A Dependency-Sharing Tool for Global Software Engineering

Douglas Lee, Allen E. Milewski, and Daniela Rosca

Dept. of Computer Science and Software Engineering,
Monmouth University West Long Branch, New Jersey, USA
{s0784833,amilewsk,drosca}@monmouth.edu

Abstract. This project explores the design of a tool to facilitate a common task that software engineers find difficult – the identification and management of dependencies between the many heterogeneous entities created in the course of a software development project. The focus of this tool is the value it might have during the maintenance phase. Maintenance engineers learn and understand the project differently from the original authors of the artifacts. Typically, they come to understand the project by investigating dependencies between entities- a task that can be very difficult and time-consuming. To deal with these differences, the Global Software Traceability (GST) Tool was designed and prototyped to explore improvements in the usability of maintaining dependency links after the project has been deployed. The GST Tool is a proof-of-concept design prototype used to investigate how to make such a tool both useful and usable. The tool was successful in creating an environment whose overhead was low enough to make it likely that it would be used despite the severe time constraints found in software maintenance.

Keywords: dependencies, traceability, maintenance.

1 Introduction

A significant and difficult task in software engineering is to keep track of dependencies between the multitudes of heterogeneous entities that constitute a large software project. These entities include such things as system goals, contract details, system requirements, architectural and design specifics, source code modules and test cases with their results. Entities are typically produced and maintained by different engineers who are often in different organizations that are not collocated. Their documentation is almost always distributed across a large number of different project artifacts: requirements are maintained in a System Requirements Specification, software design entities in a Software Design Specification, etc.

Even in projects where documentation of these entities is well-done, it can be difficult to keep track of and document the dependencies between them. The problem of tracking dependencies between project entities has been researched from several

A. Marcus (Ed.): DUXU/HCII 2013, Part IV, LNCS 8015, pp. 37–46, 2013.

different domains in software engineering. Requirements Engineering, for example, has explored traceability[1] methods [1] [2] primarily to maintain links from requirements to implemented function. The study of Impact Analysis has focused on code evolution and risk [3] [4]. Dependency Analysis between software modules is something that has concerned software architects and programmers since it affects reusability and code comprehension and has been shown quite clearly to be a good predictor of faults [5] [6] [7].

Looking across these different domains where the understanding of dependencies between entities is critical, two things are of special note. First, these domains have often treated the problem as specific to their own area rather than recognizing, as suggested in [8], that dependency management is an issue that crosses all phases and roles in a software project. Second, it is generally agreed that dependencies between project entities are difficult for software engineers to discover and identify [5] [8]. Indeed, there have been persistent attempts to automate or partially automate the processes [1] [9] [10] [7]. While some significant advances have been made in automation, many of the attempts have relied upon unique contexts or restrictive assumptions that may make the approach difficult to apply in general development environments. In the area of code dependencies, one example of how difficult it may be to automate entirely the detection of relationships comes from work on logical dependencies [11] [5] [12]. Logical dependencies are detected by co-occurrences of Modification Requests during the maintenance phase rather than by inspection of the code itself. Indeed, there may be little or no syntactic evidence for a dependency. Yet, logical dependencies are often more critical for quality than syntactic dependencies.

The place where this difficulty with dependencies takes its largest toll is in the maintenance phase of the project. It is during the maintenance phase that engineers and developers need to understand the system from a variety of viewpoints and levels of analysis. A significant portion of maintenance work is spent in analysis/isolation tasks which consist of impact analysis, cost benefit analysis, and isolation. Impact analysis and cost benefit analysis require analyzing different implementation alternatives and comparing their effect on schedule, cost, and requirements-fulfillment. Side-effect prevention is also tantamount Isolation refers to the time spent trying to understand the problem or the proposed enhancements to the system. All these tasks deal with figuring out how the various parts of the system contribute to making the system work [13]. The amount of work for developers supporting these systems to manually find the dependencies between components and between the requirements and their corresponding components is immense [14].

