Theseus: Tool Support for Managers of Distributed Software Development Projects

Cleidson R. B. de Souza, Sebastião B. Fonseca

Faculdade de Computação
Universidade Federal do Pará (UFPA)
66.075-110 – Belém – PA – Brasil

cdesouza@ufpa.br

Abstract. *In this paper we present a tool to facilitate the work of managers of global software development projects. This tool explores the relationship between software dependencies and coordination of work and uses social networks to suggest potential coordination problems for managers. The theoretical and empirical motivations for the tool focusing on the relationship between software dependencies and the coordination of software development work.*

1. Introduction

In the past years, more and more organizations have distributed their software development projects in different sites spread around the globe. Researchers and practitioners have proposed strategies, tools, and approaches to facilitate this scenario which faces social, technical and cultural challenges [Herbsleb and Moitra 2001]. In this paper we present a tool, called Theseus, that aims to facilitate the work of managers of global software development projects. This tool is based on theoretical predictions [Conway 1968][Parnas 1972] and empirical observations [Morelli et al. 1995][Grinter 1995][Sosa 2002][de Souza et al. 2004][Cataldo et al. 2006] about the nature of software development work. Through our tool, a manager could potentially monitor distributed software developers and anticipate coordination problems among them. This paper briefly describes the design of this tool and our theoretical motivations.

2. Background and Motivation

One way to manage complexity of today's growing systems is to decompose these systems into subsystems [Simon 2006]. This has been known as modularity. Basically, modularity means that a complex system can be decomposed into smaller pieces called *modules*. Modularity reduces complexity because it allows (i) one to deal with the details of each part of the system in isolation (by ignoring the details of other modules), and (ii) one to deal with the overall characteristics of all modules and their relationships in order to integrate them into a coherent system [Ghezzi, Jazayeri et al. 2003]. Modules are thus parts of a large system and need to interact in some coordinated way for an effective system to exist. These interactions imply that these modules are dependent upon one another. In order words, modules rely or depend upon other modules to create the larger system. These dependencies make it easier or more difficult to understand,

extend, modify, and reuse these different modules. Therefore, their study is of fundamental importance for software development efforts.

Software engineers have long recognized the need to deal with dependencies in development efforts. For example, there are different techniques for program dependency analysis [Podgurski and Clarke 1989]. These approaches are used, among other goals, to improve software testing, evolution, and understanding. Similar approaches have been proposed to component-based systems [Vieira and Richardson 2002], and software architectures [Stafford and Wolf 2001]. Another approach adopted by researchers and practitioners is the creation of mechanisms in programming languages to reduce dependencies between software elements. In this case, the most important principle is Parnas' information hiding [1972]. Parnas proposed more than just a technical approach: he recognized the relationship between software dependencies and coordination when he suggested that by reducing dependencies between modules, it is possible to reduce developers' dependencies on one another, a managerial advantage. Nowadays, this is a well-known argument cited in software engineering textbooks [Ghezzi, Jazayeri et al. 2003]. Conversely, but also supporting this relationship between dependencies and coordination, Conway [1968] postulated that the structure of a software system would reflect the communication needs of software developers. In short, whereas Parnas argues that dependencies *shape* the coordination and communication activities performed by software developers, Conway argues the converse: that dependencies *reflect* these coordination and communication activities. That is, technical dependencies between components create a need for communication between developers, and similarly, dependencies between the development tasks are reflected in the software. Both Parnas' and Conway's arguments have been validated by different qualitative and quantitative studies in collocated and distributed software projects [Morelli et al. 1995][Grinter 1995][Sosa 2002][de Souza 2004] [Cataldo et al. 2006].

Despite this acknowledged relationship between dependencies and communication and coordination needs, this relationship has not been exploited to facilitate and understand software development activities. One of the few exceptions is the work of [Cataldo et al. 2006]. 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 of those developing it. In this paper we explore this relationship to support software development projects through the description of a software development tool to help managers of distributed projects. This tool is described in the following section.

3. Theseus

Among the empirical studies that describe the relationship between software dependencies and coordination, some of them are more relevant to the work described in this paper. To be more specific, [Morelli 1995] and [Sosa 2002] conducted quantitative studies where they found a strong correlation between dependent components in a software system and the frequency of communication among the team members dealing with these components. Their result suggests that developers dealing with dependent components are more likely to engage in communication than developers implementing independent components. According to these authors,

