ISR Institute for Software Research
University of California, Irvine

A Survey of Visualization Tools that Promote Awareness of Software Development Activities

Erik Trainer
University of California, Irvine
etrainer@uci.edu

David F. Redmiles
University of California, Irvine
redmiles@ics.uci.edu

December 2009

ISR Technical Report # UCI-ISR-09-5

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

A Survey of Visualization Tools that Promote Awareness of Software Development Activities

Erik H. Trainer and David F. Redmiles
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
{etrainer, redmiles}@ics.uci.edu

ISR Technical Report #UCI-ISR-09-05

December 2009

Abstract:

In recent years, research attention in the software engineering community has shifted from process management and workflow tools that aim to plan for all coordination activity and eventualities before development begins to a new generation of more flexible tools that saturate the developer's workspace with information at varying degrees of granularity and in different visual, and often interactive, representations. The common thread that runs through these tools is the objective of supporting awareness of software developers' activities, in order to put one's own activities in context. Despite the glut of such tools, little work has been done to assess to what extent they address well-understood coordination needs.

This survey symbolizes a critical first step in that process. Its primary goal is to study the relationship between coordination and awareness as empirically explored in the software engineering literature, identify important aspects of awareness from that same body of literature, and, with respect to these aspects, compare tools representative of those used in academia and industry that are built to support awareness of development activities. An analysis of the tools was performed and a table was constructed that maps the tools to important dimensions of awareness.

This table is the central contribution of this survey. It is a mechanism through which researchers can perform careful comparisons of each tool as well as develop a more critical understanding of how each addresses components of awareness as identified in the literature. *Therefore, this survey crucially links two bodies of literature in software engineering: empirical and theoretical findings on how developers maintain awareness and tools that visualize human activities over the course of software development projects.*

A Survey of Visualization Tools that Promote Awareness of Software Development Activities

Erik H. Trainer
University of California, Irvine
etrainer@ics.uci.edu

Introduction

Professional software development is, and has always been, a human-driven activity. Even small-scale software projects require the coordination of multiple individuals, typically grouped into teams, working in parallel on different components of the system for up to many weeks, months, or years (Brooks 1995). Software design and development activities typically produce a "project memory," (Cubranic and Murphy 2003) of archival artifacts such as source-code, e-mail lists, design documentation, problem reports and change histories of all this information that developers use as a means toward coordinating the smooth flow of work. Successful coordination often requires not only an understanding of the tasks to be performed but also of the internal components of the system under development, their interactions, and dynamic behavior over the course of the project (Parnas 1972; Curtis et al. 1988; de Souza et al. 2004; Cataldo et al. 2006; de Souza et al. 2007).

Yet the inherent complexities of software make design and development a difficult undertaking for the individuals (and teams) involved for a number of reasons (Brooks 1995). Although it has an architecture (Perry and Wolf 1992; Shaw and Garlan 1996), software is not a physical entity like a building or monument, but rather an abstract encapsulation of functionality that serves to fit a set of stakeholders' requirements. It has no visible representation in the real world. As such, a natural management task like identifying "progress" is difficult to perform. Software is not built once-and-for-all, but emerges incrementally from changing requirements and is constantly subjected to rigorous verification and validation. It is not possible for the casual observer to gauge the "size" of a software system in situ. Rather, the metric for the size of a system requires a source-code perspective and is often measured in lines-of-code (LOC). Even then, a line-of-code measurement fails to take into account other important measures of complexity such as the coupling between components or the number of possible paths through a program's source-code (McCabe 1976). No single measurement can, in of itself, describe the complexity of a software system.

Dealing with this complexity requires careful management of human resources and their allocation to development tasks over the course of the project. It is extremely difficult, if not impossible, for an individual or groups of individuals to comprehend systems in their entirety (Soloway and Ehrlich 1984). As such, assigning enough developers to cover all of the implementation work is an important management decision.

While having sufficient human resources is a fundamental requirement for completing development work, it does not follow that adding more personnel will get the work done faster. In Brooks' seminal work (Brooks 1995), he empirically observed that merely adding more people to a project will not speed up development time, but *delay* it instead. The extra time required derives from the additional effort (and thus time) required to understand the software and coordinate activities with other developers. In general, software development organizations take more careful and structured approaches toward assigning work. The research literature has shown that organizations carry out complex tasks by dividing them up into smaller, interdependent work items and assigning them to teams. Coordination needs arise from the interdependencies between these tasks (Malone and Crowston 1994) and change as progress is made on the system under construction (Cataldo et al. 2006).

In response to these technical and organizational issues, software development researchers and practitioners have developed techniques and tools for coordinating work. The most common way to cope with this complexity is to adopt and follow a *software process* (Fuggetta 2000). Software processes frame and organize the technical and social aspects of developing software. More specifically, a software process can be described as a "coherent set of policies, organizational structures, technologies, procedures, and artifacts that are needed to conceive, develop, deploy, and maintain a software product" (Fuggetta 2000, pp. 28). Fuggetta identifies four aspects of software development processes:

1. Software development technology—tool support such as infrastructures, environments, and visualizations
2. Software development methods and techniques—guidelines and rules-of-thumb on how to use technology to complete tasks
3. Organizational behavior—the science of organizing people, coordinating and managing work activities
4. Marketing and economy—being able to address customer needs in specific market situations, understanding the context in which software is developed and sold

All four aspects have been subject to much attention in the context of software development by both researchers and practitioners. Studies of software development projects have revealed organizational behavior and structure is largely influenced by interdependencies between software modules and run-time components (Parnas 1972; Curtis et al. 1998; Morelli and Eppinger 1995; Sosa and Eppinger 2002; de Souza et al. 2004; de Souza et al. 2007; Valetto et al. 2007; Avritzer et al. 2008). Much of the work related to coordinating the implementation of those components requires an understanding of the ongoing activities of others and how those activities might impact shared tasks. Yet the increase in global software development (Carmel 1999; Sangwan et al. 2006) poses many difficulties for developers and managers who wish to manage these aspects of the software development process. As a result, particular methods and techniques such as architecting systems to require less communication overhead, outlining principles of awareness, and identifying awareness networks have been developed to cope with these difficulties. In turn, researchers have engineered a host of

software development tools to minimize communication requirements and enhance developers' understanding of the activities being performed over the lifecycle of the project.

Due to limitations of scope and space, this survey will address the first aspect of software processes identified above: software development technology. The primary objective of this survey is to study the relationship between coordination and awareness as empirically explored in the software engineering literature, identify important aspects of awareness from that same body of literature, and, with respect to these aspects, compare tools representative of those used in academia and industry that are built to support awareness. Because surveying all existing tools and technologies is impractical, this survey addresses a subset of these tools. The systems were primarily selected on the frequency of their appearance in the literature, both as *standalone publications* and as *references* that lend support to and motivate work done by other researchers in the field.

This survey was performed according to the following process: first the existing scientific literature on empirical studies of software development coordination and awareness activities was searched. Literature was selected from premier peer-reviewed research conferences and journals in the areas of: software tools and environments, empirical software engineering, computer-supported cooperative work, awareness, and visualization (e.g., at the ICSE, CSCW, ECSCW, SOFTVIS, AVI, and VL/HCC venues). Careful attention was devoted to identifying common themes in the information developers need and the strategies they use to stay aware of each others' activities. In general, these papers describe, analyze, or theorize about problems, but do not mention a specific tool. As such, forward citation searches of this research were conducted to identify software tools with the purpose of promoting awareness for a variety of software development activities. Backward citation searches of the tool literature were performed to identify other principles of awareness and empirical work not revealed by original searches, yet relevant to the topic, from which these tools were created. Finally, these publications were combined with the first set of literature describing awareness needs to identify crucial principles of awareness, providing a context in which to evaluate the tools.

When tools could not be found using backward and forward citation searches, searches were made via the web using the keywords "software development," "awareness," and "visualization." These searches yielded tools that addressed broad usage scenarios instead of specific tasks, such as "showing evolution," for example. In general, the tools identified addressed one or more tasks, such as making a change to a portion of source-code or identifying who to contact about some technical issue. Despite the fact that these tools were directly or indirectly motivated by awareness needs, the publications rarely discussed the extent to which they addressed awareness as it relates to the tasks they were built to support.

Knowing how well these tools encompass the aspects of awareness they are designed to support is an important step in gauging their usefulness, understanding their positive and negative qualities for the purpose of improving them, and facilitating their adoption by users. As such, analysis of the tools was performed and a table was constructed that maps the tools to crucial aspects of awareness. The table allows the visualization and comparison of the characteristics of each tool and explains how each addresses components of awareness. *Therefore, this survey crucially links two bodies of*

literature in software engineering: empirical and theoretical findings on how developers maintain awareness and tools that visualize human activities over the course of software development projects.

In the next sections, the role of awareness and different aspects of awareness relevant to software development activities are discussed, framing the content of this survey. Several fundamental themes in the awareness literature that are relevant, yet not fully explored by visual tools for promoting awareness, are identified. In the remainder of the paper existing gaps in the current tool support available are exposed with respect to these themes and implications for future iterations of these tools as well as novel research avenues are discussed.

The Role of Awareness in Software Development

Coordinating work activities is easiest done in environments where individuals are co-located (Olson and Olson 2000). Members of projects who are co-located have opportunities for more informal interactions, including overhearing (Heath and Luff 1992) and unplanned discussions over meals or in hallways (Whittaker et al. 1994). In an empirical study on time management by software developers, Perry and colleagues (Perry et al. 1994) were struck by the fact that developers spent an average of 75 minutes per day in unplanned personal interactions. In general, co-located individuals have a common view of the way work should unfold, either because of an in-place software development process (Fuggetta 2000; Nutt 1996), or a shared vocabulary or perspective about the work to be done and how that work is assigned among individuals and project teams.

While co-location may be the ideal configuration of software developers from an organizational perspective, the reality is that software development at a distance is increasingly becoming commonplace. This is due to a number of reasons beyond the scope of this survey, including the globalization of markets and production, cost concerns, and the desire for mixed expertise and skills.

People at a distance typically communicate infrequently and less effectively (Herbsleb and Grinter 1999). Thus it is often difficult to know what colleagues are doing day to day and thus whether they are available to work on dependent tasks. The research literature has referred to this sense of other people's availability and their activities as "awareness." More precisely, "awareness" has been described as "an understanding of the activities of others, which provides a context for (one's) own activity" (Dourish and Bellotti 1992). Yet awareness is not only about understanding others' actions, but knowing how one's own action may impact others as well (de Souza and Redmiles 2007).

Awareness might answer questions such as "Who is doing what," "Who's who," and "Who do I talk to about issue x?" In a study of open-source software developers, Gutwin and colleagues identify two types of awareness: general awareness (e.g. who is doing what) and specific awareness (e.g. who do I talk to?) (Gutwin et al. 2004). Similarly, in an empirical study of distributed software development teams, de Souza found that despite the fact developers had a general idea of whose code affected whom, they still had questions of the form "Who do I talk to about problems with this component?" "Is my code being called?" and "Who is implementing this interface?" (de

Souza et al. 2007).

Thus awareness involves a perspective, or information space of the people involved, the activities they must perform, and the resources with which they work. However, awareness information must also be managed over time. Although technical strategies such as decoupling the system and designing stable APIs can limit the coordination required upfront, even they fail to hold up against the many changes that occur over the development cycle (de Souza et al. 2004).

Aspects of Awareness

Content—Artifacts

Promoting "awareness" necessarily implies the action of delivering contextually relevant information to interested parties. Before that information can be passed along, however, the content of that information must first be identified. In general, typical data sources of awareness information include change management/version control systems, defect and issue (i.e. bug) trackers, program source-code, documentation, and informal communication channels such as IM and chat.

Coordination requirements, and thus the content of awareness information, change over time due to the dynamic and iterative nature of dependencies between software development tasks (Cataldo et al. 2006). There are many reasons for these changes. For example, it is not uncommon for clients to request more features than originally communicated. As a consequence, parts of the design and thus the implementation might well change. The new requirements create additional implementation tasks to which management must allocate developer resources. As developers implement features and check them in to a versioning repository, bugs invariably emerge. As a result, developers create and file bug reports, creating work to which bug fixers must attend. Newly written code is subject to the same rigorous testing and verification as the original code it replaces. In short, changes imply other activities that must be performed downstream.

Artifacts of the development process such as code are subject to constant change and different versions must be maintained by developers. Awareness of changes may need to be communicated daily as new code is checked into a repository. Dependencies in source-code emerge over time by their nature; one component may have no dependencies for a day or weeks and then suddenly, one day, a portion of source-code may reference it. If a developer fails to know their code is being referenced, they may change it and break the very code that depends on it. Failing to address the requirements of consuming code can cause problems for the developers involved, especially if a project deadline looms near (de Souza et al. 2007). Reverse engineering and maintenance tasks may require information over a longer period of time such as monthly releases of the software. Thus, awareness information should reveal information about activities that occurred in the recent past, activities that are currently happening, or activities that occurred over the history of the whole project.

The above scenarios are by no means exhaustive, but serve simply to illustrate the dynamic nature of coordination requirements. Software development is an incremental,

iterative process so this is not unexpected. Artifacts of the software development process are not the only resources in flux. As explained in the following section, the individuals with whom one must coordinate change as well.

Content—People

In addition to development artifacts, people use each other as resources to maintain an awareness of what is going on. In software development, awareness and information-seeking go hand in hand. Coworkers are the most frequent sources of information about design artifacts and developer expertise (Ko et al. 2007). Developers prefer to pick the brains of others and find what they need by utilizing their personal networks, often going outside of their assigned teams to do so (LaToza et al. 2006; Ehrlich and Chang 2006). Documentation is often not kept up to date and design knowledge is often distributed across teams by the very nature of modular decomposition (Parnas 1972) and the assignment of teams to different aspects and components of the system. In an empirical study of software developers, Ko and colleagues (Ko et al. 2007) identified the types of knowledge sought over several different work tasks: writing code, submitting changes, triaging bugs, reproducing failures, understanding execution behavior, and reasoning about design. They found that co-worker awareness, i.e. what people were doing, was among the top information needs over these categories. Individuals often deferred searches about implementation choices, program behavior, and impacts of changes to code until they could find the right co-worker to talk to about a particular problem.

Numerous researchers have tried to characterize the dynamic nature of people's relationships over time. de Souza proposes a social network (Wasserman and Faust 1994)-oriented view with the term "awareness network," which refers to the set of people who need to be aware of one's actions as well as the set of people whose actions one needs to monitor (de Souza and Redmiles 2007). The network expands and constricts over time as tasks evolve and require individuals to coordinate. Similarly, Nardi and colleagues refer to "intentional networks," collections of contacts that an individual constructs, maintains and activates as the work requires (Nardi et al. 2002). Awareness networks may be the prerequisite for constructing intentional networks.

Before one can maintain a network of those of whom they need to be aware, individuals must first identify what set of people should actually be in the network itself. Empirical studies have shown that, in some cases, at least 80% of all coordination and communication activities can be predicted in advance by analyzing dependencies in the system architecture (Morelli and Eppinger 1995; Sosa and Eppinger 2002). In general, however, much of the coordination work individuals do is in response to needs that emerge from changing code, tasks, and availabilities of co-workers. These needs cannot be known before development work begins. Empirical studies (Cataldo et al. 2006) have validated this observation.

In an effort to support the process of being aware of others' activity and finding expertise, software researchers have developed tools that show the relationships between people based on dependent tasks they share as well as the artifacts they use. For example, using a matrix-multiplication method derived from the PCANS model (Carley and Krackhardt 1998), de Souza and colleagues showed how dependencies in source-

code create dependencies between people working on the code. They describe how, at any release or snapshot of a project, a social call-graph, or sociogram, visualizing connections between individuals based on the code they write, can keep developers aware of how changes to that code may affect them or others (de Souza et al. 2007). The graphs can also help developers identify multiple people to talk to about portions of the code based on their usage of it. TESNA, a tool developed by Amrit (Amrit 2008), also uses sociograms to show links between developers based on dependencies in the code but combines the graphs with social network analysis (SNA) metrics (Wasserman and Faust 1994) to assign developers relative measures of prominence and reputation (e.g. developer ownership, involvement in knowledge exchange). Recommender systems (Mockus and Herbsleb 2002; McDonald and Ackerman 2000; Ackerman and McDonald 1996) are a set of tools that address this problem by automatically suggesting people with whom to talk based on the history of their interactions with the source-code. The common feature of these tools is that they show individuals with whom they might coordinate based on the history of their interactions with the code. They identify the "who" dimension of awareness.

When individuals use awareness information to get help or ask a question, that information should give an indicator of that individual's willingness to respond.

