Usability Inspection Method-based Analysis of a Socio-Technical Visualization Tool

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

Stephen Quirk
University of California, Irvine
squirk@ics.uci.edu

Cleidson de Souza
Universidade Federal do Pará, Brazil
cdesouza@ufpa.br

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

June 2012
ISR Technical Report # UCI-ISR-12-6

Usability Inspection Method-based Analysis of a Socio-Technical Visualization Tool

ISR Technical Report #UCI-ISR-12-6
June 2012

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

[1]*Institute for Software Research*
University of California, Irvine
Irvine, CA, USA – 92697
[etrainer, squirk, redmiles]@ics.uci.edu

[2]*Faculdade de Computação*
Universidade Federal do Pará
Belém, PA, Brazil – 66075
cdesouza@ufpa.br

Abstract

Ariadne is a visualization tool that allows end users to explore the socio-technical relationships in software development projects. Essentially the visualization is a variant of a social network graph. It is based on the observation that dependencies between software components create dependencies between the developers implementing those components. This relationship emerged in our own and other researchers' field studies of software projects. Large software development projects require management of dependencies by managers and developers to ensure the smooth coordination of work. We sought to evaluate our visualization to assess its utility. Although we had some informal trials with potential end users, we sought a deeper analysis before further refinement of the tool and evaluation on a larger scale. Usability inspection methods provided one potential avenue. Moreover, such inspection methods yield a kind of rationale not directly derived from human subjects evaluations. We report on the application of these inspection methods, the results of evaluating Ariadne in particular, and implications for evaluating visual information interfaces.

1. Introduction

It has been long recognized that breakdowns in communication and coordination efforts constitute a major problem in collaborative software development [8]. 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.

Parnas was one of the first researchers to recognize the relationship between software dependencies and coordination: he suggested that by reducing dependencies among software development artifacts, it is possible to reduce developers' dependencies on one another, creating a managerial advantage [12, 19]. Nowadays, this is a well-known argument among researchers and practitioners that can even be found in textbooks [10].

Conversely, but also supporting this relationship between dependencies and coordination, Conway [6] postulated that the structure of a software system would reflect the communication needs of the people performing the work. In short, while Parnas argues that dependencies *shape* the coordination and communication activities software developers perform, 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.

Both Parnas' and Conway's arguments have been validated by a host of different empirical studies [1, 4, 8, 12, 21, 24], including our own [9].

Ariadne's visualization, the target of our evaluation in this paper, was created with the aim of reducing the acknowledged gap between software dependencies and coordination by exploring socio-technical relationships to support software developers' activities. During early development of the tool, we performed two key field studies, each 2-3 months in duration, that provided us insight into several types of communication and coordination problems in distributed software development projects. Of these issues, we derived several representative scenarios that revealed the types of dependency relationships managers and developers need to understand in order to coordinate their work. Next, we designed and implemented the visualization.

The visualization went through iterative development and was demonstrated to colleagues and visiting researchers for general suggestions and improvement. Initially, the visualization used a graph-based approach traditionally used by practitioners in the social network analysis domain [25], 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 inconsistency of the graph layout. We therefore began experimentation with a new visualization. In order to keep the visualization linked to human needs, we applied several usability inspection methods and cognitive theories to evaluate it against typical usage tasks we observed earlier. This paper reports on our inspections and suggests important avenues for future work.

The rest of this paper is structured as follows. In the following section we briefly describe the process by which Ariadne infers dependencies between developers based on the code they write. Next, in section 3, we present Ariadne's visualization. We follow up in sections 4 and 5 with the results of our evaluation and discuss implications for evaluating visual information interfaces. We conclude in section 6.

2. Ariadne's process

Ariadne uses APIs from the popular Eclipse IDE to infer dependencies between developers based on the code they write. It calculates dependencies between source-code artifacts before run-time. As such, these dependencies represent a static call-graph. Ariadne annotates this graph with authorship information for each line of code by connecting to the project's configuration management repository. Finally, Ariadne calculates a sociogram [25], representing dependencies between developers, using a matrix multiplication method described in [4, 9].

The visualization is a stand-alone application that users launch from the development editor. In the early stages of design, this allows us to test the tool with publically available projects and refine it further. The intended users of Ariadne, however, will want to see the visualization in the context of current software development activities. We intend to more carefully explore this tension as we conduct eventual trials with human subjects.