The present investigation explored design elements of a general purpose, highly flexible tool to facilitate the documentation and understanding of dependencies between project entities. The Global Software Traceability (GST) Tool assists software engineers in maintaining relational information between a wide variety of heterogeneous entities, including use cases, requirements, design diagrams, test cases, blocks of

[1] It has been suggested [16] that the term “dependency” be used to encompass the various terms, such as trace, link, coupling and impact, used in the different software domains. We have adopted this practice despite confusions due to the prevalent use of “dependency” to describe software code modules.

code and others. Design of the GST Tool is based on the premise that while dependencies may be difficult to discover after the fact, there are times in the project lifecycle when a dependency does become clear to a project participant. For example, when designing a specific module to meet a set of requirements, the dependency between the two is clear to the designer of the module. Similarly during testing, when a fault is found to result from a heretofore unknown logical dependency between two blocks of code, that dependency is now prominent to the tester. The problem in both these cases is communicating the dependency to the rest of the project members so that it is salient and persists across the project. For the GST Tool, accumulation of relational information is a collaborative effort with entries potentially coming from and being made available to different project roles and organizations. While the sharing of dependency information may already be part of the collaborative tradition of software engineers, Cataldo [6] [11] has shown that, for example, dependencies are especially difficult to deal with when team members are not collocated. We expect a tool like the GST Tool to be of exceptional value in a global software development environment.

2 Global Software Traceability Tool Goals

There was a consistent set of functional and non-functional requirements that guided the iterative design and development of the GST Tool throughout its prototyping.

2.1 Pervasiveness

One key goal was to prototype a tool that can facilitate the management of dependencies between many heterogeneous entities and do so all within the same framework. This includes entities developed in very different phases of development, structural and behavioral entities, and textual and graphical entities. There already exist a variety of tools specially designed for requirements traceability, impact analysis and syntactic dependencies in code [8] [11], etc.. The GST Tool is designed to work across these domains. In addition, we sought to manage dependencies across different levels of detail. For example, it was important to be able to show dependencies between a requirement and portions of code within a module, as compared with showing the dependencies only at the code module level. While not represented in the current version of the prototype, users also need simple procedures for adding their own, customized types of entities so that dependencies can be maintained even if they were not foreseen prior to a project.

2.2 User Centered Design

Rather than utilizing intricate methods of automatic analysis, the GST Tool focuses on making the manual entry of relational information natural and uncomplicated. This strategy put the burden on the user experience design to reduce cognitive and motor overhead as much as possible. A “drag-and-drop” style of interaction was adopted in the GST Prototype so that dependencies could be created with a simple mouse movement. Moreover, a multi-window approach was taken to display information about most entities in separate windows, which could be positioned separately for maximum flexibility in the software development environment.

2.3 Facilitates Information Sharing

Several sources have noted the difficulties that an individual engineer can have in perceiving software dependencies, and have suggested that these problems could be reduced with tools that display to the software engineers dependencies other software engineers have found or even conjectured based on their own experience [15]. The GST Tool is envisioned for collaborative use by contributors in different project roles and organizations. Indeed, the GST Tool relies upon the load of information-contribution being spread across groups of project members, in much the same way that consumer-oriented social media applications rely on inputs from many diverse participants to build a network of dependencies.

2.4 Can Be Integrated with Standard Development Tools

The goal for the initial GST Tool prototype was that it should have a similar feel to the applications that the user are already familiar with. Eventually, the tool should have the ability to be integrated with different vendors' tools or at least be architected for a common Open Source environment (e.g. the Ariadne plugin in Eclipse, [8]).

2.5 Can Facilitate Emergent Insights about Higher-Order Dependencies

Of the various forms of entity dependencies that have been delineated in a software project (e.g. horizontal/vertical, unidirectional/bidirectional, etc. [16], potentially, the most informational are the indirect dependencies. An indirect dependency, for example, might be when entity A is related to entity B and entity B is dependent upon entity C. The transitivity of the dependencies between entities A and C are difficult to detect, and dependencies more than two steps removed, while valuable to know about [17], are even much more difficult to detect. The GST tool is built to manage and highlight indirect dependencies to help engineers understand the broad implications of changes. While, of course, the mapping of indirect dependencies could be carried out through many orders (e.g., 1st order, 2nd, order, 3rd order) the GST Tool tracked only through the 2nd order since the number of dependencies grew too large to comprehend. Moreover, while dependencies in the GST Tool are treated as being unidirectional, it is be possible to trace links backwards, so that, for example, when two dependent software modules result from two seemingly independent requirements, it can be seen that those requirements are actually related.

3 Prototype Description

The GST prototype is a C# application composed of a series of related forms, where each form provides a unique function.

3.1 Project List

The Project List displays all the projects for which dependencies are being managed. The user would select a project before adding or exploring any dependencies, since dependencies are project-specific. In addition to being useful for separate projects,

this functionality can be used to manage separate releases of the same product. To facilitate this, it will be possible to make a copy of an entire project and begin modifying it when a new version begins.

