3 Visualization and the evolutionary nature of software development

Software is expected to evolve if it is to continue to satisfy the needs of its users [57]. While some views are ephemeral and meant to be thrown away, many visualizations are expected to remain up-to-date as the system evolves. Others are intended to be saved as a record, documenting the system at a particular time. These archival views, like many other artifacts created during software development, can be used to explain how a system has evolved. To achieve this, they should be stored with the source code and other documentation using a version controlled repository.

Visualizations often lack metadata such as their creation date or whether the visualization is ephemeral, should be archived, or is expected to be updated as the system evolves. This information becomes crucial when visualizations are to be shared or saved for future use.

Archiving visualizations is not trivial. Visualizations are often stored in a proprietary format. Such visualization require access to the software used to create them. Over time, this software may become unavailable, perhaps due to license restrictions or changes to the surrounding development environment.

Alternatively, an archived view might use a programmatic description of how the visualization was computed, and might require re-analysis of the software system to render it (e.g. necessitating access to the source code as it was at the moment the visualization was created). This might be difficult or impossible to achieve unless anticipated by the tools and infrastructure within the maintenance environment. We propose that visualizations should not be trapped in their formats, nor should they be dependent on the tools that were used to create them.

In addition to archived visualizations, some visualizations are expected to stay synchronized with an evolving system. Such visualizations are living documents, and should require little attention from the developer, yet provide an always up-to-date view of the system. This is obviously a significant challenge to designers of visualization tools, and some manual manipulation may be necessary.

4.3 Research Opportunities

The realms of social computing, Web 2.0, and today's familiar and yet sophisticated software packages, suggest a number of opportunities that may be worth exploring to improve collaborative visualization in software maintenance. We look at some of these opportunities, once again organizing these by the areas introduced in Section 2.

4.3.1 Collaborative system understanding: lightweight and malleable visualizations

Software visualization tools and techniques can be complex to learn and to use. Lightweight visualizations can improve adoption. As their name implies, lightweight visualizations are expected to be simple and require few resources to be created and manipulated. Malleable visualizations are those that can be easily altered and adapted by the user for her specific needs.

Today's use of the internet has resulted in a paradigm shift, referred to as Web 2.0, in how we use computing, especially over the web and on mobile devices. Speaking at IBM CASCON 2007, Carol Jones [58] suggested that Web 2.0 is the intersection of three things: simple and efficient user interfaces (e.g. REST and AJAX), the delivery of software as a service rather than a product (e.g. RSS feeds, mash-up capability), and the support of community (e.g. through blogs and wikis). She went on to describe the characteristics of successful Web 2.0 applications. Individuals must benefit from a tool, regardless of broader participation, but they must also see improved value when others participate. A simple user experience is more important than advanced or complex features. The use of self-organizing methods (e.g. tagging) can build community knowledge. Users must be able to remix available services (e.g. through mash-ups). Finally, access to data is critical and applications are typically highly data-driven (e.g. Google Search, Google Maps and Facebook).

Web 2.0 approaches and technologies could be used to create lightweight visualization services. Such services can be easily configured and adapted to the particular needs of a user. The web already plays a key role in the collaborative authoring of text through, for example, wikis and Google Docs. Online visualization tools such as ManyEyes and Swivel provide collaborative visualizations of general data. This lightweight, inherently distributed approach can assist software maintainers with the design and sharing of collaborative visualizations.

Taking this idea a step further, lightweight visualization components and data sources could be combined to create mash-ups to satisfy new requirements and individual user's needs. Mash-ups often lead to functionality never imagined by the original service providers. Moreover, developers could share their mash-ups and collaboratively construct them.

4.3.2 Collaborative system understanding: borrowing from the desktop

The web as a creation and delivery platform may not offer sufficient integration with the tools that maintainers already use, so other avenues for improving tools should also be explored. We suggest that tool designers consider borrowing ideas for collaborative visualization from today's powerful client side software. One idea from Adobe Photoshop software worth exploring is the use of *layers.*

Different layers can be added on top of a visualization, each providing new information, such as annotations or new visual elements. Layers can also be used to hide areas of the visualization that are of no interest and even apply operations on the layers to generate a new visualization (such as using a new rendering algorithm on a previously rendered graph). When used by an individual, each layer, or group of layers, can correspond to a different task. The user could hide or show a layer depending on the task at hand.

In a shared visualization, each member could create personalized layers. Layers could also be used as a communication mechanism between different members of a team. For example, a layer could contain annotations that document how the visualization was created and what information is being communicated. Layers could also facilitate simultaneous editing of visualizations.

Another idea we can borrow from common desktop tools is the use of lightweight viewers for visualizations, such as Acrobat Reader for PDF files. Lightweight viewers should be easy to install and use (perhaps as plug-ins to typical tools such as web browsers or Eclipse). This would make it simple for a casual user to inspect or compare archived visualizations.