3. Visualization

Ariadne's 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 (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 (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] which emphasize the work authors must undertake in order to determine the code and other developers that impact their own code.

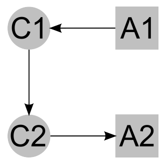

Figure 1. Old conceptual socio-technical dependency representation.

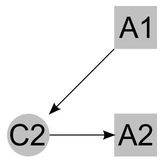

Figure 2. New conceptual socio-technical dependency representation.

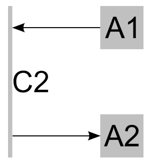

Figure 3. New visualization's socio-technical dependency representation.

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 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. The visualization lays out code units organized by package, much how a programmer or manager might expect to see them in a development editor. To see dependencies within these packages (Figure 4), users can Ctrl+click on a package. Similarly they can click on classes to see method dependencies.

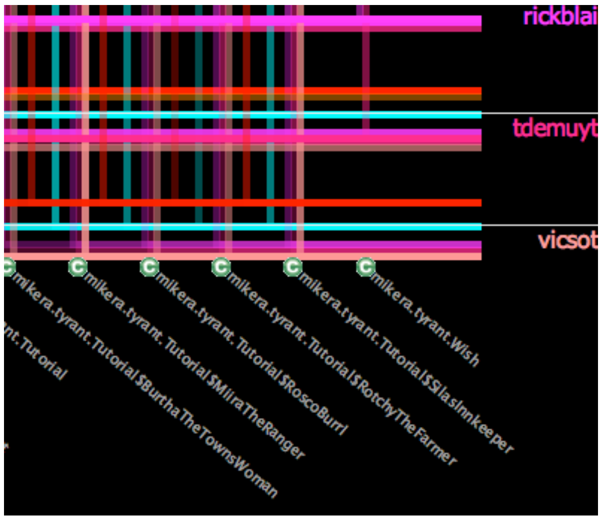
rickblai
tdemuyt
vicsot
nt.Tutorial
mikera.tyrant.Tutorial$BurthaTheTownsWoman
mikera.tyrant.Tutorial$MiraTheRanger
mikera.tyrant.Tutorial$RoscoBurrl
mikera.tyrant.Tutorial$RotchyTheFarmer
mikera.tyrant.Tutorial$SlasInnkeeper
mikera.tyrant.Wish

Figure 4. Closeup of socio-technical dependencies in the "main" package of open-source Java project "Tyrant."

Users can also 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 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.

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. 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.

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, as indicated by our field studies [9]. Focusing on an artifact may allow managers and developers to locate other developers affecting or affected by changes to that artifact.

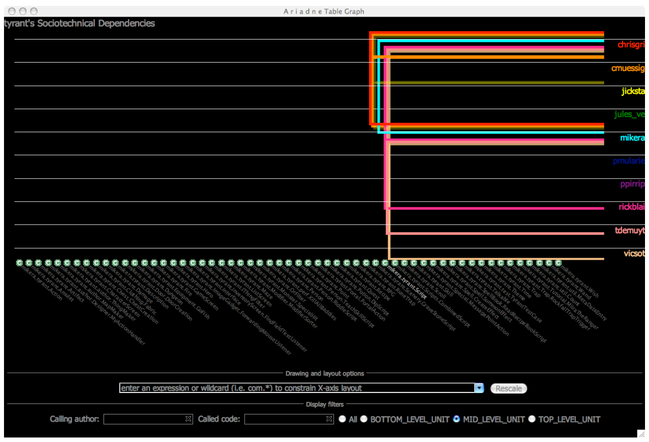

Figure 5. Filtering the overview to show socio-technical dependencies for the artifact "mikera.tyrant.Scripts."

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.

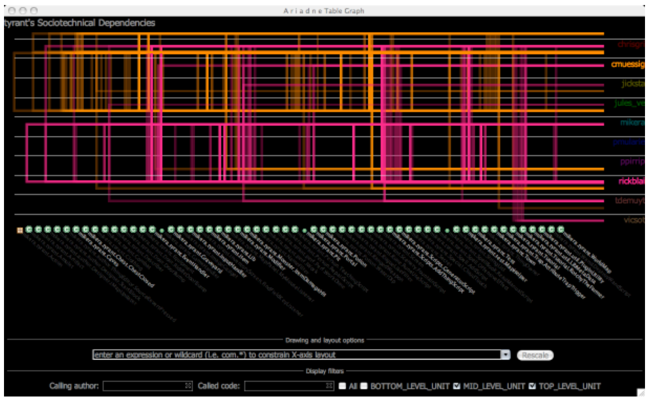

Figure 6. Filtering the overview to show two authors' socio-technical dependencies.

