Supporting Collaborative Software Development through the Visualization of Socio-Technical Dependencies

Cleidson R. B. de Souza[1], Stephen Quirk[2], Erik Trainer[2], David F. Redmiles[2]

[1]Universidade Federal do Pará
Belém, PA, Brazil
66075-110
+55-91-3201-7835
cdesouza@ufpa.br

[2]University of California, Irvine
Irvine, CA, USA
92607-3425
+1-949-824-0247
{squirk, etrainer, redmiles}ics.uci.edu

ABSTRACT
One of the reasons large-scale software development is difficult is the number of dependencies that software engineers face. These dependencies create a need for communication and coordination that requires continuous effort by developers. Empirical studies, including our own, suggest that technical dependencies among software components create social dependencies among the software developers implementing those components. Based on this observation, we developed Ariadne, a plug-in for Eclipse. Ariadne analyzes software projects for dependencies and collects authorship information about projects relying on configuration management repositories. Ariadne can "translate" technical dependencies among components into social dependencies among developers. We have created visualizations to convey dependency information and the presence of coordination problems identified in our previous work. We believe the information conveyed in the visualizations will prove useful for software developers.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – user interfaces. H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces – *collaborative computing, computer-supported cooperative work*; H.1.2 [**Models and Principles**]: User/Machine Systems – *human factors, human information processing*.

General Terms
Design, Human Factors.

Keywords
Collaborative software development; socio-technical dependencies; program dependencies; social dependencies; visualization; awareness; coordination.

GROUP' 07, November 4 – 7, 2007, Sanibel Island, Florida, USA.

1. INTRODUCTION
Researchers and practitioners have long recognized that breakdowns in communication and coordination efforts constitute a major problem in software development (Curtis, Krasner et al. 1988). One of the reasons for this problem is the large number of dependencies that any software development effort involves: dependencies among activities in the development process and dependencies among different software artifacts. To overcome this problem, the field of software engineering has developed tools, approaches, and principles to manage dependencies. Configuration management and issue-tracking systems are examples of such tools. The adoption of software development processes (Nutt 1996; Fuggetta 2000) exemplifies an organizational approach, while the information hiding (Parnas 1972) illustrates a fundamental principle that underpins several mechanisms in programming languages (Larman 2001).

In any one of these cases, the underlying goal is to make dependencies more manageable. By minimizing dependencies it is possible to reduce required communication and coordination by software developers. This was recognized by Conway (1968) and Parnas (1972) over 30 years and validated by different empirical studies more recently (Morelli, Eppinger et al. 1995; Sosa, Eppinger et al. 2002; Grinter 2003; de Souza, Redmiles et al. 2004).

Despite the acknowledged relationship between dependencies and coordination needs, this relationship has not been explored to facilitate software development activities. Indeed, software development is a strong candidate for exploring this relationship since (i) dependencies among software components can be automatically identified, and (ii) software is malleable (i.e. dependencies, if so desired, can be more or less easily changed, and consequently the coordination of those developing the software). Ariadne aims to fill the gap between dependencies and communication and coordination needs and explore this socio-technical relationship. This article describes Ariadne's motivation, underlying architecture, and visualization-based approach. Ariadne's contribution is the usage of software dependency analysis to facilitate the coordination and execution of software development activities. By identifying the "social" dependencies among software developers extracted from technical dependencies, Ariadne is able to identify developers who are more likely to be communicating, developers whose similar dependencies make them likely to collaborate. Furthermore, our approach uses visualization as a cognitive tool to aid developers in

understanding these relationships as they arise from dependencies in the project. Visualization of socio-technical dependencies is Ariadne's major strength..

The rest of the paper is organized as follows. We begin by presenting the results of previous field studies that motivated our approach and the construction of Ariadne. Then, we describe Ariadne and our approach to extract code dependencies from the source-code, and how we infer social dependencies between software developers from source-code dependencies. We detail Ariadne's architecture in the next section. After that, we present the visualizations that Ariadne provide and explain how they relate to the problems that we observed in our field studies. After that, a discussion is presented, followed by conclusions about our work and avenues for future work.

2. MOTIVATION

This section describes a set of four scenarios identified during our previous field studies of collaborative software development (de Souza, Redmiles et al. 2004)(de Souza 2005)(de Souza, Hildenbrand et al. 2007)(de Souza and Redmiles 2007). They were chosen because (i) they illustrate that software developers are aware of the relationship between dependencies and coordination, and, more importantly, (ii) they illustrate problematic situations that emerge when these dependency relationships are unknown. We identify two major problems uncovered in the scenarios, notably a lack of awareness among developers of one another's work and the challenge of finding developers of interest in software projects.

In the scenarios, specific names of individuals, organizational units, and software have been changed to provide anonymity.

2.1 Introduction to the Setting

All scenarios take place at the same organization, BSC, which recently adopted an organization-wide reuse program. Through this program, each team in the organization is responsible for developing particular components and, whenever possible, they should reuse software components that provide the services they require. That is, instead of implementing a particular feature, a developer (or team) should reuse the component that provides this same (or close enough) feature, if a component is available.

The scenarios described below are based on observations from two different software development teams: MCW and MBL. MCW is responsible for developing a client-server application that had not yet been released during the period of the study. These developers are divided in five sub-teams. Because of the reuse program, MCW developers need to interact with other software developers in other parts of the organization to reuse their components and, whenever necessary, request changes in these components. The second team is called MBL, which is responsible for developing a mobile application. The project staff was distributed over five different sites spread in 3 different countries: Raleigh, US; Westford, US; Beijing, China; Shanghai, China; and Taipei, Taiwan. The main coordination of the project and the project manager for this project was located in Westford, US, where all the data was collected.

