Supporting Program Comprehension in Agile with Links to User Stories

Sukanya Ratanotayanon
Department of Informatics
University of California, Irvine
Irvine, USA
sratanot@uci.edu

Susan Elliott Sim
Department of Informatics
University of California, Irvine
Irvine, USA
ses@ics.uci.edu

Rosalva Gallardo-Valencia
Department of Informatics
University of California, Irvine
Irvine, USA
rgallard@uci.edu

***Abstract*—Agile software development involves continuously making iterative and incremental changes to source code. When making changes, developers quickly focus on parts of code that they consider important, and sometimes miss other relevant parts. Therefore, tool support is needed to help developers locate conceptually related sections of code. We present Zelda, a tool designed to work with Agile practices that captures and maintains links between high-level information and source code. We evaluated Zelda with a pilot study where subjects were required to make a change to a small web application (10KLOCs). They were given a task description either on paper or in Zelda. We found that the Zelda Group made more accurate changes and were less likely to become disoriented.**

Keywords-component; user stories; program comprehension; link evolution; traceability links

I. INTRODUCTION

Projects using Agile processes are frequently confronted with fast-paced technical advances and rapidly changing requirements. Consequently, developers must make changes to source code quickly and accurately. Previous studies of program comprehension found that professional developers sought only to understand portions of code that were relevant to the task at hand [1, 2]. However, their perceptions of what was relevant were not always correct or complete. This mismatch is particularly problematic when an implementation of a concept or a feature spread across different parts of the code or artifacts. The scattering of code is common in modern software architectures because they often have multiple layers, and use frameworks, or existing services. The need for effective tools to support program comprehension is particularly high in Agile, where the software written today is next week's legacy code.

We seek to address this problem by providing links from high-level concepts to their locations in source code. These links can help developers to recognize scattered pieces of a concept, build mental models, and make connections between them. These links are similar to traceability links in plan-driven software development. Due to the distinctive software artifacts and the order in which they are produced, Agile presents both opportunities and challenges for using traceability links to support program comprehension.

On one hand, we cannot apply traditional traceability techniques, as there are usually no documents from which to trace. On the other hand, the sequence of artifact creation provides us a code-centric and lightweight mean to capture links. In Agile, units of work are defined as user stories and tasks. When using test-driven development, a developer begins by writing test cases and then writes just enough program code to pass the test cases. By recording traceability links among user stories, test cases, and source code, we can map information from high-level concept such as features and tasks to source code.

To this end, we created Zelda, an Eclipse plug-in that helps developers create links from these user stories and tasks to their source code, test cases and various text-based files. Zelda can manage user stories information or retrieves user stories and tasks from an external tool, such as XPlanner [3]. The process for creating links requires little effort, and the links are kept up to date automatically, so that developers can remain focused on coding. When the underlying artifacts are changed, links are updated by extracting information from a Revision Control (RC) system. By analyzing the results from differencing subsequent versions of artifacts, we can automatically determine the correct locations of links. Zelda also provides different types of link visualizations using file decorations, markers in the Eclipse editor, and a SeeSoft-style [4] visualization. These visualizations help with browsing the source code and provide a visual aid for creating a working set [5].

We evaluated Zelda in a small pilot experiment with two conditions and four subjects. Subjects were given a task description, either on paper or using Zelda, and subjects were asked to implement changes to existing software. We found that subjects in Zelda group were able to complete the task slightly faster and more accurately than those in the Paper group. While this result is difficult to generalize due to the small number of subjects, it is encouraging.

The remainder of the paper is organized as follows. Our approach for providing software comprehension support in Agile processes is presented in Section 2. Section 3 gives an overview of how we record and evolve traceability links. In Section 4, we introduce our prototype tool: Zelda. An evaluation of Zelda is given in Section 5. Section 6 summarizes related work and we conclude with future work in Section 7.

