Exploiting the Relationship between Software Dependencies and Coordination through Visualization

Cleidson de Souza[1,2] Erik Trainer[1] Stephen Quirk[1] David Redmiles[1]

[1]Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA, USA – 92667

[2]Departamento de Informática
Universidade Federal do Pará
Belém, PA, Brazil – 66075

cdesouza@ufpa.br, {etrainer, squirk, redmiles}@ics.uci.edu

ABSTRACT

Large software development projects require management of dependencies by managers and developers to ensure the smooth coordination of work. Based on theoretical predictions and empirical observations (ours and from others) that dependencies between software components create dependencies between the developers implementing those components, we created Ariadne, a visualization tool designed as a plug-in for Eclipse. Ariadne aims to explore this socio-technical relationship by translating technical dependencies into dependencies among software developers presented in a social network graph. Here, we describe a revised visualization that preserves the ease of identifying connections found in social network graphs but also facilitates efficient identification of dependency information needed to coordinate developers' work. We have evaluated the visualization using multiple inspection methods appropriate for visual interfaces. The results of our evaluation indicate significant improvements over the graph-based approach and suggest important avenues for future work.

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; visualization; awareness; coordination.

1. INTRODUCTION

It has been long recognized that breakdowns in communication and coordination efforts constitute a major problem in collaborative software development [5]. One of the reasons for these problems is the large number of dependencies among activities in the software development process and the dependencies among different software artifacts. To overcome this problem, the field of software engineering has developed tools, approaches, and principles to deal with dependencies. Configuration management and issue-tracking systems are examples of such tools, while the adoption of software development processes [14] exemplifies an organizational approach. One of the most important and influential principles used to manage dependencies is Parnas' information hiding [24]. According to this principle, software modules should be both "open (for extension and adaptation) and closed (to avoid modifications that affect clients)" [20]. Information hiding aims to decrease the dependency (or coupling) between two modules so that changes to one do not impact the other.

This relationship between software dependencies and the coordination of the work holds even when modular decomposition is applied, i.e., software developers who are supposed to work independently because of the usage of software interfaces, still need to communicate and coordinate to guarantee a smooth flow of work [11]. Despite this acknowledged relationship between dependencies and communication and coordination needs [4,24], this relationship has not been explored to facilitate software development activities. In fact, 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., their dependencies, if so desired, can be more or less easily changed, and consequently the coordination effort of those developing it. Ariadne, the tool described in this paper, was created with the aim of reducing this gap and exploring this socio-technical relationship to support software developers' daily activities. The contribution of this paper is the presentation of Ariadne. Initially, Ariadne used a traditional graph-based visualization approach, but our attempts to visualize the complete set of this information proved to be unmanageable for large software projects due to the number of connections and variability of the graph layout. Therefore, we designed and evaluated a new visualization using multiple inspection methods. The results of this evaluation indicate significant improvements over the traditional graph-based approach and suggest important avenues for future work.

The rest of this paper is structured as follows. In the following section we review previous research efforts into the socio-technical relationship between software dependencies and coordination of software development. Then, in section 3 we briefly present Ariadne and its approach for analyzing dependencies in software development projects. After that, we present our revised visualization method. Section 5 and 6 present the results of the evaluations that we have performed of the visualization. We conclude in section 7 and present avenues for future work.

2. BACKGROUND AND RELATED WORK

2.1 Software Dependencies

Software engineers have long recognized the need to deal with dependencies between software components. For example, dependency analysis techniques have focused on programs [13, 25], component-based systems [30], and software architectures [28]. Minimizing dependencies facilitates several tasks in software development. For instance, program dependencies are used to improve software testing, maintenance, parallelization, computer security, and code optimization [25]. Component dependency analysis is crucial to effective maintenance, evolution, testing, debugging, and management of component-based systems [30]. In addition, architectural dependency analysis techniques can be used to support architectural reuse, change impact analysis, regression testing, and software understanding.

Dependence relationships in software engineering have also been studied under the name traceability. In this case, instead of focusing on the source code or the software architecture, the focus is on dependency relationships between artifacts. Software traceability is defined as "the ability to relate *artifacts* created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other, the stakeholders that have contributed to the creation of the artifacts, and the rationale that explains the form of the artifacts" [27]. In a survey of the area, Spanoudakis and Zisman [27] identified seven possible types of relationships between software artifacts, dependence being one of them. The existence of a dependence link between the requirements and the analysis, and later to a design document that implements this requirement, is, seen as something positive or beneficial, because it indicates that this particular requirement has been addressed by the software being implemented. Furthermore, some authors argue that dependencies between requirements can support software reuse: if similar requirements are identified when the stated requirements are compared with existing requirements, then this indicates a possible reusable component [6].

2.2 Software Dependencies and Coordination

Parnas was one the first researchers to recognize the relationship between software dependencies and coordination. More than 30 years ago, he suggested that by reducing dependencies at the artifact level, it is possible to reduce developers' dependencies on one another, creating a managerial advantage [18, 24]. Nowadays, this is a well-known argument among researchers and practitioners that can be found in textbooks [16,p.241].

Conversely, but also supporting this relationship between dependencies and coordination, Conway [4] postulated that the structure of a software system would reflect the communication needs of the people performing the work. MacCormack and colleagues [21] exemplify Conway's argument when they compare commercial and open-source software development. Because software developers were collocated in the commercial project, it was easier to build tight connections between software components, therefore producing a software system more coupled compared to the similar open-source project with distributed developers. In short, while 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 product dependencies.

2.3 Empirical Studies

Both Parnas' and Conway's arguments have been validated by several different empirical studies. In 1988 for example, Curtis et al. [5] discussed, among other things, 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" [5, pg. 1280].