The following scenarios were extracted from either the 32 semi-structured interviews (McCracken 1988) with MCW and MBL team members or the field notes that we collected using non-participant observation (Jorgensen 1989) over a period of 11 weeks at the field site. All interviews were transcribed and alongside with the field notes were coded using grounded theory techniques (Strauss and Corbin 1998). These scenarios illustrate problems faced by software developers at BSC. The reason for these problems can be found elsewhere (de Souza, Redmiles et al. 2004) (de Souza 2005)(de Souza, Hildenbrand et al. 2007)(de Souza and Redmiles 2007).

2.2 Problems with Software Development at BSC

2.2.1 Problems with Awareness of software developers

Scenario 1 – Manager's Lack of Awareness of Evolving Social Dependencies

Cathy is the project manager for one of the MCW sub-teams. She leads eight developers developing the client-side of the MCW application. This means that Cathy's team's work is especially tightly integrated with the MCW server side team. In fact, she has just had a meeting with the manager of this other team. They discussed the new schedule, since both teams' schedules have started to slip. Now, at the weekly group meeting with her team, Cathy wants to find out about her team members' progress so she can make decisions about the next release of the software. Each developer reports to Cathy what they think will be ready by the deadline. When she finds out that Francis, a developer in her team, has not started to integrate his code with two other developers in her own team, Alfred and Denise, she realizes that Francis' deadline is not "realistic": he will not be able to finish his implementation before the deadline.

Scenario 2 – Developers' Lack of Awareness of Evolving Code Dependencies

Jacqueline is developing a component that provides services to the user-interface layer in the multi-tier architecture adopted by the MBL software. Meanwhile, Alfred is implementing a software component in the UI that requests services from Jacqueline's code. That is, Alfred's code is dependent upon Jacqueline's code. Jacqueline has already finished the implementation of her component. However, she does not know if Alfred has already started integrating his code with her code, or the way she put it if her "API is being exercised". She is concerned because the deadline for this integration is coming. But more importantly, if Alfred finds some problem in her implementation, she might not have much time to change her implementation. She wishes she could know about Alfred's integration status without having to keep asking him about it.

Again, according to MBL developers, it is important for them to know who is consuming their code and when this integration starts, i.e., when a developer starts to use another developer's code. This information is useful because it allows the developer to anticipate the work that will be requested of him before the deadline. This information is necessary from both collocated and distributed colleagues. Indeed, the distributed nature of the MBL team project was particularly relevant in this case. The informal conversations that are afforded by the collocation simplify this process of finding information among local colleagues. In contrast, it is much more difficult for a developer to find this status information when his colleagues are distributed over different countries – across space and time. For instance,

Jacqueline reported that in one occasion a developer in China was already using her code and she did not know. Similarly, Chinese developers reported not knowing when their American colleagues started using their code.

2.2.2 Finding Software Developers

Scenario 3 – Developers finding the "right" developer

Fred is a software developer in the MCW team working on a client component that requests services from a server component. The server developer assigned to implement the server component is Peter. There is a software interface between the client and server software components, which is used by Fred and Peter to divide and organize their work: Peter is implementing the services described in the interface, while Fred is using the services provided by this interface. Fred needs to contact Peter in order to find out how to use Peter's component. However, Fred and Peter have never met and Fred does not know how to contact him. Furthermore, Fred has only access to a "dummy" implementation of the interfaces and that's what he has been programming against. This implementation and the interfaces were designed by the software architect, Jack.

To find information about Peter, Fred needs to contact his colleagues who might know Peter. Another option for Fred would be to look up the information in the CM system. However, in this case, he would only find Jack (the software interface designer) in the CM database, not the actual developer implementing this interface. Fred would have then to look up Jack's contact information and query as to whether he knew Peter. Another alternative to Fred is to contact his manager, Cathy, and let her find out Peter's contact information. In all cases, while Peter is only "two degrees separated" from Fred, Fred is required to navigate this chain of communication to reach Peter.

Scenario 4 – Developers finding "similar" developers

Jake is a software developer in the MCW team who needs a particular service in order to implement his own component. He has recently contacted Bob, the developer responsible for the component that provides the service he needs. Bob works in a team, not MCW, which has a different deadline. Because of the deadline, Bob has not been very responsive to Jake's requests. To make things worse, there are other developers in the same organization who also need to use Bob's component and who also have been sending requests to him. Because Bob has not been very responsive to Jake's requests, Jake has not been able to make a lot of progress and his schedule is starting to slip. Jake meets (by accident) the three other developers from BSC that also require services from Bob. Those four developers start to interact and divide the work, which previously was redundant. Furthermore, they start to request features from Bob that will be useful for the four of them.

2.2.3 Observations

Scenarios 1 and 2 draw attention to an important point in software development: the lack of awareness among software developers (Grinter 1995; de Souza, Redmiles et al. 2003; Sarma, Noroozi et al. 2003). For instance, scenario 1 illustrates how the software manager is not aware of the current level of integration between her team members and other software developers in the project. When she finds out that integration has not started, she realizes that this particular developer is not going to be able to make the deadline. Similarly, in the second scenario, Jacqueline is concerned with the integration that Alfred is going to perform because he might ask for changes in her code and there will be little time left for her. This scenario illustrates that Jacqueline, to some extent, might be aware of who depends on her code, especially when these developers are collocated, but might not be aware when these developers are distributed. More importantly, Jacqueline wants more information; she wants to be aware of when these dependencies are made concrete, or when Alfred starts integrating his code with hers. Overall, this suggests that analysis of the source-code in combination with CM information could have given an answer to Jacqueline and Cathy: by inspecting a data structure that contains the dependencies of each component in the project they would know if the integration had already started. They only need to find out if the strength of the connection between their code and the other developers' code is changing over time.