II. SUPPORTING PROGRAM COMPREHENSION WITH LINKS TO USER STORIES

When making changes to source code, developers need to be able to work quickly, while preserving the integrity of

the original design decisions and respecting the structure of the software. A program comprehension tool that directs developers to look at all and only relevant sections of code can aid this work. We hypothesize that this can be done by creating and maintaining traceability links to associate high-level information from user stories, task descriptions, and concepts, to its implementation in the source code.

In Agile, only minimal documentation is created, so we cannot employ existing traceability approaches, as there are no documents across which to trace. Since Agile is a code-centric process, we need a mechanism to ground the links in the source code. In addition, the overhead of capturing traceability links with traditional techniques is too high to be suitable for Agile. Instead, we need a process to capture traceability links that is lightweight and does not incur much overhead to the daily work. Lastly, Agile developers are normally working in a fast evolving environment, which means captured traceability information will deteriorate quickly, if not maintained.

On the other hand, the Agile environment presents a unique opportunity for using lightweight traceability for the purpose of supporting program comprehension. As software development is test-driven, and source code-centric, a development environment can be leveraged as a vehicle for traceability. In Agile, the artifacts that are produced are lightweight, for instance, units of work are expressed as user stories [6]. When using Test-Driven Development [7], a developer starts with a user story and begins by writing test cases and then writes just enough program code to pass the test cases. Because artifacts for a feature are closely associated chronologically, we can exploit the sequence in which artifacts are created to capture links. The linking process can be as simple as activating a user story or a task and associating test cases and source code with it.

The high-level information recorded in user stories and tasks can help developers answer questions such as what is the purpose of a section of code or test cases, and why are they implemented that way. Linked test cases can give information about expected behavior of classes and modules in the system. These links also provide an effective means to tie together the parts of an implementation of a user story that is scattered across various modules and artifacts. By helping developers answer these questions, these links help developers to discover parts of the code relevant to the required changes, and to understand the original design of the program. In addition, mappings between a user story and its implementation can present an example of how a similar, new user story should be implemented.

Since Agile emphasizes working software, the mechanism for linking user stories and tasks to underlying artifacts must require little effort. Consequently, we have incorporated the mechanism to create, maintain, and present the traceability links into an integrated development environment (IDE). Integrating traceability features into an IDE takes advantage of source code, and similar text documents, as central artifacts in software development. In the same manner that development activities center around source code, so too, do the representation and maintenance of user stories and links. Visualizations of these links help developers to easily see and access the scattering implementation of a user story or a task.

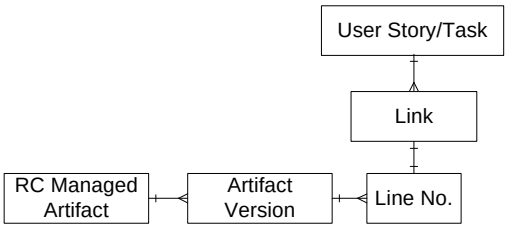

Figure 1. Model of Links Data

III. Recording and Evolving Links

Our approach takes advantage of key practices in Agile to provide a lightweight and code centric method to capture traceability links. It captures and evolves the links between user stories, tasks and their implementation, including test cases and source code. It is a common practice to keep source code under a RC system, and we leverage this practice to record and evolve links between source code and other work artifacts.

A. Link Recording

We use the following example scenario to help illustrate the mechanics and the benefits our approach.

Mike was working on a web-based application. Mike started working with a user story, named 'maintain user data', which stated "As an admin, I can create and save user accounts, including username and password, so users can access the system.." With our tool, Mike activated the user story from his IDE. To implement this user story, he created test cases and several new source files. He also modified some existing configuration files containing captions on the user interface in different languages.

When he finished, Mike checked in his changes to the RC system. At this point, our tool recorded links from the user story to lines that were added or changed in the checked in files. Mike also manually associated lines in the configuration file with the user story. He did not modify these lines, but felt that they were relevant to the task. Having completed the task, Mike deactivated the user story.

Our tool creates and stores links by integrating together different parts of an Agile developer's working environment. To facilitate in recording link, our tool provides an interface to communicate to a user story management tool, so that a developer can access user stories and tasks right in the IDE. The link recording occurs when a user checks in or manually selects lines to be linked. At check in, our tool queries the RC system for lines that are added or modified and creates links from the active user story to these lines.