4. Application of usability inspection methods

In order to assess the presentation, usability, and ease of learning of Ariadne's visualization, we evaluated it using the Heuristic Evaluation [17], the Cognitive Walkthrough [26], and the Cognitive Dimensions of Notations [11]. First, we checked the interface against well-established usability principles with Nielsen's Heuristic Evaluation. Second, 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. Third, we used Cognitive Dimensions to uncover further mentally demanding operations.

We performed each inspection method with the help of four research colleagues. 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.

4.1. Heuristic evaluation

In this section we describe how the visualization meets or fails to meet each usability heuristic. A complete description of each heuristic can be found in [17]. Below, we just summarize the evaluation.

4.1.1. 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 contrast between the filtered out data and the data left on the screen.

4.1.2. 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 reused Eclipse's icons for code granularity to more completely describe the code granularity of the artifacts as they appear on the horizontal axis.

4.1.3. User control and freedom. Currently, 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 they just change the view – these functions are not critical in the interim. The user can always clear a filter. But at the same time, as they perform many filtering operations, it may become difficult to remember the whole chain of filters they have 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. For example, managers may want to see dependencies between teams rather than individual developers themselves. Demonstrations of Ariadne's visualization to colleagues and visiting researchers in software engineering have corroborated this idea. As such, we have marked this as an important addition to the next iteration of the tool.

4.1.4. 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 may lead the user to believe they have not applied the filter correctly.

4.1.5. Error prevention. The visualization only allows users to load files of the type ".graph" to prevent errors that may occur when loading a project.

4.1.6. Recognition rather than recall. The most critical problem we found is that it is not obvious to the user that they can click on a code or author to filter on only that object. Instead, 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.

4.1.7. Flexibility and efficiency of use. The closest thing to a shortcut 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 4.1.3, is to allow the user to save configurations, including layout and filters, to speed up interactions with the tool.

4.1.8. Aesthetic and minimalist design. As discussed in 4.1.4, the filters provided by Ariadne do a sufficient 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.

4.1.9. Help users recognize, diagnose, and recover from errors. The only errors we identified occurred when the visualization loaded a malformed graph file. 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.

4.1.10. 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, and show how to trace social dependencies from one author to another through the code.

4.2 Cognitive walkthrough

For the second part of our interface inspection, we employed the Cognitive Walkthrough. A complete description of the process and the questions asked at each step can be found in [26]. 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.

We constructed our tasks based on the data we collected during our field studies [9]. We categorized our observations and distilled them into four scenarios that describe coordination problems in large software development projects with code being reused by different teams. These scenarios share a common theme: software developers' usage of dependency information to facilitate software development tasks. In the interest of space, we report on the analysis of only one task here.

4.2.1. Developer's lack of awareness of evolving code dependencies. In this scenario, the developer wants to find out when others begin exercising their code, because they want to make sure they will have enough time to fix code in case an integration problem occurs. This was observed among both collocated and distributed developers. 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; and
3. Determine the recency of the dependencies.

In step 2, if the developer does not know the names of the developers calling his code, they can either click on the code of interest or perform a search and filter. If they choose the former, they 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."

Next, we noticed some potential problems with associating dependencies with developers. 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 further.

If the developer knows the name of the developers that should be calling their code, they 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.

4.3 Cognitive dimensions of notations

The Cognitive Walkthrough uncovered issues that made learning how to use the visualization difficult. We chose to complement this analysis with the Cognitive Dimensions of Notations Framework [11] (referred to hereafter as "Cognitive Dimensions") in order to further uncover mentally demanding operations with the visualization. Cognitive Dimensions provides a vocabulary for analyzing the usability of tools, programming languages, and environments that has been used to evaluate systems such as the Z formalism in TranZit [16]. Two other examples include [15] and [7]. Like the other inspection methods presented here, Cognitive Dimensions are designed for non-usability specialists and can be applied in the early stages of design before experiments with human subjects.