Other ideas that might be worth exploring include borrowing design and collaboration techniques from the realms of collaborative scientific visualization software and multi-player games.

4.3.3 Understanding workflow: Empirical Studies in Software Maintenance

Evaluating CSCW tools can be a challenge because lab studies often factor out behaviour and tasks that are fundamental to collaborative group activity. For example, McGrath [40] noted that group functions of

improving group health and providing member-support do not surface in a lab setting. Yet factors such as these are critical if the new tools are to be adopted in a collaborative setting.

Research methods from the social sciences are therefore more suitable for understanding tool requirements and evaluating tools than approaches from cognitive psychology. One methodology for studying group behaviour is grounded theory [59], which does not presume that a theory is true or false. Instead, it matches the data that has been collected so far and the theory may have to be adapted to match new findings. Some computer scientists struggle with the use of these methodologies (due perhaps to the less prescriptive nature of the findings). However, such approaches can provide a rich understanding of how a tool may support team work practices, for example by documenting the work flows that provide context for a tool's use.

Studies that focus on tool-related requirements in software maintenance are becoming more prevalent. Ko *et al*. [2] discuss requirements for software maintenance environments. Further studies are needed to explore how diagrams, and even text documentation, are used in teams to enhance program understanding. The findings from these studies, using methods from the social sciences, can then be used to improve the design of collaborative software visualization tools.

4.3.4 Supporting view evolution: versioning and differencing visualizations

Like software artifacts, when visualizations become historical artifacts, they need to be archived and versioned. The current practice is to store visualizations as binary files in a version control system, with little or no metadata to describe their creation or contents. This makes it difficult to automate the comparison or evaluation of two visualizations. Comparing the differences between non-textual software artifacts is another opportunity for research that has been mostly overlooked. The lightweight viewers mentioned in Section 4.3.1 could facilitate manual inspection of differences between saved views, if no differencing tool is available.

When visualizations are archived, it is also important to be able to query them. Metadata is one potential solution, but such metadata usually provides only a high-level description. It might be worth studying methods for querying archived or current visualizations to discover their meaning and content (e.g. "find all visualizations that show the interactions of this class in the system"). Such a query would require that the visualization be stored in an open format (such as XML) that includes a description of the visualization's semantics.

Another research opportunity is to develop a mechanism that computes how an archived visualization can be applied to a later version of a system. Related to this would be a facility to reapply an archived view to a new version of the system such that the results would still be informative.

5. Concluding Remarks

In this paper, we proposed that designers of visualization tools for software maintenance could benefit from looking at the social and collaborative aspects of their field. It may be true that a visualization can portray a thousand words, but if it does not effectively support communication across teams, it may fall short of its goal. Visualization tools that are cumbersome to use, or that fail to support the way that software maintainers work, may be rejected or lead to flawed conclusions about their potential benefits. Improving how visualization tools work in a collaborative setting has the added benefit of helping individual maintainers understand a system authored by others.

Current visualization tools for software maintenance are rarely designed to provide explicit support for collaboration. We suggest that researchers adopt methodologies from CSCW and the social sciences during requirements gathering, and for the evaluation of collaborative visualizations. Researchers can also learn from the rise of social computing in general, as exemplified by recent trends in Web 2.0 technologies and tools.

Most of today's software maintenance environments are designed from the perspective of a *space,* where information on the code and development processes are stored. We suggest that these tools could be improved if they were designed as a *place,* that supports, encourages, and enhances collaboration [38]. The effectiveness and adoption of visualization tools for software maintenance could be dramatically improved if researchers and tool designers make this shift in perspective.

6. References

[1] J. Sillito and E. Wynn, "The social context of software maintenance," in *Proceedings of IEEE International Conference on Software Maintenance (ICSM07)*, Oct. 2007, pp. 325–334.

[2] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of the 29th international conference on Software Engineering (ICSE'07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 344–353.

[3] J. Whitehead, "Collaboration in software engineering: A roadmap," in *Future of Software Engineering (FOSE'07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 214–225.

[4] R. Fjeldstad and W. Hamlen, "Application Program Maintenance Study: Report to Our Respondents," in *Tutorial on Software Maintenance, IEEE Computer Society Press*, 1982, pp. 13–30.

[5] M. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen, "SHriMP views: an interactive environment for information visualization and navigation," in *Proceedings of Conference on Human Factors in Computing Systems*. ACM Press New York, NY, USA, 2002, pp. 520–521.

[6] R. Lintern, J. Michaud, M. Storey, and X. Wu, "Plugging-in visualization: experiences integrating a visualization tool with Eclipse," in *Proceedings of the 2003 ACM symposium on Software visualization*. ACM Press New York, NY, USA, 2003, p. 47.