Scenario 1 also illustrates how software developers are, at least to some extent, aware of the coordination effort necessary to integrate code. To be more specific, because Cathy finds out that Francis needs to integrate his code with two other developers and because of the coordination effort necessary to perform such integration, she knows that Francis will not be able to meet the deadline.

Scenarios 3 and 4 illustrate one common problem in software development: finding people (McDonald and Ackerman 1998; Herbsleb, Mockus et al. 2001). In the first scenario (number 3), it describes the need to find the "right" developer, the one actually implementing the API. The next scenario (number 4) illustrates a situation where developers with similar dependencies were performing redundant work because they did not know about each other. What is most striking about these scenarios is that they describe situations in which developers could be found through dependency analysis of the source-code combined with information from the configuration management repository. Jake and Fred have the same dependency on the code than the people they were trying to find:

- Fred depends on Jack's code, and Peter depends on Jack's code. Fred wants to find Peter to remove his dependency on Jack's code (the dummy implementation); and

- Jake depends on Bob's code and found out that three other software developers also depend on Bob's code.

To summarize, these scenarios suggest that software dependency analysis, in addition to facilitating software reuse, software understanding, and other technical aspects (see next section); can be used by software developers to facilitate the coordination and execution of software development activities. These results are also supported by additional work described in (de Souza, Hildenbrand et al. 2007). In the next section, we will describe similar previous work.

3. Related Work

Many sub-disciplines of software engineering have researched various aspects of software dependencies. For example, dependency analysis techniques have focused on programs (Ferrante, Ottenstein et al. 1987; Podgurski and Clarke 1989), component-based systems (Vieira and Richardson 2002), and software architectures (Stafford, Wolf et al. 1998). Minimizing dependencies facilitates software reuse, understanding and testing.

For instance, program dependencies are used to improve software testing, maintenance, parallelization, and computer security. Another approach adopted by researchers and practitioners to deal with software dependencies is the creation of mechanisms in programming languages to reduce dependencies between software elements. In this case, the most important principle is information hiding (Parnas 1972), which motivates several mechanisms in programming languages, including data encapsulation, interfaces, and polymorphism; and is one of the key principles behind object-oriented programming (Larman 2001).

All these approaches, however, are purely technical. They do not take into account the relationship between software dependencies and the coordination of the work, a socio-technical relationship. Parnas, about 30 years ago, was one of the first researchers to recognize this relationship. He suggested that by reducing dependencies at the artifact level, it is possible to reduce developers' dependencies on one another, creating a managerial advantage (Parnas 1972; Herbsleb and Grinter 1999). Nowadays, this is a well-known argument among researchers and practitioners and is even cited in software engineering textbooks (Ghezzi, Jazayeri et al. 2003). Conversely, but also supporting this relationship between dependencies and coordination, Conway (1968) postulated that the structure of a software system would reflect the communication needs of the people performing the work. In short, whereas Parnas argues that dependencies shape the coordination and communication activities performed by software developers, Conway argues the converse: that dependencies reflect these coordination and communication activities. That is, technical dependencies between components create a need for communication and coordination between developers, and similarly, dependencies between the development tasks are reflected in the software.

Both Parnas' and Conway's arguments have been validated by several different empirical studies. Curtis et al. (1988) discussed how the system architecture affected the communication required among project personnel, and at the same time, he recognized that "occasionally, the partitioning [of components to reduce dependencies between components] was based not only on the logical connectivity among components, but also on the social connectivity among the staff". More recently, Herbsleb and Grinter (1999) discussed the influence of the software architecture in the coordination of distributed software development. They argued that "the more cleanly separated the modules, the more likely the organization can successfully develop them at different sites", because this will remove the communication required among the different sites. Finally, Sosa and colleagues (2004) found a strong correlation between dependent components in a software system and the frequency of communication among the team members dealing with these components.

In general, what can be observed is that despite the acknowledged relationship between dependencies and communication and coordination needs, this relationship has not been explored to facilitate software development activities. Software development is indeed a strong candidate for exploring this relationship since (i) dependencies among software components can be automatically identified, and (ii) software is malleable (i.e. dependencies, if so desired, can be more or less easily changed, and consequently the coordination of those developing the software). Ariadne, the tool described in this paper, aims to fill the gap between dependencies and communication and coordination needs and explore this important and powerful socio-technical relationship. In this paper, we describe Ariadne's underlying architecture and, more importantly, how Ariadne addresses the problems identified in the scenarios that ultimately motivate our work. Ariadne's most important contribution is the usage of software dependency analysis to facilitate the coordination and execution of software development activities. By identifying the "social" dependencies among software developers extracted from technical dependencies, Ariadne is able to identify developers who are more likely to be communicating as well as developers whose similar dependencies make them likely to collaborate. Furthermore, it can even facilitate people finding. Ariadne is described in the next section. An earlier version of Ariadne appeared in (Trainer, Quirk et al. 2005).

4. Ariadne

A simple restatement of the above arguments and scenarios is that software developers working on dependent pieces of code are more likely to engage in communication and coordination activities than developers working on unrelated pieces of code. Ariadne aims to explore this relationship to facilitate the coordination of software development efforts. Specifically, Ariadne is designed to perform automatic dependency analysis on software projects shared in configuration management repositories and generate visualizations of social dependency information. The visualizations generated by Ariadne can be used by software developers to identify two important pieces of information: who they depend on and who depends on their work. We hypothesize, based on our field studies, that by identifying these set of developers, developers can more easily coordinate their work.