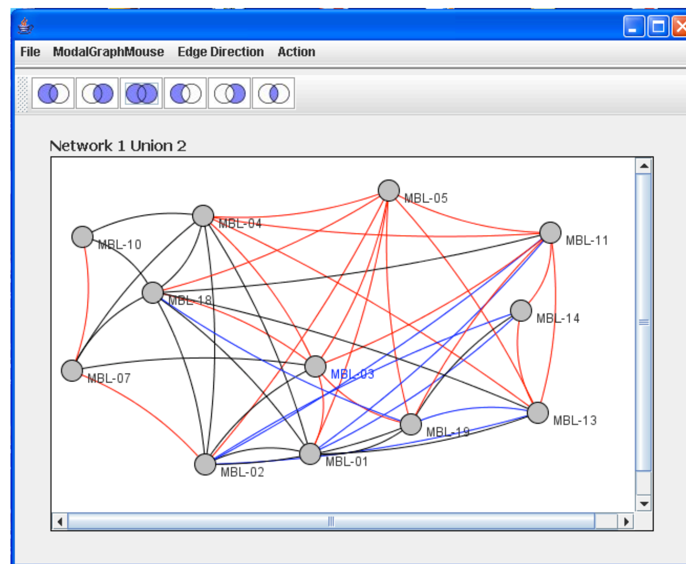
technical dependencies could then be used to predict communication frequency among team members. That is, given two dependent software modules, the developers responsible for developing those modules need to interact to coordinate their work, despite the usage of interfaces and other mechanisms to minimize dependencies. It is this insight that guides the design and usage of our tool.

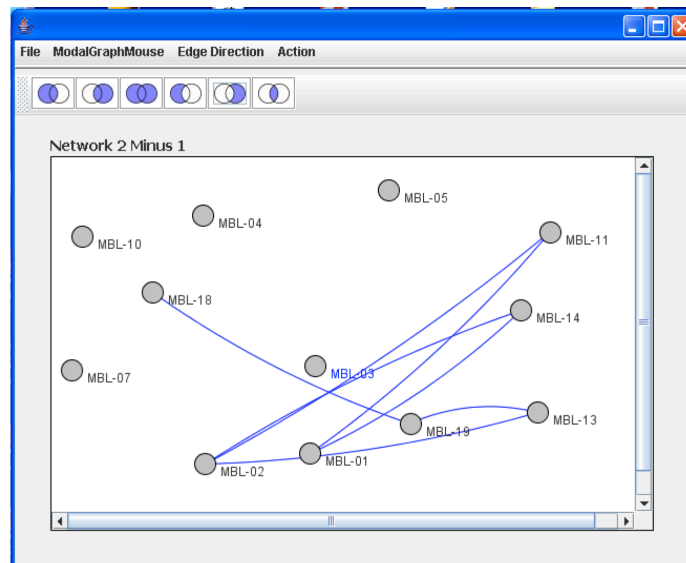
However, in order to explore the relationship between software dependencies and coordination, it is necessary to identify dependent pieces of software and communication events among the software developers. Our tool automatically identifies dependent pieces of code and their authors using Ariadne [Trainer et al 2005][de Souza 2007], so that it is possible to create a social network [Wasserman and Faust 1994] of software developers that identifies which developer depends on the code of another software developer. Communication events are more difficult to be identified since developers can use different media to communicate: emails, instant messages, phone calls and so on. Our current implementation registers email exchanges through an event-notification server that receives all emails exchanged among pairs of developers, i.e., copies of work-related emails sent by software developers are also sent to our tool, which aggregates all emails thus creating a communication network of developers. These two social networks – dependency and communication – are then combined by our tool.

The goal of our tool is to automatically identify situations when there is a *mismatch* between the dependency and communication networks. This includes two situations. In the first case, there is a dependency between two components, but the software developers dealing with them are not engaging in communication events. This might mean that those developers are not aware of each other, a usually problematic situation [de Souza et al. 2004]. The second case happens when two developers are communicating with some frequency, but there is not a dependency between their components. This situation might suggest a need for re-structuring the architecture of the system (that's why they are communicating) or that possibilities for software reuse are being lost.

Our tool supports the visualization of different social networks: the communication and dependency networks, the network that highlights the matches between the communication and dependency networks, and finally, the network of communication (dependency) not accompanied by dependency (communication). This way it provides to managers easy access to the information of interest. Figure 1 below presents three of these views as provided by our tool: (i) the union of the two networks, (ii) the dependency network minus the communication network, and (iii) the communication network minus the dependency network. Again, the overall idea is to identify the mismatches between the networks, which is achieved by presenting the difference between the networks.

An important design decision of our tool is just to present information to the manager, letting him to decide how to handle it. That is, the tool just indicates the mismatches between the networks; it does not automate tasks for the manager. The reason is that the manager might be aware of additional information, which would help him to make sense of the reported mismatches: e.g., a refactoring of the code that is about to happen might justify the communication that is going on among developers despite the fact that their components are independent.