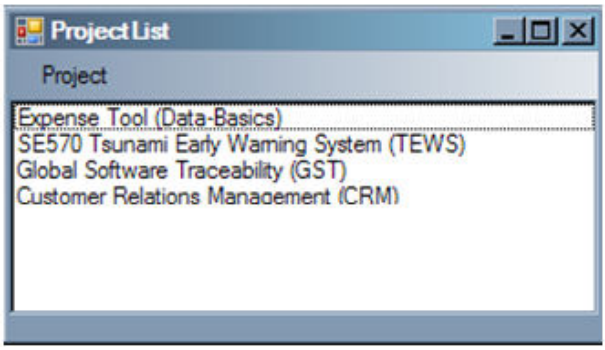

Fig. 1. Project List

3.2 Entity List

The Entity List is the central hub of the GST Tool. It consists of a treeview to represent parent-to-child dependencies and show entities and subentities in a hierarchical fashion. While not implemented in the initial prototype, entity types will be added to the GST manually with an "Add Entity" button or by importing entities from other projects. As can be seen in Figure 2, selecting one of the entities produces a Details window in which can be entered a text description of the entity, and a name. Two additional fields are managed automatically. One is an entity ID, which is used internal to manage entities. The other is a list of dependencies (or links) relevant to the current entity.

The GST Tool implements a Drag and Drop interface style. To manually create the links, the user drags any entity from a form on top of another entity. Multiple entities can be dropped on the same entity. Any entity can also be used multiple times. The Drag and Drop feature can be used within the Entity List or across other Entity forms. For example, dragging a block of code in a Code form and dropping it onto a specific Requirements form establishes a link.

As the links are added, the entity detail form (e.g. Figure 2) will be automatically updated with the dependency information. Also if a third entity is linked to an entity that is linked to an entity already in the linked form, it is considered an indirect link and is added as a sub-node of the entity with the direct link. With the entity list, then, the user can get an overall picture of all the entities and the general structure of their dependencies. The hierarchical structure of direct and indirect links is also represented in the Details Forms for each entity.

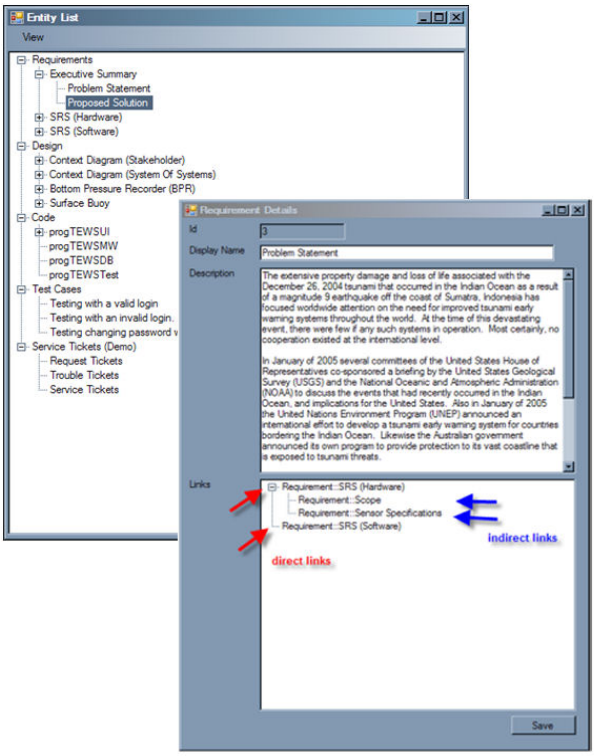

Fig. 2. Entity List with Details Form for Selected Entity

3.3 Requirement Details

The Requirements Detail Form , shown in Figure 2, is one of several "Details" forms. There are similar forms for Projects, Design Diagrams and Code, as well as Trouble Tickets, Test Cases, etc. As shown above, these details include, besides the entity name and description, the dependency links, which are categorized as direct links and indirect links. The indirect links are those obtained from transitivity of the dependencies with other entities.

3.4 Code and Diagram Details

The final two detail forms implemented in the current version of the prototype were for Code and Diagrams (Figure 3). The Code Form is implemented as a treeview just as are the other entities in order to simplify creation of links between entities and blocks of code smaller than an entire window. Code can be imported from external files into a Code Form. The Diagram Form permits users to create a range of software design forms. This facility is currently a freeform drawing package and does not permit importation from external sources since there are no generally agreed-upon import standards for most design artifacts, such as UML diagrams.

