Bridging the Gap between Technical and Social Dependencies with Ariadne

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

[etrainer, squirk, cdesouza, redmiles]@ics.uci.edu

ABSTRACT

One of the reasons why large-scale software development is difficult is the number of dependencies that software engineers need to face: e.g., dependencies among the software components and among the development tasks. These dependencies create a need for communication and coordination that requires continuous effort by software developers. Empirical studies, including our own, suggest that technical dependencies among software components create social dependencies among the software developers implementing these components. Based on this observation, we developed Ariadne, a Java plug-in for Eclipse. Ariadne analyzes a Java project to identify program dependencies and collects authorship information about the project by connecting to a configuration management repository. Through this process, Ariadne can "translate" technical dependencies among software components into social dependencies among software developers. This paper describes the design of Ariadne, how it identifies technical dependencies among software components, how it extracts information from configuration management systems and, finally, how it translates this into social dependencies. Ariadne's purpose is to create a bridge between technical and social dependencies.

Categories and Subject Descriptors

H.4.1 [Information Systems Applications]: Office Automation – Groupware; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces - Computer-supported cooperative work.

General Terms

Design, Human Factors

Keywords

Collaborative software development, program dependencies, social dependencies.

1. INTRODUCTION

Researchers and practitioners have long recognized that breakdowns in communication and coordination efforts constitute a major problem in software development [4]. One of the reasons for this problem is the large number of dependencies that any software development effort involves: dependencies among activities in the development process and dependencies among different software artifacts. To overcome this problem, the field of software engineering has developed tools, approaches, and principles to manage dependencies. Configuration management and issue-tracking systems are examples of such tools. The adoption of software development processes ([11, 22]) exemplifies an organizational approach [9] to managing dependencies. Finally, information hiding [23] illustrates a fundamental principle that has been implemented as several mechanisms in programming languages, e.g. interfaces and polymorphism [18].

In any one of these cases, the underlying goal is the same, to make dependencies more manageable. By minimizing dependencies it is possible to reduce the required communication and coordination of software developers. This relationship between coordination and dependencies has long been recognized. Parnas [23], for instance, recognized over 30 years ago that the principle of information hiding also brings managerial advantages: by dividing the work in independent modules, it is also possible to assign the implementation of these modules to different developers that can work on them in parallel. More recent ethnographic studies (e.g., Grinter [14] and de Souza et al. [7]) found that technical dependencies in source code create "social dependencies" among software developers. That is, given two dependent pieces of code, the developers responsible for developing each piece need to interact and coordinate in order to guarantee the smooth flow of work. In a quantitative approach, Morelli, Eppinger and Gulati [20] found out that these same dependencies can be used to predict communication frequency among team members in a manufacturer of electrical technologies:

> "Analyzing the frequency of each communication linkage reveals that nearly all of the frequent and most of the occasional coordination-type communications were predicted. … Such predictability suggests that regularly occurring communication linkages could be reliably planned with this project."

Later, similar results were found in the software development industry in a study of a telecommunications organization [24].

Despite this acknowledged relationship between dependencies and communication and coordination needs, this relationship has not been explored to facilitate and understand software development activities. Software development is indeed a strong candidate for exploring this relationship since (i) dependencies among software components can be automatically identified, and (ii) software is malleable, i.e., their dependencies, if so desired, can be more or less easily changed, and consequently the

coordination of those developing it[1]. Ariadne, a plug-in for Eclipse, aims to fill this gap and explore this socio-technical relationship. In this paper, we describe Ariadne's underlying architecture and API. By identifying these "social" dependencies, Ariadne is able to identify developers who are more likely to be communicating, as well as, developers whose similar dependencies make them likely to collaborate. Furthermore, it can even facilitate expertise identification [19] [8].

The rest of the paper is organized as follows. We begin by presenting the three types of dependencies that Ariadne supports, namely, technical, socio-technical, and social dependencies. More importantly, we describe our approach to extract program dependencies from the source code and how from code dependencies, we infer social dependencies between software developers. In the following section, we describe Ariadne's architecture, including its configuration management (CM) module, dependency generation module and its visualization module. Finally, we make conclusions about our work and describe avenues for future work.