Herbsleb and Grinter [18] discuss the influence of the software architecture in the coordination of distributed software development. They argue 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.

Sosa and colleagues [26] found a strong correlation between dependent components in a software system and the frequency of communication among the team members dealing with these components. Based on this result, they suggest that technical dependencies can be used to predict communication frequency among team members. Similarly, ethnographic studies [11, 17] suggest that technical dependencies among pieces of code create "social dependencies" among software developers. That is, given two dependent pieces of code, the developers responsible for developing those pieces need to interact and coordinate in order to guarantee the smooth flow of work.

TESNA is a tool developed by Amrit and colleagues [39] that identifies Socio-Technical Structure Clashes (STSCs), instances where the social network of the development team does not match technical dependencies in the system architecture. The tool calculates the team's current social network from chat data rather than from code in a CM repository. It also does not provide technical dependency information at the same resolution (i.e. the artifact level) as Ariadne.

Finally, Cataldo and colleagues [3] used these ideas to develop a way of computing coordination requirements, i.e., who needs to coordinate with whom in order to accomplish a unit of development work. Valleto and colleagues [29] compare not the tasks, but the structure of the software organization itself with the structure of the software product (its dependencies).

2.4 Research Approach

These different empirical studies only confirm a unsurprising correlation: software engineers developing *dependent* pieces of code are more likely to engage in communication and coordination activities than developers working in unrelated pieces of code[1]. *What is surprising, however, is that such an obvious relationship has not yet been leveraged as much as possible to facilitate software development activities, especially because software systems allow the automatic identification of their dependencies.* Ariadne, the tool presented in this paper, aims to support this kind of leverage.

Several ethnographic studies also suggest that software developers in their daily work acknowledge this socio-technical relationship and, indeed, make use of it to get their work done. *In that regard, Ariadne aims to facilitate this software developers' invisible work.* We acknowledge that Ariadne will not replace software developers' approaches, it will *complement* them. Ariadne then is especially useful for software developers involved in large software development efforts who fail to make use of the relationship between coordination and technical dependencies. It is important to emphasize that Ariadne was inspired by examples of these situations identified in our studies [7-11] of large software projects. Ariadne is described in the following section.

3. ARIADNE

3.1 Features

Ariadne is implemented as a Java plug-in to the popular Eclipse IDE. As such, Ariadne is integrated as its own Perspective (called "Social Graph") and makes use of several of the services it provides. Initially, the plug-in uses Eclipse's SearchEngine APIs to extract dependencies from Java projects' source-code. These dependencies represent a static call-graph (i.e. before runtime). This graph is annotated with authorship information when Ariadne connects to the configuration management repository associated with a project and retrieves this information for each artifact selected for analysis. Finally, Ariadne calculates the sociogram for the project using the matrix multiplication method described in section 3.2.

Ariadne let users set artifacts to include in the dependency analysis under its "Element Includes" View in Eclipse. This view displays all artifacts for projects in the workspace currently linked to CM repositories. It lays out the artifacts for each project as a tree. That is, artifacts composed of other artifacts are displayed hierarchically. Toggleable checkboxes next to each artifact allow the user to selectively include only elements of interest. Once the user is satisfied with the artifacts he has selected, he can run the analysis by clicking the sociogram generator icon from the toolbar.

The current implementation can present technical and socio-technical dependency visualization at three different levels of abstraction, based on the programming language's hierarchy (e.g. packages, classes, methods in Java). Essentially, information is aggregated at each hierarchy level also to, potentially, average the different results provided by diverse call-graph extractors [23]. For instance, class dependencies are displayed as the aggregation of method dependencies (i.e., the call-graph).

3.2 Generating Social Dependencies from Technical Dependencies

Our approach for generating social dependencies from technical dependencies has been described in details elsewhere [9], therefore, we will only briefly describe our approach here. In Ariadne, dependencies are represented as static call-graphs. A call graph "summarizes the dynamic invocation relationships between procedures. The nodes of the call graph are the procedures in the program. An edge (pl, p2) exists if procedure pl can call procedure p2 from some *call site* within pl. Hence, each edge may be thought of as representing some call site in the program" [2]. A call graph, then, reveals the potential dynamic structure of a software system, although it can be derived using static analysis techniques. We then populate the call-graph with 'social information,' (in this case authorship information) to create an intermediate structure called a 'social call-graph' [11]. Social call-graphs are similar to ownership architectures [1], but they are built automatically.

The information from call-graphs and social call-graphs can be used to generate a sociogram, a graphical representation of a set of items, vertices, or nodes, connected to one another via links or edges [31]. For our purposes, this sociogram describes the dependencies between software developers through the code they write. Software developers can use these sociograms to find two important pieces of information: who they depend on and who depends on their work. As our previous work [9-11] illustrate, this situation is not uncommon in large software projects.

We can represent both graphs as matrices, where entries represent the strength of a connection between code and code, or developer and the code he authored, respectively. To infer the sociogram, we multiply the matrix produced from the social call-graph by the matrix produced from the call-graph to produce an author by code matrix, describing which code authors depend on [3]. Next, we multiply the result by the transpose of the social-call graph matrix to obtain an author by author matrix, representing the dependencies between authors (as a result of their code dependencies).

3.3 Architecture

Ariadne was initially implemented to analyze only Java projects and extract information from CVS repositories. We re-designed it to be general to support various source languages, configuration management (CM) systems, and visualizations. By default, Ariadne has no knowledge of the source language to be analyzed or the type of CM repository where the source-code is stored. Ariadne's architecture allows multiple dependency generators, CM tools, and visualizations may be installed at the same time. We leverage Eclipse's features to use the user's context in Eclipse to determine which code generator and CM subsystem is used to extract the relevant information to Ariadne.