Similar to [16] we used the Cognitive Dimensions Questionnaire Optimised for Users [3] as a starting point to identify the relevant dimensions as a basis for the evaluation. The questionnaire clearly presents the concepts and introduces a set of questions that map to each cognitive dimension. We used these questions and details from [11] to complete our analysis. In this section we present a summary of the evaluation.

4.3.1. Visibility and juxtaposability. In general, it is easy to tell what has been changed or created. User actions including filtering by author or code will result in only those elements and dependencies displayed on the screen after some delay. During an update of the visualization by the user, status indicators on top of the window display actions being performed by the visualization (e.g. updating axes and drawing dependencies).

Calling code (code that calls the modules on the x-axis) is more difficult to see because it is not explicitly represented as an element of the visualization. Instead, users may hover over dependencies for a tooltip that displays this information.

Users can view combinations of elements (authors and code) at the same time by using the filtering mechanism.

4.3.2. Viscosity. Changes to the visualization mean changes in the way the data is presented, since users will always be performing some sort of filtering of the information. As such, making a change is as simple as clicking on objects or using a text-based search to display only information of interest on the visualization

As explained in our Heuristic Evaluation, changes that undo previous filtering may be complicated, especially if the user performs a large chain of filters and decides to undo only a small set of them. One solution, but not the only, might be to display the changes in a list fashion, similar to image editing programs such as Adobe Photoshop, where users can click on individual actions they have performed and consequently undo them.

4.3.3. Diffuseness. The visualization's notation allows us to display all the relationships we have identified in our field studies and surveys of related literature, namely the relationship between people based on the code they write.

Calling code takes more space to describe because it can only be viewed by hovering over a particular dependency.

4.3.4. Hard mental operations. When the number of authors is large, the number of distinct colors the user must keep track of becomes daunting, because some colors are too similar to others to really distinguish on the visualization. In our future work, we intend on adding the capability to display teams on the y-axis in addition to authors, effectively aggregating team members and reducing the total number of colors as a result.

Users can reduce the difficulty in understanding multiple sets of dependencies between different authors and code by filtering the data to display only connections of interest. If the user performs a text-based filter for called code and the code does not show on the display, it is difficult to tell if the code has not been called or the code does not exist.

4.3.5. Closeness of mapping. The notation is highly related to the information the visualization is intended to convey, namely dependencies between developers through code. Labels of code represent actual artifacts in the project, keeping the same naming scheme used for Java, namely packages, classes, and methods. Labels with developer names represent CM logins. These labels can be augmented with real names when such information is available, as in environments like IBM's Jazz [13]. The bracket metaphor to represent dependencies is different from traditional representations such as matrices or sociograms, and requires some initial explanation.

4.3.6. Role expressiveness. It is easy to identify each component in the notation insofar as the dependencies are distinguishable and matchable to the developers the user has identified as relevant.

4.3.7. Hidden dependencies. Dependencies are first class objects in this visualization. Instead of assuming that some dependencies are more important than others, we allow users to find the ones of interest by performing filtering operations. Dependencies between authors are clear, but dependencies between code artifacts require an extra step to find the calling code. The visualization provides this as an easily readbale tooltip text, however.

4.3.8. Premature Commitment. The visualization does not assume a specific order of operations. Instead, users can view information and perform filters in any order they like.

4.3.9 Consistency. See our description of consistency in section 4.1.4.

5. Discussion

The combination of inspection methods allowed us to tease out the most important problems with the visualization. For example, both the Cognitive Walkthrough and Cognitive Dimensions analyses pointed out problems with color picking. Possible solutions include using general color design guidelines [5] and selecting colors to support colorblind users [14, 20]. The Heuristic Evaluation and Cognitive Dimensions revealed the potential need to allow users to undo certain filtering actions in order to trace back their steps, as well as the option to view different configurations of developers (into teams, for example) and system components. All three methods suggested the need to improve feedback, whether to indicate that specific dependencies have not been created, to display the calling code for a given dependency, or to show progress bars when the visualization undergoes a screen refresh.

Each usability inspection has its particular focus, so it is not surprising that the problems we found were problems the methods were intended to reveal. The Cognitive Walkthrough and Cognitive Dimensions focus on actions with the visualization that are mentally demanding. Accordingly, they revealed problems with keeping track of different colors and filters applied across use of the visualization. The Heuristic Evaluation, serving as a broad checklist of good usability principles, reinforced these findings and helped to identify improvements to be made in the future (e.g. help and documentation and correction of visual inconsistencies). Although not reported on in this paper, we complemented the Heuristic Evaluation with a general visual inspection, using Tufte's principles of information visualization [22, 23].