2. TYPES OF DEPENDENCIES

2.1 Technical Dependencies

In software engineering, program dependence graphs (PDGs) are used to allow explicit representation and manipulation of program dependencies. According to Horwitz and Reps [16], formally, a PDG for a program P is a directed graph whose vertices are statements of P connected by edges that represent control and data dependencies. For simplicity purposes, researchers initially explored the construction of PDGs for simple programs: isolated procedures and programs that contain a single procedure. Later, interprocedural approaches were explored considering several procedure calls,their parameters and return types; which originated the term system dependence graph [1]. These graphs can be used to construct *call graphs* [17] that are used for interprocedural program optimization and program understanding [21]. According to Callahan and colleagues, 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" [3].

2.2 Socio-Technical Dependencies

By extracting dependencies in the source-code, a call-graph potentially unveils dependencies among software developers responsible for the software components [5-7]. For instance, assume that a software component *a* depends on another software component *b* and that *a* is being developed by *developer A* and *b* is being implemented by *developer B*. If *a* depends on *b*, we similarly find that *developer A* depends on *developer B*. That is, these software developers need to coordinate and communicate to guarantee the smooth flow of work [15, 24-26], even when programming constructs, like interfaces, are used [8]. The results of these empirical studies suggest that product dependencies create and reflect task dependencies between software developers, that is, product dependencies create a need for communication and coordination between developers, and, similarly, task dependencies are reflected in the product dependencies. This translates into the need to populate the call-graph with 'social information.' The goal is to create a data structure that describes which software developers depend on which other software developers for a given piece of code [7]. An example of this data-structure, called a social call-graph, is presented in Figure 1. A directed edge from package A to B indicates a dependency from A to B. Directed edges between authors and packages indicate authorship information.

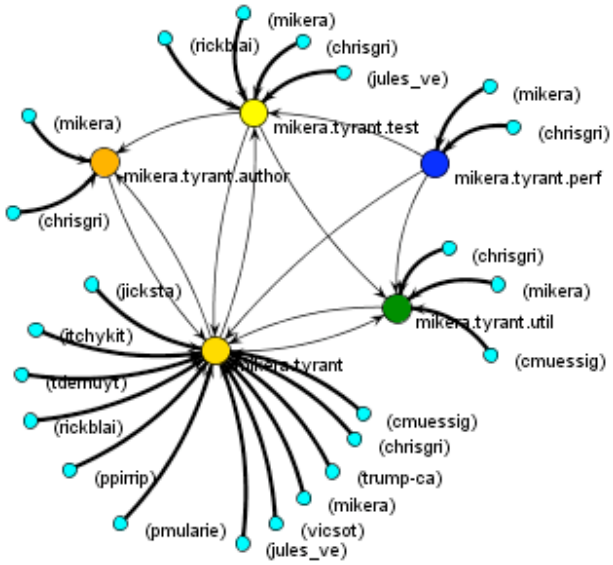

Figure 1 - Socio-technical dependencies.

2.3 Social Dependencies

Because social call-graphs describe both technical dependencies *and* authorship information, they can be used to generate sociograms describing the dependence relationship *only* among software developers, that is, dependencies between social developers *because of* dependencies in the source-code they are working on. A sociogram, as used in social network analysis [27], is a graphical representation of a set of items, vertices or nodes, connected to one another via links or edges. Figure 2 below presents an example of a sociogram created using Ariadne.

Software developers can now use these sociograms to find out two important pieces of information: who they depend on and who depends on their work. We hypothesize that by identifying this "impact network", developers can more easily coordinate their work. Indeed, we plan to test this hypothesis through a series of interviews (see section 4). We have used these sociograms to understand open/free source software development [6].