This is achieved through the usage of a layered architecture with a Visualization layer on top of a CM and Dependencies APIs. Below this API, programming language and CM "plug-ins" can be found. The configuration management and dependency management API is used to isolate the programming language and configuration management tools from the visualizations provided by Ariadne. Through this approach, independent developers can contribute new functionality (configuration management tools and programming languages) to Ariadne, while reusing previous visualizations. And, at the same time, it is possible to easily design new visualizations to already supported programming languages and CM tools. In fact, this paper describes a new evaluation that we designed and evaluate to represent socio-technical dependencies.

[1]Note that, as other researchers [22] have pointed out, this relationship is not unique to software engineering.

3.4 Subsystems

Ariadne is divided into three main subsystems: program dependency, configuration management, and visualization. In this section we explain each.

3.4.1 Program Dependency Analysis

Ariadne has been designed to represent the relationship between programming language elements by using the composite design pattern [15]. This pattern allows us to represent part-whole hierarchies as well as treat individual and composite objects in the same way. Being able to represent hierarchies in different programming languages facilitates making Ariadne independent of programming language. Ariadne's API uses the concepts of Nodes and Edges to model any graph and programming language so that Nodes can represent program elements (such as methods, attributes, classes, and packages in Java) and Edges represent dependency relationships between these elements.

Using the program dependency part of our API, we have implemented a code dependency infrastructure that analyzes Java source code and Eclipse's manifest and "plugin.xml" files.

3.4.2 Configuration Management

Ariadne models CM repositories in a generic way that allow views of a project's data at one or many points in time, regardless of the CM system used. This module is integrated with the dependency generator module so that it is possible to find dependency information for any element being versioned. *Ariadne uses this information to query the code dependency generator module for any language elements in the region.*

Currently, we have CVS and SVN extractors that are used to automatically connect to a project's CVS or SVN repositories (using Eclipse's Team API) and extract CVS or SVN annotations (change and authorship information). The extraction results are parsed into Ariadne and used to create social call-graphs and, ultimately, sociograms.

3.4.3 Visualization

Ariadne's visualization subsystem allows developers to explore dependency information and authorship information. Ariadne's default visualization is a simple directed graph with nodes representing authors and edges representing dependencies between authors. Alternatively, the developer may implement his visualization of choice – that may be a line-oriented approach as in the SeeSoft project [12] , treemaps, or however else he chooses to visualize dependencies.

Our focus for this paper was to deviate from the default graph-based visualization Ariadne provides, choosing instead to implement and subsequently evaluate a new one. As such, we extended the visualization subsystem with a new visualization. It is important to note that, unlike the default visualization, our new visualization does not run within Eclipse. It exists instead as a standalone Java application, intended to occupy the user's whole screen. We give an overview of the visualization in the next section.

4. VISUALIZATION

Our new visualization takes a graph-based approach to visualizing a reduction of the dependency information collected by the tool. We implemented it using Prefuse, a Java-based visualization toolkit (http:///www.prefuse.org). In the past, we represented complete socio-technical dependency information as a series of three edges connecting a dependent author to the author they depend upon through the code units authored by each (see Figure 1). Our attempts to visualize the complete set of this information proved to be unmanageable for large software projects due to the number of connections and variability of the graph layout.

Recognizing the challenges of displaying all three elements of the socio-technical relationship, we removed one of the relationships (see Figure 2) reducing the number of connections needed to be displayed, but still allowing a consistent layout of the dependency information. The rationale for eliminating the C1 to C2 relationship has to do with the scenarios of usage that we identified for Ariadne in our previous work [9-11] which emphasize the work authors must undertake in order to determine the code and other developers that impact their own code.

The visualization interface allows users to easily reveal information about the technical dependency information, meaning no information is lost. The layout of this reduced dependency graph keeps important graph characteristics and benefits from a consistent layout that helps highlight information required to reason about coordination needs.

To take advantage of available screen real estate, Ariadne lays out dependency information in a type of table-based fashion placing the most numerous data items along the longest screen dimension. Called code units occupy the x-axis and authors occupy the y-axis, with both ordered alphabetically by default. Users can reorder the axes based upon queries against all the data and its associated meta-data. We draw connections from a dependent author to the code unit they are dependent upon and back to the author responsible for that code unit (Figure 3) and repeat for each set of socio-technical dependency information in the project (the connections explained in the previous paragraph, also see section 3.2). The color of each line (or dependency) denotes the directionality of the dependency and shares its color with the originating (dependent) author. For example, if A1 is blue, a blue line connecting A1 to C2 to A2 denotes an outbound dependency from A1's code (C1, not shown) to A2's code (C2, shown). The opacity of each line color denotes how many duplicate technical dependencies exist between two authors.

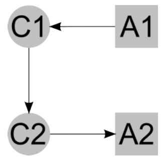

Figure 1. Old conceptual socio-technical dependency representation.

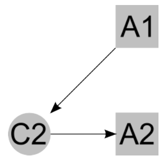

Figure 2. New conceptual socio-technical dependency representation.

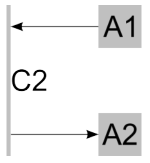

Figure 3. New visualization's socio-technical dependency representation.

Viewing dependency information using this hybrid table- and graph-based approach offers pattern recognition capabilities, easy filtering, and comparisons. An unfiltered overview of the dependency information allows us to show the state of dependencies for an entire project at once (Figure 4). From this perspective it is possible to recognize patterns in the way developers call other developers code, prominent code modules, and prominent authors even for a specific area of the code.