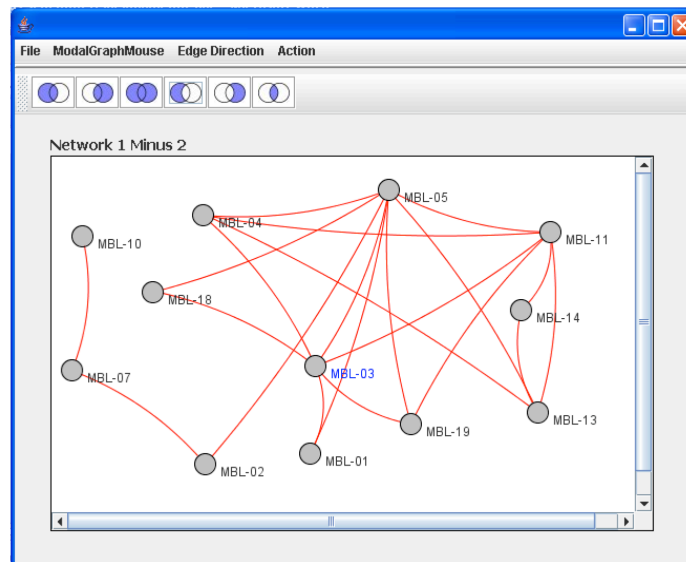


Figure 1: Tree different views provided by the tool

Our tool is implemented in Java and uses Elvin [Fitzpatrick et. al 1999] as the event-notification server that monitors and receives email information exchanged among software developers. The dependency network from Ariadne is imported as CSV files. The usage of Elvin and Ariadne means that our tool currently supports communication networks based on emails among software developers and Java projects hosted in CVS repositories, respectively. Furthermore, Ariadne is an Eclipse plug-in that uses call-graphs and data dependencies graphs to identify dependencies in Java projects [Trainer et al. 2005][de Souza et al. 2007]. The comparison between the social networks is done using matrix operations, as in standard social network software implementations [Wasserman and Faust 1994] and the visualization of networks is done using JUNG (http://jung.sourceforge.net).

4. Conclusions and Final Remarks

The goal of this paper is to present a tool developed by the authors to facilitate the work of managers dealing with software development projects. This tool is based on the relationship between software dependencies and the coordination of software development work. This relationship has been predicted in the past and corroborated by different empirical studies. Managers could use the tool to monitor interactions among distributed software developers and therefore anticipate potential problems.

We plan to continue improving the tool and also to conduct empirical studies with it. For instance, by using it to analyze real data from global software development projects or deploying it in global teams. Another approach to be explored is the visualization of the information that it displays. Currently, our tool presents its information as sociograms [Wasserman and Faust 1994], i.e. graphs, however we plan to explore additional visualizations that are more meaningful to project managers.

5. References

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

Conway, M. E. How Do Committees invent? (1968), Datamation, vol. 14, pp. 28-31.

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

de Souza, C. R. B., Quirk, S., Trainer, E., Redmiles, D. F. Supporting Collaborative Software Development through the Visualization of Socio-Technical Dependencies (2007). To appear In Proceedings of the International ACM SIGGROUP conference on Supporting group work, Sanibel Island, FL, USA.

Fitzpatrick, G., T. Mansfield, et. al. Augmenting the workaday world with Elvin (1999). in 6th European Conference on Computer Supported Cooperative Work.. Copenhagen, Denmark: Kluwer.

Ghezzi, C., M. Jazayeri, and D. Mandrioli, Fundamentals of Software Engineering. Second Edition ed. 2003: Prentice Hall.

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

Herbsleb, J. D. and Moitra, D. Global software development (2001), *IEEE Software*, vol. V18, pp. 16-20.

MacCormack, A. et. al., Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code (2004), Harvard University, 05-016.

Morelli, M. D. et. al., "Predicting Technical Communication in Product Development Organizations (1995), *IEEE Trans. on Engineering Management*, vol. 42, pp. 215-222.

Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules (1972), *CACM*, vol. 15, pp. 1053-1058.

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

Simon, H.A., The Architecture of Complexity: Hierarchical Systems (1996), in The Sciences of the Artificial. The MIT Press: Cambridge, MA. p. 183-216.

Sosa, M. E. et. al., Factors that influence Technical Communication in Distributed Product Development: An Empirical Study in the Telecommunications Industry (2002), *IEEE Trans. on Engineering Management*, vol. 49, pp. 45-58.

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

Stafford, J.A. and Wolf, A.L. Architecture-Level Dependence Analysis for Software Systems (2001). International Journal of Software Engineering and Knowledge Engineering, 11 (4). 431-453.

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

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

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