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

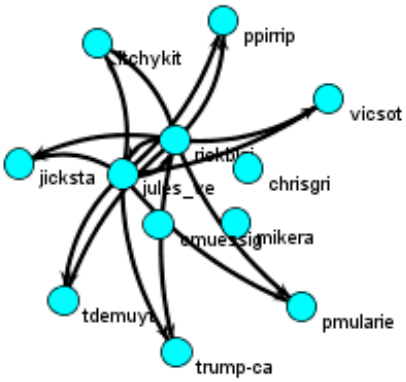

Figure 2 - Sociogram

3. ARIADNE

3.1 Features

Ariadne is implemented as a Java plug-in to the popular Eclipse IDE. As such, Ariadne is integrated into this environment and makes use of several of the services it provides. Initially, the plug-in uses Eclipse's SearchEngine class to extract dependencies from a Java project's source-code. Ariadne connects to the configuration management repository associated with a project to retrieve authorship information about the project. After that, the plug-in annotates the call-graph with the extracted authorship information to create a social call-graph (see section 2.2). Finally, the social-call graph is used to generate a sociogram that is displayed using the graphical framework JUNG (Java Universal Network/Graph Framework)[2].

Ariadne presents developers with three visualization options: technical dependencies, socio-technical dependencies and social dependencies. Our 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 [21]. For instance, class dependencies are displayed as the aggregation of method dependencies (i.e., the call-graph). All visualizations provided by Ariadne can be exported to Comma Separated Values formatted files, while sociograms can be exported to files suitable for use with social network tools like UCINet.

Ariadne also supports the temporal analysis of all dependencies, similarly to TeCFlow [13]. That is, Ariadne can generate visualizations for graphs of snapshots in time, which allows us to study the evolution of a project's technical and social dependencies.

3.2 Ariadne's Architecture

Ariadne was initially implemented to analyze only Java projects and extract information from CVS repositories. We recently re-designed it to be general enough 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. This is achieved through the usage of a layered architecture presented in Figure 3. As expected, the most important part is the configuration management and dependency management API. This 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.

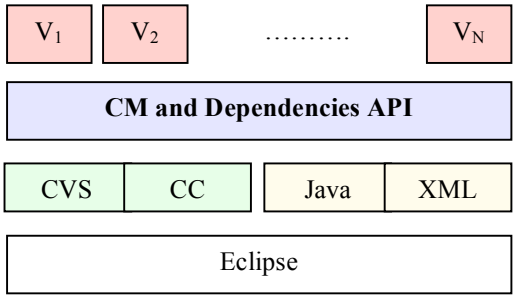

Figure 3 – Ariadne's architecture

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.

Currently, we have implemented a code dependency infrastructure that analyzes Java code and Eclipse's manifest and "plugin.xml" files. We built a CVS extractor used to connect to a project's CVS repository (using Eclipse's Team API), that annotates the dependencies with authorship information, and creates visualizations based on directed graphs. We have also built an infrastructure that imports source-control annotations from Rational Clearcase. These annotations are parsed and used to create social call-graphs and, ultimately, sociograms.

To facilitate the understanding and usage of this API, Ariadne utilizes the façade design pattern [12] that aggregates methods to be used to query program dependency, authorship information and the combined information (the social call-graph). For example, developers may query the classes that depend on a particular class, the authors of a particular piece of code, all the authors of a file, how the ownership of a class changes from one release to the next, etc.

Figure 4 below presents a UML class diagram for the program dependency and visualization parts only of this API. Both parts, as well as the visualization subsystem will be described in the following sections.

[2] http://jung.sourceforge.net

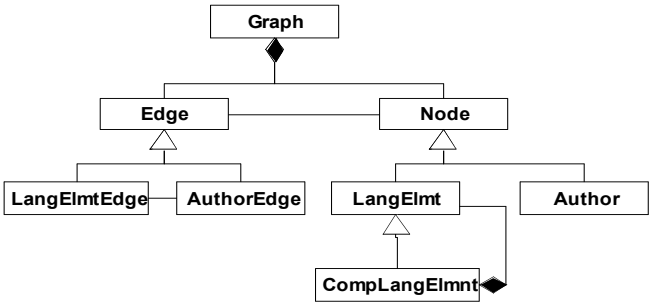

Figure 4 - Class diagram

3.3 Program Dependency Information

Ariadne has been designed to represent hierarchy levels in various programming languages. These different levels can be thought of as two different types of code units: *Language Elements* as well as *Composite Language Elements*. *Language Elements* are defined as pieces of source-code that are *not* composed of smaller code units. For example, consider the case that a developer has chosen to analyze dependencies in a software project written in Java. In this source-code there is a class A and a method of that class, b. In our approach, method b is considered a *Language Element* because methods are the lowest level of the hierarchy in Java. On the other hand, class A is a *Composite Language Element* because it is composed of methods – one of them being b – and possibly attributes. This is basically an implementation of the composite design pattern [12] to represent the relationship between programming language elements, in this case, *Language Elements* and *Composite Language Elements.* This pattern allows us to represent part-whole hierarchies as well as treat individual and composite objects in much the same way.