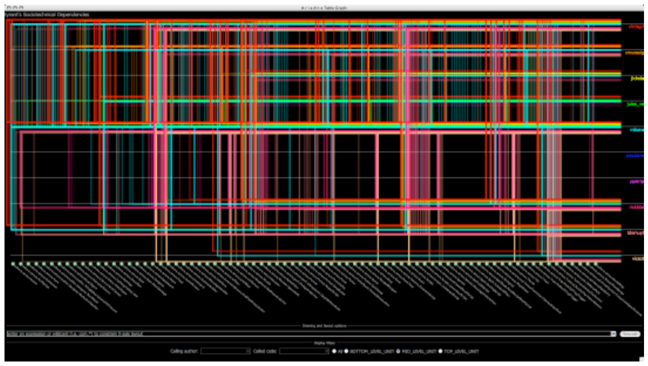

Figure 4. An overview of the socio-technical dependencies in the open-source Java project "Tyrant."

Filtering the overview by artifact reveals connections only from authors using that artifact (Figure 5). Managers and developers can focus on artifacts at different granularities that may be undergoing many changes in order to determine developers' progress. Focusing on an artifact may allow managers and developers to locate other developers affecting or affected by changes to that artifact.

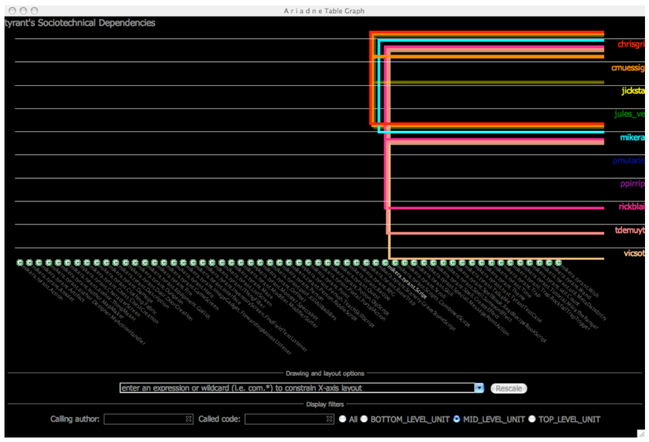

Figure 5. Filtering the overview to show socio-technical dependencies for the artifact "mikera.tyrant.Scripts."

Using an additive approach, we can compare the calls on code units made by one author with those made by another author. The user can click on authors' names to reveal only their dependency information (Figure 6). Ariadne's visualization technique preserves the ease of identifying connections between authors found in simple social network graphs of developers. Looking at only the y-axis, users can readily determine the inbound and outbound connections between a project's developers. The presence of a color corresponding to an author's name indicates an outbound dependency, while the presence of other authors' colors indicates an inbound socio-technical dependency from those other authors. While Ariadne's visualization makes a significant departure from a more traditional graph-based approach, it does not eliminate the advantages of that method of data display.

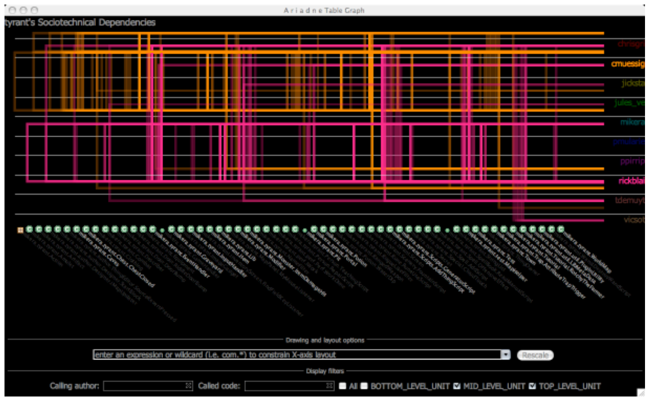
A r i a d n e Table Graph
tyrant's Sociotechnical Dependencies
cmuessig
rickbla
Drawing and layout options
enter an expression or wildcard (i.e. com.*) to constrain X-axis layout
Rescale
Display filters
Calling author:
Called code:
All
BOTTOM_LEVEL_UNIT
MID_LEVEL_UNIT
TOP_LEVEL_UNIT

Figure 6. Filtering the overview to show two authors' socio-technical dependencies.

5. EVALUATION

In order to assess the presentation, usability, and ease of learning of Ariadne's new visualization, we have evaluated it using Tufte's principles of information visualization [36, 37], the Heuristic Evaluation [34], and the Cognitive Walkthrough [38]. First, we analyzed it against principles of good information visualization to understand how well the visualization presents its intended information to the user for optimum understanding and analysis. Second we checked the interface against well-established usability principles with Nielsen's Heuristic Evaluation. Third, we evaluated the interface with the Cognitive Walkthrough, a method that is particularly good at focusing on the user's role, a priori assumptions, and what they can accomplish with and without training.

With each inspection method, we evaluated the tool with the help of at least two other of our colleagues who varied from session to session. For the most part, they had no experience using the new visualization. This unfamiliarity helped us to identify problematic assumptions about users' expectations and perceptions of the tool. In short, it helped to broaden the collective experience and expertise brought to bear on the evaluation.

5.1 Principles of Information Visualization

In this section we analyze Ariadne's visualization in terms of information visualization principles and explain how it meets or fails to meet each. A description of each principle can be found in [37]. For the sake of space, we only describe our analysis.

5.1.1 Comparisons

We have shown the comparisons that the new visualization affords in the following table:

Table 1. Comparisons afforded by the visualization and how to recognize them.