Availability/Willingness to Help

Awareness information is often used to identify individuals with whom to talk (de Souza et al. 2007). Yet identifying the person to talk does not necessarily mean they will provide the information the asker needs. Unlike design artifacts such as architecture diagrams, code, and design documents, people only become resources when they consent to do so, often under conditions of their choosing (Illich 1971). In order for an individual to get help, the information-provider must be willing to cooperate. This willingness might depend on the helper's demeanor, their history of interactions with the information-seeker, and any perceived benefits as a result of helping. Asking someone for help usually comes at a cost: the disruption of flow and continuity of ongoing work, which reduces the productivity of the helper (Szóstek and Markopoulos 2006). As such, helping may require precious time for little reward, aside from a possible increase in social reputation. Yet appearing unwilling by responding "no" can degrade the helper's relationship with their peers because declining to help violates social norms, especially if the culture of the workplace promotes sharing. Ultimately, knowing whether someone is reachable is not enough; the seeker must rely on social context for when and where to make contact. To the author's knowledge, this issue commands further attention and exploration in the literature.

It is not enough to know whose actions one needs to watch. Software development is an information-intensive process already and any additional awareness information that becomes available may distract developers. As such, awareness information should facilitate decision-making related to tasks at hand within some larger task and corresponding phase of the software development process.

Mapping Content to Tasks

Awareness content needs to be directly tuned to a developer's current task

because, in the interest of time, a developer is often only seeking information that is required to complete that task (Kersten and Murphy 2006). If information acquisition requires use of a stand-alone system, it increases the cognitive cost of that information, requiring the developer to switch between different workspaces, lose short-term memory, and perform less efficiently and productively as a result. Gutwin and Greenberg also made this observation in their framework for workspace awareness (Gutwin and Greenberg 1999). They articulate a "what" dimension of awareness (e.g. "What are people doing?" "What goal is that action part of?" What object are they working on?" etc.).

Thus for awareness information to be considered as relevant and useful as possible, it should directly relate to the task or subsets of tasks at hand. De-contextualized information, such as Microsoft's "Tip of the Day," is more annoying to users than it is helpful (Fischer 2001). Software developers spend much of their time using popular development environments, or IDEs, such as Eclipse and Jazz for example. These IDEs have reached a critical mass of users engaged in both software development research and practice. As such, many tools for awareness have been designed as plug-ins to these environments as to not distract from the flow of work (Sarma et al. 2003; Hupfer et al. 2004; DeLine et al. 2005). These tools aggregate resources and events already in the environment, but usually time consuming to access, in order to makes sense of the work currently being done. For example, some tools inspect deltas from versioned files to show timeline views of activity that convey change and evolution. Others identify parallel workspace events generated by multiple developers to detect conflicts in order to determine who will be impacted by, and should be notified of, changes. Information that other developers are doing parallel work on the same files is most useful, for example, when the target user is working on those files concurrently, not after they have checked the files in to a CM system (Conradi and Westfechtel 1998). Thus identifying the task a particular piece of awareness should support and delivering that awareness in a timely manner are fundamental requirements for displaying information.

In turn, the task that the awareness information is designed to support should be representative of some real-world task as identified by some phase of the software development process. Examples include fixing bugs, understanding source-code, adding features in the maintenance phase of software development, and implementing a component or submitting a change in the implementation phase of software development. Many of the tools designed for supporting awareness focus on activities in the maintenance phase of software development, such as program comprehension and reverse engineering.

When awareness content is intended to support decision-making involved in completing a task, a measure of the confidence of the accuracy of the information provided is crucial. To effectively use awareness content in order to complete a task, the individual must understand its relevance and trust its accuracy.

The "Interpretive Gap"

In general, there is a gap between providing awareness information and actually being able to express confidence in the validity of the information and its usefulness toward completing a task. A similar gap exists in the field of information visualization,

the "use of computer-supported, interactive, visual representations of data to amplify cognition" (Card et al. 1999). The tension rests in what is being shown in the visualization compared with what *needs* to be shown to make a straightforward decision (Amar and Stasko 2004). Most approaches toward providing awareness rely on the user to unquestioningly unpack the awareness information and put it to use. There is usually no indication of the validity of the information or the data from which it came. As a result, decision-making is often informed by individual user experience and incomplete information rather than a rich set of cues the developer can use to tradeoff the consequences of certain courses of action.

In his discussion of the user activities involved in carrying out a task via a computer, Don Norman (Norman 1986), a distinguished researcher in the field of usability, observes the tension between the user's psychological perception of the system's state and the system's actual state. When a user performs a task, they specify a list of action sequences, or interactions with the software, needed to achieve some result. After the user executes the actions, they perceive a visual result (e.g. a blinking status update). In response, the user goes through with what Norman terms a phase of "interpretation." Interpretation involves the cognitive process of giving meaning to the perceived information and subsequently comparing it to what was expected when the action sequence was initially specified.

Yet perceptions are subjective and can lead to incorrect interpretations. Awareness support can only be as good as the source of the awareness information itself. For example, Ariadne (Trainer et al. 2008), a visual software tool that shows connections between developers based on the shared dependent code they use shows varying levels of activity of different developers in the project based on the number of calls they make. The more thick the connections are, the more calls to the code developers are making One could potentially use this visualization to get a sense of certain people's "expertise," the argument being that the more connections from a developer to the code with which they work, the more knowledge that individual has about the code. However, it could also be the case that that developer is only using a small set of features (e.g. instance variables or helper methods) and thus would not be very helpful answering questions relating to the majority of the implementation. This information is not clearly available via the interface, so relying on the information provided is not adequate for forming a complete and correct interpretation. To more correctly evaluate the possibility of someone being an expert on a code module, the user would need more information, such as a view of the code being called juxtaposed with the total size of the module and other related code.

Borrowing from Norman, the mismatch between the user's interpretation of the awareness content and the meaning implicitly assigned to the content by the process through which it was derived will be referred to hereafter as the "interpretive gap."

Thus awareness content should include various forms of evidence to support its correct interpretation and meaning, including strengths and certainties of relationships, support for user hypothesis testing (Shneiderman 2002), and alternate views of that awareness should be conveyed in awareness delivery mechanisms. For example, a system that displays connections to individuals with particular source-code expertise should provide rationale for and trade-offs between asking different individuals such as their history of responding to questions (indicating their willingness to help), their actual

experience using the code, their preferred method of contact, their location (which might determine their response speed), and so on and so forth.

Temporal Unit of Analysis

As software artifacts are produced as a result of the development process, they are typically automatically archived in versioning repositories. These repositories typically contain all versions of items and associated meta-information such as who created them, when they were created, and textual descriptions summarizing the items. Some tools even log real time user interactions with items in the development workspace, such as editing or checking in source-code. As a result, information about items can potentially be extracted at any point in time over the trajectory of the project to reveal insights into the activities that occurred at that point. For example, the entire history of a project might be needed to complete reverse engineering or maintenance activities while only the recent project history like daily code commits might be needed to assess the impact on others' dependent code. Real time awareness information like ongoing parallel changes to the same artifact by multiple developers would be needed to detect code commit conflicts. Awareness information, then, can be expressed in several temporal granularities: the entire project history, recent project history, and real time project activities.

Visual Representations

Another characteristic of awareness is the visual form through which it is conveyed. In general, most systems to support awareness deliver it in three forms: textual, graphical, or both. Fitzpatrick et al. found that something as simple as a lightweight tickertape mechanism was sufficient for conveying changes made to a software system (Fitzpatrick et al. 2006). In a study of open-source software development, Gutwin and colleagues found that by using mailing lists and chat systems, developers could actively maintain awareness of what their colleagues were doing (Gutwin et al. 2004).

Much of the work on supporting awareness, however, focuses on the use of graphics. Tool designers use graphical representations of people and artifacts like source-code and design diagrams to convey structure and overall patterns in development activity. Software is by its nature (Brooks 1995) complex and invisible. Graphical representations, or visualizations, have been used extensively to give visibility to software and make the process of understanding software easier as a result. It is generally accepted that the advantages of successful visualizations over textual representations include allowing users to process more data in parallel, facilitating identification of significant recurring patterns, and identifying the high-level relationships among data critical to decision-making (Bertin 1982; Ball and Eick 1996; MacKinlay and Shneiderman 2000).

Tool designers use several common visual representations of design artifacts and people involved in software development activities. One widely-used representation is a simple network graph composed of nodes and edges. In software development graphs are often used to represent relationships between different system components, and thus their

structure (Herman et al. 2000). For example, a task like finding a particular file is most usually performed by navigating a hierarchical tree (a type of graph). Developers use call graphs to understand dependencies in the code. Nodes represent source-code artifacts and edges represent calls between source-code.

Alternatively, small visual cues or decorators can be used to augment existing visualizations and displays in the environment in unobtrusive and lightweight ways. These icons are typically too small to convey much information themselves. As such they are often used to convey a single property, such as the status of an item (e.g. buggy, conflict, valid, etc.). Yet their small size means many can be viewed in the same window or dialog box at once, giving a sense of the "overall" status of the information they annotate. Typical examples in software development practice include the red squiggly underlines in the code editor that indicate typographical errors, red boxes next to lines of code that indicate compile problems, or green lights next to a test case indicating it passed.

Researchers have used decorators for the purpose of conveying awareness as well. Sarma et al. use colored arrows and text to annotate source-code in the Eclipse editor, indicating the potential for conflicting code check-ins and measurements of the code's impact on other code in the system (Sarma et al. 2003). In the Team Tracks visualization, DeLine and colleagues use a textual list view combined with decorators to indicate development artifacts related to the current source-code displayed in the editor (DeLine et al. 2005). The lengths of horizontal bars next to the related items indicate the strength of the relation while small arrow icons represent directionality of dependencies. In other visualizations, user avatars or icons are also typically used to represent developers, managers, or other stakeholders involved in the development process (Hupfer et al. 2004).

Such visual representations are typically, but not necessarily, integrated into the development environment along with standard views such as hierarchical source-code trees, the source-code itself, and UML diagrams. On the other hand, visualizations that exist as stand-alone systems increase the cognitive cost of using them because of constant task-switching users must perform. Some awareness visualizations try to visualize whole systems (Eick et al. 1992; Froehlich and Dourish 2004). In these cases, such visualizations may require more screen real-estate than one monitor can provide. Allocating stand-alone visualizations to a second monitor adjacent to the developer's primary monitor represents one promising approach to this problem (Speier et al. 1997; da Silva et al. 2006).

The following table summarizes the different aspects of awareness covered up to this point:

Awareness Aspects	Description
1. Content: • Artifacts • People	--Data sources and associated awareness information (e.g., CM repository and change history, IDE and real time activity)
2. Mapping Content to Tasks	--Relating awareness content to practical tasks and phases of software development
3. Interpretive Gaps	--Uncertainty in interpretations of awareness content needed to make decisions (e.g. recommendations)
4. Temporal Unit of Analysis	--Granularity of time: entire project history, recent project history, real time
5. Visual Representations	--How awareness is conveyed through the interface: graphics, text, or both

Table 1—Aspects of awareness used to support software development activities.

Summary

As the previous sections have illustrated, organizing the interactions involved in software development is a complex problem due to the coordination effort required. Software managers and developers use a software process to manage this complexity. Within it, designers and architects use a variety of organizational and technical strategies to reduce coordination needs. These strategies work more effectively when developers are co-located than when they are distributed in time and space. Yet global software development is increasingly becoming the rule, not the exception. As system knowledge and developer expertise become spread out over different time zones, distances and individuals with different cultural backgrounds, it becomes increasingly difficult to maintain the same awareness to which co-located groups have access.

To address the problem of staying aware, the research community has developed a host of visualization tools to either augment existing views in development environments or provide specialized views of individual activity combined with information about software artifacts. The vast majority of these tools use the project versioning repository as a project memory to uncover what individuals have done and what changes to artifacts have been made. There are literally dozens of examples of such tools. Despite this fact, there is still relatively little known about how the vast majority of these tools differ, with the exceptions of the broad categories of information they display, their visual representations, whether they have been evaluated or not, and the types of interactions they support (Storey et al. 2005). Identifying similarities and differences in these tools can lead to an understanding of their best and worst aspects as well as how they may be combined to better support awareness.

Existing Visual Tools for Supporting Awareness

In this section, forty (40) tools in total are introduced and surveyed. From them, a

subset was chosen and subsequently analyzed using the aspects of awareness reported in the previous section. In an effort to categorize and analyze the tools, classifications describing the broad tasks (i.e. tasks to be completed by developers) the tools support were chosen:

- Understanding change management and evolution
- Recommending people and artifacts
- Avoiding conflicts
- Determining individual availability
- Understanding developer activities

A direct link was found between these classifications and the tasks developers perform as identified in the surveyed research literature on empirical software engineering and awareness (e.g. determining individual availability and recommending people and artifacts). Within categories, tools were selected based on their research merits. This process resulted in twelve (12) tools.

One way to measure the merit of a concept or tool is the extent to which it appears—as a publication itself or as a reference in another publication—in the research literature. The more a piece of work has been published, the more it has been peer reviewed and refined to address various issues raised by scientists in the same research community. As the ideas are refined, the more rigorous they become and the more firmly they hold up to scientific critique. Thus number of publications—everything from workshop papers to book chapters—was used as one metric to quantify research value. The value of theories, frameworks, tools, and empirical results can also be assessed by the number of citations to that work, an indicator of its relevance to and influence on ongoing related, but different, research in the field. As such, citation count was also considered a measure of a tool's worth. **Table 2** below lists the tools and their paper and citation counts, respectively.

Tool Name	Paper Count	Citation Count[1]
ELVIN/Tickertape (Fitzpatrick et al. 2006)	4	69
Command Console (O'Reilly et al. 2005)	2	29
softCHANGE (German et al. 2004)	2	102
Expertise Recommender (McDonald and Ackerman 2000)	3	120
Hipikat[2] (Cubranic and Murphy 2003)	4	106
Team Tracks (DeLine et al. 2005)	2	39
Palantír (Sarma et al. 2003)	5	45
TUKAN (Schummer and Haak 2001)	2	10
Awarenex (Begole et al. 2002)	3	139
Community Bar (Tee et al. 2006)	3	11
FASTDash (Biehl et al. 2007)	2	16

[1] From portal.acm.org--when citation counts were not available they were counted manually using Google scholar search results.

[2] Hipikat became Mylar (Kersten and Murphy 2006) and is now Mylyn (http://eclipse.org/mylyn/start/)

Jazz[3] (Hupfer et al. 2004)	3	37

Table 2—Tools chosen for analysis, the number of published papers, and the number of times the papers have been cited.

The table below lists the tools categorized according to the five aspects of awareness summarized in Table 1. The tools ultimately chosen for analysis appear in **bold**. Some tools appear in more than one category because they support multiple tasks. The list is by no means exhaustive, but at the same time covers tools that support a range of tasks empirically identified as central to software development processes.

[3] The version of Jazz surveyed here is the older (and only published) version, as described by Hupfer and colleagues (Hupfer et al. 2004). A newer version can be downloaded at http://www.jazz.net.