In the first implementation of Ariadne, due to the design of the dependency generation subsystem, we were not able to identify in the sociogram *the piece of code responsible for a social dependency*. Therefore, we redesigned Ariadne to address this issue as described in Figure 4. Our current design defines a superclass *Edge*, which abstracts the two different possible types of edges, *Author Edge* and *Language Element Edge*. The first type of Edge models social dependencies, while the second one models program dependencies in the source-code. These two edges are connected by a relationship that is used to allow bi-directional navigation: given a technical dependency, which are the authors involved in the corresponding social dependency, *and*, given a social dependency, which are the programming elements involved in the corresponding technical dependency.

The usage of the abstract class *Edge* allows us to abstract away the difference between the different edges in the visualization module, providing a generic way to draw edges. Furthermore, an edge can be queried for information about what piece of information it links. We describe the visualization subsystem in more details in section 3.5.

3.4 CM Information

CM systems offer tremendous amounts of data that Ariadne aims to abstract into generic formats that developers can mine to produce informative visualizations. For our purposes, Ariadne models CM repositories in a generic way that allow views of a project's data at one or many points in time, no matter which CM system is used. We believe we designed an API that is generic enough to capture the essential functionality *that Ariadne requires* of systems such as CVS, Subversion, and Clear Case, while still providing detailed authorship information from repositories. This is possible because the CM subsystem consists of a hierarchy of related classes that share a common resource heritage and exist inside a repository. Ariadne associates one repository with each project in the workspace. Repositories consist of branches, which represent the state of code in the repository at specific points in time (releases). Branches do not exist until users dictate how the repository should be populated from the CM system. Implementers may choose to have their plug-in select dates by which to break up the development timeline into meaningful states. Branches are broken into collections of commit sets that group changes made at arbitrary points in time. An example commit set could be all the resources a developer commits to the repository after fixing a bug. Commit sets hold a collection of deltas that represent a set of changes made to a file. Deltas represent individual changes made to different parts of a file and contain the line number information for where a change began and ended. The Ariadne core module uses this information to query the code dependency generator module for any language elements in the region.

3.5 Visualization

Ariadne's visualization subsystem allows developers to access information from the CM repository as well as the dependency information. In order to create visualizations, a developer must query Ariadne's API for an instance of the Graph object. Our visualization framework utilizes some of the same design principles found in the JUNG project - specifically that we represent a Graph object as a generic container of Edges and Nodes. As such, Ariadne is capable of displaying any type of visualization that can be represented as entities and their connections. By doing that, we can reuse the same algorithms to draw technical and social dependency graphs since the Author and Language Element classes are subclasses of class Node (see Figure 4).

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 [10] , treemaps, design structure matrices [2] or however else he chooses to visualize dependencies.

4. CONCLUSIONS AND FUTURE WORK

This paper described Ariadne, a plug-in to the Eclipse IDE that aims to reduce the gap between technical and social dependencies, and therefore facilitate the coordination of software development work. Ariadne was motivated by our own field studies of large-scale software development and reflects some of the insights that we learned from these studies. We described Ariadne's features as well as architecture and presented parts of its API, which allows software developers to have access to source control and dependency information provided by multiple configuration management systems and programming languages.

Furthermore, all visualizations are based on this API, therefore they can be easily reused. We plan to extend this API to fully explore the Eclipse plug-in model, so that, new visualizations can be created as new Eclipse plug-ins. Finally, we plan to adapt our plug-in so that developers can choose from many visualizations ranging from directed graphs, annotated class diagrams, or decorators inside the Eclipse workbench. Decorators are simple visual clues (usually in the form of an icon) to developers that display additional information about resources in the workspace.

Currently, we are in the last planning stages of a field evaluation of Ariadne with software developers from a large software development company and an open-source project. We want to understand the coordination problems faced by these developers and whether Ariadne can be used to minimize some of these problems. After this initial evaluation, we will make more improvements in Ariadne before releasing it to the public as an open-source tool.

5. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under awards 0205724 and 0326105, IBM through the Eclipse Innovation Program, and by the Brazilian Government under CAPES grant BEX 1312/99-5.

6. REFERENCES

1. Aho, A.V., Sethi, R. and Ullman, J.D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. Browning, T.R. Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions. *IEEE Transactions on Engineering Management*, *48* (3). 292-306.
3. Callahan, D., Carle, et. al. Constructing the Procedure Call Multigraph. *IEEE Transactions on Software Engineering*, *16* (4). 483-487.
4. 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.
5. de Souza, C.R.B., Dourish, P., Redmiles, D., et. al., From Technical Dependencies to Social Dependencies. in *Workshop on Social Networks for Design and Analysis: Using Network Information in CSCW*, (Chicago, IL, 2004).
6. de Souza, C.R.B., Froehlich, J. and Dourish, P., Seeking the Source: Software Source Code as a Social and Technical Artifact (to appear). in *ACM Conference on Group Work*, (Sanibel Island, FL, USA, 2005).
7. 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.
8. de Souza, C.R.B., Redmiles, D., Cheng, L.-T., Millen, D. and Patterson, J., Sometimes You Need to See Through Walls - A Field Study of Application Programming Interfaces. in *Conference on Computer-Supported Cooperative Work (CSCW '04)*, (Chicago, IL, USA, 2004), ACM Press, 63-71.
9. de Souza, C.R.B., Redmiles, D., et. al., Management of Interdependencies in Collaborative Software Development: A Field Study. in *International Symposium on Empirical Software Engineering*, (Rome, Italy, 2003), 294-303.
10. 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.
11. Fuggetta, A., Software Processes: A Roadmap. in *Future of Software Engineering*, (Limerick, Ireland, 2000).
12. Gamma, E., Helm, R., et. al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
13. Gloor, P.A., TeCFlow - A Temporal Communication Flow Analyzer for Social Network Analysis. in *Workshop on Social Networks for Design and Analysis: Using Network Information in CSCW*, (Chicago, IL, USA, 2004).
14. Grinter, R.E. Recomposition: Coordinating a Web of Software Dependencies. *JCSCW*, *12* (3). 297-327.
15. Grinter, R.E., Recomposition: Putting It All Back Together Again. in *Conference on Computer Supported Cooperative Work*, 1998, 393-402.
16. Horwitz, S. and Reps, T., The use of program dependence graphs in software engineering. in *International Conference on Software Engineering*, (Melbourne, Australia, 1992), 392-411.
17. Lakhotia, A., Constructing call multigraphs using dependence graphs. in *SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Charleston, South Carolina, USA, 1993),, 273-284.
18. Larman, G. Protected Variation: The Importance of Being Closed. *IEEE Software*, *18* (3). 89-91.
19. McDonald, D.W. and Ackerman, M.S., Just Talk to Me: A Field Study of Expertise Location. in *Conference on Computer Supported Cooperative Work '98*, (Seattle, Washington, 1998), 315-324.
20. Morelli, M.D., Eppinger, S.D. and Gulati, R.K. Predicting Technical Communication in Product Development Organizations. *IEEE Transactions on Engineering Management*, *42* (3). 215-222.
21. Murphy, G., Notkin, D., Griswold, W.G. and Lan, E.S.-C. An Empirical Study of Static Call Graph Extractors. *ACM TOSEM*, *7* (2). 158-191.
22. Nutt, G.J. The evolution toward flexible workflow systems. *Distributed Systems Engineering*, 1995.
23. Parnas, D.L. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, *15* (12). 1053-1058.
24. Sosa, M.E., Eppinger, S.D. 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.
25. Sosa, M.E., Eppinger, S.D. et. al. Identifying Modular and Integrative Systems and Their Impact on Design Team Interactions. *ASME Journal of Mechanical Design*, *125*. 240-252.
26. Sosa, M.E., Eppinger, S.D. et. al. The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. *Management Science*, *50* (12). 1674-1689.
27. Wasserman, S. and Faust, K. *Social Network Analysis: Methods and Applications*. Cambridge University Press, Cambridge, UK, 1994.