Comparison	How to locate in visualization
Intensity	Brightness of dependency
Depended upon code	Many dependencies intersecting artifact
Depended upon developers	Many dependencies – not in the developer's color – incoming to developer name
Amount of code called	Many dependencies in developer's

by developer	color spanning artifacts
Authors calling same code	Dependencies in multiple colors passing through same artifacts

The intensity of a dependency refers to the amount of calls one developer is making to another's code, which may indicate a significant need to coordinate work as intensity increases [11]. A module is more depended upon than another if developers are making more calls to it. A heavily depended upon code module may alert developers or managers to code re-use opportunities. Heavily depended upon code and authors, as well as the amount of code called by developers, can allow users to gauge code integration. The identification of developers calling the same artifacts can be useful for developers seeking help using those artifacts.

5.1.2 Causality, Mechanism, Structure, Explanation

First, the information presented in the visualization answers the question about why two developers may need to coordinate their work. The dependency line is the main indicator. It answers the "why" by highlighting the code module responsible for the social dependency. Second, developers and managers may be able to uncover why the project is slipping behind schedule. The absence of dependencies, or indications of weak dependencies (from the intensity), may indicate that developers have failed to integrate code before a deadline, for example.

5.1.3 Multivariate Analysis

The new visualization presents a number of dimensions from which to reason. The first is the intensity of the dependency, which as discussed earlier, may play a role in assessing the integration status of code. Second, the visualization presents called code along the horizontal axis and authors on the vertical axis. The visualization also uses color to distinguish authors from each other. The number of different colors also indicates the number of developers. Granularity is used by the visualization to help users distinguish between artifacts of varying resolutions.

5.1.4 Integration of Evidence

Ariadne's visualization uses text to convey code module names as well as developer names. In addition, images alongside each code module indicate its granularity. It is important to note that these correspond to the same images used by the Eclipse IDE to represent the granularity of the source-code construct (e.g. packages, classes, and methods). Graphic lines represent socio-technical dependencies.

5.1.5 Documentation

In its current state, the visualization clearly indicates that it displays socio-technical relationships for the name of the project under analysis. It is not so clear, however, that the data comes from a CM repository, nor that it comes from one point in time. Additionally, the visualization assumes that no gatekeeper controls check-ins. If a gatekeeper does in fact control check-ins, as in some open-source projects, users of the visualization will mistake code authored by others as code checked in by a gatekeeper.

5.1.6 Content

The important content in Ariadne's visualization are the socio-technical dependencies. Since these dependencies are the source of the communication breakdowns we observed in the data collected from our field studies [9-11], they are the necessary component in the visualization. As such, the tool dedicates the majority of the available screen space to the visualization of these dependencies.

5.2 Heuristic Evaluation

In addition to evaluating our new visualization as a visualization, we evaluated it as a user interface as well. As in section 5.1, we describe how our interface meets or fails to meet each usability heuristic. A complete description of each heuristic can be found [34]. Below, we just present the results of our evaluation.

5.2.1 Visibility of System Status

In general, we found that the visualization needs improvement with regards to reporting system status. For example, once the user loads a project dependency graph to analyze, there is no progress reporting bar alerting the user how much load time is left. This problem is compounded by the fact that large graphs can take several minutes to load. The user may in fact believe that an error has occurred and give up waiting for the tool to finish. Similarly, redrawing dependencies after the user has filtered data can be unnecessarily slow at times. Last, when hovering over dependencies to see more information, the visualization does not always highlight the dependency the user expected until after some delay.

While there are delays in the feedback presented to the user, the feedback itself is obvious. Generally, the user will manipulate the interface by filtering data to display only dependencies, code, or authors of interest. When the interface responds, the difference in the look of the interface is clear. Many items on the screen that were there before will not be there. For example, bright colors will fade into the black background, creating a stark contrast between the filtered out data and the data left on the screen.

5.2.2 Match Between System and Real World

Ariadne displays the name of the project, the code modules in the project, and the CM login names of the developers in the project. However, managers may not know the CM login names of their developers or the names of fine-grained code modules. The latter can be mitigated by filtering the visualization to see the code at a higher abstraction, such as packages in Java.

Since the tool is intended to be used in conjunction with Eclipse, we have reused Eclipse's icons for code granularity to more completely describe the code granularity of the artifacts as they appear on the horizontal axis.

5.2.3 User Control and Freedom

In its current state, the visualization does not support undo or re-do. We believe that because the visualization is exploratory and the user never really "manipulates" data – rather he just changes the view – these functions are not critical in the interim. The user can always clear a filter. But at the same time, as he performs many filtering operations, it may become difficult to remember the whole chain of filters he has applied. It also might be beneficial to give the user the freedom to rearrange data on the axes into a configuration he desires, and then save this configuration for future use. As we collect more data on the usage of the visualization and user preferences, we may add these capabilities.

5.2.4 Consistency and Standards

The only major inconsistency we found is how the visualization responds to filtering by typing and filtering by clicking the desired artifact, author, or dependency. When it detects a filter by click, the visualization highlights the results and all other elements fade to grey against the black background. However, when the tool detects a filter by typing, it highlights the dependency results but fails to fade out the names of the other labels (code and authors). This is clearly inconsistent behavior and may lead the user to believe he has not applied the filter correctly.

5.2.5 Error Prevention

The tool does not display error messages in its current state. The only place an error message would be useful is when the interface loads the project dependency graph. Because the visualization can take awhile to load, the user may not know whether there was an error loading the graph, or whether it has not yet completed loading. Ideally, an appropriate error message should not only tell the user that the graph is malformed, but *what about* the graph is malformed.

5.2.6 Recognition Rather than Recall

The most critical problem we found in this area is that it is not obvious to the user that he can click on a code or author to filter on only that object. We believe that a status bar could update when the user hovers over filter-enabled toggleable labels, for instance, to communicate what can be done with that object. Another option would be to have the mouse cursor change shape to indicate that the object is clickable.