4.1 Creating Social Dependencies

Creating social dependencies involves collecting code dependency information and retrieving authorship information of the source-code to identify the authors associated with a code dependency. Initially, Ariadne identifies the technical dependencies in the source-code by constructing call-graphs. According to Callahan and colleagues, a call-graph "summarizes the dynamic invocation relationships between procedures." (Callahan, Carle et al. 1990). However, in our approach we generate invocation information by leveraging Eclipse's existing SearchEngine API, which parses source-code to generate a call-graph (i.e. before runtime). As such, we generate static call-graphs rather than dynamic ones. A call-graph can be represented as a square matrix (Technical matrix) where entries represent the number of relationships between two procedures, or units of code (i.e. packages, classes, etc.). By describing dependencies in the source-code, a call-graph potentially unveils dependencies among software developers responsible for the software components (de Souza, Froehlich et al. 2005). In order to reveal dependencies among developers, it is necessary to annotate the call-graph with "social information." The associations between authors and code can be represented as a matrix (Sociotechnical matrix) where entries represent a value for the strength of a connection between author and code.

As the Technical and Sociotechnical matrices describe both technical dependencies and authorship information, they can be used to generate sociograms describing the dependence relationship only among software developers. That is, social dependencies between developers that exist because of dependencies in the source-code they are working on. A sociogram, as used in social network analysis (Wasserman and Faust, 1994) is a graphical representation of a set of items,

vertices or nodes, connected to one another via links or edges. Ariadne uses well-established social network operations to produce information about which developers depend on which other developers. To infer social dependencies, and create a sociogram, requires first multiplying the Sociotechnical matrix by the Technical matrix to produce an author by code matrix indicating which code units authors depend on. Multiplying this product by the transpose of the Sociotechnical matrix yields an author by author matrix (Social matrix) representing the extent of code dependencies between a pair of authors (Cataldo, Wagstrom et al. 2006). Figure 1 presents an example of a sociogram of the Sourceforge.net project Tyrant, created using Ariadne version 0.1. The sociogram is presented as a View within the Eclipse enviroment, so that it is easily accessible by software developers from their own work environment.

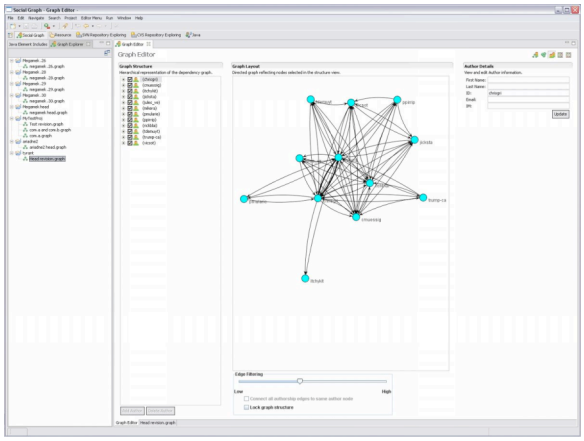

Figure 1 - Tyrant sociogram

4.2 Ariadne's Architecture

Ariadne is implemented as a Java plug-in to the popular Eclipse IDE. As such, Ariadne is integrated into this environment and makes heavy use of Eclipse functionality and its plug-in model. The dependency processing functionality is encapsulated in a main control plug-in that delegates source-code analysis, annotation of the source-code analysis data, and visualization of the created data structure to sub plug-ins. As a result, Ariadne offers users the flexibility to use dependency generators for a diverse set of source languages, configuration management repositories, and methods of visualization.

Ariadne automatically selects (while offering users the ability to override this choice) appropriate plug-ins for analyzing the user's project based on the project's context. Once the control plug-in has located appropriate sub plug-ins to analyze the project's source-code and query the project's configuration management, the control plug-in automatically generates social dependencies for that project. Using one of the installed visualization plug-ins, it is possible to display all three types of dependency information to the user: technical dependencies, social call-graph (call-graphs annotated with authorship information), and sociograms.

Our current implementation can present call-graphs and social call-graphs at three different levels of abstraction, based on the programming language's hierarchy (e.g. packages, classes, and methods in Java). Essentially, information is aggregated at each hierarchy level to, potentially, average the different results provided by diverse call-graph extractors (Murphy, Notkin et al. 1998). For instance, class dependencies are displayed as the aggregation of method dependencies (i.e., the call-graph).

5. Our Approach: Visualization

As exemplified by the software scenarios above, a key issue in collaborative activities, particularly software development, is awareness or "knowing what is going on" (Dourish and Bellotti 1992; Heath and Luff 1992).

Awareness of colleagues' activities facilitates the coordination of collaborative efforts and is achieved by social actors through different channels. Depending on several factors such as the geographical distance between teams, cultural differences, and rules and norms in an organization, it may be difficult for developers to discern the activities of their colleagues. As such, various tools have been developed by the software engineering research community to both provide and augment developers' awareness in software projects. Many tools rely on visualization techniques as a means to provide awareness (Al-Ani, Sarma et al. 2006). Visualization is generally accepted as a good candidate for conveying awareness information because of its ability to capitalize on perceptual effects familiar to users, such as foreground/background effects and use of color to make the information of interest more salient. However, merely simplifying the logical representation of the data is not a panacea. As noted by Petre, Blackwell, and Green, the complexity of the visualization technique is linked to not only the information to be visualized, but also to its context of use (Petre, Blackwell et al. 1997).

In other words, the visualization should be designed to support the particular roles of users and the specific tasks they need to accomplish. For example, the same bit of information that is highly important for one user may be completely uninteresting for another user in the same situation. As such, we have designed our visualizations to not only address the specific problems outlined in the aforementioned scenarios, but also to present the information in a way that integrates seamlessly with developers' activities in Eclipse.