The four analyses in total have allowed us to identify problems in the early stage of the development of Ariadne, before trials with human subjects. Eventually, we will run new trials with human subjects, though, generally speaking, human subject evaluations yield only performance data and not rationale that may affect design, especially in the early stages of design.

Some experimenters obtain rationale through Talk Aloud methods. Nielsen and colleagues provide a recent, detailed discussion of applying this method and extensions to limit certain biases [18]. The rationale obtained in Talk Aloud protocols is expensive in terms of obtaining subjects and performing the subsequent extensive analysis. The complexity and cost make it less appealing to early design.

Interestingly, some of the original authors of the Cognitive Walkthrough applied it to a visual interface [2]. The work used the Cognitive Walkthrough to eliminate categorically different design alternatives for a domain specific visual language. In our work, we seek to refine our design further based on the results; refining design decisions at a lower level. We also seek to complement the one inspection method with others, a total of four as mentioned above.

6. Conclusions and future work

This paper described Ariadne, a visual software tool that translates technical dependencies in source code to social dependencies between developers implementing that code. Ariadne has been motivated by our own empirical studies of software development projects and others'.

We chose to evaluate the visual interface with usability inspection methods. To a degree, this approach is somewhat novel as these methods are normally applied to *user interface* components and not so often to workspace or *information interface* components.

In conclusion, the inspection findings will lead us to improve the design of Ariadne before additional testing with human subjects. Moreover, the findings were sufficient to confirm the usefulness of these inspection methods in early design. Finally, inspection methods yield design explanations, answering questions about how and why an interface can be used to achieve its intended objectives.

Our work is a mid point for researchers interested both in visual interfaces to socio-technical data and evaluation methods for visual information interfaces.

7. References

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

[2] Bell, B., Rieman, J., and Lewis, C. 1991. Usability testing of a graphical programming system: things we missed in a programming walkthrough. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Reaching Through Technology* (New Orleans, Louisiana, United States, April 27 - May 02, 1991). S. P. Robertson, G. M. Olson, and J. S. Olson, Eds. CHI '91. ACM, New York, NY, 7-12.

[3] Blackwell, A F, Green, T.R.G (2000), 'A cognitive Dimensions Questionnaire Optimised for Users. " in A.F.

Blackwell & E. Bilotta (Eds.) Proceedings of the twelfth Annual Meeting of the Psychology of Programming Interest Group, 137-152

[4] 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.

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

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

[7] Cox K (2000), 'Cognitive Dimensions of Use Cases – feedback from a student questionnaire'. In A F Blackwell, E Bilotta (Eds). Proceedings of the twelfth Annual Meeting of the Psychology of Programming Interest Group, 99-122.

[8] 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.

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

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

[11] Green, T. (1989), "Cognitive Dimensions of Notations", In A. Sutcliffe & L. Macaulay (Eds.), People and Computers V Proceedings of HCI'89, Cambridge University Press.

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

[13] Hupfer, S., Cheng, L., Ross, S., and Patterson, J. 2004. Introducing collaboration into an application development environment. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work* (Chicago, Illinois, USA, November 06 - 10, 2004). CSCW '04. ACM, New York, NY, 21-24.

[14] 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.

[15] Kadoda G (2000), 'A Cognitive Dimensions view of the differences between designers and users of theorem proving assistants'. In A F Blackwell & E Bilotta (Eds), Proceedings of the twelfth Annual Meeting of the Psychology of Programming Interest Group, 33-44.

[16] Khazaei, B. and Triffitt, E. 2002. Applying cognitive dimensions to evaluate and improve the usability of Z formalism. In *Proceedings of the 14th international Conference on Software Engineering and Knowledge Engineering* (Ischia, Italy, July 15 - 19, 2002). SEKE '02, vol. 27. ACM, New York, NY, 571-577.

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

[18] Nielsen, J., Clemmensen, T., and Yssing, C. 2002. Getting access to what goes on in people's heads?: reflections on the think-aloud technique. In Proceedings of the Second Nordic Conference on Human-Computer interaction (Aarhus, Denmark, October 19 - 23, 2002). NordiCHI '02, vol. 31. ACM, New York, NY, 101-110.

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

[20] 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

[21] 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.

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

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

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

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

[26] 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.