The recorded links are stored in their own database. Benefits of this approach are that it is simple and it requires the fewest modifications to existing software tools and work practices. Fig. 1 is the data model for the link records.

B. Updating Link Locations

The continuation of our scenario is used to explain the mechanics of how links are updated and presented.

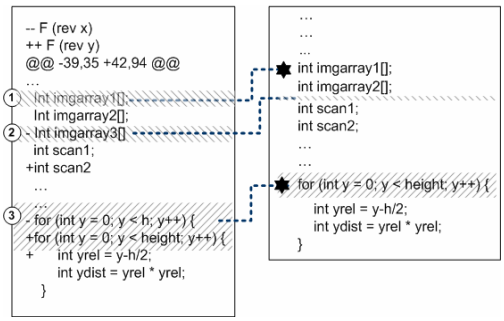

Figure 2. Locating Links with Diff result

Five months later, another developer, Ken, needed to modify the feature for maintaining user data. In the intervening time, many changes have been made to the code base, including a number of new features related to the 'maintain user data' story and a bug fix. To complete this task, he needed to locate all relevant parts of code to be modified. He used our tool to follow the links from the 'maintain user data' user story to lines in the most recent version of the code base.

The links that Mike created were helpful to Ken. It is easy to see from description of the user story that the implementation of 'maintain user data' story will be related to Ken's task. Despite time passing and numerous changes to the code, we were able to show the links in the most recent version of the file, because we automatically maintain links over successive changes.

We keep our links up to date by piggy backing on an RC system. To update links, our tool request RC system to produce difference information ('diff') between the original linked versions of files and their current versions. An example of a diff result in the unified format can be seen in the box at the left of Fig. 2. We analyze the difference result to identify updated position of a link in the most recent versions of the target file. Detailed information about our algorithm for evolving traceability links can be found elsewhere [8].

The user story can have links added at different times, so it is possible to have links pointing to different versions of the same file. Our tool updates these links starting from the version where they were injected and then accumulates them so that they can be presented together.

The updated links can be "unchanged," "removed," or "modified" as shown by the lines labeled 1, 2, and 3, respectively in Fig. 2. An "unchanged" line has the same content, but may have moved to another location within the file, for instance, the line 1. Link 2 is an example of a "removed" link that cannot be found, and is not shown to the user. A "modified" link is one where the content has changed, and may have moved. See the link number 3 for an example of an unchanged link. The 'modified' links will be presented to a developer, but will look different from ones that are 'unchanged.'

IV. ZELDA

A prototype platform for recording and maintaining traceability links in Agile environment, called Zelda, has been implemented as an Eclipse plug-in. Zelda comes with a built-in system that allows users to maintain user stories and tasks. The information of user stories and tasks is stored in MySQL database. It is also possible to import user stories and tasks from an external source, such as XPlanner [3]. An interface has been provided to communicate with these external systems.

Fig. 3 shows a screenshot of the Zelda tool. The available user stories and tasks of a developer can be seen in the Browser view, labeled ***C***. A user stories or tasks that the developer is interested in or currently working on can be sent to the Virtual Stack view, labeled ***E***. This view presents a list of current working set of user stories and tasks. The user can activate and de-activate user stories and tasks in the list using actions on the view to created links to files.

When present the links related to a user story or a task, Zelda analyzes the link information stored in the database and the diff results obtained from Subversion to determine the current locations of the links. The updated link locations are shown in the following visualizations.

Overview: Visualiser plug-in from the AJDT group is used to implement this visualization, labeled as ***B***. It presents an overview of all links to files and their associated lines. The files are shown as long blocks and the associated lines are shown as horizontal stripes within the blocks. This overview provides easy access to the implementation of a user story. The developer can jump to a section of the code by double clicking on either the block or the stripe.

File Decorations: The links of a user story or a task are presented at the file-level granularity using a file decoration in the package explorer view in Eclipse, as shown in the area labeled ***A***.