For displaying data entities and the relationships among them, two popular representations have been commonly espoused by researchers: matrices and graphs. In an empirical study by Novick and Hurley that examined the use of three spatial diagrams, including graphs and matrices, and reasons to use one over the other, matrices were deemed desirable for representing associative links (non-directional) and the absence of a link between two elements. On the other hand, graphs were found to be useful when any node can be linked to another node without specific constraints. Graphs were also determined to be appropriate for visualizing the fact that any number of links can start from and terminate at a given node (e.g. many-to-many, one-to-many relationships). Additionally, it is much easier to traverse a given path through the data when the representation is a graph rather than a matrix (Novick and Hurley, 2001). Moreover, in judging the applicability of graph visualizations to data, Herman (Herman, 2000) ask whether there are inherent relations among the data elements. If the answer is "yes", then the data can be represented by nodes of a graph with edges representing the relationships between the data.

Based on the considerations above, we use a graph-based visualization to represent the three different types of dependency information generated by Ariadne. Because Ariadne deals with dependencies, or more specifically, the relationships between code modules and code modules, authors to code modules, and authors, an appropriate visual representation should convey both these relationships and the data entities themselves. As illustrated in the previous paragraph, one of the most important utility of graphs, as opposed to matrices, is the explicit emphasis on the relationships between the data and their semantic meanings. Because Ariadne seeks to extract meaningful information from dependencies that developers and mangers can leverage to increase their awareness of others' work activities, we choose graph-based visualizations to represent our data rather than matrices.

When using a graph-based visualization to represent data, the layout of the graph is an important cue for understanding underlying structures in the data (Petre, Blackwell et al. 1997; Herman, Melancon et al. 2000; Otjacques and Feltz, 2005). Because Ariadne is used primarily for uncovering the meaning of relationships between developers and their code rather than their structural properties, the layouts do not always provide added value. In each scenario we explicitly state whether or not the layout of the visualization is meant to convey additional information to developers.

In order to rapidly test and refine our graph-based visualizations, we use the JUNG (Java Universal Network/Graph) framework (O'Madadhain and Fisher, 2003). JUNG provides a general, flexible API for creating, manipulating, and visualizing graph and network data. Although it is not a mature, fully-featured framework and is largely intended to be extended by developers, JUNG does come with some standard libraries for rendering a limited number of node and edge shapes. As such, developers can quickly produce graph-based visualizations comprised of nodes and edges. However, because JUNG is a general framework and provides limited functionality without being extended, it is difficult to quickly experiment with some visual affordances such as the shapes of edges and the specific positions of nodes and edges. On the other hand, JUNG's extensibility has allowed us to easily utilize other visual cues such as node and edge size, color, and different views of abstraction to convey meaningful, contextual information to developers.

This section describes four different visualizations, each tailored to the problems uncovered in the scenarios described in section 2. In each visualization:

- Code modules are designated by green nodes in keeping with Eclipse's Java coloring scheme;
- Software interfaces are designated by purples nodes in keeping with Eclipse's Java coloring scheme;
- Authors are represented as icons, code modules are designated by circular shapes;
- The local developer in the local workspace is designated as an orange node with his name in bold;
- Other developers are designated by cyan nodes;
- A directed edge from a code module to code module indicates a technical dependency; and
- A directed edge from a developer to a code module indicates a socio-technical dependency (authorship) .`

5.1 Visualization / Scenario 1

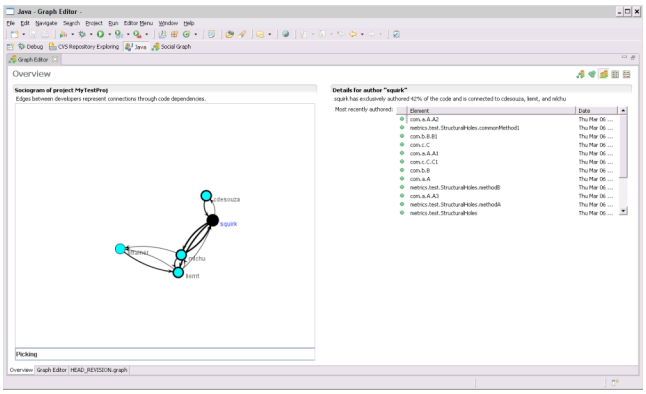

Figure 2 - Manager Awareness View

In order to meet project deadlines, managers need to know *when* team members have started integrating their code with other team members' code. To this end, the Manager Awareness View presents a project's sociogram with recency information encoded in the shading of edges between authors. More recent dependencies created between authors appear as more deeply shaded connections, while older connections are lighter. The graph and recency information help managers create an understanding of the state of integration between project developers by highlighting recent connections between authors. Recent connections signify to managers that two developers who must integrate code with each other are, in fact, integrating their code. To discover the specific artifacts through which those developers are connected, managers can drill down into the connections for more detailed information. Users of this visualization can interact with the graph by clicking on authors and edges to find information on what an author has most recently authored and the code dependencies underlying an edge, respectively.

5.2 Visualization / Scenario 2

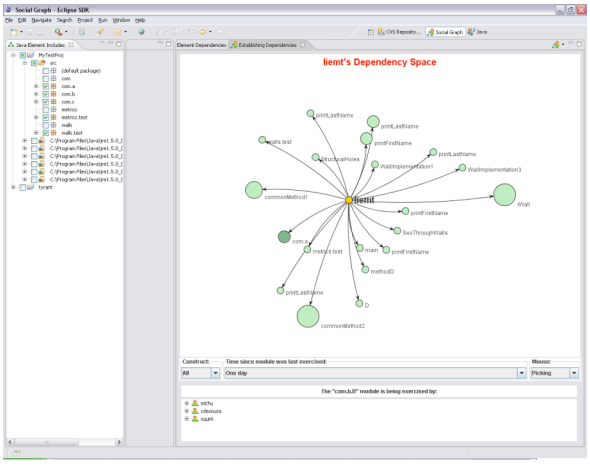

Figure 3 - Developers' Awareness of Real-time Code Dependencies

In both collocated and distributed software engineering environments, it is important for developers to be aware of who is a consumer of their code and when the "consumption" starts. The Establishing Dependencies View (Figure 3) provides a means through which developers can see such information. As we noted earlier, it is important to contextualize the information displayed in the visualization in a way that flows with the user's normal activities. Consequently, the focal point of the Establishing Dependencies View is the developer (indicated by the "person" icon and accompanying user name, e.g. liemt) surrounded by the code modules that he has authored. By displaying only a subset of developer-specific information, the developer can see how the code he writes is being utilized by other developers without having to traverse socio-technical dependency relationships for the whole project.