5.2.7 Flexibility and Efficiency of Use

The closest thing to a shortcut as of now is dynamic single-character filtering with complex SQL-like queries. As the user enters each character into the filter text box, the visualization actively searches for matches and displays them. An avenue for future work, as mentioned earlier in section 5.2.3, is to allow the user to save configurations, including layout and filters, to speed up interactions with the tool.

5.2.8 Aesthetic and Minimalist Design

As discussed in 5.2.4, the filters provided by Ariadne do an excellent job of pruning information not of importance to the user. However, the inconsistency in the behavior of the filters results in clearly readable labels of no interest to the user. As a result, they should not be displayed.

5.2.9 Help Users Recognize, Diagnose, and Recover from Errors

The only errors we identified occurred when the visualization loaded a malformed graph file. As mentioned in section 5.2.5, the visualization should clearly indicate the malformed section(s) of the graph and provide a solution, such as re-running the dependency analysis plug-in, or notifying the user of errors and showing a subset of the information that the visualization managed to parse correctly. In this case, the visualization should notify users that they are only viewing part of the project.

5.2.10 Help and Documentation

While there is no documentation for the tool yet, it is definitely part of our future work. We are aware that the visualization takes some learning before one can be proficient with it. In the documentation we plan to cover how to select dependencies, perform filtering, explain the concept of dependency intensity, and show how to trace social dependencies from one author to another through the code.

5.3 Cognitive Walkthrough

For the second part of our interface inspection, we employed the Cognitive Walkthrough. In short, the Cognitive Walkthrough involves specifying tasks that users will attempt to accomplish with an interface and then analyzing the ease with which users can perform those tasks. Evaluators perform the analysis by asking a set of four questions at each step to uncover usability and learning issues. A complete description of the process and the questions asked at each step can be found in [38].

We constructed our tasks based on the data we collected during our field studies. We translated our observations into four scenarios that describe typical coordination problems in large software development projects with code being reused by different teams. These scenarios share a common theme: the usage of dependency information to facilitate software development tasks. They are described in details elsewhere [9], so in this section, we briefly describe each scenario and the problems with the interface we subsequently found.

5.3.1 Managers' Lack of Awareness of Evolving Social Dependencies

The goal in this scenario is for the manager of a software development team to find out whether or not a set of people have started to call each other's code. We have observed this need especially in both collocated and distributed teams. Briefly, the series of steps for accomplishing this goal are:

1. Select the project to analyze
2. Find the involved developers
3. Determine the presence/absence of connections between the developers

The first problem we noted was that, on first load of the interface, the action to select a project of interest is not obviously available. Instead, the immediately available actions on the interface are form and text boxes on the bottom for filtering and options to rescale the interface. The "Select Project..." button is hidden in the top left corner of the interface.

If users are able to find the correct button to open a project, we noticed that they might be confused about which files to open. The dialog filters out all files without the ".graph" extension, but as we realized, users might in fact be looking for files of type ".project."

Once the manager loads the correct file, there is no indication of progress for the load. Because loading a whole project can take some time, a manager might think that the program has crashed and, as a result, may close the window. There is also no error reporting in the case of an error while parsing the project file.

Next, we noticed some potential problems with finding involved developers. When managers go to find developers of interest on the vertical axis, we have made the assumption that the manager knows the developers' CM login names. This may be true or not, depending on the manager's involvement with his team. When looking for dependencies originating from the developers of interest, there is a chance that colors may not be distinguishable enough from each other. This will make matching authors to their dependencies almost impossible without filtering the data.

When determining the presence or absence of connections between developers, managers may find it difficult to tell which connections begin or terminate at the developer of interest. However, we mitigate this problem by always displaying incoming and outgoing dependencies, marked by individual colors, in the same order from author to author. This way, managers can easily tell when the incoming or outgoing dependencies for each author begin and end. Managers can also selectively view only developers of interest when determining integration status by clicking the name of each developer. However, it is not clear that developer names are clickable (see section 5.2.6).

5.3.2 Developer's Lack of Awareness of Evolving Code Dependencies

In this scenario, the developer wants to find out when others begin exercising his code, because he wants to make sure he will have enough time to fix his code in case an integration problem occurs. This was observed among collocated developers, but especially among distributed ones. We assume that the developer knows the name of the code of interest and may or may not know the developers who are calling the code. The steps involved are:

1. Select the granularity of the code of interest
2. Select the target code if the calling developers are not known and associate resulting dependencies with calling developers OR filter by code and developer if the calling developers are known and associate resulting dependencies with calling developers
3. Determine the recency of the dependencies

In step 2, if the developer does not know the names of the developers calling his code, he can either click on the code of interest or perform a search and filter. If he chooses the former, he will get good feedback because the interface will highlight connections through the code of interest and fade out the other connections. If no dependencies show up, then no one has started to call that code. In the case that the developer decides to search with the filter box, however, we discovered that it is impossible to tell if the dependency doesn't show up because the code does not exist, or because it has not yet been called by anyone. A status message should either indicate, "Code not found" or "Dependencies not found."

If the developer knows the name of the developers that should be calling his code, he can instead perform a filter on both code and authors. We have noted the problems with filtering by code, and they are the same for filtering by developer. However, not knowing whether a developer exists is likely to be less worrisome, at least compared to the problem with code, because there will generally be significantly fewer developers than code.

We have not yet implemented the ability to determine recency of the connections. One idea is to change the coloring semantics. The developer could theoretically define or choose a predefined coloring scheme based on a date parameter.