Tool Name	Content	Task and Corresponding Phase of Lifecycle		Interpretive Gaps	Temporal Unit of Analysis	Visual Representations
Understanding Change Management and Evolution						
VRCS (Koike and Chu 1997)	Visual representations of files and versions	Version control and module management; building a system	Implementation-Maintenance	N/A	Whole state of repository—not real time	2-D network graphs laid out in 3-d, explicitly model files, versions, and releases; edges indicate which versions should be compiled together
ADVIZOR (Eick et al. 2002)	Software changes, authors, issue requests	Version control, module management	Maintenance	N/A	Whole state of repository—not real time	Matrix displays, 2-D and 3-D bar charts, pie charts, line oriented displays, network graphs
XIA/CREOLE (Wu et al. 2004a)	Architecture/source -code changes and associated time, location and authorship	Version control, module management	Implementation-Maintenance	N/A	Current state of repository—not real time	Architecture diagram, call graphs, and data flow views
softCHANGE	Metadata from CVS and BugZilla (versions and bugs)—Files, bugs, authors	Version control, module management	Maintenance	Method for categorizing issues based on changes is not transparent	Whole state of repository – not real time	Network graphs, histograms showing files, authors, author-file relationships
EVOLUTION MATRIX (Lanza 2001)	Software releases and versions, number of additions/removals of modules over time	Show software evolution	Maintenance	N/A	Whole state of repository – not real time	2-D Matrix view with classes on y-axis and time ordered releases on x-axis
BEAGLE (Tu and Godfrey 2002)	Software releases and modules, evolution metrics	Understand evolution: structural changes	Maintenance	N/A	Whole state of repository – not real time	Call graphs, tree views and scatter plots
SPECTOGRAPH (Wu et al. 2004b)	Software releases/revisions, authorship	Punctuation—sudden and discontinuous change; which parts of system were frequently modified; Identify developer coding behavior	Maintenance	N/A	Whole state of repository—not real time	"Spectograph" with time ordered releases on x-axis, files authors or directories on y -axis
REVISION TOWERS (Taylor and Munro 2002)	Change history: revision information and authorship	Change management	Maintenance	N/A	Whole state of repository—not real time	"Tower" metaphor compares revisions of two files with width attribute indicating size of change; map towers to timeline

WORKSPACE ACTIVITY VIEWER (WAV) (Ripley et al. 2007)	Real time workspace changes, types of changes and who made them	Detect workspace changes--Project management	Implementation-Maintenance	N/A	Whole state of workspace and repository—real time	3-D towers, developers or artifacts, with attributes representing change size, type, age
COMMAND CONSOLE	Change history, structure of source-code, ongoing changes	Project management—identifying ongoing change and reduce conflicts	Implementation-Maintenance	Conflicts are color-coded by the likelihood they will occur; rationale behind the likelihoods is not conveyed through the interface.	Current state of repository/workspace; real time	8 linked screens; complexity thumbprint, hierarchical files/folders view, stacked layout showing relative size and ongoing changes to artifacts
GEVOL (Collberg et al. 2003)	Source-code, developers, change history	Change management, understanding evolution	Maintenance	N/A	Whole state of repository—not real time	Novel graph format showing control flow, inheritance, and call graphs annotated with author colors showing authorship
CODE CRAWLER (Lanza 2003)(Lanza et al. 2005)	Source-code metrics, change history	Reverse engineering	Maintenance	N/A	Whole state of repository—not real time	Hierarchical graphs, attributes like width and height indicate number of methods/fields
RelVIS (Pinzger et al. 2005)	Change history, evolution metrics across releases, module dependencies	Change management, understanding evolution	Maintenance	N/A	Whole state of repository—not real time	Kiviat diagrams that show multiple releases of software per diagram and different metrics for each release
ELVIN/TICKERTAPE	CM commits, source-code and authors involved, developer discussion	Coordinate around changes to the system	Implementation	N/A	Current state of repository—real time	Scrolling text-based ticker tape at top of development environment, link to chat window
CODESAW (Gilbert and Karahalios 2007)	Change history, source-code, newsgroup discussion lists	Reveal group dynamics, compare work and discussion done by multiple developers	Implementation-Maintenance	N/A	One year's time of repository—not real time	Stacked timeline view of commits by author (above timeline) and corresponding discussions (below timeline) annotated with developer "post-it" notes
Recommending People and Artifacts						
TUKAN	Related artifacts (e.g., variables, classes, etc.), developers, and potential conflicts	Find developers who are knowledgeable about code/avoid parallel conflicts	Implementation-Maintenance	Weather symbols indicate potential for conflict but no quantifiable measure	Current state of repository as well as concurrent changes	Graph of artifacts and relationships (e.g. inheritance, composition, etc.)
SIJ (STeP-IN Java) (Ye et al. 2007)	Source-code, documentation, discussion archives,	Support information-seeking during development	Implementation-Maintenance	If question is answered, asker can see the helper's technical	Current state of repository—real time	Web interface, standard menus, code browser

	developer and code profiles			profile/expertise and helping history		
EXPERTISE BROWSER (Mockus and Herbsleb 2002)	Developers and their code commits/bug reports for particular artifacts	Find expert on aspect of software system	Implementation-Maintenance	Asker can see distribution of helper's code commits and activity on files	Whole state of repository—not real time	List view of individuals from organizational chart, horizontal box representing code; width/height attributes indicate magnitude of commits
EXPERTISE RECOMMENDER	Individuals and their contact information	Find expert on aspect of software system	Maintenance	Asker can specify what groups should get a request, but the process of recommending is done on the server; not transparent, only details provided are contact information	Whole state of repositories—near real time	List of individuals by department/social network and general problem topic
ANSWER GARDEN 2 (Ackerman and McDonald 1996)	Individuals and their contact information	Find expert on aspect of software system	Implementation-Maintenance	N/A	Current state of repository—not real time	Web interface, standard menus in web browser
HIPIKAT	Relevant source-code artifacts, bugs, documentation, reasons for recommending the artifact	Information-seeking; finding artifacts related to task; software change task for example	Maintenance	Textual description of why the artifact is related and confidence measure from 0-1	Whole state of repository—not real time	Tabular list view integrated into Eclipse window
CODEBROKER (Ye and Fischer 2002)	Relevant source-code examples, documentation	Information-seeking; finding reusable components	Implementation-Maintenance	Description of component received, and confidence measure from 0-1	Current state of repository—real time	Editor windows for code examples, HTML pages for documentation, list tabular view for recommended components
TEAM TRACKS	Source-code	Information-seeking, code navigation and understanding	Implementation-Maintenance	Arrows indicate dependency relationships as a surrogate for explicit rationale; horizontal bars predict relative ranking of results in conjunction with rank-ordered list	Current state of repository—not real time	Tabular list view integrated into Eclipse Window, decorators indicate dependency directionality, favorites window shows most visited code, related code window should code related to the selected code in the editor
Avoiding Conflicts						
PALANTÍR	Annotations on source-code indicating potential conflicts resulting	Reduce check-in conflicts	Implementation	Text in-line with decorators indicate impact severity and types and sizes of	Current state of repository as well as concurrent changes	Visual cues and textual indications of severity in code editor, hierarchical view shows pairwise conflicts

	from ongoing changes			conflicts		
STATE TREEMAP (Molli et al. 2004)	Real time workspace changes; what artifacts are being changed by whom	Detect ongoing parallel work	Implementation	No quantifiable measure of divergence	Current state of workspace—real time	Treemap of workspace indicates different state (e.g. checked out and modified, checked out and newly committed) of artifacts by shading
COMMAND CONSOLE (O'Reilly et al. 2005)	Change history, structure of source-code, ongoing changes	Project management—identifying ongoing change and reduce conflicts	Implementation-maintenance	Conflicts are color-coded by the likelihood they will occur; rationale behind the likelihoods is not conveyed through the interface.	Current state of repository/workspace—real time	8 linked screens; complexity thumbprint, hierarchical files/folders view, stacked layout showing relative size and ongoing changes to artifacts
COLLABVS (Dewan and Hegde 2007)	Change history, ongoing changes	Reduce check-in conflicts	Implementation-Maintenance	Details of conflict in inbox	Current state of repository; ongoing changes—real time	Conflict inbox, visual cues/notifications, chat window
TUKAN	Related artifacts (e.g., variables, classes, etc.), developers, and potential conflicts	Find developers who are knowledgeable about code/avoid parallel conflicts	Implementation-Maintenance	Weather symbols indicate potential for conflict but no quantifiable measure	Current state of repository as well as ongoing changes—real time	Graph of artifacts and relationships (e.g. inheritance, composition, etc.)
FASTDash (Biehl et al. 2007)	Documentation, source-code, files and modules people are editing in real time	Who has file checked out, what files are being viewed and edited by who, detect conflicts	Implementation-Maintenance	N/A	Current state of repository—real time	Integrated window with sections for source-code, documentation, annotated with user avatars of who is working on what; runs on shared display
MIRAMAR (Hancock et al. 2006)	Users' workspace and the artifacts with which they are working	Detect conflict and divergence with 3-D feedback	Implementation-Maintenance	N/A	Current state of repository—real time	3-D visualization of different users' workspaces and "stretchy" connections between them when there are conflicts due to modifying shared resources
Determining Individual Availability						
ACTIVE MAP (McCarthy 1999)	Location and movement of people	Locating people and their movement	Whole process	Only as good as location sensors	Real time; last updated	Map background, user avatars overlaid on buildings
AWARENEX	Computer activity, appointments, computer activity with appointments, and gaps with no activity at all	Identify rhythms in people's availability to form a shared sense of time	Whole process	N/A	History of all activity in 3 week intervals	"Actogram" 12 hr timeline view on horizontal axis and 3 week timeline on vertical axis; availability blocks within a 12 hr span laid out horizontally; also an

						aggregate view over 10 months
JAZZ (Hupfer et al. 2004)	Source-code, change history, developers, chat mechanisms, workspace activity	Reveal activities of team members	Implementation-Maintenance	N/A	Whole state of repository; ongoing changes—real time	Jazz band links developer avatars to resources they are working on, decorators indicate file and resource status (e.g. checked out, being modified, checked in, etc.), chat windows anchored in code
COMMUNITY BAR	Artifacts/documents people are working on	Determining availability, monitoring and coordinating, opportunistic collaboration	Whole process	Only as good as screen sharing app; privacy controls restrict view of artifact content	Real time; last updated	Vertical bar with chat windows, screen sharing
FRACTAL FIGURES (D'ambros et al. 2005)	Change history, magnitude of changes, authorship information	Project management–Understand development effort and distribution of effort over developers	Implementation-Maintenance	"Fractal values" from 0-1 note how distributed the module is over a set of developers; can correlate bugs with fractal value for example	Whole state of repository—not real time	Fractals/tree maps characterizing extent of developer effort
TEAM-SCOPE (Jang et al. 2000)	Team documents, emails, files, events, repositories	General team communication and awareness	Implementation-Maintenance	N/A	Current state of repository, not real time	Web-interface, calendar view with events, standard hierarchical file layout
Understanding Developer Activities						
FASTDash	Documentation, source-code, files and modules people are editing in real time	Who has file checked out, what files are being viewed and edited by who, detect conflicts	Implementation-Maintenance	N/A	Current state of repository—real time	Integrated window with sections for source-code, documentation, annotated with user avatars of who is working on what; runs on shared display
TESNA (Amrit 2008)	Source-code dependencies, developer dependencies, chat log communications	Project management–Identify gaps in coordination	Implementation-Maintenance	N/A	Current state of repository—not real time	Social network diagrams depicting dependencies between code and developers, chat log correspondence
ShriMP VIEWS (Storey et al. 2001) (Storey et al. 1997)	Software module structure, documentation, source-code	Program comprehension	Maintenance	N/A	Current state of repository—not real time	Integrated views of call graphs, java documentation, and source-code
TARANTULA (Jones et al. 2001) (Jones and	Test cases and lines of code that execute	Fault localization	Testing	Color code portions of code that may be	Current state of repository—real	Line-oriented view illustrates involvement of each line in

Harrold 2005)	during those test cases			responsible for faults	time	failure/passing of test cases
SHO (Ellis et al. 2007)((Halverson et al. 2006)	Change requests, developers assigned to and resolving them, classifications of bugs	Coordinate bug tracking reporting, assignment and resolution	Maintenance	N/A	Whole state of repository—not real time	Novel visualization: change requests laid out horizontally by component, color-coded by assignee, tool tips for description of change request
ELVIN/TICKERTAPE (Fitzpatrick et al. 2006)	CM commits, source-code and authors involved, developer discussion	Coordinate around changes to the system	Implementation	N/A	Current state of repository—real time	Scrolling text-based ticker tape at top of development environment, link to chat window
AUGUR (Froehlich and Dourish 2004)	Source-code, CM comment logs, authorship, change history	Monitoring activity, Understanding activities over time	Implementation-Maintenance	N/A	Whole state of repository—not real time	Line-oriented view of source-code colored by author, network graphs and line charts
CODESAW (Gilbert and Karahalios 2007)	Change history, source-code, newsgroup discussion lists	Reveal group dynamics, compare work and discussion done by multiple developers	Implementation-Maintenance	N/A	One year's time of repository—not real time	Stacked timeline view of commits by author (above timeline) and corresponding discussions (below timeline) annotated with developer "post-it" notes
JAZZ	Source-code, change history, developers, chat mechanisms, workspace activity	Reveal activities of team members	Implementation-Maintenance	N/A	Whole state of repository—real time	Jazz band links developer avatars to resources they are working on, decorators indicate file and resource status (e.g. checked out, being modified, checked in, etc.), chat windows anchored in code
SEESOFT (Eick et al. 1992) (Ball and Eick 1996)	Source-code change history, authorship, software metrics	Program management	Implementation-Maintenance	N/A	Whole state of repository—not real time	Line-oriented view of source-code with authorship information and associated software metrics, call-graphs, control flow

Table 3—List of visualization tools for awareness and their mappings to different aspects of awareness from **Table 1**.

Understanding Change Management and Evolution

ELVIN/Tickertape

ELVIN/Tickertape is a lightweight chat mechanism and publish/subscribe notification tickertape integrated into CVS, the popular control versioning software (Fitzpatrick et al. 2006). Developers subscribe to groups (e.g. structured by software component) and messages are generated and sent by the system to the tickertapes belonging to each individual in the group whenever code is committed to the repository (see Figure 1a). Developers can compose custom messages and send them to members of their group or others'. Developers use the messages sent to their tickertape to initiate chat dialogs with other developers in response to the CVS message (Figure 1b). ELVIN/Tickertape was evaluated using quantitative statistical analysis of the CVS logs combined with a qualitative analysis of chat logs and interviews. There was evidence of the tool supporting stimulated focused discussion around the changes and supplementing log information with contextual pieces of information such as significance of the changes and corresponding developer discussions. The tool plays an important role in supporting coordination by combining changes to the code and dialog around those changes.

Figure 1a—Tickertape message of the form: group: CVS committer: file modified: comment.

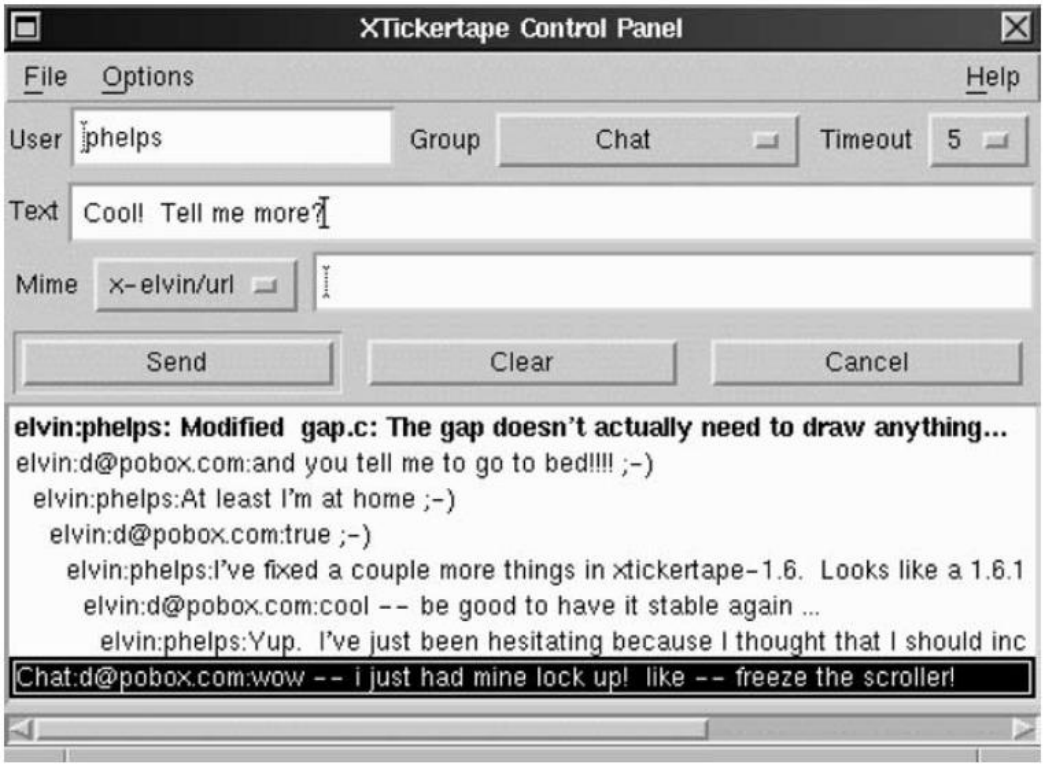

Figure 1b—Threaded chat message around an ELVIN/Tickertape commit message service.

Content: The tickertape displays a message every time a developer commits code to the repository noting the developer who made the change, the source-code artifact that was changed, and the commit comment. Users can also open a dialog box to start a threaded conversation related to the content of the tickertape message. The dialog box serves as a mechanism to associate individuals with the source-code they are working on. Dependencies between people and between source-code must be inferred by the user.
Mapping Content to Tasks: The tool was built to support developer-developer interactions and coordination of development activities around committed changes made to a software system. Code commits are representative of real tasks in software development and happen on a daily basis. The tool is for use by—and was subsequently tested with—developers in the software implementation phase.
Interpretive Gaps: There is no uncertainty in the information displayed. It comes directly from CVS logs and recorded chat transcripts.
Temporal Unit of Analysis ELVIN/Tickertape displays messages in real time as developers make commits to the repository. Notifications and chat logs can be archived using the client interface. However there is no timeline view presenting the changes and the individuals who worked on them so it is difficult to relate the two over a non-trivial length of time.
Visual Representations: ELVIN uses text and simple dialog boxes only. It displays a scrolling "ticker" window with messages and a threaded chat messaging log showing conversations relating to the commits. The representations are light-weight and the tickertape metaphor is well-suited toward displaying abbreviated notifications of all types (e.g., stock prices and sports scores). The representations do not distract greatly from

developer work because reading CVS commit messages and project mailing lists are familiar activities.

