Towards Traceable Test-Driven Development

Jane Huffman Hayes
Computer Science
University of Kentucky
Lexington, KY 40506
U.S.A.
hayes@cs.uky.edu

Alex Dekhtyar David S. Janzen
Computer Science Department
California Polytechnic State Univ.
San Luis Obispo, CA 93407
U.S.A.
{dekhtyar, djanzen}@csc.calpoly.edu

Abstract

Key among the Grand Challenges in Traceability are those that lead to achieving traceability as a by-product of the natural software development life cycle. This position paper profiles test-driven development (TDD), an emerging software development practice, in which automated tests and code satisfying them are developed in rapid succession over multiple iterations. Our position is that the nature of TDD offers unique opportunities for collecting traceability information throughout the TDD life cycle and that the provision of traceability information to the software developers during TDD will improve the process and the resulting software. We discuss the opportunities, challenges, and plans for the synthesis of TDD and traceability.

1. Introduction

Despite the many benefits realized when traceability information is generated and maintained[1], and despite the fact that traceability is mandated for many fields[2], traceability information is still not commonplace in software projects. A long term vision for the traceability research community is that of "one click tracing." This goal envisions a future where software engineers are developing and maintaining traceability information as they perform their normal duties, with traceability being captured and updated as a by-product of their work and completely unbeknownst to them (or perhaps as a result of "one click" of the mouse). At least two challenges must be overcome before this can be a reality, as documented in the traceability research community's Grand Challenges of Traceability [1]: *"(1) C-GC2Develop incremental, almost real-time, traceability recovery approaches to be integrated into Integrated Development Environments*;" and "*(2) C-GC3 Develop change management systems that effectively support the evolution of traceability links across multiple artifact types* [1]."

To date, a number of advances have been made toward one click tracing. For example, automated tracing using information retrieval (IR) methods has been shown to be capable of recovering traceability information for structured (such as source code) and unstructured (such as natural language requirements) artifacts [2,3,4,5]. Event-based tracing has been introduced to assist with the maintenance of traceability information (to capture changes to the traceability information when the underlying artifacts change) [6]. A tool called TraceAnalyzer acknowledges the importance of models and recovers traceability between software systems and models using scenarios (test cases) during program execution [19]. Frameworks have emerged that attempt to capture traceability information as an integral part of a detailed development process, such as Jazz [7]. However, there is still no general purpose mechanism for achieving this long term goal for any and all software projects and artifacts.

Yet another step toward one-click traceability is seamless integration of tracing within traditional and emerging software development paradigms. In this paper, we discuss the need, the potential, and the challenges of such integration of tracing into the test-driven development (TDD) [8] framework.

We posit that the TDD process is well-suited for achieving traceability as a by-product of development, **and** that availability of timely traceability information has the potential to improve TDD itself. Indeed, the key aspect of TDD (correlated (i.e., connected) co-

[1] Such as performing satisfaction assessment, performing change impact analysis, ensuring no unintended functions, etc.

[2] It is required by ISO/IEC 15504, Sarbanes-Oxley Act (SOX), the FDA requires it for medical device software, and the FAA requires it via DO-178B.

changes in tests and code/design throughout the development cycle) can be viewed as the source of (almost) free traceability information. In turn, if such information is available to the developer, it may improve the efficiency with which tests are produced and code is written for each iteration of the process. Of course, issues do exist with this approach. But, in our view, the approach is worthy of examination to see what might be learned toward the broader, longer term goal of one click (or by-product) tracing.

As such, we examine this long-term goal in the context of TDD and traceability. This paper addresses: (a) our position on the issue; (b) analysis supporting our position; and (c) suggestions for future research in the area.

The rest of the paper is organized as follows. In Section 2, we express our position concerning by-product tracing and how TDD might assist. In Section 3, we discuss the emerging relationship between traceability and TDD as well as obvious challenges. In Section 4, we discuss the challenges and questions that the proposed research will address in the future.

2. Position

Test-driven development is an emerging software development practice that has been shown to improve software quality in terms of lower defect density [16,11], higher test coverage, and smaller, simpler code [Janzen]. We believe that:

1. The nature of TDD should provide for seamless integration of collection of traceability information, and
2. There are significant benefits for the TDD process when augmented with traceability information.

In other words, traceability researchers should welcome test-driven development as one of the software development paradigms in which the goal of one-click traceability, set forth by the C-GC2 and C-GC3 Grand Challenges in Traceability [1], is imminently achievable. At the same time, incorporation of traceability information will be a *bona fide* improvement of TDD.

3. Test-Driven Development

Test-driven development (TDD) is a disciplined development practice that involves writing automated unit tests prior to writing the unit under test. By writing a test first, the software developer must make detailed design decisions such as determining the interface and expected behavior of a unit before actually implementing the unit. A common misconception is that ALL of the tests are written prior to implementing the code [11]. Rather, TDD involves short, rapid iterations of "write a test, write the code to make the test pass, and refactor." These short iterations provide rapid feedback. Refactoring of both the test and code ensures that everything is performed to ensure simplicity and readability of emerging code.