In any visualization, it is important that the mapping from the visual variables to the attributes of the data is appropriate and meaningful given how humans visually perceive things (Ware, 2000). As such, in our visualization we make use of both size and color to guide the developer toward the information he desires. According to the second scenario, the developer's interest lies in both figuring out who is utilizing code and when the code is being called. In order to facilitate the developer's understanding of the code modules that are being consumed, we highlight code modules that are more highly depended upon than others by displaying them in a larger size. In this way, the developer's attention is immediately focused on potential code modules of interest and diverted from others. Indeed, according to the second scenario, the developer is more likely to be interested in certain modules that are being heavily consumed rather than code modules with little to no activity. Once the developer has identified code modules of interest, he can look at the darkness of the node's color to determine if the module has been recently utilized by another developer. To view authors and code modules consuming a particular module of interest, the user/developer left-clicks on the desired code module surrounding him. A panel below displays a list of authors and the code modules they have written that consume the code module in question.

By contextualizing a project's socio-technical dependency relationships and making use of color and size to highlight particular areas of interest, the Establishing Dependencies View visualization has potential to increase developers' awareness of evolving code dependencies. If by examining a code module that should be the center of activity for many other developers in the project, a developer finds that no one has started utilizing it, he can determine that immediate communication with those other developers is necessary to avoid delays, or other potential breakdowns in the project.

5.3 Visualization / Scenario 3

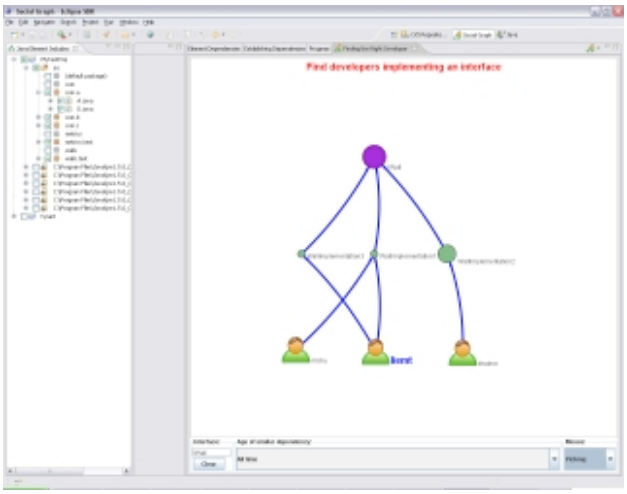

Figure 4 - Finding the "right" person

A common problem in software development is finding the right person to talk to about a piece of code, an interface's implementation for instance. Rather than merely identifying the author of an interface, this view allows the developer to identify others who implement an interface of interest and the code modules through which they do so. As in the previous visualizations, we make use of visual cues such as color and size to convey important contextual information.

As we indicated before, the layout of the visualization can influence interpretations of the data. In our visualization, we leverage this ability of a graph layout to facilitate the developer's discovery of paths to an interface. We choose to represent interfaces, implementing code modules, and authors as a hierarchical static ordering (Figure 3). This approach has two advantages. First, because the layout is static, users do not have to spend time arranging the nodes and edges to discover how other developers are connected to the same interface. Rather the layout ensures that the same information will be presented to the user in the same exact way each time. Second, the layout reinforces the user's understanding that the process of uncovering the information of interest involves three major components: authors, implementing code modules, and the interface. Authors are connected to code modules which are connected to the interface that they implement. By breaking each component into its own level, developers can more easily trace each step in the process.

5.4 Visualization / Scenario 4

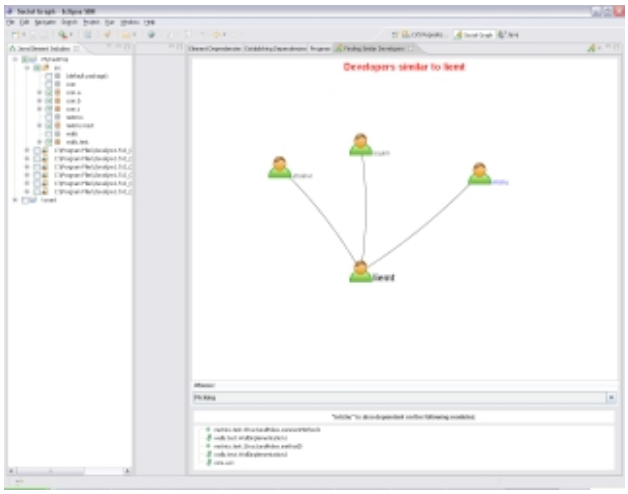

Figure 5 - Finding "similar" developers

In order to eliminate redundant work and to ask other developers for help with particular code modules, developers should be able to identify other developers who share similar socio-technical dependencies. Once the developer identifies others who are performing similar work, he can coordinate with them accordingly. The Finding Similar Developers View (Figure 4) aims to facilitate this process. Similar to the second visualization, this view aims to contextualize the information displayed to the developer by placing him in the center of the graph. However, rather than showing code modules the developer has authored, we display other developers who are dependent on the same pieces of code. Moreover, to direct the developer's attention toward other developers who have many dependencies in common - developers who may potentially be performing overlapping work - we display these developers in a larger size than others. Because there is not necessarily a social dependency between ego and another "similar" developer, we use undirected edges to represent the relationship between similar developers. To view the dependencies that are shared between developers, the user can click on an edge of interest.