Command Console

Command Console is a set of linked visualizations on eight consoles designed to help gauge project progress, reveal conflicts, and build a shared understanding of software development activities (O'Reilly et al. 2005). The consoles update in real time in accordance with developers' activities. The system was evaluated during a 5-week long study of an industrial-sized software project. Project managers said that it gave a good high-level view of where action happens in the code and that it helped bring new developers up to speed. Command Console also helped project managers "understand the impact that changes were making." (O'Reilly et al. 2005).

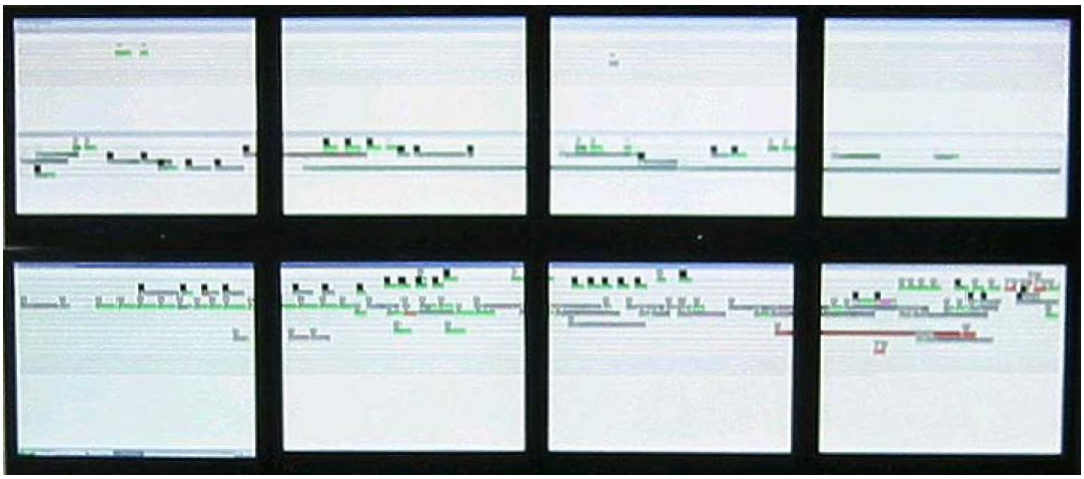

Figure 2a—The Command Console Display.

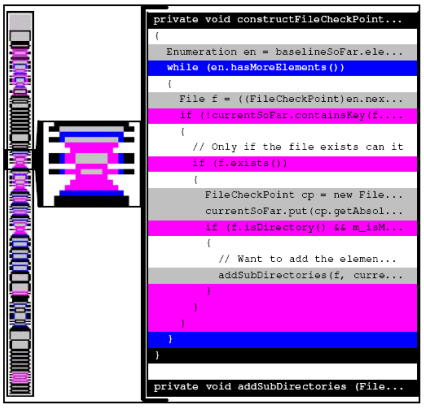

Figure 2b—The "Complexity Thumbprint," a visualization that displays source-code size and structure.

Content: The linked displays show attributes of source-code revisions, including size and structure, using "complexity thumbprints" (Figure 2b). One display shows groups of complexity thumbprints organized by the modules and system-level components they belong to while another displays the ongoing changes to artifacts and warnings of any

potential conflicts as a result. It is possible to color the complexity thumbprints by authorship and relate developers to portions of the code they implement, yet dependencies must be inferred from the visualizations.

Mapping Content to Tasks: Command Console was engineered to help project managers and developers gauge the progress of a system under development, understand its complexity and size, and alert them to conflicting development efforts. These are activities that generally occur during implementation and maintenance phases of software development.

Interpretive Gaps: There is slight uncertainty in the information displayed. It comes directly from CVS repositories and events generated from developer workspaces. Conflicts are color-coded by the likelihood they will occur (potential vs. certain) although the rationale behind the likelihoods is not conveyed through the interface.

Temporal Unit of Analysis Command Console shows activities in real time as they occur. There are no views of activities and developers involved in those activities over the course of the software project.

Visual Representations: Command Console uses graphics and text to convey awareness information. It lays out artifacts in familiar hierarchical list views but makes use of unconventional display setups and representations of source-code developers do not typically use. The tool uses a "war room" shared display, best-suited for co-located teams, yet today's teams are increasingly becoming distributed. It is not clear how or even, if, the Command Console could work in such configurations of teams. Assigning a Command Console unit to each location would be costly. One solution might be to broadcast the large image from a central location to computers that can simply project the image on shared walls or projector screens belonging to remote team members. On the other hand the decrease in resolution associated with most projectors might make it difficult to see and interpret important details in the visualization.

The "complexity thumbprints" convey size, structure, and ongoing changes in developers' workspaces. They are novel representations and require some initial learning. However when aggregated, they, crucially, show important patterns in structure at the system-level in concert with tasks color-coded with the source-code that references them.

SoftCHANGE

SoftCHANGE (German et al. 2004) extracts metadata from a CVS repository and a team's corresponding issue tracking system and correlates both. It analyzes different revisions of the same files to determine the exact nature of the changes made and attempts to classify changes based on the issues they address (e.g. new features, code defects). The tool provides graphical views of source-code, authors, and their relationships, such as what source-code was modified together and who modified what files when.

Project Mozilla

Details of Modification Request

By Date | By Author | By File Name | By Bugzilla Bug Number | Search

MR id	Author	Files Modified	Date	Time	Description
mikep%oeone.com:2003/01/13 14:11:52	mikep%oeone.com	4	2003-01-13	14:11:52	Fixing bug 109476, in mc Fixing bug 188888, in mc Fixed thanks to patches

Files in MR

Filename	RevisionId	Lines Added	Lines Removed	Lines Total	State
calendar/resources/content/calendarEvent.js	1.45	27	0	27	Active
calendar/resources/content/calendar.xul	1.124	21	14	7	Active
calendar/resources/content/monthView.js	1.49	85	15	70	Active
calendar/resources/content/monthView.xul	1.19	12	10	2	Active
Total		**4**	**145**	**39**	**106**	

Figure 3a—A hypertext view of the details of a change request.

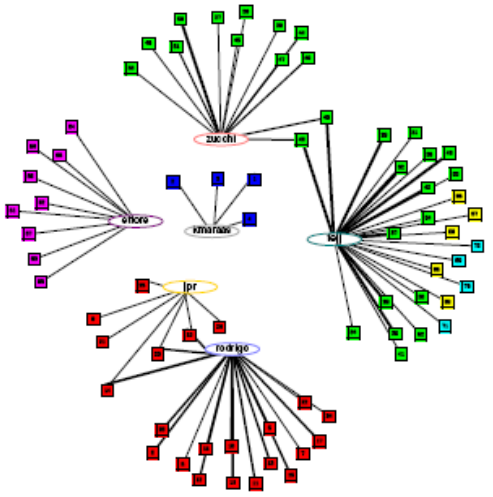

Figure 3b—A network graph of authors and the files they modified, color-coded by module.

Content: SoftCHANGE displays changes from CVS along with issues managed by a project issue tracker and the developers involved. The tool displays relationships between them, such as relationships between number of files and number of open issues, numbers of functions added versus number and types of change issues, and developer activity compared with the number of change issues. The graphs can also be used to show who is working on what code, but not who depends on whose code.

Mapping Content to Tasks: SoftCHANGE was built to support project managers, developers, and researchers as they try to reason about the evolution of a software project in terms of changes made and issues introduced. These are tasks that typically occur in the maintenance phase of software development.

Interpretive Gaps: There is slight uncertainty in the validity of the information presented, particularly the association of bugs with change requests. The process of categorizing issues based on changes is not transparent to the end-user.
Temporal Unit of Analysis The tool allows extraction of data over the course of the whole project. Some of the views present time-ordered scatter plots of relationships between source-code, individuals, and bugs. However the views do not update in real time.
Visual Representations: SoftCHANGE uses a combination of textual descriptions and graphical charts to convey information. It uses a hypertext view to show details related to specific change requests, including when and why a change was made, the type of change, and if it was fixed. Graphical views such as scatter plots of time ordered information allow users to inspect relationships between bug rates and attributes of files changed. Color-coded network diagrams are used to show relationships between developers and the files they have modified. It is not clear whether these visualizations are linked together or how they are used in concert to reason about changes made to the system.

Recommending People and Artifacts

Expertise Recommender

Expertise Recommender is a tool for locating expertise needed to solve difficult technical problems (McDonald and Ackerman 2000). Development of the tool was preceded by a field study that resulted in analytical guidelines for locating expertise: expertise identification, expertise selection, and escalation. Expertise Recommender provides support for this model of expertise. A user specifies a knowledge request via a client interface (Figure 4a) and sends it to a server that processes the request and makes recommends of individuals with matching expertise. Before sending the request, the user selects a location heuristic (e.g. "change management," "tech support") to define the repository that should be searched (CM system or tech support database respectively) and a filter (e.g. "social network," or by department) to filter out people not associated with particular groups in the organization. If results returned by the tool are not sufficient, the expertise-seeker can "escalate" the request to other contacts in different departments who have higher authority or access to more resources.

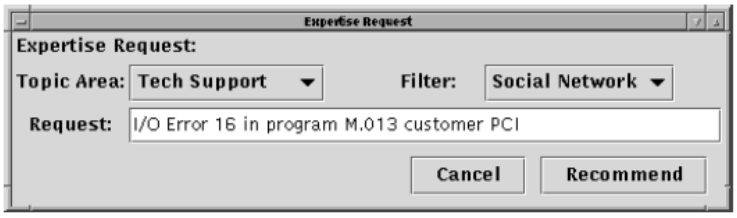

Figure 4a – An expertise request dialog window.

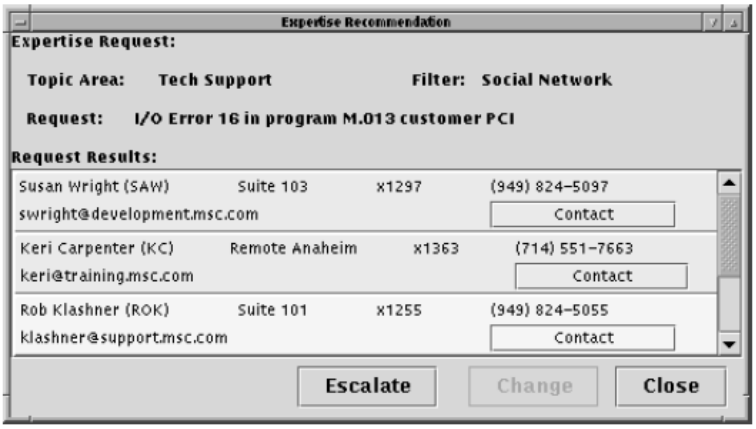

Figure 4b – Recommendation results returned by Expertise Recommender with the option to escalate the request.

Content: Expertise Recommender links people and the extent with which they have worked on technical artifacts to offer recommendations of people who are best-suited to address problems and requests defined by the expertise-seeker. Recommendations are based on data contained in CM repositories and technical support databases as well as individuals' social networks. The recommendations returned by the tool include contact information for each individual and their location yet no description of the work they have performed relative to the technical issue. It also gives no indication of whether or not that person may be currently available.

Mapping Content to Tasks: The tool was developed in response to field studies that elicited analytical steps in the process of expertise location: expertise identification, expertise location, and escalation. It supports each activity individually through the use of a client interface. The problem of finding expertise is framed in the context of resolving bugs found in the source-code and solving technical support issues. These activities are representative of tasks in the maintenance phase of software development after a version of the software has been deployed.

Interpretive Gaps: There is minor uncertainty in the information displayed. There is flexibility in the request process: users can direct the requests to specific groups and classify the requests. Results are returned in ranked order, but the ranking process is not transparent and no description of it is provided on the interface. There are no indications of whether or not individuals still have particular expertise, in spite of the fact that projects end and technical knowledge erodes over time.

Temporal Unit of Analysis The Expertise Recommender client interface maintains a history of all requests and makes recommendations based off the entire history of the CM repository and technical support database.

Visual Representations: The tool's UI makes very little use of graphics, displaying small dialog windows with the familiar "look and feel" (text fields and buttons) for filling out and sending expertise requests. It returns recommendations via another dialog window in a scrollable list with textual descriptions of experts and their contact information. It is a stand-alone system instead of integrated into the tools used by technical support representatives or the development environments used by developers. On the other hand, it is relatively lightweight as well. No usability issues are reported by the authors.

Hipikat

Hipikat is an awareness tool that leverages project archives to make recommendations of related artifacts related to developer tasks, such as changing a piece of source-code, to support developer productivity (Cubranic and Murphy 2003). Given a change request, a developer queries the Hipikat interface for related artifacts and is returned a list of related source-code and bug reports (Figure 5). Upon inspecting the results and their relevance criteria, the developer drills down to each recommendation to find the information of interest.

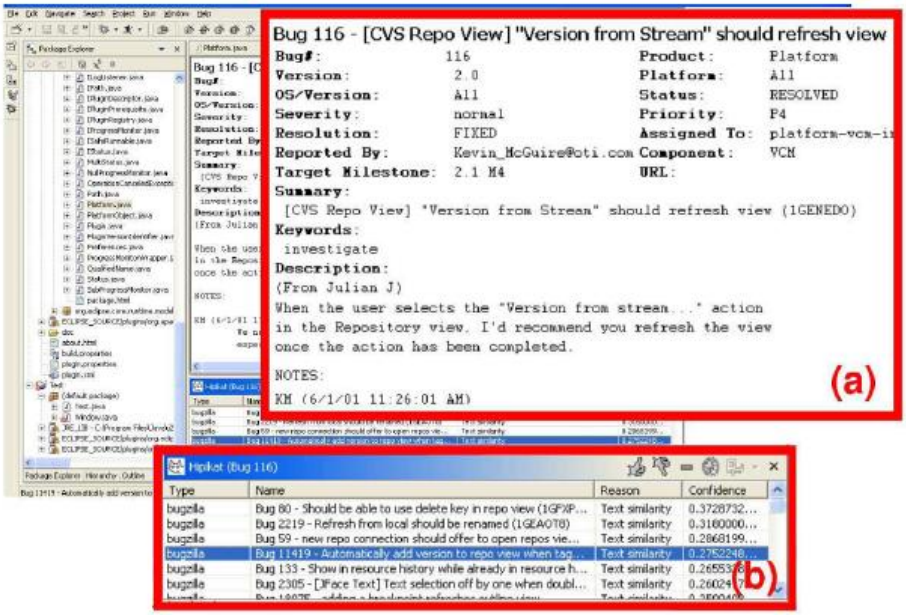

Figure 5—Hipikat UI integrated into Eclipse displaying the change task (a) and a list of related artifacts (b)

Content: Hipikat crawls CM repositories and issue tracking systems and recommends related artifacts (e.g. source-code, change requests) based on the task the user (i.e. developer) is currently performing. It explicitly models relationships between versions of source-code and bugs but source-code authorship and bug assignment information is not shown. These relationships must be inferred.
Mapping Content to Tasks: Broadly, the tool supports developers who wish to learn about a code base by recommending artifacts relevant to common development tasks like resolving bugs and making change modifications. In particular, the authors describe a scenario of Hipikat's use for addressing change requests contained in an issue-tracking system. These are activities typically performed in the maintenance phase.
Interpretive Gaps: There is some uncertainty in the recommendations made by the tool. It displays evidence using confidence intervals from 0-1 in addition to textual descriptions (e.g. text similarity in the change request and the source-code itself). Yet researchers have shown that numeric values of confidence make little sense to users (Herlocker et al. 2000). The authors try to make up for this shortcoming by using textual

descriptions as much as possible but even their usefulness is uncertain (Cubranic and Murphy 2003).

Temporal Unit of Analysis The recommender searches all versions of artifacts and displays all versions seemingly relevant to the current task. There are no timeline views that might assist developers by displaying who else used artifacts and for how long to complete the same tasks. It might then be possible to gauge relevant sets of artifacts based on their length of use.

Visual Representations: Hipikat primarily uses textual decorators within the development environment to convey recommendations. It uses lists and windows fully integrated into the Eclipse development environment, thus the interface has the same "look and feel." Queries are performed by right-clicking on artifacts in the familiar hierarchical source-code view and entering search terms into Eclipse dialog boxes. Results and confidence are expressed via textual descriptions in the same list dialogs used by Eclipse. Because of this design, context-switching is noticeably reduced and no substantial time spent "learning" the interface is required.

Team Tracks

Team Tracks is a visualization that unveils developers' patterns of navigation through source-code in an effort to support comprehension of that code by users who are new to the code (DeLine et al. 2005). The visualizations are integrated into Microsoft Visual Studio dialogs much in the same way Hipikat is integrated with Eclipse. Team Tracks is based on two assumptions: 1) parts of the code developers visit more frequently are more important to someone new to the code and 2) the more two snippets of code visited are visited in succession, the more likely they are to be related. As such, the visualization is composed of two displays within the development environment: "Code favorites" (e.g. frequently visited code) and "Related Items." The tool was evaluated quantitatively through a program comprehension quiz and user satisfaction ratings as well as qualitatively through interviews with developers who used it to complete typical development tasks such as change requests.