Markers: Markers are used to show links at the line level, as shown in the area labeled ***D***. A marker is used to present the most recent location of each link. By using markers, links can be presented in a manner that is grounded in the source code.

Program Element Tree: This view raises the level of abstraction in which we show links to source code. Program elements containing lines that are linked to a user story or a task is shown in a tree view using its hierarchical structure as shown in the area labeled ***F***.

V. EVALUATION

We conducted a pilot study to evaluate Zelda. The goal was to conduct an initial assessment of our tool for helping developers to deal with scattered implementation of a user story or a task. In the study, we asked subjects to modify an existing web-based survey management application. There were a total of four subjects. We divided them into two equivalent groups, each consisting of one undergraduate student and one professional software developer. All subjects reported having experience in Java and web development. However, none of them had experience with either JSP (Java Server Page) or XML.

To ensure that all subjects were working from the same starting point, we provided them with a description of what would be involved in the required changes. This description would be analogous to preliminary research done by a

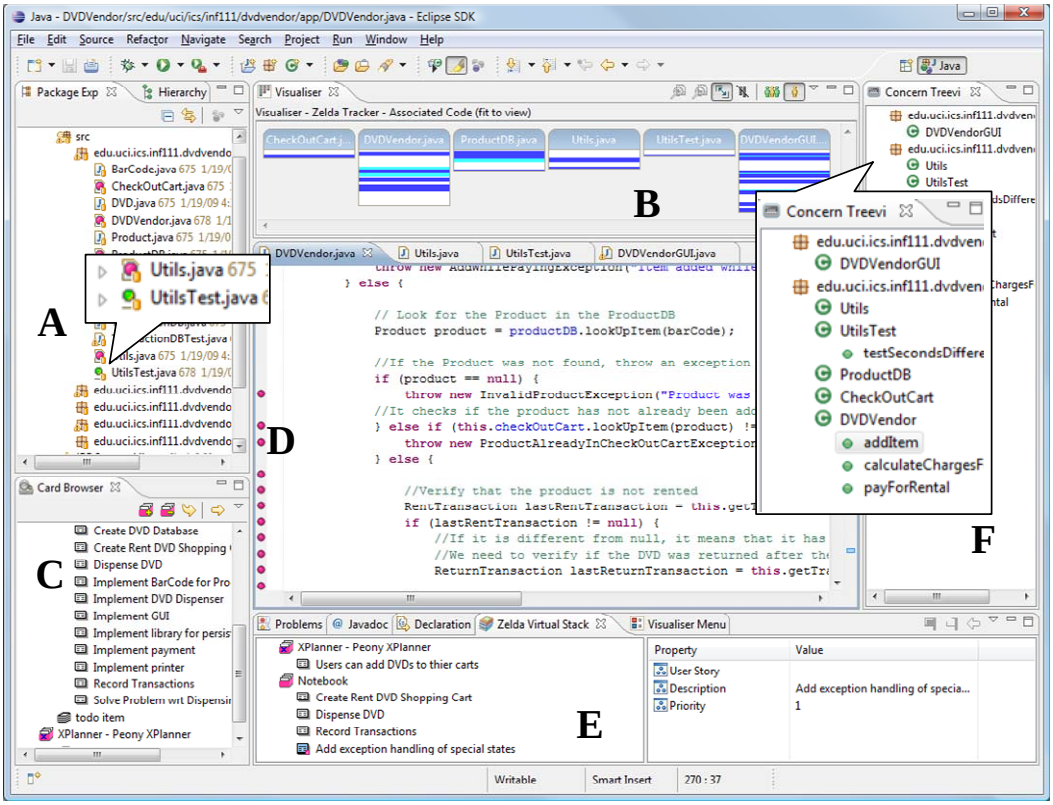

Figure 3. Screenshot of Zelda User Interface and Link Visualizations

developer to create an accurate effort estimate or risk assessment. The subjects received the task description and the preliminary research result either in Zelda or on paper, and they were required to implement the changes. In addition, subjects were asked to think aloud as they worked to provide us with additional insight into their behavior.