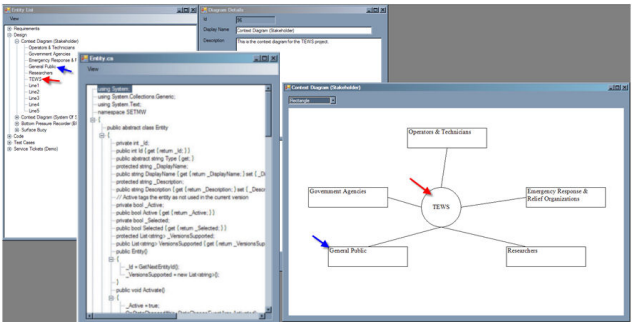

Fig. 3. Code and Diagram Details Forms

3.5 Additional Forms (Not Implemented)

During the development and evaluation of the GST Tool, it became clear that it may be useful to maintain dependencies with and between additional kinds of entities. For example, during the maintenance phase, it is common practice to track trouble tickets and modification requests to manage their status. There could be clear value in tracing their dependencies to other entities in the software project. Further, several writers have recently noted the "Socio-Technical" aspects of dependencies and of project development in general [11] and have shown that analyzing dependencies between technical elements such as code modules and social elements such as organizations and workflow elements significantly improve fault estimation. As a result, we envision the GST Tool potentially providing facilities for tracking social and organizational information as well.

4 Evaluation

An informal evaluation of the GST Tool was done with 4 software engineers. They were given two maintenance tasks to complete with the assistance of the GST Tool. One was a standard forward engineering task (i.e. from requirements to code), while the other was reverse engineering (i.e. from code to requirements) task. A survey asked questions about their experience in completing the task and if a tool such as the GST Tool would be useful to them. Besides the surveys, respondents were interviewed about adding traceability to their development process and documentation in general.

All the respondents worked for small-sized software companies of 50 or fewer employees. The majority of the projects these respondents worked on had little or no project documentation. None worked or was working with a traceability tool of any sort. All had programmed for the majority of their career with a minimum of 5 years. And one was currently a manager. All worked primarily with C#, but some had also done projects in VBScript, VB6, and Java. Although none of the respondents had formal forward engineering process experience, they did not find the forward engineering task especially difficult. Nonetheless, because working with projects with

little or no documentation was the norm for them, they were more familiar with reverse engineering strategies.

All respondents felt that having a tool like GST would save time in the long-term. There was somewhat of a difference in attitudes between different roles. The developers felt that the process would be too tedious if it were to be combined with their current responsibilities. The project manager on the other hand, did not find the process to be tedious. One survey item had respondents rank the priority of time, cost and quality as factors in deciding what tools and process elements were critical in their environment. Three responded that time was the highest priority; one person selected cost. This emphasis on time factors resulted in a general "corner-cutting" attitude so that tasks like creating and maintaining documentation of any sort was thought to add too much overhead to the development process. This might not be a surprise taking into consideration the companies' low level of maturity on the CMMI scale. It would be interesting to repeat the same survey with a group of developers from a higher level of maturity company.

Given the focus on time factors, automation and reduced overhead were found to be critical keys to the adoption of a traceability tool for their team. Most of the respondents interviewed felt that the stress to complete current tasks as quickly as possible was more important than quality and cost. The distinction between what tasks increased overhead and which did not centered around whether the task actually added to their understanding beyond what current processes and tools could provide. For example, drawing or editing class diagrams to represent changes they made in code was considered tedious because (i) the problem had already been solved, and (ii) their IDE's were fairly good at automatically drawing these diagrams. In other words, the task was largely transcribing known information from one form into another. However, creating links between classes with the GST Tool was considered useful and less tedious because it was the dependencies that they were trying to understand during the maintenance task and dragging-and-dropping the relevant code took little extra time. Even if the dependencies are automatically generated (e.g. in a class diagram), they would still need to go through the links to learn how the system currently works. Thus time spent creating the traces between entities is not as tedious as creating classes and objects. Nonetheless, the interviews did indicate a strong desire to make more of the process automated since that could add even more to the speed of understanding.

5 Summary and Conclusion

The development and evaluation of the GST Tool prototype provided important insights into the importance of entity dependencies in a software development project and into how a full-blown dependencies-management tool might be constructed. The drag-and-drop user interface style appeared to reduce the overhead of creating links to a point where software engineers might be likely to use the tool. The multi-windowed interface also seemed to fit well with the development environment. The strategy of manual creation of links appeared acceptable as well, although automatic creation is an attractive feature for those entities where it is possible. Finally, the fact that the

tool was capable of tracking dependencies between many different kinds of entities in different development phases seemed valuable.