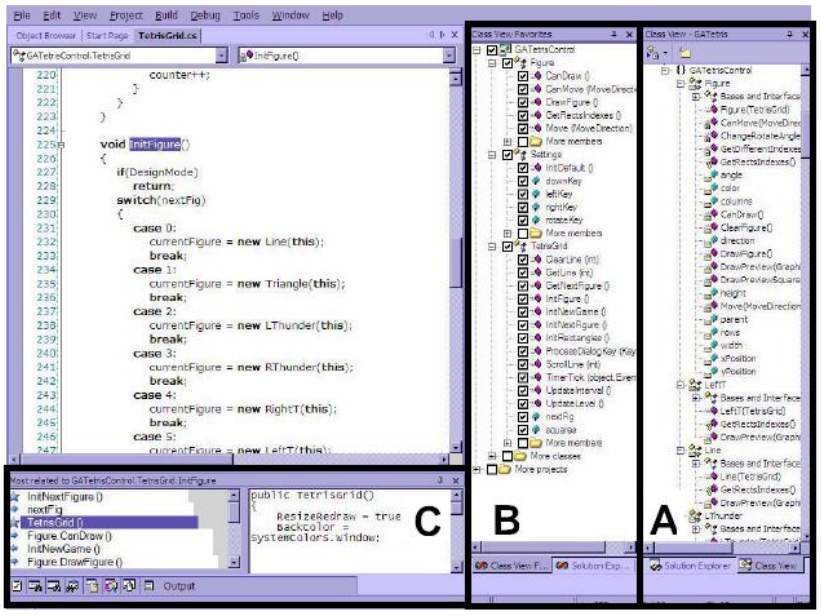

Figure 6—Team Tracks interface in Visual Studio displaying A) A standard directory/file view, B) Code Favorites and C) Related Items to source-code selected in the development editor.

Content: Team Tracks uses windows integrated into Visual Studio to display the source-code most viewed by developers and rank-ordered related methods to the source-code currently in focus in the development editor. The tool uses icons to show whether the methods have incoming or outgoing dependencies to the source-code in the editor but the call graph for a method is not displayed. This could make traversing the call path easier for the user. No authorship information is shown by the tool so it is not possible to determine who navigated dependent code without querying a CM repository. Navigation data from a senior architect, for example, would possibly be more revealing than a programmer.

Mapping Content to Tasks: The tool supports the broad activity of understanding source-code based on the navigation patterns of other developers so the user can complete implementation tasks such as changing code or maintenance activities such as adding features or resolving bugs.

Interpretive Gaps: There is some uncertainty in the recommendations made by the tool. Although horizontal bars and ordering of items in the list convey rankings of relatedness, evidence for ranking one suggestion over the other is not explicitly provided. Instead, the basis for interpretation is team navigation patterns, which is not a reason for related code the same way dependencies, author ownership, or inheritance relationships are, for example. As a substitution for explicit rationale, the authors use arrow icons indicating incoming or outgoing dependencies from the related code to the code selected in the editor.

Temporal Unit of Analysis Team Tracks uses navigation patterns from the most current snapshot of the CM repository and provides recommendations based off those navigation patterns. The navigation data is not collected and used to assist the user in real time. It is not clear how often the navigation data should be collected.

Visual Representations: Team Tracks uses a combination of textual decorators and visual icons to convey awareness information. Like Hipikat, Team Tracks' visualizations

are integrated into familiar windows part of the everyday implementation work of software developers. The views are anchored around where developers spend most of their time in the editor: the source-code. The "Class Favorites" window displays favorite code items the same way in which the Visual Studio environment hierarchically displays directory structures. The "Related Items" view uses list structures also found in the environment interface. In addition, the interface uses good information presentation principles: small horizontal bars appear next to source-code items indicate relative ranking, inviting direct comparisons and contrasts to other ranked items (Tufte 1990; Tufte 2006).

Avoiding Conflicts

Palantír

Palantír is an Eclipse plug-in that supports developers in identifying and avoiding conflicts that arise from committing different versions of the same file to a CM repository (Sarma et al. 2003). The tool increases awareness by continuously sharing information regarding other developers' actions on files in the workspace, for example the potential for conflicts and the severity and impact of changes to those files. The tool was empirically evaluated using a lab experiment (Sarma et al. 2008b). The results indicate that Palantír increased self-coordination among users and, as such, led to fewer conflicts.

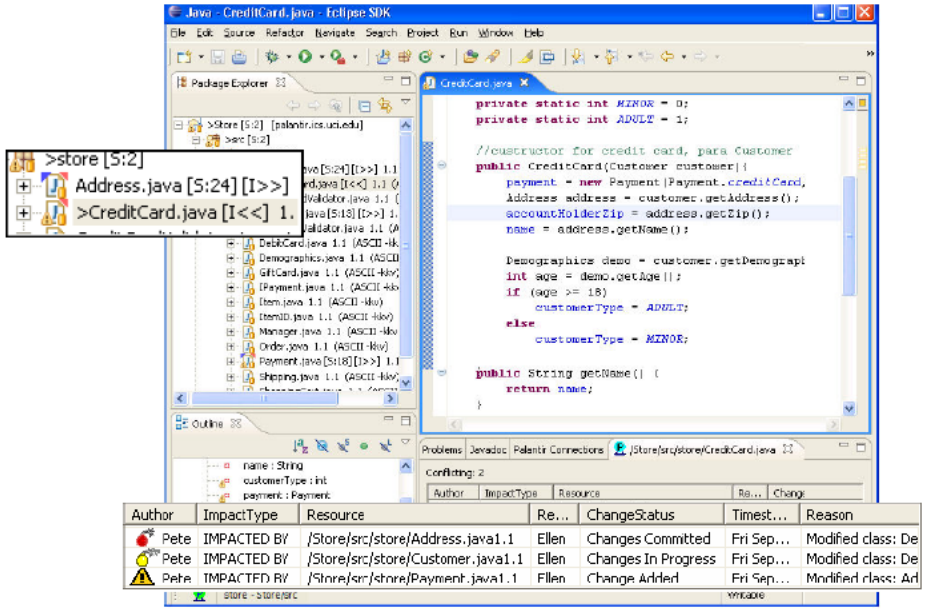

Figure 7—Palantír-enhanced Eclipse workspace with annotations to files in conflict (left) and description of impacts of changes (bottom).

Content: Palantír extracts information from a number of control versioning repositories and monitors activities in the workspace for changes to the local and repository versions of source-code. It analyzes the differences between files in order to compare the number

of lines changed and calculate a measure indicating the severity of the changes. The tool then translates these activities to events that it subsequently shares with all affected user workspaces. Palantír shows textual descriptions of these events, including who is changing what, whose code will be impacted by the change, and when the change happened or is currently in progress. These events describe dependencies of the type "impacted by" rather than dependencies of the type "is called by."

Mapping Content to Tasks: The tool supports developers as they implement source-code and make changes to a versioning repository, a standard daily task in software development. It supports coding activities by passively annotating the files in the workspace as potential conflicts emerge in real time. These activities constitute the implementation and maintenance phases of software development.

Interpretive Gaps: Palantír displays various pieces of evidence for potentially troublesome parallel work. It displays three measurements: the potential for and sizes of conflicts, and the impact the changes will have on dependent code. Small, blue triangles next to files in the resource view indicate parallel changes to the same artifact in different workspaces, signifying a direct conflict. Red triangles indicate parallel changes to another artifact in another workspace that will affect the current artifact. The sizes of the triangles indicate the magnitude of the change. Arrows (>> or <<) indicate whether the artifact affects or is affected by other changes. Finally, a percentage value indicates the severity of the changes in terms of the lines-of-code affected. Users can quickly view differences in source-code if they desire to see the actual impacts of the changes.

Temporal Unit of Analysis Palantír focuses on activities occurring in developer workspaces in real time. It continually extracts information from previous revisions of files and logs events as they occur over time to provide ongoing awareness. Over time, patterns emerge as activities continue and developers can use these patterns to self-coordinate their work and avoid running into conflicts. Thus it supports activities occurring in the present and in the past.

Visual Representations: Like Hipikat and Team Tracks, Palantír's interface is integrated with the development environment so as not to distract from current tasks. The source-code editor anchors interactions with Palantír because much of the work involved in analyzing conflicts revolves around looking at source-code. It uses visual decorators in the resource view of Eclipse to indicate types and sizes of emerging conflicts and textual representations to indicate impact severity and give details about who is involved, what code was affected, and when the changes occurred.

TUKAN

TUKAN is a collaborative development tool with the broad goal of orienting developers around code and promoting awareness of other developers' activities (Schummer and Haake 2001). It displays graphs of artifacts that are semantically related and extracts versioning information from a CM repository to determine the severity of potential conflicts (Figure 8).

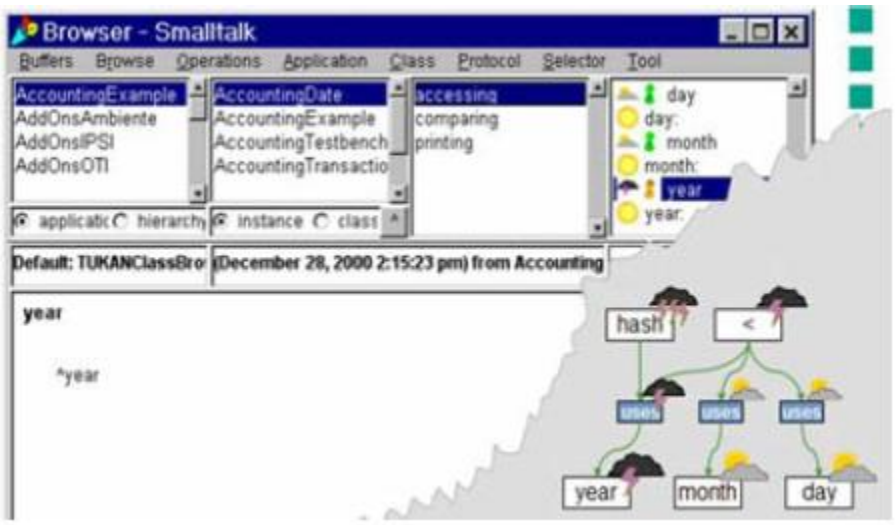

Figure 8—Graph of related source-code with weighted relationships signifying "use" relationships and weather icons indicating potential conflicts.

Content: TUKAN connects to a source-code repository and semantically analyzes source-code artifacts and determine which ones are related due to dependencies, use, and inheritance relationships. It collects information from developers' Smalltalk workspaces to identify who is working on what files and whether conflicts will emerge. It annotates the graphs of source-code artifacts with icons signifying active developers and weather icons corresponding to the severity of parallel work on those same artifacts. Developers can get a feel for who will be impacted to changes to their code and whether they will be affected by changes to others' code.

Mapping Content to Tasks: TUKAN supports the broad goal of orienting developers around code, being aware of who is working on what, and identifying conflicts—all important parts to development work within both the implementation and maintenance phases of software development.

Interpretive Gaps: There is some ambiguity in the mechanism that calculates potential conflicts. Although weather symbols can show the severity of conflicts relative to one another (lightening vs. clouds and sunshine vs. lightning), they do not map to absolute, quantifiable measures of conflict or combined with other metrics like impact severity and conflict type (indirect or direct), as in Palantír.

Temporal Unit of Analysis Notifications of ongoing changes and potential conflict are in real time. Users use synchronized editors to make changes to the system and can view multiple versions of source-code from the code repository. Thus, like Palantír, TUKAN supports exploration of current and past activities.

Visual Representations: TUKAN primarily uses graphics to convey awareness information. It is integrated with an online SmallTalk development editor so as not to distract from development activities by way of context-switching. The tool uses notations familiar to software developers like dependency graphs and graphs showing use and inheritance relationships. The graphs are annotated with intuitive visual decorators like "people" icons showing developers who are working on source-code as well as weather metaphors that convey "sunny" (and thus positive) or "stormy" (and thus negative) states with respect to potential conflicts.

Determining Individual Availability

Awarenex

Awarenex (Begole et al. 2002) is a visual awareness tool that reveals patterns in people's work schedules such as: where they are during times of the day, what times of the day they are available, when they are busy with appointments, what times they arrive and depart for work, what time zones they are working in, and when they break for lunch. The tool logs input received by the user's keyboard to determine when they are active and available for contact and inactive. It collects data from online calendars to infer when users are in appointments. The visualizations show activity over a 12 hour day during the 10 months the data were collected.

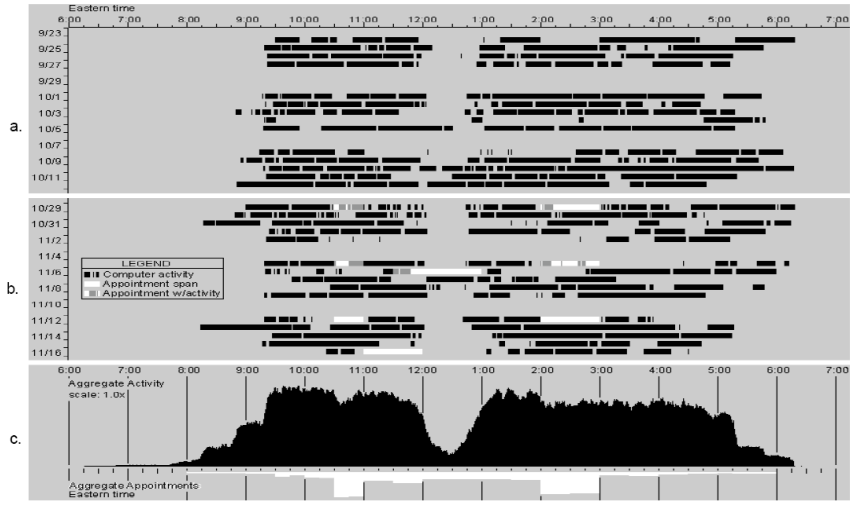

Figure 9—Visual interface showing a) 3 weeks of activity with active periods in black, b) another 3 weeks with white indicating appointments, and c) an aggregate view over 10 months of daily activity. Arrivals and departures can be seen in a) and b).

Content: Awarenex uses input from the keyboard to determine when users are active or inactive. It logs whether the user is reading or sending e-mails and analyzes their online calendar to determine when the user is in an appointment and unreachable. Location data is also collected (e.g. office, home, lab). People can be mapped to their general work activities as well as when and where they do work. Thus their availabilities can be inferred from the visualizations.

Mapping Content to Tasks: The tool supports locating patterns in people's availability such as when they are switching offices, on lunch breaks, working to meet appointment deadlines, or leaving for the day. These patterns can be used, for example, to suggest when someone will return from being active. Although the visualization is not representative of typical visualizations used by software developers, it nicely structures the data. It reveals insights into the work patterns of groups working in any domain, contributing to a shared sense of time and the availability of workers. Awarenex can seemingly be used at any phase of the development process.

Interpretive Gaps: There is uncertainty in determining whether someone is available. Just because someone is reachable does not mean they will be receptive to interruption. That depends on their current tasks, history of interaction with the asker, and their perception that helping will benefit them in some way. At the same time, established, long-term patterns of availability are good predictors of future availability.
Temporal Unit of Analysis Awarenex retrospectively visualizes snapshots of daily computer interactions over a 10 month period, not as they occur in real time. Use of the timeline is critical because it is only over an extended period of time, by their definition, that patterns can be identified. Developers could use changes in the rhythms as a result of project deadlines, for example, in concert with a social dependency graph (de Souza et al. 2007) to suggest times to meet with other developers to coordinate dependent work up to a release of the software.
Visual Representations: Awarenex uses graphics and minimal text to visualize activity using a time-ordered x-y axis. The x axis shows a 12 hour day while the y axis shows days increasing from top to bottom (Figure 9). Activity is graphically represented by horizontal bars with length representing time and different colors indicating certain types of activity (e.g. appointments, reading e-mail, etc.). Reading downward, the horizontal bars invite direct comparisons and contrasts (Tufte 1990; Tufte 2006) in availability and types of activity day-to-day.

Community Bar

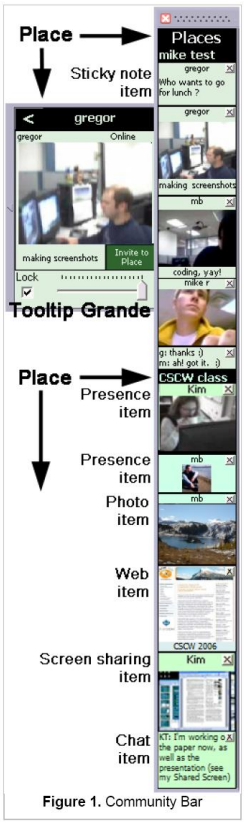

Figure 10—Community Bar content is composed of places, presence information, chat dialogs, sticky notes, photo items, and web items (e.g. webpages)

Community Bar (Tee et al. 2006) is an application that allows users to share their screens with one another, engage in real time chat messaging, share video information about where they are, and post digital objects with which they are working. The tool supports progressive visibility of information users choose to reveal. Users can control the visibility of their information by blurring sensitive parts of content. Feedback from usage of the tool suggested that it was useful for opportunistic interactions, monitoring when people could be interrupted, and to measure progress on collaborative tasks.