A. *Software and Requested Change*

The application used in this study was an open source web-based survey management tool called VTSurvey [9]. It is a typical three-tiered web application running on a Tomcat application server. VTSurvey consists of 38 Java files, 74 JSP files, and 4 DTD (Document Type Definition) files. There are a total of 10,342 lines of code in the application.

VTSurvey allows an administrator of the application to create, delete, and edit users of the system. Initially, it did not maintain each user's email address. Subjects were requested to modify the system to accept, maintain, and present this data. We gave the subjects an outline of the solution, which included: i) change description, ii) system description, iii) files affected by the change (including description of work required in each file), iv) risks, v) effort estimation, and vi) test cases.

In the Zelda Group, subjects were given the task description with links to relevant information in Zelda and they were allowed to use Zelda as they wished. In the Paper Group, subjects were given the same information on paper and did not have access to Zelda. Both formats of task description contained the same information and were structured similarly.

B. *Procedure*

Each study session had four parts: background questionnaire, familiarization task, implementation task, and debriefing. The background questionnaire asked subjects about their educational background, employment history, and level of experience with some software technologies. In the familiarization task, we showed the subjects the programming environment, and how to compile and run the project. Subjects were given a step-by-step instructions for implementing a 'Hello, World' JSP page and were asked to implement the page. In addition, the Zelda Group received training on how to use Zelda in this task. In the implementation task, subjects were given the task description in a format according to their group and were asked to implement changes specified in the task description. The change was divided into two sub-tasks:

Task A: The system shall allow the administrator to provide an email address upon adding a new user.

Task B: The system shall present the user's email address as another column on the user account listing page.

The subjects had a maximum of 90 minutes to complete the task. At the end of that time or once they had finished the implementation, whichever occurred first, we debriefed the subjects. They were asked to evaluate their own performance. As well, subjects in the Zelda Group also had the opportunity to express their opinions about the tool.

During the study, subjects were asked to think aloud as they worked to provide us with additional insight into their behavior. The subject's activities were recorded by a web camera, a microphone, and screen capture software.

C. Results

On the average, the time subjects in both group spent in completing each of the two subtasks is comparable. To analyze the accuracy of the implementation, the work of each subject was scored out of 60. The average scores for both groups are the same. The results are shown in Table 1. Subject2 and Subject3 who received high scores are both professional software developers. The results show that individual differences were the biggest factor in the success of the subject, more so than the treatment factor. This result is similar to those obtained by Robillard and Murphy [10].

We also analyzed the number of relevant and irrelevant files viewed by the subjects. A relevant file was defined as a file containing information that could help a subject to perform the implementation task, but may or may not need to be changed. As presented in Table 1, we observed that the subjects in the Paper Group viewed more irrelevant files. We hypothesized that Zelda helped subjects to focus on relevant files. Upon further inspection, we found that both subjects in the Paper Group missed the same relevant file mentioned in the outline of the solution, which caused them to miss the internationalization strategy in the software. As a result, both subjects hard coded the captions on a JSP page which would cause an inconsistency in the application in languages other than English.

TABLE I. IMPLEMENTATION SCORE AND NUMBER OF FILES VIEWED

	Zelda Group		**Paper Group**	
	Subject1	***Subject2***	***Subject3***	***Subject4***
Score (60)	25	55	50	30
#Relevant Files	10	11	13	13
#Irrelevant Files	0	1	6	5

In another instance, both subjects from the Paper Group had trouble finding the correct DTD file to modify. The application has two DTD files with the same name, but in different locations. We provided the location of the correct file in the task description as a part of the preliminary investigation result. However, one subject in the Paper condition modified the wrong file and the other subject had trouble finding the DTD file using Eclipse's IDE and had to search for it. None of these difficulties was encountered by subjects in the Zelda Group.

Another instance that showed how Zelda's links provides better support for accessing relevant information is how subjects did or did not make use of architectural information. The task description in both formats had a URL for a web page that contained information about the system architecture. Both subjects using Zelda followed the links and viewed the architecture information, while subjects using paper version of the task description did not.