There were several limitations to this initial prototype evaluation. In particular, the tool was evaluated and demonstrated in the context of relatively small development projects. It is not clear what the tool's applicability in very large projects would be, where the chances might be high of becoming lost in a complex web of many dependencies.

In the course of iterative prototyping, several potential enhancements were revealed that we plan on exploring in the future. The first of these is a revamped architecture to provide a web-based interface. This enhancement will make the tool more sharable, transforming it into a truly collaborative tool. As part of the collaborative enhancements, a filtering facility is needed so that software engineers working in different roles can flexibly limit the scope of dependencies viewed to those that pertain to the problem at hand. A second enhancement has to do with the concern over comprehending dependencies when their number becomes large. There have been many advances in the display of large amounts of data and the GST Tool requires the ability to display the dependencies in a variety of graphical ways, such as variants of network graphs [18] [8], and potentially others. A final enhancement will be to explore the addition of automatic dependency detection where feasible, and potentially to combine that with the use of indirect links. There are a number of potentially useful tools that can be used within software engineering domains (see e.g., [11]). The possibility exists that some dependencies information obtained from automatic tools could be propagated through the indirect links to reveal dependencies in other entities for which automation is more difficult.

References

1. Neumuller, C., Grubacher, P.: Automating Software Traceability in Very Small Companies: A Case Study and Lessons Learned. In: Automated Software Engineering, ASE 2006, Tokyo (2006)
2. Seibel, A.: From Software Traceability to Global Model Management and Back Again. In: 15th European Conference in Software Maintenance and Reengineering, Oldenburg, GR (2011)
3. Vora, U.: Change Impact Analysis and Software Evolution Specification for Continually Evolving Systems. In: Fifth International Conference on Software Engineering Advances, ICSEA (2010)
4. Imtiaz, S., Ikram, N., Imtiaz, S.: Impact Analysis from Multiple Perspectives: Evaluation of Traceability Techniques. In: The Third Conference on Software Engineering Advances, Sliema, Malta (2008)
5. Cataldo, M., Mockus, A., Roberts, J., Herbsleb, J.: Software Dependencies, Work Dependencies and their Impact on Failures. IEEE Transactions on Software Engineering 35(6), 864–878 (2009)
6. Cataldo, M., Nambiar, S.: Quality in Global Software Development Projects: A Closer Look at the Role of Distribution. In: Fourth IEEE International Conference on Global Software Engineering (2009)

7. Nagappan, N., Ball, T.: Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In: The First International Symposium on Empirical Software Engineering and Measurement, Madrid, Spain (2007)
8. de Souza, C.: On the Relationship between Software Dependencies and Coordination: FIeld Studies and Tool Support. University of California, Irvine (2005)
9. Dam, H., Ghose, A.: Automated CHange Impact Analysis for Agent Systems. In: 27th IEEE International Conference on Software Maintenance, ICSM (2011)
10. Briand, L., Labiche, Y., Soccar, G.: Automating Impact Analysis and Regression Test Selection Based on UML Designs. In: Proceedings of the International Conference on Software Maintenance, ICSM 2002 (2002)
11. Cataldo, M.: Identifying and Managing Dependencies in Global Software Development. In: Oram, A., Wilson, G. (eds.) Making Software: What Really Works, and Why we Believe it, pp. 349–371. O'Reilly, Sebastopol (2011)
12. Gall, H., Hajek, K., Jazayeri, M.: Detection of Logical Coupling Based on Product Release History. In: Proceedings of the International Conference on Software Maintenance, pp. 190–198 (1998)
13. Basili, V., Briand, L., Condon, S., Kim, Y.-M., Melo, W., Valett, J.: Understanding and Predicting the Process of Software Maintenance Releases. University of Maryland, College Park (1995)
14. Ghazarian, A.: Coordinated Software Development: A Framework for Reasoning about Trace Links in Software Systems. In: International Conference on Intelligent Engineering Systems, Barbados (2009)
15. Thomas, N., Murphy, G.: How Effective is Modularization? In: Sebastopol, C.A. (ed.) Making Software, What Really Works, and Why We Believe It, pp. 389–408. O'ReillyMedia (2010)
16. Spanoudakis, G., Zisman, A.: Software Traceability: A Roadmap. In: Chang, S. (ed.) Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishng Co. (2004)
17. Cleland-Huang, J., Chang, C.: Event-Based Traceability for Managing Evolutionary Change. IEEE Transactions on Software Engineering 29(9), 796–810 (2003)
18. Quante, J.: Do Dynamic Object Process Graphs Support Program Understanding?- A Controlled Experiment. In: The 16th IEEE International Conference on Program Comprehension, Amsterdam (2008)