Content: Community Bar allows the sharing of multiple types of artifacts such as video frames, images, web resources and even the users' screens (see Figure 10). It shows individuals and the artifacts they choose to share but not who is monitoring, and thus dependent on, the availability of whose artifacts. Shared content is visible to everyone connected to Community Bar except when privacy controls are used to selectively display the content. For example, someone might want to share an unfinished draft of a document only with their collaborator(s) and then release it with full visibility once it is completed. One's availability can be determined by looking at their desktop screen to determine what they are working on, their status message, or their webcam.
Mapping Content to Tasks: The tool gives insight into the artifacts with which people are currently working and thus a basis from which to initiate spontaneous interactions and coordinate shared activities such as working on documents and gauging work progress. The tool could be used at any phase of the software process.
Interpretive Gaps: One's knowledge of the artifacts with which one is working is only as good as the artifacts the latter *decides* to display. If a user forgets to update what they are currently working on, another user's perception of that work may be irrelevant since it came from older data. Privacy controls restrict content displayed, clouding others' knowledge of their availability.
Temporal Unit of Analysis Community Bar displays artifacts users have most recently posted. There is no time ordered view of shared postings.
Visual Representations: Community Bar's interface consists of one long vertical window that lays out posted artifacts from top to bottom. It uses graphics and when applicable, text corresponding to peoples' status, short announcements to others in the group, and chat logs.

Understanding Developer Activities

FASTDash

FASTDash (Biehl et al. 2007) is a visual "dashboard" widget embedded on a single monitor or large, shared screen that allows software developers and their teams to monitor all ongoing activities including what files are checked out, what is being changed and by whom, what files are being viewed, and what files are undergoing debugging. The tool's requirements were gathered from interviews with 13 developers. In response to these interviews the tool was built and qualitatively and quantitatively evaluated using 6 of the original 13 programmers as users. It was shown to improve team awareness, reduce reliance on shared artifacts, and increase team communication.

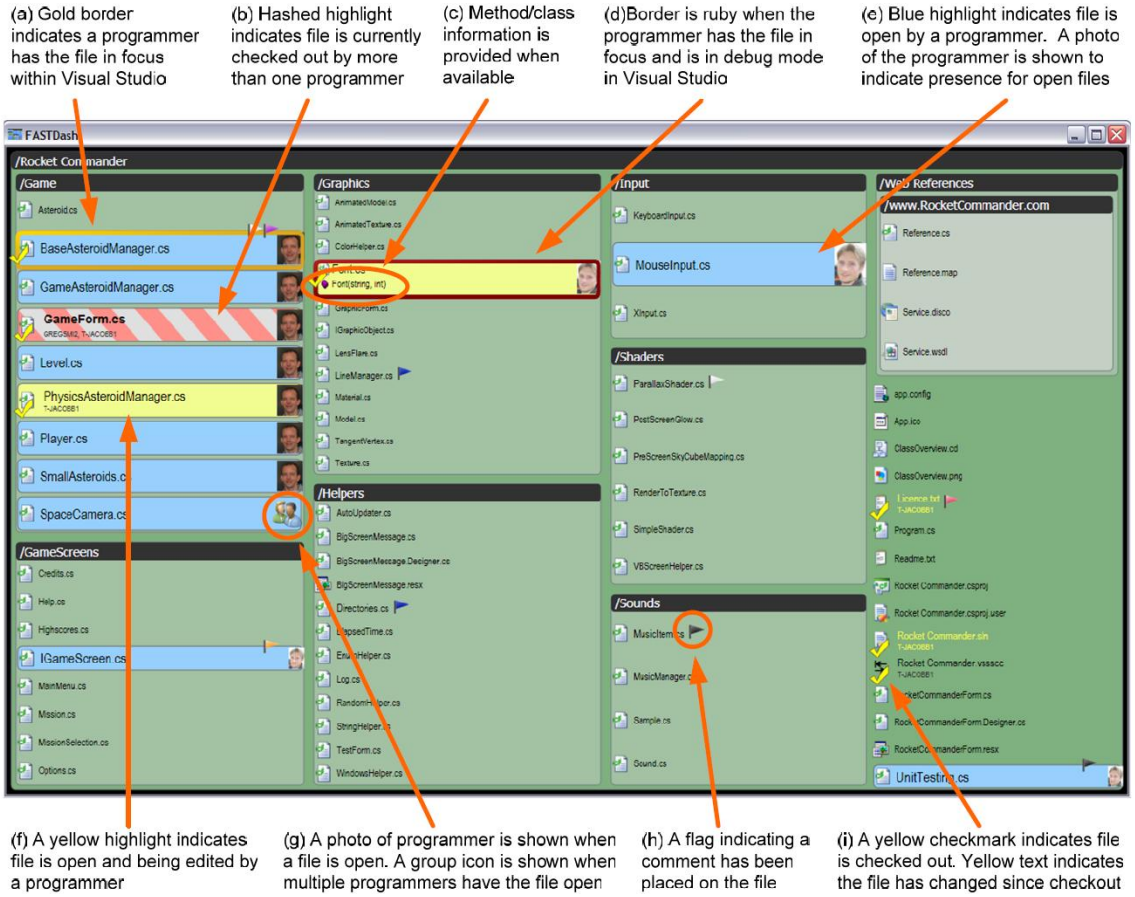

Figure 11—The dashboard interface shows active files grouped by module and activities performed on them by active developers.

Content: FASTDash displays a software project's files organized by module and graphical annotations to the files corresponding to activities being performed on them by developers, such as making changes, viewing the file, debugging it, adding documentation, etc. The interface alerts the user to who is working on what but gives no evidence of whether that work will affect the former's own work. The user must infer that themselves.

Mapping Content to Tasks: The tool supports developer understanding of activities being performed by their peers on the code base. Potential check-in conflicts can be inferred by monitoring changes to the same file by multiple developers. Developers working with the same sets of files can initiate conversation and coordinate implementation work on those files. Such activities are characteristic of the implementation and even maintenance phases of software development.

Interpretive Gaps: The tool shows such low-level actions (e.g. a file being in focus, a file being viewed) that it is difficult to second-guess the accuracy of information being presented. However, from these scattered observations alone it is hard to paint a picture of the tasks actually being accomplished because no details are provided (e.g. through tool tips). FASTDash does not provide guidance on how to organize and combine observations described in the interface over time to reveal broad detail of what individuals are working on and the sequence they have followed. It was not tested

between teams so there is no evidence for the scalability of its usage and how multiple items of information might compete to reveal insights into meaningful developer activity.
Temporal Unit of Analysis The tool displays real time ongoing changes but no history of those changes over time. It is only possible to know the most recent activity. Thus it is difficult to get a sense of others' patterns of work over time but it is easy to initiate conversation around source-code that is currently being modified. The awareness gained from knowing who is working on what at any point relieves developers of having to explain the context of their dependent work to one another as they coordinate to complete tasks.
Visual Representations: FASTDash uses a combination of graphics and text, but mostly graphics, to show awareness information. Unlike other visualizations such as Palantír, Team Tracks, or Hipikat, FASTDash exists as a standalone application. It uses familiar file icons for resources in the workspace (e.g. photos, .source-code files, authors). However a problematic aspect of the interface design is that it uses inconsistent forms of visual cues to indicate the different states of files: hashed highlighting to indicate files checked out by multiple developers and thus potential sources of conflict, yellow highlighting to indicate files that are open and being edited, gold borders to indicate files that are in focus in the editor, and checkmarks superimposed on files to indicate files that are checked out. The variety in different graphical annotations for related states of activity makes it difficult to remember the meaning of each. This is an interface that will require time to learn before it can be used efficiently by teams, yet this observation was omitted from the authors' field study of the tool.

Jazz

Jazz is a collaborative development environment (Booch and Brown 2003) that enhances the existing Eclipse platform with collaboration mechanisms for use in small-team settings (Hupfer et al. 2004). It extracts activities from user interactions with the interface and the local history of source-code and work items (i.e. reports that detail work to be accomplished) to 1) monitor what and how source-code is being changed and 2) push this information to other developers in the workspace who have subscribed to be notified of these ongoing changes. The Jazz Band (bottom of Figure 12) is a shared buddy-list from which users can initiate interactions with others (e.g. screen sharing, chat sessions) without the overhead of leaving the IDE and launching other applications.

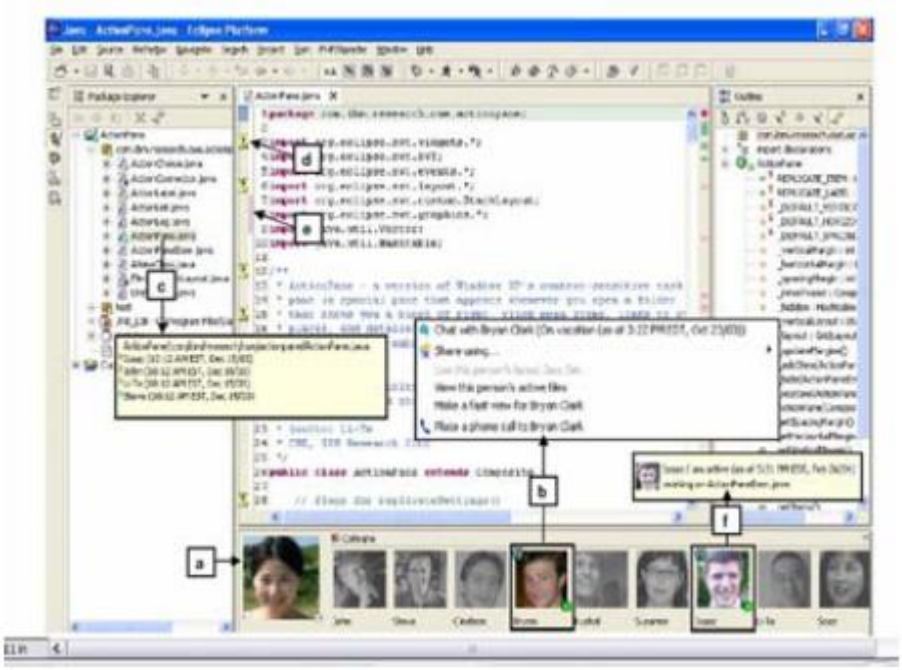

Figure 12—The Jazz environment showing a) team members, b) communication options for interacting with a team member, c) workspace files and resources annotating their current states and who is changing them, d) an anchor for a chat transcript pertaining to the opened code, e) a recently modified portion of code, and f) a team member's status/location.

Content: Jazz displays ongoing activities performed by team members within the development environment, including the changes they make to source-code, comments and documentation relevant to the code, and focused group discussion. By structuring activities around team mates, Jazz enhances developer awareness of others' contextually relevant interactions with the code and each other. When a developer selects a snippet of source-code, for example, and initiates a chat everyone else immediately knows broadly what will be discussed. No one needs to direct them to another area in the code. Jazz can also be used to gauge developer availability through the use of Jazz Band buddy list status messages and tooltips in the hierarchical source-code window describing who is changing what files.

Mapping Content to Tasks: Jazz enhances Eclipse's standard source-code editing capabilities by providing mechanisms for tracking development activities directly in the user-interface (e.g. embedded instant messaging, group chat, comments, and milestone progress indicators). As such, it eliminates much of the overhead involved in setting up communication mechanisms (e.g. chat and e-mail servers) and bringing people's perspectives into alignment in order to engage in collaborative tasks. The authors describe use of the tool in the implementation phase, but Jazz appears to be useful in maintenance phases as well, such as when developers use work items and other developers as resources for changing bugs and adding features.

Interpretive Gaps: Jazz thoroughly annotates the development environment with notifications corresponding to developer activity, prompting developers to initiate collaborative tasks. The notifications persist, allowing developers to visit them at times they deem suitable and relevant. Any ambiguity in updates or annotations can be

resolved by using the communication features embedded into the editor to check for the responsible developer's availability and subsequently contact them.

Temporal Unit of Analysis Jazz allows real time monitoring of activity, but provides no time ordered views of actions and collaboration corresponding to resources in the environment. As such, it is difficult to identify patterns of work and availability (see Awarenex) in order to improve team practices and productivity.

Visual Representations: Like Palantír, Team Tracks, and Hipikat, Jazz visualizes awareness data in the context of current development activities unraveling in the development environment. FASTDash and Jazz are similar in their goals and motivations yet FASTDash uses novel visualizations in place of visualization that have the same "look and feel" as the environment. Jazz, on the other hand, makes use of easily identifiable visual cues and annotations to source-code in the editor (as in Palantír) to signify the status of files, chat logs around the code, and documentation. The biggest enhancement to the existing editor is the Jazz Band (Figure 12a), which features avatars of team members augmented with mechanisms for *different forms* communication. The Jazz Band gives users a choice for what communication medium to use (e.g. chat, phone (VOIP), or e-mail). Providing different options is critical—yet not addressed in the majority of the tools surveyed in this paper—since it has been shown that different users prefer to be contacted different ways (Herbsleb and Grinter 1999). For example, non-native speakers prefer e-mail because it gives them time to compose their thoughts and responses. Responding in real time can become time consuming and frustrating.

Discussion

From the detailed analysis of visualization tools that support awareness in software development, numerous observations follow. They are presented below according to the principles of awareness identified previously in this survey and suggest possible areas for future research.

Content: Need for common data models. Current visualizations for awareness during software development extract information from a common set of repositories, including version control systems, issue trackers, and data from developer workspaces. More rarely, they use input from the keyboard to detect real time activities, and developer availability and location. Extracted data from the CM systems include the revisions themselves, their structural properties and relationships with other module revisions, the changes that were made in each revision, and by whom the changes were made. The acquisition of workspace data requires more detail to link individuals to source-code they are working on at that moment as well as what they have opened in the background. Despite the commonalities of the types of data collected by these tools, to the author's knowledge, no common data model or collections of APIs exist to provide a standardized way to extract awareness data at different details and levels of abstraction. Researching a common data model would promote more detailed inspections of these tools and eventually lead to the development of flexible, reusable components for extracting awareness information.

Need to support individual preference. Some of the tools that use real time data can prompt immediate communication opportunities which are more difficult in distributed software development settings. Yet aspects of communication preferences, the conditions under which people choose to make themselves responsive (Illich 1971), were only addressed in one tool, Jazz. Despite being flagged as available in tools like Awarenex, TUKAN, softCHANGE, FASTDash, ELVIN and Community Bar, people may have numerous reasons for choosing to make themselves available to communicate. People have different preferences for the ways in which they would like to be contacted (Herbsleb and Grinter 1999; Nardi et al. 2002). For example, some people prefer the persistency of e-mail since it can act as a receipt or documentation that can be referred to later. Others prefer e-mail because it gives them time to coherently construct their thoughts. Some tools like ELVIN and FASTDash facilitate opportunistic real time chat interactions yet do not offer the asynchronous support necessary for interactions across time zones that e-mail does. In order to appeal to a more diverse range of users, these tools should support multiple communication channels such as IM, e-mail, video, and chat.

Need to leverage personal relationships and history of helping. Tools that identify who to talk to, such as Expertise Recommender, and who is actually available like Awarenex and Community Bar rely on the receptiveness of the helper to be thoroughly effective. Yet helping comes at the cost of being interrupted, bringing a temporary halt to the flow of work. One way to determine one's receptiveness toward helping might be to build in a "helping history" to tools that can be used to prompt developer-developer interactions. The visualizations could be augmented with developers' history of and preferences for helping each other. For instance, developers who have a good rapport and history of sharing expertise are more likely to help one another than two developers who never see each other yet need to coordinate despite being on the same team or on different teams working on interdependent system components. Additionally, developers are more likely to respond to their superiors than their peers since they could be reprimanded for not doing so. For a recommender that relies on previous interactions to be useful, however, developers would have to initially provide a critical mass of interaction history. One way would be to have a profile filled out by each developer that showcases their skills and expertise. Every other developer might have a version of that profile on their client application and could rate developers based on their past personal interactions with them. Given this initial data as a baseline, communications could be logged by the tool and suggestions made based on that interaction history.

Mapping Content to Tasks: Need to map features to tasks. Despite the utility of supporting general awareness, of the "Change Management and Evolution" tools surveyed, very few mapped tasks representative of "change management" to features in the tools themselves. Of the 14 change management tools surveyed, only one (Wu et al. 2004b) defined and showed specific change activity for which the tool could be used to monitor: "Punctuation," moments of sudden and discontinuous change that could be used to spot buggy/inefficient code that needs refactoring. For example, Codesaw was built with the intentions of "revealing group dynamics around changes," but this is a broad statement and reveals no insight into how the tool would be used to do so. In addition, no specific group dynamics relevant to software development are defined by the authors.