In the debriefing, subjects who used the Zelda gave two main reasons for its success. One, Zelda helped them to concentrate on relevant files and to quickly access the code needed to be changed. Two, Zelda allowed subjects to see the task description and the code in the same environment. As one subject said,

". . . I did find it (link overview visualization) useful. Mostly because it showed me what files were relevant for the change and having the associated code is great. It is nice to swap back and forth while I was doing the changes. . . . "

He enthusiastically thought of how the tool could be useful in other contexts,

". . . for example, in one of the internships I did, like the first day I was said. Here is a problem, you need to solve this. I had no idea where to start. It would have been wonderful to have something where the guy who was my mentor had prepared something. This is the part of the system that you need to look at. It would have been great. . . ."

D. Threats to Validity

One of our main threats to validity is a small number of subjects. Although our results were positive, we could not generalize the results. However, we did gain insight into aspects of Zelda that subjects found useful.

The laboratory setting could also be a threat to internal validity of this study. Zelda relies on having links between source code and user stories and tasks. In this study, we provided the links and knew that they were accurate. This may not be the case in more naturalistic settings. Another factor that may have affected performance was that participants were unfamiliar with the tools, the domain, and the programming languages used. Despite these factors, Zelda showed promising results in helping programmers focusing on the relevant information. However, to obtain more accurate results, we need to conduct a long-term study of Zelda usage in a practical software project using Agile processes. A longitudinal study could reveal more nuanced aspects of how Zelda is used and how well it supports program comprehension in Agile.

VI. RELATED WORK

To our knowledge, Zelda is the only tool designed specifically to support program comprehension in Agile environments. In addition to paying attention specifically to Agile work practices, our approach builds on existing research in both program comprehension and traceability.

The concept of using mappings between high-level concepts and source code has been implemented in a number of tools. For instance, both FEAT [11] and Mylyn [12] create links to source code, in particular, links and annotations are associated with program elements. Non-code files were linked only at the file level. In contrast, Zelda links to specific lines in text files, including non-source documents. Also, these tools do not aim to maintain their links after underlying artifacts change. Zelda keeps link locations up to date, by taking advantage of an existing RC system.

There is a large body of research in traceability, with some adaptations specifically to Agile. The most common definition of traceability is requirements traceability, which is defined as "the ability to describe and follow the life of a requirement, in both a forwards and backwards direction"

[13]. Typically, research in traceability is not compatible with Agile, because they mainly concerned with linking high-level documents [14-16]. The main drawbacks of these tools is the high cost of defining and capturing traceability relationships between artifacts, a steep learning curve, and no support for linking source code. There is a traceability approach [17] intended to work with Agile, but has similar limitations. The traceability features of these tools are more sophisticated than those in Zelda. For instance, they can help manage requirements and analyze the consistency of requirements between documents. Instead, Zelda uses a relatively simple links and in exchange the tool is more lightweight, simpler to use, and tailored for Agile.

There are other approaches in the traceability literature that may be applicable to Agile software development and Zelda. Some tools can automatically discover traceability links from available artifacts to source. A variety of approaches have been used, including analyzing the runtime trace of a scenario [18] and analyzing comments in source code [19]. It may be possible to add this functionality to Zelda in the future.

VII. Conclusion and Future Work

We present an approach for providing software comprehension support in Agile software development using traceability links, and present our prototype tool, Zelda. We leverage the sequence in which artifacts are created in Agile development to provide a lightweight, code centric means to capture links between user stories, test cases, and source code. These links provide explicit mappings to help developers to find information necessary to complete the task, including details about the scattered implementation of concepts. These links reduce cognitive load in locating and forming a working set of task-relevant code fragments. It also provides a means to reuse existing knowledge by presenting examples of code usage and strategies for implementing a feature. To ensure the long-term value of these links, Zelda can automatically maintain the correct location of these links in the later versions of the artifacts, even after they were affected by successive changes.