6. DISCUSSION

The visualizations provided by Ariadne have been heavily influenced by the scenarios drawn from the MCW and MBL project teams field data, and, as such, aim to solve specific problems in software development. The first two visualizations aim to increase both managers' and developers' awareness of code dependencies and when they are established in a software project. The first visualization allows managers to gain an understanding of the status of code integration among developers, enhancing managers' ability to assess developers' progress and to subsequently determine whether or not project deadlines will be met in time. Similarly, the second visualization allows a developer to determine if and when other developers have started to integrate code with theirs. If, for example, developer A knows that developer B has not started to integrate his code within the last month, more work may be requested of developer A by developer B before the deadline. The last two visualizations are aimed at helping a developer find other developers who are related through the code they write. In the third visualization, a developer can identify other developers who implement a particular interface and the specific code modules through which they do so. In the event that developers are performing overlapping work or need assistance in programming against specific code modules, the last visualization directs the developer toward other "similar" developers and the underlying technical dependencies they have in common.

Increased awareness provided by the tool and the transparency it brings to developers' work could have effects on the way developers work, and even the tool's results. Ariadne allows developers and supervisors greater insight into the work done by their colleagues and collaborators, respectively, allowing them to gauge their peers' progress, or the lack thereof. Although developers' code becomes public when checked into a source control system (or sometimes before), dependency and authorship information of the kind produced by Ariadne is not usually so easily available. Access to such information usually requires significant effort and is confined to individuals or pairs of individuals (when shared with the target of the investigation). As such, developers may consider such information personal and could take steps to "game" the results of the tool if they feel their personal information might be used against them. Managing the use of Ariadne is certainly a challenge for the future.

We believe that the principles highlighted in Ariadne's visualizations will enhance developers' and managers' awareness of their colleagues' development activities. By capitalizing on a visualization's ability to perceptually highlight information of interest through visual affordances such as color, size, different views of abstraction, and layout, Ariadne both brings the most relevant information to the foreground and presents it to the user in a form that is cognitively easier to process. Moreover, by gearing the visualizations toward particular coordination and awareness problems in software development, Ariadne's visualizations present information that is highly contextualized toward specific project activities. The same information relevant to a developer's work may not be of interest to a project manager for example, and vice versa. Finally, because the tool and its visualizations are seamlessly integrated into the Eclipse IDE, developers can explore how they are linked to other developers through the code they write, without disrupting their current work.

7. CONCLUSIONS AND FINAL REMARKS

This article described Ariadne, a plug-in to the Eclipse IDE that aims to reduce the gap between technical and social dependencies, and therefore facilitate the coordination of software development work. Ariadne was motivated by our own field studies of software development and reflects some of the insights that we learned from these studies. We described Ariadne's features as well as architecture. Ariadne currently offers developers a variety of visualizations. In the future, we plan to offer developers a choice of many visualizations ranging from directed graphs, annotated class diagrams, or decorators inside the Eclipse workbench. Decorators are simple visual clues (usually in the form of an icon) to developers that display additional information about resources in the workspace. Eventually we plan to release Ariadne generally as an open source tool.

Currently, Ariadne supports software developers working with source-code, which allows us to perform automatic identification of dependencies. We plan to investigate how to support other software development artifacts. In this case, the dependency analysis would be performed using traceability (Spanoudakis and

Zisman 2004) which describes dependency relationships among different software development artifacts. In addition, we are evaluating Ariadne using usability inspection methods and user-studies.

8. ACKNOWLEDGMENTS

This research was supported by the U.S. National Science Foundation under grants 0534775 and 0205724, by an IBM Eclipse Technology Exchange grant, and by the Brazilian Government under CAPES grant BEX 1312/99-5 and CNPq grant 479206/2006-6. We also gratefully acknowledge comments by our colleague Steve Abrams.

9. REFERENCES

B. Al-Ani, A. Sarma, G. Bortis, I. Almeida da Silva, E. Trainer, A. van der Hoek, and D. Redmiles , Continuous Coordination (CC): A New Collaboration Paradigm. CSCW Workshop on Supporting the Social Side of Large Scale Software Development, Banff, Canada, November 2006, pages 69-72.

Callahan, D., A. Carle, et al. (1990). "Constructing the Procedure Call Multigraph." IEEE Transactions on Software Engineering **16**(4): 483-487.

Cataldo, M., P. A. Wagstrom, et al. (2006). Identification of Coordination Requirements: implications for the Design of Collaboration and Awareness Tools. 20th Conference on Computer Supported Cooperative Work. Banff, Alberta, Canada, ACM Press.

Conway, M. E. (1968). "How Do Committees invent?" Datamation **14**(4): 28-31.

Curtis, B., H. Krasner, et al. (1988). "A field study of the software design process for large systems." Communications of the ACM **31**(11): 1268-1287.

de Souza, C. R. B., D. Redmiles, et al. (2003). Management of Interdependencies in Collaborative Software Development: A Field Study. International Symposium on Empirical Software Engineering (ISESE'2003), Rome, Italy, IEEE Press.

de Souza, C. R. B., D. Redmiles, et al. (2004). How a Good Software Practice thwarts Collaboration - The Multiple roles of APIs in Software Development. Foundations of Software Engineering, Newport Beach, CA, USA, ACM Press.

de Souza, C. R. B., D. Redmiles, et al. (2004). Sometimes You Need to See Through Walls - A Field Study of Application Programming Interfaces. Conference on Computer-Supported Cooperative Work (CSCW '04), Chicago, IL, USA, ACM Press.

de Souza, C. R. B., J. Froehlich, et al. (2005). Seeking the Source: Software Source-code as a Social and Technical Artifact (to appear). ACM Conference on Group Work, Sanibel Island, FL, USA.

de Souza, C. R. B. (2005). On the Relationship between Software Dependencies and Coordination: Field Studies and Tool Support. Department of Informatics, Donald Bren School of Information and Computer Sciences. Irvine, CA, University of California, Irvine. Ph.D.: 186.

de Souza, C. R. B., T. Hildenbrand, et al. (2007). Towards Visualization and Analysis of Traceability Relationships in Distributed and Offshore Software Development Projects (to appear). Software Engineering Approaches for Offshore and Outsourced Development, Zurich, Springer.

de Souza, C. R. B. and D. Redmiles (2007). The Awareness Network: To Whom Should I Display my Actions and Whose Actions Should I Monitor? (to appear). European Conference on Computer-Supported Cooperative Work. Limerick, Ireland, Springer.