[7] A. Hamou-Lhadj, T. C. Lethbridge, and L. Fu, "SEAT: A usable trace analysis tool," in *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*. St. Louis, USA: IEEE Computer Society, 2005, pp. 157–160.

[8] T. Systä, "Understanding the Behavior of Java Programs," in *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE)*. Brisbane, Australia: IEEE Computer Society, 2000, pp. 214–223.

[9] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, "Visualizing the Execution of Java Programs," in *Revised Lectures on Software Visualization, International Seminar*. London, UK: Springer-Verlag, 2001, pp. 151–162.

[10] O. Greevy, M. Lanza, and C. Wysseier, "Visualizing live software systems in 3D," in *Proceedings of the 2006 ACM symposium on Software Visualization*. ACM Press New York, NY, USA, 2006, pp. 47–56.

[11] T. Jacobs and B. Musial, "Interactive visual debugging with UML," in *Proceedings of the 2003 ACM symposium on Software Visualization (SoftViz'03)*. New York, NY, USA: ACM, 2003, pp. 115–122.

[12] The Eclipse Foundation, "Eclipse Test & Performance Tools Platform Project," http://www.eclipse.org/tptp/, [April 2008].

[13] H. A. Müller and K. Klashinsky, "Rigi-a system for programming-in-the-large," in *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 80–86.

[14] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Muller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong, "The Portable Bookshelf," *IBM Systems Journal*, vol. 36, no. 4, pp. 564–593, 1997.

[15] P. Kaminski, "Reef," http://www.ideanest.com/reef/index.html, [June 2008].

[16] Dimitri van Heesch, "Doxygen - source code documentation generator tool," http://www.doxygen.org, [June 2008].

[17] J. Bieman, A. Andrews, and H. Yang, "Understanding change-proneness in OO software through visualization," in *Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, 2003, pp. 44–53.

[18] T. Systa, P. Yu, and H. Muller, "Analyzing Java software by combining metrics and program visualization," in *Proceedings of 4th European Conference on Software Maintenance and Reengineering (CSMR 2000)*, 2000, pp. 199–208.

[19] T. Trese and S. Tilley, "Documenting software systems with views V: Towards visual documentation of design patterns as an aid to program understanding," in *Proceedings of the 25th annual ACM international conference on Design of communication*. ACM Press New York, NY, USA, 2007, pp. 103–112.

[20] "IBM Rational Rose XDE," http://www-306.ibm.com/software/awdtools/developer/rosexde/, [June 2008].

[21] N. Zhu, J. Grundy, and J. Hosking, "Pounamu: A Meta-Yool for Multi-View Visual Language Environment Construction," in *Proceedings of IEEE symposium on Visual Languages and Human Centric Computing*, 2004, pp. 254–256.

[22] T. Panas, R. Berrigan, and J. Grundy, "A 3D metaphor for software production visualization," in *Proceedings of Seventh International Conference on Information Visualization (IV 2003)*, July 2003, pp. 314–319.

[23] A. Jermakovics, M. Scotto, A. Sillitti, and G. Succi, "Lagrein: Visualizing User Requirements and Development Effort," in *Proceedings of 15th International Conference on Program Comprehension (ICPC'07)*, 2007.

[24] X. Wu, A. Murray, M. Storey, and R. Lintern, "A reverse engineering approach to support software maintenance: Version control knowledge extraction," in *Proceedings of 11th Working Conference on Reverse Engineering (WCRE'04)*, 2004, pp. 90–99.

[25] A. Sarma and A. van der Hoek, "Visualizing parallel workspace activities," *IASTED International Conference on Software Engineering and Applications (SEA)*, pp. 435–440, 2003.

[26] R. Frost, "Jazz and the Eclipse Way of Collaboration," *Software, IEEE*, vol. 24, no. 6, pp. 114–117, 2007.

[27] S. Eick, J. Steffen, and E. Sumner Jr, "Seesoft-a tool for visualizing line oriented software statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, 1992.

[28] J. Jones, M. Harrold, and J. Stasko, "Visualization for Fault Localization," in *Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto, Ontario, Canada*, 2001, pp. 71–75.