We performed a pilot study to evaluate Zelda with a small web-based survey management application. Subjects were given a task description containing preliminary investigation results for required changes, either as in Zelda tool or on paper. We found that although subjects who used Zelda perform only marginally faster, they were more focused, and were more willing to make use of additional information.

We have only focused on using Zelda with a newly created project so far. Another aspect that needs to be examined is how to use Zelda with projects that are in progress. Zelda must be able to support comprehension for the purposes of adaptive, corrective, and perfective maintenance. As well, adoption of Zelda must be able to proceed incrementally and in a manner that is compatible with existing tools. To this end, features are needed to import existing information from various sources to create links. For example, an existing check-in comment and a bug report can be associated with the code from the change set for a particular commit operation in a revision control system.

VIII. Acknowledgments

Many thanks to Derek Raycraft for his help in editing the paper. Also thanks to our subjects for their participation in the study. This research was supported in part by a grant from the Agile Alliance Academic Research Program.

References

[1] A. Lakhotia, "Understanding Someone Else's Code: Analysis of Experiences,"Journal of System Software, vol. 23, pp. 269-275, 1993.

[2] V. Subramaniam and A. Hunt, Practices of an Agile Developer: Working in the Real World, Pragmatic Bookshelf, 2005, 097451408X.

[3] XPlanner http://www.xplanner.org/ 2009.

[4] S.G. Eick, J.L. Steffen and E.E. Sumner, "Seesoft-A Tool for Visualizing Line Oriented Software Statistics," IEEE Transactions on Software Engineering, vol. 18, pp. 957, 1992.

[5] A.J. Ko, H.H. Aung and B.A. Myers, "Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks," in Twenty-Seventh International Conference on Software Engineering, pp. 126-135, 2005.

[6] M. Cohn, User Stories Applied: For Agile Software Development, Addison-Wesley, 2004, 0-321-20568-5.

[7] K. Beck, Test-Driven Development by Example,2002,0-321-14653-0.

[8] S. Ratanotayanon, S.E. Sim and D.J. Raycraft, "Cross-Artifact Traceability using Lightweight Links," in Proceedings of the 5th International Workshop on Traceability in Emerging Forms of Software Engineering, 2009.

[9] VTSurvey http://vtsurvey.sourceforge.net/ 2009.

[10] M. P. Robillard and G. C. Murphy, "A Study of Program Evolution Involving Scattered Concerns." Technical Report CS-2003-06, Department of Computer Science, University of British Columbia, March 2003.

[11] M.P. Robillard and G.C. Murphy, "Representing Concerns in Source Code," ACM Transaction on Software Engineering Methodology, vol. 16, pp. 3, 2007.

[12] M. Kersten and G.C. Murphy, "Mylar: A Degree-of-Interest Model for IDEs," in Proceedings of the 4th International Conference on Aspect-Oriented Software Development, pp. 159-168, 2005.

[13] O.C.Z. Gotel and C.W. Finkelstein,"An Analysis of the Requirements Traceability Problem," in Proceedings of the 1st International Conference on Requirements Engineering, pp. 94-101, 1994.

[14] L. Naslavsky, T.A. Alspaugh, D.J. Richardson and H. Ziv, "Using Scenarios to Support Traceability," in Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering, pp. 25-30, 2005.

[15] T. Hughes and C. Martin, "Design Traceability of Complex Systems," in Proceedings of the Fourth Symposium on Human Interaction with Complex Systems, pp. 37, 1998.

[16] B. Azelborn, "Building a Better Traceability Matrix with DOORS," in Telelogic INDOORS Europe, 2000.

[17] C. Lee, L. Guadagno and X. Jia, "An Agile Approach to Capturing Requirements and Traceability," pp. 17-23, 2003.

[18] A. Egyed, "A Scenario-Driven Approach to Traceability," in Proceedings of the 23rd International Conference on Software Engineering, pp. 123-132, 2001.

[19] J. Sayyad-Shirabad, T.C. Lethbridge and S. Lyon, "A Little Knowledge can Go a Long Way Towards Program Understanding," in Proceedings of the 5th International Workshop on Program Comprehension, pp. 111, 1997.