Dourish, P. and V. Bellotti (1992). Awareness and Coordination in Shared Workspaces. Conference on Computer-Supported Cooperative Work (CSCW '92), Toronto, Ontario, Canada, ACM Press.

Ferrante, J., K. J. Ottenstein, et al. (1987). "The program dependence graph and its use in optimization." ACM Transactions on Programming Languages and Systems (TOPLAS) **9**(3): 319-349.

Fuggetta, A. (2000). Software Processes: A Roadmap. Future of Software Engineering, Limerick, Ireland.

Ghezzi, C., M. Jazayeri, et al. (2003). Fundamentals of Software Engineering, Prentice Hall.

Grinter, R. E. (1995). Using a Configuration Management Tool to Coordinate Software Development. Conference on Organizational Computing Systems, Milpitas, CA.

Grinter, R. E. (2003). "Recomposition: Coordinating a Web of Software Dependencies." Journal of Computer Supported Cooperative Work **12**(3): 297-327.

Heath, C. and P. Luff (1992). "Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Control Rooms." Computer Supported Cooperative Work **1**(1-2): 69-94.

Herbsleb, J. D. and R. E. Grinter (1999). "Architectures, Coordination, and Distance: Conway's Law and Beyond." IEEE Software: 63-70.

Herbsleb, J. D., A. Mockus, et al. (2001). An Empirical Study of Global Software Development: Distance and Speed. International Conference on Software Engineering, Toronto, Canada, IEEE Press.

Herman, I., G. Melancon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. IEEE Transactions on Visualization and Computer Graphics, 6(1):24-43, 2000.

Jorgensen, D. L. (1989). Participant Observation: A Methodology for Human Studies. Thousand Oaks, SAGE publications.

Larman, G. (2001). "Protected Variation: The Importance of Being Closed." IEEE Software **18**(3): 89-91.

McCracken, G. (1988). The Long Interview, SAGE Publications.

McDonald, D. W. and M. S. Ackerman (1998). Just Talk to Me: A Field Study of Expertise Location. Conference on Computer Supported Cooperative Work (CSCW '98), Seattle, Washington.

Morelli, M. D., S. D. Eppinger, et al. (1995). "Predicting Technical Communication in Product Development Organizations." IEEE Transactions on Engineering Management **42**(3): 215-222.

Murphy, G., D. Notkin, et al. (1998). "An Empirical Study of Static Call Graph Extractors." ACM Transactions on Software Engineering and Methodology **7**(2): 158-191.

Novick, L. and S. Hurley, To Matrix, Network, or Hierarchy: That is the Question, Cognitive Psychology, Vol. 42, 2001, pp. 158-216

Nutt, G. J. (1996). "The evolution toward flexible workflow systems." Distributed Systems Engineering(3): 276-294.

Otjacques, B. and Feltz, F. 2005. Representation of Graphs on a Matrix Layout. In Proceedings of the Ninth international Conference on information Visualisation (Iv'05) - Volume 00 (July 06 - 08, 2005). IV. IEEE Computer Society, Washington, DC, 339-344.

Parnas, D. L. (1972). "On the Criteria to be Used in Decomposing Systems into Modules." Communications of the ACM **15**(12): 1053-1058.

Petre, M., A. Blackwell, T. Green, Cognitive questions in software visualization, in Software Visualization: Programming as a Multi-Media Experience, MIT Press, Cambridge, MA, 1997, pp. 453–480.

Podgurski, A. and L. A. Clarke (1989). The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance. Symposium on Software Testing, Analysis, and Verification.

Sarma, A., Z. Noroozi, et al. (2003). Palantír: Raising Awareness among Configuration Management Workspaces. Twenty-fifth International Conference on Software Engineering, Portland, Oregon.

Sosa, M. E., S. D. Eppinger, et al. (2002). "Factors that influence Technical Communication in Distributed Product Development: An Empirical Study in the Telecommunications Industry." IEEE Transactions on Engineering Management **49**(1): 45-58.

Sosa, M. E., S. D. Eppinger, et al. (2004). "The Misalignment of Product Architecture and Organizational Structure in Complex Product Development." Management Science **50**(12): 1674-1689.

Spanoudakis, G. and A. Zisman (2004). Software Traceability: A Roadmap. Handbook of Software Engineering and Knowledge Engineering. S. K. Chang, World Scientific Publishing Co.

Stafford, J. A., A. L. Wolf, et al. (1998). Architecture-Level Dependence Analysis for Software Systems. International Workshop on the Role of Software Architecure in Testing and Analysis (ROSATEA), Marsala, Sicily, Italy.

Strauss, A. and J. Corbin (1998). Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory. Thousand Oaks, SAGE publications.

Trainer, E., S. Quirk, et al. (2005). Bridging the Gap between Technical and Social Dependencies with Ariadne. Eclipse Technology Exchange, San Diego, CA.

Vieira, M. R. E. and D. J. Richardson (2002). The Role of Dependencies in Component-Based System Evolution. International Workshop on Principles of Software Evolution, Orlando, Florida.

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