[29] M. Godfrey and Q. Tu, "Growth, evolution, and structural change in open source software," in *Proceedings of the 4th International Workshop on Principles of Software Evolution*. ACM New York, NY, USA, 2001, pp. 103–106.
[30] M. Lanza and S. Ducasse, *Tools for Software Maintenance and Reengineering, RCOST/Software Technology Series*, ch. Codecrawler-an extensible and language independent 2D and 3D software visualization tool, pp. 74–94.
[31] D. German and A. Hindle, "Visualizing the Evolution of Software Using Softchange," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 1, pp. 5–21, 2006.
[32] M. Ogawa, K. Ma, C. Bird, P. Devanbu, and A. Gourley, "Visualizing social interaction in open source software projects," in *Proceedings of 6th International Asia-Pacific Symposium on Visualization (APVIS'07)*, 2007, pp. 25–32.
[33] M. Storey, D. Cubranic, and D. German, "Ón the use of visualization to support awareness of human activities in software development: a survey and a framework," in *Proceedings of the 2005 ACM symposium on Software visualization*. ACM Press New York, NY, USA, 2005, pp. 193–202.
[34] S. Bassil and R. Keller, "Software visualization tools: Survey and analysis," in *Proceedings of International Workshop on Program Comprehension (IWPC'01)*, 2001, pp. 7–17.
[35] T. Hendrix, S. Maghsoodloo, and M. McKinney, "Do visualizations improve program comprehensibility? experiments with control structure diagrams for Java," *ACM SIGCSE Bulletin*, vol. 32, no. 1, pp. 382–386, 2000.
[36] H. Ishii, "Tangible bits: designing the seamless interface between people, bits, and atoms," in *Proceedings of the 8th international conference on Intelligent user interfaces (IUI'03)*. New York, NY, USA: ACM, 2003, pp. 3–3.
[37] Wikipedia, http://en.wikipedia.org/wiki/CSCW, [June 2008].
[38] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *Proceedings of the 1992 ACM conference on Computer-supported cooperative work (CSCW'92)*. New York, NY, USA: ACM, 1992, pp. 107–114.
[39] L. C. Rosenberg, "Update on national science foundation funding of the "collaboratory"," *Commun. ACM*, vol. 34, no. 12, p. 83, 1991.
[40] J. McGrath, "Time, Interaction, and Performance (TIP): A Theory of Groups," *Small Group Research*, vol. 22, no. 2, p. 147, 1991.
[41] C. A. Ellis, S. J. Gibbs, and G. Rein, "Groupware: some issues and experiences," *Communications of the ACM*, vol. 34, no. 1, pp. 39–58, 1991.
[42] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Classics in software engineering*, pp. 139–150, 1979.
[43] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, pp. 543–554, 1983.
[44] G. Olson and J. Olson, "Distance Matters," *Human-Computer Interaction*, vol. 15, no. 2/3, pp. 139–178, 2000.
[45] Y. Ye, Y. Yamamoto, and K. Nakakoji, "A socio-technical framework for supporting programmers," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE'07)*. New York, NY, USA: ACM, 2007, pp. 351–360.
[46] G. Johnson, "Collaborative visualization 101," *ACM SIGGRAPH - Computer Graphics*, vol. 32, no. 2, pp. 8–11, may 1998.
[47] S. A. Bly, "A use of drawing surfaces in different collaborative settings," in *Proceedings of the 1988 ACM conference on Computer-supported cooperative work (CSCW'88)*. New York, NY, USA: ACM, 1988, pp. 250–256.
[48] K. Brodlie, D. Duce, J. Gallop, J. Walton, and J. Wood, "Distributed and Collaborative Visualization," *Computer Graphics Forum*, vol. 23, no. 2, pp. 223–251, 2004.
[49] M. Wattenberg, J. Kriss, and M. McKeon, "ManyEyes: A Site for Visualization at Internet Scale," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1121–1128, 2007.
[50] "Swivel website," http://www.swivel.com/, [June 2008].
[51] J. Heer, F. B. Viégas, and M. Wattenberg, "Voyagers and voyeurs: Supporting asynchronous collaborative information visualization," in *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'07)*. New York, NY, USA: ACM, 2007, pp. 1029–1038.
[52] "DecisionSite website," http://spotfire.tibco.com/products/decisionsite.cfm, [June 2008].
[53] "Creole website," http://www.thechiselgroup.org/creole, [June 2008].
[54] J. Tullio and E. Mynatt, "Use and Implications of a Shared, Forecasting Calendar," *Lecture Notes in Computer Science*, vol. 4662, p. 269, 2007.
[55] T. Schümmer and J. M. Haake, "Supporting distributed software development by modes of collaboration," in *Proceedings of the seventh conference on European Conference on Computer Supported Cooperative Work (ECSCW'01)*. Norwell, MA, USA: Kluwer Academic Publishers, 2001, pp. 79–98.
[56] G. C. Bowker and S. L. Star, *Sorting Things Out: Classification and Its Consequences (Inside Technology)*. The MIT Press, October 1999.
[57] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sept. 1980.
[58] "Cascon 2007 Speakers," https://www-927.ibm.com/ibm/cas/cascon/speakers/index.shtml, [June 2008].
[59] B. Glasser and A. Strauss, "The discovery of grounded theory," *The discovery of grounded theory*, vol. 16, no. 1, 1967.