5.3.3 Developers Finding the "Right" Developer

The goal of the developer in this situation is to find the actual implementer of a specified interface instead of its dummy implementation. This is necessary because, in the organization that we studied [11], the software architect designs the interfaces and provides dummy implementations for all of them. However, a different software developer will be responsible for the actual implementation of these interfaces. Furthermore, we observed that developers were often not aware of who was consuming or providing services to their code. As in the previous scenario, we assume the developer knows the name of the interface of interest. The required steps are:

1. Select the granularity of the code of interest
2. Select the code of interest
3. Associate calls to the code with developers
4. Hover over each dependency to determine the calling code

We covered the problems associated with steps 1-3 in the previous sections.

One problem we uncovered in the last step is that the action for hovering over a dependency is not obviously visible on the interface. When the user hovers over a dependency, it highlights in green and pops up a tooltip that displays the calling code (in this case the implementation of the interface). Since there is no indication that one can hover over a dependency, the user will not know it is possible until he actually does it. However, we expect that users will move their mouse over the visualization as they analyze the data. As soon as they perform this action, the feedback should be clear, and the user should gain insight into the meaning of the hover and highlight. While the green highlight may be the same color as an author on the vertical axis, the user should recognize the color of the dependency before it was highlighted, and associate it with the correct developer.

5.3.4 Developers Finding "Similar" Developers

The developer's goal in this scenario is to find other developers who are calling the same code units in order to request help and divide up work among each other. This situation was identified because of the size of the organization studied and the organization-wide software reuse program adopted, which prescribed that software components should be reused as much as possible in the entire organization. That meant that, in some occasions, different software developers were reusing the same code – and facing the same problems – without being aware of each other. In this case, the assumptions are the same as in the last scenario. The steps are:

1. Select the granularity of the code of interest
2. Select the code of interest
3. Associate calls to the code with developers
4. Factor out developers who are not the user

Again, the problems with steps 1-3 have been covered in previous sections. We did not uncover any additional problems with this task.

6. DISCUSSION

Our evaluation has given us insight into aspects of the visualization that need improvement. In short, as designers, we need to make better use of color as a visual indicator to users of the visualization. In addition, our evaluations suggest that the tool should provide better feedback to users in the form of status indicators. In spite of these problems, however, we believe the new visualization provides a strong foundation of actionable dependency information from which developers and managers can better coordinate their work efforts. On top of this foundation, we can sufficiently address the concerns we have uncovered in our evaluations.

The major improvement we need to make to the visualization is its color-picking algorithm. In order to support tasks with the tool, the new visualization relies on the user's ability to use color to identify developers and dependencies of interest as well as make comparisons and find patterns. We have identified two major issues subsumed under color usage. The first is the visualization's ability to choose colors that are distinguishable enough as the number of developers increases. One approach is to use color design guidelines [32]. Such guidelines work to mitigate visual problems related to poor color selection. They are typically based on past experiences and best practices, not unlike, the information visualization principles we have explored as part of our evaluation. Another approach would be to let users define and tune their own colors, or color palettes, through the use of a color parameter.

Our second issue of concern related to color is the visualization's inability to support colorblind users. A significant number of the population views color differently from the norm. Seemingly identifiable colors to a normal color viewer can appear unidentifiable to people with color blindness. In addition to relying on good color use guidelines, a promising approach is to use algorithms to modify colors so they are suitable for colorblind users as described in [33, 35].

In addition to color selection, our inspections have also indicated the need to clearly convey status information. For example, we have learned from the heuristic evaluation and the cognitive walkthrough that the visualization does not convey progress when loading graph data. Moreover, actions such as clicking developers or artifacts to filter the information presented are not obviously available. While as designers we have become familiar with the visualization's behavior, we cannot expect the same from novices.

We believe the visualization we have presented in this paper strongly facilitates understanding and reasoning about dependencies between source-code artifacts, and as a result, the developers implementing those artifacts. The new visualization provides consistent layouts for analysis and the ability to display many dependencies along the longest screen dimension. A high-level overview of the project can reveal patterns in code usage and integration. Depending on the type of user (developer or manager) and his coordination needs, the user can filter against known code modules or developers to reveal fine-grained information of interest. While future work certainly suggests improvements in the areas of color selection and status indication, we believe these problems are incidental to our method for visualization. In other words, the evaluations we have conducted have not suggested a need to change the fundamental ways in which we visualize dependencies. Rather, the evaluations suggest *additions* to the visualization that can improve usability.

7. CONCLUSIONS AND FUTURE WORK

This paper described Ariadne, a plug-in for Eclipse, that translates technical dependencies in source-code to social dependencies between developers implementing that code. This work has been motivated by our own empirical studies, and complemented by studies and theoretical predictions from other researchers. All these studies suggested scenarios, and as a result, developers' tasks used as a basis for the evaluation presented here. Unsatisfied with our previous graph-based method of visualizing dependencies, we have contributed a revised visualization. We have evaluated this visualization with inspection methods appropriate for both visualizations and user interfaces. The results of our evaluation support our claim that the visualization can be used to support the tasks derived from our empirical studies. Moreover, the evaluation methods we have adapted to Ariadne and documented in this paper not only serve our own research interests, but provide a basis for other researchers evaluating visual software tools as well.

As part of our future work, we plan to address the problems with respect to color and status uncovered by our evaluation here. After that, we plan to test the tool with end users to further assess its utility and usability.

8. REFERENCES

[1] Bowman, I.T. and Holt, R., Reconstructing Ownership Architectures To Help Understand Software Systems. in International Workshop on Program Comprehension, (Pittsburgh, PA, USA, 1999), IEEE Press, 28-37.

[2] Callahan, D., Carle, A., Hall, M.W. and Kennedy, K. Constructing the Procedure Call Multigraph. IEEE Transactions on Software Engineering, 16 (4). 483-487.