Command Console helped project managers reveal "where the action is" but that too is a high-level observation. Writing code may constitute "action," but so may writing documentation or communicating with developers responsible for part of the source-code. These are "invisible" yet crucial aspects of work that often go undetected and thus unrewarded by managers. Specific patterns of action and what they mean for project managers and developers are unclear.

Insights that emerge from tools like CodeSaw, including VRCS and softCHANGE do so more by individual user experience and intuition rather than from a process or defined method of use. In general, all of the surveyed tools identified awareness data that *could* be used to support tasks. This is an important first step for very few change management tools were evaluated with respect to specific tasks. As researchers improve their tools, their focus should not remain on "what" information can be revealed but instead turn to *how the tool can be used to reveal information central to completing expected activities* defined in software development projects—in other words, the process of using the tool itself to reveal useful information.

One interesting observation is that *availability* is the least supported, yet development projects are increasingly becoming globally distributed. This suggests that new technologies that promote awareness should include mechanisms that facilitate opportunistic interaction and focus less on analysis of past work, relatively speaking.

Three tools, Community Bar, Awarenex, and Active Map support finding people across the whole software development process. One tool, Tarantula, (Jones et al. 2001; Jones and Harrold 2005) addresses fault localization, an activity characteristic of the testing phase of software development (see Figure 13). That is not to say that awareness visualizations do not exist for other phases of development such as gathering requirements or creating high-level architectures.

Identifying the tasks and phases of development for which tools are appropriate is an important step in describing their intended usage and considering their deployment. Of the awareness visualizations for software development surveyed, most of them support tasks in the maintenance phase; the rest support activities representative of the implementation, phase.

One explanation for the gap across development phases might be that these tools resulted from a selection bias in the software engineering literature surveyed. To the contrary, the most integral venues in the field were selected and searched with respect to *awareness needs* identified in the very same venues. Few tools were found to address awareness in the context of testing, requirements, and design because requirements for awareness in these phases have not been defined or described greatly in detail in the software engineering literature. Thus, it is likely that if there is a selection bias, it is in the interests of researchers and the peer review processes adopted by the organizers of these venues, not this survey.

A more intuitive interpretation is that most artifacts are generated in the implementation phase(s) of software development, and thus, more tools are developed to provide support for activities that fall into those phases. One of the benefits of a well-designed visualization is the ability to see relationships in complex and large streams of input data (Bertin 1982). The artifacts stored in team repositories can themselves be seen as input to these tools. Artifacts—especially source-code, since it is the ultimate representation of the system--accrue quickly and in parallel (the very reason why conflict

detection mechanisms are developed) after the work is divided up. Visualizations are also used to *compare* activities to expected activities, implementation to design; design to requirements, and so on. For the most part, source-code implementation is strictly a human behavior, and humans make mistakes. Take bugs for example—they are deviations in the expected behavior of the implemented system, linked to the design, and back to the requirements. These deviations in expectations are what programmers and managers care about in the real world because the required work to "correct" the deviations costs time and money. This may help explain why the locus of study has been the implementation and maintenance phases of software development.

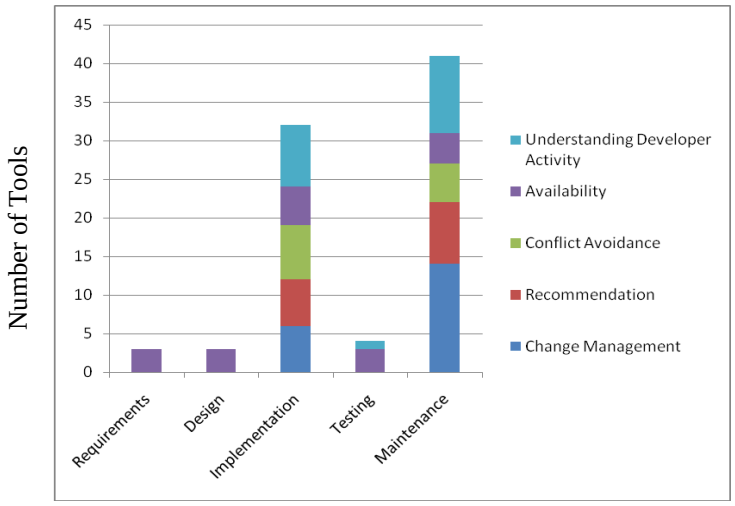

Figure 13—Number of tools surveyed in each phase of software development broken down (i.e. color coded) by category as identified in Table 3. Some tools appear in more than one category.

Interpretive Gaps: Need for multivariate and transparent evidence of rationale. When applicable, tools that give a recommendation or suggest the potential for coordination breakdowns should provide multiple forms of evidence to assist the user in making a correct interpretation of the content and an actionable decision. Interpretive gaps showed up the most often in the "Recommending People and Artifacts" and "Avoiding Conflicts" categories. Systems like Hipikat that give a confidence rating from 0-1, for example, are of little use to users because no rationale for the rating is provided. Similarly, systems like Expert Recommender simply list individuals' contact information without providing transparency into the history of their expertise.

Yet Palantír and Team Tracks, for example, helped with interpretation by providing multiple forms of evidence: Palantír uses relative size of graphics and colors to indicate size and types of conflicts, left-and-right arrows to indicate local or remote code affected by changes, and percentages to indicate the amount of code affected. While percentages are relative to the code size, users can see a visualization of the size of affected code to help them determine the severity of impacts. Along the same lines, although team navigation patterns are not an indicator of code relatedness, Team Tracks uses a combination of cues to strengthen the confidence of its recommendations: rank-ordered lists of related source-code combined with graphics depicting relative relatedness of the items and icons showing dependency relationships between the suggested items and the source-code in focus. In the evaluations of both these systems, the rationale was shown to assist users in taking appropriate action: avoiding conflicts and understanding the code well enough to implement change requests, respectively.

Thus Hipikat, and similar systems, might be improved by adding additional units of rationale to support the end-user process of interpretation. One approach might be to also show the percentage of people who also used artifacts based on similar tasks or the percentage of people who found a similar recommendation to be less useful than another one. The recommendation might group the people by team or assigned components so that the user can put their work in the perspective of others' and make better-informed

decisions (Cartwright et al. 2002). When using an interface is not part of the normal flow of work as in Expertise Recommender for example, details from the original work may be lost in the recommendation process. A developer who wishes to know about why a certain implementation was chosen over another for example might wish to include detailed comparisons and running times of different algorithms. Yet if the recommended individual worked on significant portions of that code (the reason for the recommendation) but has since moved on to other projects, it is unlikely they will have remembered much and time that could have been spent addressing the problem will need to be spent bringing the helper up to speed.

Temporal Unit of Analysis: Need to support understanding dependencies over time. In the earlier sections of this survey it was made clear that, dependencies, or relationships between developers and the artifacts with which they work, change over time. Designing clean APIs is one solution to managing dependencies between artifacts but not between people. One way to show how developer and artifact dependencies change over time is by quantifying them. Tools like Palantír and TUKAN make it possible to associate authors and artifacts at any one point in time for the purpose of coordinating and resolving potential conflicts, yet they fail to quantify associations when the dependencies are not explicitly determined at the time of conflict. The metrics used to calculate socio-technical congruence (Cataldo et al. 2006) and dependencies (Carley and Krackhardt 1998) could be used to provide additional information, like a weighted measure to dependencies between artifacts and people that might be relevant to a particular task at any time. For example, a dynamic buddy list augmented with the buddy list in Jazz could update every time a specific part of the code (e.g. defined by the user) is changed. This buddy list would act as an "awareness network" (de Souza et al. 2007). In this way, the developer could more easily track how their network evolves without the use of a large and distracting separate visualization application.

Of the tools surveyed here, especially the change management and evolution tools, no tools except Workspace Activity Viewer (see Table 3) used animation to show changes to a software system over time. Similarly, in their survey (Storey et al. 2005), Storey and colleagues found that no tool used animation to play back activity. Some obvious uses of animation are to show flurries and stagnation in activity as well as continuous changes in authorship and artifacts (can signal potential unstable and buggy source-code (Wu et al. 2004b)). This information could help managers identify when task assignments are unclear or when parallel work (and thus a possible conflict) occurs.

Visual Representations: Map visualizations to the tasks they should support. In Jazz, Team Tracks, and Palantír, all awareness visualizations are integrated with the existing editor. Current resources in the environment are annotated with graphics that show status as it pertains to a specific activity, such as conflict avoidance (e.g. shown with colored arrows) and communication (e.g. clicking on an individual's avatar to launch a chat dialog). They enhance standard features in a manageable way that does not involve a lot of context switching. On the other hand, FASTDash, described to support the same kind of tasks (i.e. conflict avoidance) is a stand-alone application that requires use outside of the editor. The unique visualizations it provides are not based on any pre-existing standards and there is no clear way to orient or relate the activities of others to one's own work. Conflicts emerge as developers are focused on writing code. As such, warnings that appear should do so where developers are likely to see them (i.e. in the code), not on

a separate screen or wall. Doing so puts an unnecessary burden on the developer, requiring them to periodically interrupt themselves and work less efficiently as a result.

Use scalable representations that can be used to identify patterns. Despite the context-switching issues associated with using large amounts of screen real-estate to convey awareness information, such techniques better convey patterns *over time* using visualization techniques, novel or otherwise, that scale, such as those implemented in Awarenex and Command Console. By their very nature, these tools visualize massive amounts of data in order to identify meaningful patterns of activity. As such, they utilize significant portions of the developer's workspace and can be a distraction. As a side effect however, patterns such as who modified what, what parts of code are affected, and who is available at what times of the day become easily recognizable. These patterns are seen in the work rhythms of individuals displayed by Awarenex and the complexity thumbprint visualizations in Command Console. They increase the extent to which one can predict future activity and thus make better-informed decisions as a result.

Organizational Strategies

While they are certainly an important part of enhancing coordination, tools are not the only strategies for promoting awareness. The guidelines proposed above serve to improve the tools' abilities to augment awareness and therefore enhance coordination activities during software development. Yet using tools is but one part of a larger strategy aimed at increasing the awareness needed to improve the ability to coordinate, or to achieve what some might call an ideal state of coordination, or "congruence." Congruence is defined by Cataldo et al. (Cataldo et al. 2006) as "a state in which an organization has sufficiently aligned their coordination capabilities to meet the coordination demands of the technical products under its development:" in other words, what the organization *can* do versus what it *needs* to do. Crucially, congruence implies not only the use of technology but also organizational strategies like restructuring teams, providing training, and even allocating conflict resolution engineers and liaisons (McCord et al. 1993) to resolve technical issues and provide continuous, intensive information exchange between and among teams.

Assessing an organization's compliance with congruence assumes its capability to measure or quantify actual and corresponding required levels of coordination. Sarma and colleagues (Sarma et al. 2008a) outline the objectives of measuring congruence: 1) knowing what information to collect, 2) finding the best approach to collecting this information, directly or indirectly, and 3) understanding appropriate ways of computing congruence measures. The tools surveyed here collect a variety of information in order to gauge collaboration and coordination activities, among them: direct communication from e-mails and chat logs, indirect collaboration from change logs, and artifacts of the system under development itself including source-code and bug reports. They obtain important historical information by mining repositories for explicit and derived (e.g. finding experts) data as well as contextually relevant real time data through instrumentation of project workspaces (e.g. to detect conflicts). However, less is known about how congruence can be measured and thus recognized in practice, especially since there are so many ways to coordinate work. Sarma and colleagues hypothesize that innovative visualizations that summarize vast amounts of information can facilitate the

emergence of distillable and recognizable patterns that researchers and practitioners can use to create benchmarks for assessing congruence. This observation is in line with the conclusions of this survey, yet more research remains to be conducted.

Although the rhetoric of congruence is coordination-centric, the absence of collaboration as reported by tools does not necessarily imply an undesirable state of congruence *if no coordination is required*. Thus, project managers or even other developers should not be automatically alarmed or surprised if no data is being reported by tools or no activity is being reported in visualizations. Violations in expectations of development activity based solely on information reported in tools may result in unjustified, negative performance evaluations of personnel. Moreover, developers may well regard the transparency revealed by the tools as a violation of their privacy. These situations highlight that tools, while crucial, are not an awareness panacea; they are situated within an organizational context of expectations, norms, and both horizontal and vertical dimensions of authority.

Conclusion

The objective of this survey was to: 1) motivate the need for tools that promote awareness in the face of the rise in global software development projects and associated coordination challenges, 2) identify important principles and requirements for awareness in the software engineering literature, and 3) to identify and compare existing awareness tools with respect to the identified principles.

The contributions of this work are four-fold. First, this survey extrapolates important principles of awareness based on existing empirical work yet left out in the implementation of current tools designed to promote awareness. Second, it crucially identifies a considerable subset of these tools that are representative of those used in academic and professional settings. Third, this work provides a novel categorization of the tools which acts as a basis from which to analyze them with respect to the awareness principles. Fourth, based on this analysis, it shows how the visualizations might be improved to support various aspects of awareness. The categorization proposed here is not meant to constitute a normative framework to which all other tools should be compared. Moreover, it has not been evaluated and used by other researchers. Instead it raises several important questions to ask about them. The analysis builds on established research and empirical findings from the awareness, coordination and tool research communities—rather than considering any one in isolation—to provide a structured and holistic approach toward understanding the similarities and differences in, as well as the advantages and limitations of, current tool support. As such it serves as a starting point for researchers also interested in the intersections between these communities and implications for coordination technologies.

Acknowledgements

This research is supported by the U.S. National Science Foundation under grants 0534775 and 0205724, and by an IBM Eclipse Technology Exchange grant. The author

gratefully acknowledges past and present collaboration with Cleidson de Souza and feedback from members of the CRADL research group.

References

Ackerman, M. S. and Malone, T. W. (1990): 'Answer Garden: a tool for growing organizational memory', *ACM SIGOIS and IEEE CS TC-OA Conference on Office Information Systems*, April 1990, pp. 31-39.

Ackerman, M. S. and McDonald, D. W. (1996): 'Answer Garden 2: merging organizational memory with collaborative help', *ACM Conference on Computer Supported Cooperative Work (CSCW)*, November 1996, pp. 97-105.

Amar, R. and Stasko, J. (2004): 'A Knowledge Task-Based Framework for Design and Evaluation of Information Visualizations', *IEEE Symposium on Information Visualization (INFOVIS)*, October 2004, pp. 143-150.

Amrit, C. (2008): 'Improving Coordination in Software Development through Social and Technical Network Analysis', University of Twente, Enschede, The Netherlands , Ph.D. dissertation.

Avritzer, A., Paulish, D., and Cai, Y. (2008): 'Coordination Implications of Software Architecture in a Global Software Development Projects', *IEEE/IFIP Conference on Software Architecture*, February 2008, pp. 107-116.

Ball, T. and Eick, S. G. (1996): 'Software Visualization in the Large', *Computer*, April 1996, pp. 33-43.

Begole, J., Tang, J. C., Smith, R. B., and Yankelovich, N. (2002): 'Work rhythms: analyzing visualizations of awareness histories of distributed groups', *ACM Conference on Computer Supported Cooperative Work*, November 2002, pp. 334-343.

Bertin, J. (1982): *Graphics and Graphic Information-Processing*, Walter de Gruyter, Berlin, Germany.

Biehl, J. T., Czerwinski, M., Smith, G., and Robertson, G. G. (2007): 'FASTDash: a visual dashboard for fostering awareness in software teams', *SIGCHI Conference on Human Factors in Computing Systems*, April 2007, pp. 1313-1322.

Booch, G. and Brown, A. W. (2003): 'Collaborative Development Environments', *Advances in Computers*, , vol. 59, 2003, pp. 2-29.

Brooks, F. P. (1995): *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Boston, MA.

Card, S., Mackinlay, J. and Shneiderman, B. (1999): *Readings in Information Visualization: Using Vision to Think*, Morgan Kaufmann, San Francisco, CA.

Carley, K. and Krackhardt, D. (1998): 'A PCANS Model of Structure in Organizations', *Symposium on Command and Control Research and Technology*, June 1998, pp. 113-119.

Carmel, E. (1999): *Global Software Teams*, Prentice-Hall, Upper Saddle River, NJ.

Cartwright, W., Crampton, J., Gartner, G., Miller, S., Mitchell, K., Siekierska, E. and Wood, J. (2001): 'Geospatial Information Visualization User Interface Issues', *Special Issue on Research Challenges in Geovisualization, Cartography and Geographic Information Science*, vol. 28, no. 1, January 2001.

Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., and Carley, K. M. (2006): 'Identification of coordination requirements: implications for the Design of collaboration and awareness tools', *ACM Conference on Computer Supported Cooperative Work (CSCW)*, November 2006, pp. 353-362.

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*, June 2003, pp. 77-86.

Conradi, R. and Westfechtel, B. (1998): 'Version models for software configuration management', *ACM Computing Surveys*, June 1998, pp. 232-282.

Cubranic, D. and Murphy, G. C. (2003): 'Hipikat: recommending pertinent software development artifacts', *International Conference on Software Engineering (ICSE)*, May 2003, pp. 408-418.

Curtis, B., Krasner, H., and Iscoe, N. (1988): 'A Field Study of the Software Design Process for Large Systems', *CACM*, vol. 31, no. 11, November 1988, pp. 1268-1287.

D'Ambros, M., Lanza, M., and Gall, H. (2005): 'Fractal Figures: Visualizing Development Effort for CVS Entities', *IEEE international Workshop on Visualizing Software For Understanding and Analysis*, September 2005, pp 16.

DeLine, R., Czerwinski, M., and Robertson, G. (2005): 'Easing Program Comprehension by Sharing Navigation Data', *IEEE Symposium on Visual Languages and Human-Centric Computing*, September 2005, pp. 241-248.

de Souza, C. R., Redmiles, D., Cheng, L., Millen, D., and Patterson, J. (2004): 'How a good software practice thwarts collaboration: the multiple roles of APIs in software development', *ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 2004, pp. 221-230.

de Souza, C. R., Quirk, S., Trainer, E., and Redmiles, D. F. (2007): 'Supporting collaborative software development through the visualization of socio-technical dependencies', *ACM Conference on Supporting Group Work*, November 2007, pp. 147-156.

de Souza, Redmiles, D.F. (2007): 'The awareness network: To whom should I display my actions? And, whose actions should I monitor?', *European Conference on Computer-Supported Cooperative Work (ECSCW)*, September 2007, pp. 99-118.

Dewan, P. and R. Hegde. (2007): 'Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development', *European Conference on Computer-Supported Cooperative Work (ECSCW)*, September 2007, pp. 159-178.

Dourish, P. and Bellotti, V. (1992): 'Awareness and coordination in shared workspaces', *ACM Conference on Computer-Supported Cooperative Work (CSCW)*, November 1992, pp. 107-114.

Ehrlich, K. and Chang, K. (2006): 'Leveraging expertise in global software teams: Going outside boundaries', *IEEE International Conference on Global Software Engineering*, October 2006, pp. 149-158.

Eick, S. G., Steffen, J. L., and Sumner, E. E. (1992): 'Seesoft-A Tool for Visualizing Line Oriented Software Statistics', *IEEE Transactions on Software Engineering*, vol. 18, no. 11, November 1992, pp. 957-968.

Eick, S. G., Graves, T. L., Karr, A. F., Mockus, A., and Schuster, P. (2002): 'Visualizing Software Changes', *IEEE Transactions on Software Engineering*, vol. 28, no. 4, April 2002, pp. 396-412.

Ellis, J. B., Wahid, S., Danis, C., and Kellogg, W. A. (2007): 'Task and social visualization in software development: evaluation of a prototype', *ACM SIGCHI Conference on Human Factors in Computing Systems*, April-May 2007, pp. 577-586.

Fischer, G. (2001): 'User Modeling in Human–Computer Interaction', *User Modeling and User-Adapted Interaction*, vol. 11, no. 1-2, March 2001, pp. 65-86.

Fitzpatrick, G., Marshall, P., and Phillips, A. (2006): 'CVS integration with notification and chat: lightweight software team collaboration', *ACM Conference on Computer Supported Cooperative Work (CSCW)*, November 2006, pp. 49-58.

Froehlich, J. and Dourish, P. (2004): 'Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams'. *International Conference on Software Engineering (ICSE)*, May 2004, pp. 387-396.

Fuggetta, A. (2000): 'Software process: a roadmap', *Conference on the Future of Software Engineering*, June 2000, pp. 25-34.

German, D., Hindle, A. , and Jordan N. (2004): 'Visualizing the evolution of software using softChange', *International Conference on Software Engineering and Knowledge Engineering*, 2004, pp. 336–341.

Gilbert, E. and Karahalios, K. (2007): 'CodeSaw: A Social Visualization of Distributed Software Development', *INTERACT*, 2007, pp. 303-316.

Gutwin, C. and Greenberg, S. (1999): 'The effects of workspace awareness support on the usability of real time distributed groupware', *ACM Transactions on Computer-Human Interaction*, vol. 6, no. 3, September 1999, pp. 243-281.

Gutwin, C., Penner, R., and Schneider, K. (2004): 'Group awareness in distributed software development', *ACM Conference on Computer Supported Cooperative Work (CSCW)*, November 2004, pp. 72-81.

Halverson, C. A., Ellis, J. B., Danis, C., and Kellogg, W. A. (2006): 'Designing task visualizations to support the coordination of work in software development', *ACM Conference on Computer Supported Cooperative Work*, November 2006, pp. 39-48.

Hancock, M. S., Miller, J. D., Greenberg, S., and Carpendale, S. (2006): 'Exploring visual feedback of change conflict in a distributed 3D environment', *Working Conference on Advanced Visual Interfaces*, May 2006, pp. 209-216.

Handel, M. and Herbsleb, J. D. (2002): 'What is chat doing in the workplace?', *ACM Conference on Computer Supported Cooperative Work*, November 2002, pp. 1-10.

Herbsleb, J. D. and Grinter, R. E. (1999): 'Splitting the organization and integrating the code: Conway's law revisited', *International Conference on Software Engineering (ICSE)*, May 1999, pp. 85-95.

Herlocker, J.L., Konstan, J.A., and Riedl, J. (2000): 'Explaining collaborative filtering recommendations', *ACM Conference on Computer-Supported Cooperative Work (CSCW)*, December 2000, pp. 241-250.

Herman, I., Melancon, G., Marshall, M.S. (2000): 'Graph visualization and navigation in information visualization: A survey', *IEE Transactions on Visualization and Computer Graphics*, vol. 6, no. 1, January-March 2000, pp. 24-43.

Hupfer, S., Cheng, L., Ross, S., and Patterson, J. (2004): 'Introducing collaboration into an application development environment', *ACM Conference on Computer Supported Cooperative Work*, November 2004, pp. 21-24.

Illich, I. (1971): *Deschooling Society*, Harper and Row, New York, NY.

Jang, C.Y., Steinfield, C. and Pfaff, B. (2000): 'Supporting awareness in a web-based collaborative system: The Team-SCOPE System', *Workshop on Awareness and the WWW, ACM Conference on Computer Supported Cooperative Work* (CSCW '00), December 2000.

Jones, J. A., Harrold, M. J., and Stasko, J. T. (2001): 'Visualization for Fault Localization', *Workshop on Software Visualization, International Conference on Software Engineering (ICSE)*, May 2001, pp. 71-75.

Jones, J. A. and Harrold, M. J. (2005): 'Empirical evaluation of the tarantula automatic fault-localization technique', *IEEE/ACM international Conference on Automated Software Engineering*, November 2005, pp. 273-282.

Kersten, M. and Murphy, G. C. (2006): 'Using task context to improve programmer productivity', *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, November 2006, pp 1-11.

Ko, A. J., DeLine, R., and Venolia, G. (2007): 'Information Needs in Collocated Software Development Teams', *International Conference on Software Engineering (ICSE)*, May 2007, pp. 344-353.

Koike, H., and Chu, H.-C. (1997): 'VRCS: Integrating version control and module management using interactive three- dimensional graphics', *IEEE Conference on Visual Languages*, September 1997, pp. 170-175.

Lanza, M. (2001): 'The Evolution Matrix: recovering software evolution using software visualization techniques', *Workshop on Principles of Software Evolution, International Conference on Software Engineering (ICSE)*, May 2001, pp. 37–42.

Lanza, M. (2003): 'CodeCrawler - Lessons Learned in Building a Software Visualization Tool', *European Conference on Software Maintenance and Reengineering*, March 2003, pp. 409.

Lanza, M., Ducasse, S., Gall, H., and Pinzger, M. (2005): 'CodeCrawler: an information visualization tool for program comprehension', *International Conference on Software Engineering (ICSE)*, May 2005, pp. 672-673.

LaToza, T. D., Venolia, G., and DeLine, R. (2006): 'Maintaining mental models: a study of developer work habits', *International Conference on Software Engineering (ICSE)*, May 2006, pp. 492-501.

Malone, T. W. and Crowston, K. (1994): 'The interdisciplinary study of coordination', *ACM Computing Surveys*, vol. 26, no. 1, March 1994, pp. 87-119.

McCabe, T. J. (1976): 'A complexity measure', *IEEE Transactions on Software Engineering*, vol. 2, no. 4, December 1976, pp. 308-320.

McCarthy, JF. (1999): 'ACTIVE MAP: A visualization tool for location awareness to support informal interactions', *Lecture notes in computer science, International Symposium on Handheld and Ubiquitous Computing* , 1999, pp. 158-170.

McCord, Kent R. (1993): 'Managing the integration problem in concurrent engineering', Massachusetts Institute of Technology, USA, M.S. thesis.

McDonald, D. W. and Ackerman, M. S. (2000): 'Expertise recommender: a flexible recommendation system and architecture', *ACM Conference on Computer Supported Cooperative Work (CSCW)*, December 2000, pp. 231-240.

Mockus, A. and Herbsleb, J. D. (2002): 'Expertise browser: a quantitative approach to identifying expertise', *International Conference on Software Engineering (ICSE)*, May 2002, pp. 503-512.

Molli, P., Skaf-Molli, H., Bouthier, C. (2001): 'State Treemap: an awareness widget for multi-synchronous groupware', *International Workshop on Groupware*, September 2001, pp. 106-114.

Morelli, M. D., Eppinger, S.D., and Gulati, R. K. (1995): 'Predicting technical communication in product development organizations', Working papers 3602-95, February 1995.

Nardi, B. A., Whittaker, S., and Schwarz, H. (2002): 'NetWORKers and their Activity in Intensional Networks', *Computer-Supported Cooperative Work*, vol. 11, no. 1-2, April 2002, pp. 205-242.

Norman, D. A. (1986): 'Cognitive Engineering', in D. Norman and S. Draper (eds.): *User Centered System Design: New Perspectives on Human–Computer Interaction,* Lawrence Erlbaum and Associates, Hillsdale, NJ, 1986, pp. 31-61.

Nutt, G. J. (1996): 'The evolution toward flexible workflow systems', *Distributed Systems Engineering*, vol. 3, December 1996, pp. 276-294.

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*, May 2005, pp. 57-65.

Olson, G.M. and Olson, J.S. (2000): 'Distance Matters', *Human- Computer Interaction*, vol. 15, no. 2-3, 2000, pp. 139-178.

Parnas, D.L. (1972): 'On the Criteria to be Used in Decomposing Systems into Modules', *Communications of the ACM*, vol. 15, no. 12, December 1972, pp. 1053-1058.

Perry, D.E. and Wolf, A.L. (1992): 'Foundations for the study of software architecture', *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, October 1992, pp. 40-52.

Perry, D.E., Staudenmaye, N.A., and Votta, L.G. (1994): 'People, Organizations, and Process Improvement', *IEEE Software*, vol. 11, no. 4, July 1994, pp. 36-45.

Pinzger, M., Gall, H., Fischer, M., and Lanza, M. (2005): 'Visualizing multiple evolution metrics', *ACM Symposium on Software Visualization*, May 2005, pp. 67-75.

Ripley, R.M.; Sarma, A.; van der Hoek, A. (2007): 'A Visualization for Software Project Awareness and Evolution', *International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT*, June 2007, pp. 137-144.

Sangwan, R., et al. (2006): *Global Software Development Handbook*, Auerbach Publications, Boca Raton, FL.

Sarma, A., Noroozi, Z., and van der Hoek, A. (2003): 'Palantír: raising awareness among configuration management workspaces', *International Conference on Software Engineering (ICSE)*, May 2003, pp. 444–454.

Sarma, A., Herbsleb, J. and van der Hoek, A. (2008a): 'Challenges in Measuring, Understanding, and Achieving Social-Technical Congruence', *Technical Report CMU-ISR-08-106, Carnegie Mellon University, Institute for Software Research International*, April 2008.

Sarma, A., Redmiles, D.F., and van der Hoek, A. (2008b): 'Empirical evidence of the benefits of workspace awareness in software configuration management' *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, November 2008, pp. 113-123.

Schummer, T., and Haake, J.M. (2001): 'Supporting distributed software development by modes of collaboration', *European Conference on Computer Supported Cooperative Work (ECSCW)*, September 2001, pp. 79–98.

Shaw, M. and Garlan, D. (1996): *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Upper Saddle River, NJ.

Shneiderman, B. (2002): 'Inventing discovery tools: combining information visualization with data mining', *Information Visualization*, vol. 1, no. 1, March 2002, pp. 5-12.

Soloway, E. and Ehrlich, K. (1984): 'Empirical studies of programming knowledge', *IEEE Transactions on Software Engineering*, vol. 10, no. 5, September 1984, pp. 595-609.

Sosa, M. (2008): 'A structured approach to predicting and managing technical interactions in software development', *Research in Engineering Design*, vol. 19, no. 1, March 2008, pp. 47-70.

Sosa, M.E., Eppinger, S.D., Pich, M., McKendrick, D.G., and Stout, S.K. (2002): 'Factors that influence Technical Communication in Distributed Product Development: An Empirical Study in the Telecommunications Industry', *IEEE Transactions on Engineering Management*, vol. 49, no. 1, February, 2002, pp. 45-58.

Speier, C., Valacich, J. S., and Vessey, I. (1997): 'The effects of task interruption and information presentation on individual decision making' *International Conference on Information Systems*, 1997, pp. 21-36.

Storey, M. D., Wong, K., Fracchia, F. D., and Mueller, H. A. (1997): 'On Integrating Visualization Techniques for Effective Software Exploration', *IEEE Symposium on Information Visualization (INFOVIS)*, October 1997, pp. 38.

Storey, M. D., Best, C., and Michaud, J. (2001): 'SHriMP Views: An Interactive Environment for Exploring Java Programs', *International Workshop on Program Comprehension*, May 2001, pp. 111-112.

Storey, M. D., Cubranic, D., and German, D. M. (2005): 'On the use of visualization to support awareness of human activities in software development: a survey and a framework', *ACM Symposium on Software Visualization*, May 2005, pp. 192-202.

Szóstek, A. M. and Markopoulos, P. (2006): 'Factors defining face-to-face interruptions in the office environment', *Conference on Human Factors in Computing Systems*, April 2006, pp. 1379-1384.

Taylor, C.M.B. and Munro, M. (2002): 'Revision towers', *International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT, June 2002, pp. 43-50.

Tee, K., Greenberg, S., and Gutwin, C. (2006): 'Providing artifact awareness to a distributed group through screen sharing', *ACM Conference on Computer-Supported Cooperative Work (CSCW)*, November 2006, pp. 99-108.

Trainer, E., Quirk, S., de Souza, C.R.B., and Redmiles, D.F. (2008): 'Analyzing a Socio-Technical Visualization Tool Using Usability Inspection Methods', *IEEE Symposium on Visual Languages and Human Centric Computing*, September 2008, pp. 78-81.

Tu, Q., and Godfrey, M. W. (2002): 'An integrated approach for studying architectural evolution', *International Workshop on Program Comprehension*, July 2002, pp. 127–136.

Tufte, E. (1990): *Envisioning Information*, Graphics Press, Cheshire, CT.

Tufte, E. (2006): *Beautiful Evidence*, Graphics Press, Cheshire, CT.

Valetto, G., Helander, M., Ehrlich, K., Chulani, S., Wegman, M., and Williams, C. (2007): 'Using Software Repositories to Investigate Socio-technical Congruence in Development Projects', *International Workshop on Mining Software Repositories, International Conference on Software Engineering (ICSE)*, May 2007, pp. 25.

Wasserman, S. and K. Faust: (1994): *Social Network Analysis: Methods and Applications. Structural Analysis in the Social Sciences*, Cambridge University Press, Cambridge, UK.

Whittaker, S., Frohlich, D., and Daly-Jones, O. (1994): 'Informal workplace communication: what is it like and how might we support it?', *SIGCHI Conference on Human Factors in Computing Systems*, April 1994, pp. 131-137.

Wu, J., Holt, R. C., and Hassan, A. E. (2004a): 'Exploring software evolution using spectrographs', *Conference on Reverse Engineering*, November 2004, pp. 80–89.

Wu, X., Murray, A., Storey, M. A., and Lintern, R. (2004b): 'A reverse engineering approach to support software maintenance: Version control knowledge extraction', *Conference on Reverse Engineering*, November 2004, pp. 90–99.

Ye, Y., Yamamoto, Y., and Nakakoji, K. (2007): 'A socio-technical framework for supporting programmers', *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, September 2007, pp. 351-360.