[3] Cataldo, M., Wagstrom, P.A., Herbsleb, J.D. and Carley, K.M. Identification of Coordination Requirements: implications for the Design of Collaboration and Awareness Tools 20th Conference on Computer Supported Cooperative Work, ACM Press, Banff, Alberta, Canada, 2006.

[4] Conway, M.E. How Do Committees invent? Datamation, 14 (4). 28-31.

[5] Curtis, B., Krasner, H. and Iscoe, N. A field study of the software design process for large systems. Communications of the ACM, 31 (11). 1268-1287.

[6] Dahlstedt, A.G. and Persson, A., Requirements Interdependencies - Molding the State of Research into a Research Agenda. in The 9th International Workshop on Requirements Engineering: Foundation for Software Quality - REFSQ'03, held in conjunction with CAiSE, (Velden, Austria, 2003).

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

[8] de Souza, C.R.B., Hildenbrand, T. and Redmiles, D., Towards Visualization and Analysis of Traceability Relationships in Distributed and Offshore Software Development Projects (to appear). in Software Engineering Approaches for Offshore and Outsourced Development, (Zurich, 2007), Springer, 1182-1199.

[9] de Souza, C.R.B., Quirk, S., Trainer, E. and Redmiles, D. Supporting Collaborative Software Development through the Visualization of Socio-Technical Dependencies (to appear) ACM Conference on Supporting Group Work, ACM Press, Sanibel Island, FL, 2007.

[10] de Souza, C.R.B. and Redmiles, D. 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, Springer, Limerick, Ireland, 2007.

[11] de Souza, C.R.B., Redmiles, D., Cheng, L.-T., Millen, D. and Patterson, J., How a Good Software Practice thwarts Collaboration - The Multiple roles of APIs in Software Development. in Foundations of Software Engineering, (Newport Beach, CA, USA, 2004), ACM Press, 221-230.

[12] Eick, S.G., Steffen, J.L. and Sumner, E.E. SeeSoft -- tool for visualizing line oriented software. IEEE Transactions on Software Engineering, 11 (18). 957-968.

[13] Ferrante, J., Ottenstein, K.J. and Warren, J.D. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS), 9 (3). 319-349.

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

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

[16] Ghezzi, C., Jazayeri, M. and Mandrioli, D. Fundamentals of Software Engineering. Prentice Hall, 2003.

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

[18] Herbsleb, J.D. and Grinter, R.E. Architectures, Coordination, and Distance: Conway's Law and Beyond. IEEE Software. 63-70.

[19] Kiczales, G. Beyond the Black Box: Open Implementation. IEEE Software, 13 (1). 8-11.

[20] Larman, G. Protected Variation: The Importance of Being Closed. IEEE Software, 18 (3). 89-91.

[21] MacCormack, A., et al. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code Harvard Business School Working Papers, Harvard University, Cambridge, MA, 2004, 40.

[22] Morelli, M.D., et al. Predicting Technical Communication in Product Development Organizations. IEEE Transactions on Engineering Management, 42 (3). 215-222.

[23] Murphy, G., et al. An Empirical Study of Static Call Graph Extractors. ACM Transactions on Software Engineering and Methodology, 7 (2). 158-191.

[24] Parnas, D.L. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, 15 (12). 1053-1058.

[25] Podgurski, A. and Clarke, L.A., The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance. in Symposium on Software Testing, Analysis, and Verification, (1989), 168-178.

[26] Sosa, M.E., et al. 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.

[27] Spanoudakis, G. and Zisman, A. Software Traceability: A Roadmap. in Chang, S.K. ed. Handbook of Software Engineering and Knowledge Engineering, World Scientific Publishing Co., 2004.

[28] Stafford, J.A. and Wolf, A.L. Architecture-Level Dependence Analysis for Software Systems. International Journal of Software Engineering and Knowledge Engineering, 11 (4). 431-453.

[29] Valleto, G., et al. Using Software Repositories to Investigate Socio-technical Congruence in Development Projects Workshop on Mining Software Repositories, ACM Press, Minneapolis, 2007.

[30] Vieira, M.R.E. and Richardson, D.J., The Role of Dependencies in Component-Based System Evolution. in International Workshop on Principles of Software Evolution, (Orlando, Florida, 2002), 62-65.

[31] Wasserman, S. and Faust, K. Social Network Analysis: Methods and Applications. Cambridge University Press, Cambridge, UK, 1994.

[32] Chisholm, W., et al. W3C Web Content and Accessibility Guidelines 1.0, 1999

[33] Jefferson, L. and Harvey, R. 2006. Accommodating color blind computer users. In Proceedings of the 8th international ACM SIGACCESS Conference on Computers and Accessibility (Portland, Oregon, USA, October 23 - 25, 2006). Assets '06. ACM Press, New York, NY, 40-47.

[34] Nielsen, J.K. 1994. Heuristic Evaluation. In Usability Inspection Methods, J.K. Nielson, & R.L. Mack, Eds. Wiley, NY.

[35] Song, J., et al. Digital Item Adaptation for Color Vision Variations. In SPIE, Conf. Human Vision and Electronic Imaging VIII, volume 5007, pages 96--103, 2003

[36] Tufte, E. 1990 Envisioning Information. Graphics Press, Cheshire, CT.

[37] Tufte, E. 2006. Beautiful Evidence. Graphics Press, Cheshire, CT.

[38] Wharton, C. W., Reiman, J., Lewis, C. & Polson, P. 1994. The cognitive walkthrough method: A practitioner's guide. In Usability Inspection Methods, J.K. Nielsen, & R.L. Mack, Eds. Wiley, NY.

[39] Amrit, C. and van Hillegersberg, J. Detecting Coordination Provlems in Collaborative Software Development Environments. (to appear) Information Systems Management special issue on “Collaboration Challenges: Bridging the IT Support Gap.”