Traditional TDD focuses on unit tests (methods and classes) and occurs primarily in the software construction phase, often following some level of requirements engineering and software architecture definition. Variations on TDD have emerged, such as storytest-driven development (STDD) [9] and acceptance test-driven development (ATDD) [10], which focus on requirements acceptance tests. STDD and ATDD rely on automated testing frameworks such as Fit (*http://fit.c2.com*), FitNesse (*http://www.fitnesse.org/)*, and FitLibrary (*http://sourceforge.net/projects/fitlibrary*) to specify executable acceptance tests. ATDD encourages the software professional to design in APIs behind very thin user interfaces [12].

Next, we examine how TDD might assist traceability and vice versa.

3.1. TDD and Traceability

In TDD and its variants, code, test cases and design evolve simultaneously. The automated tests should be easily traced to code as depicted in Figure 1.

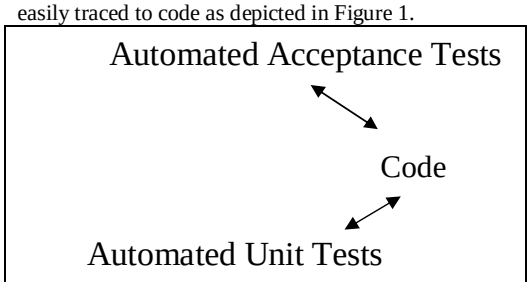

Figure 1: Tracing automated tests to code.

In agile processes, requirements are typically captured in user stories or use cases which are uniquely numbered. It seems perfectly reasonable to add a mechanism whereby automated acceptance tests include direct references to the requirements they test, extending our traceability as depicted in Figure 2.

A partial Requirements Traceability Matrix (RTM) is obtainable as a direct byproduct of the TDD process by matching new tests with changes in the code and in the design. This only requires two things: access to all versions of the artifacts for which the RTM is being constructed, and the ability to match versions of the

artifacts correctly. Both are achievable within the TDD process by using version control systems and committing all the artifacts after every successful run of the program (i.e., a run which succeeds on all current tests).

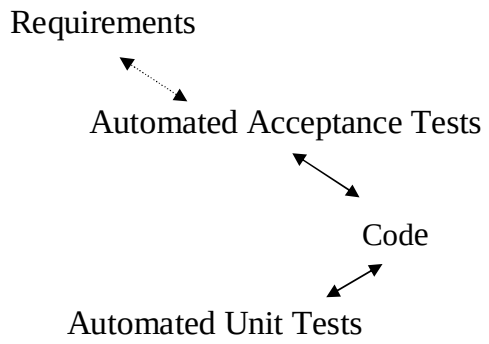

Figure 2: Extending traceability from requirements to code.

3.2. Traceability and TDD

On the flip side, the TDD process might be improved with the presence of easily accessible traceability information. Developers following the TDD process are constantly making important decisions, e.g., which feature of the product to test next or when is the right time to refactor. Traceability information available to developers in real-time as the development proceeds can inform their decision-making and help them determine what tests and code still need to be written and when refactoring needs to be triggered. Additionally, as developers refactor the tests and code, traceability information can help them with regression. An RTM link broken during refactoring may be treated as a warning for possible code regression.

Real-time traceability information has the potential to improve automated test-case generators such as Jtest (*http://www.parasoft.com/jsp/products/home.jsp?product=Jtest&itemId=14*) to help fill in some of the tedious test-writing details once we've gotten the design benefit out of TDD. For example, we use TDD to write a unit test for some yet unwritten class/method. This is where detailed design decisions about the to-be-written unit are made. After the unit is written and passes the tests, we refactor and write another test. At some point, new tests won't be adding much to the detailed design, but will remain important for unit and regression testing. With traceability information available, we envision the opportunities for traceability-aware automated test generation. An automated test generator, for example, can use the current RTM to determine untested functionality and generate test cases for it.

3.3 Challenges to Traceable Test-Driven Development

As the adage goes "do not look a gift horse in the mouth," yet we must admit that traceable test-driven development is not without challenges. That is, there is no "free traceability."

First, though we may end up with a partial RTM as a byproduct of TDD (as discussed in Section 3.1), the RTM may be incomplete due to: (a) new code mapping to existing test/design, for example, when a design element is satisfied by multiple tests/code fragments, and/or (b) refactoring. Refactoring is an important aspect of TDD, but can pose severe challenges to traceability. Refactoring of code can occur anywhere in the code base, and may lead to the appearance of new traceability links and the disappearance of old traceability links between tests/requirements and code. Additionally, refactoring may lead to temporary code degradation, when some of the existing tests fail to pass. We thus see two complementary challenges: to re-establish traceability after refactoring and to use traceability to improve refactoring.

When refactoring, the TDD developer must ensure that all automated tests continue to pass. Refactoring of both the automated tests and the code may occur, but the TDD developer never works for more than a few minutes without ensuring that all tests still pass. As a result, following some refactoring, a traceability matrix might be automatically rebuilt with every successful run of the automated tests. Or, at a minimum, failing tests could be set as triggers for updating traceability if manual traceability updates are necessary.

A second challenge is that of defining what information is readily available as part of the TDD process as opposed to what information traceability researchers would view as the 'ultimate' level of information to have available. For example, when dealing with a change to a user story and hence to tests for that user story, researchers in traceability envision having data such as what triggered a change, when was a change triggered, who triggered the change, what is the impact of the change, etc. In reality, readily capturable information may merely include the IDs of test cases validating the changed user story and perhaps methods that are executed as a test case is run (if some sort of dynamic analyzer is used).

Another challenge that is more dogmatic than technical is that of "burdening" the otherwise lightweight and agile process of TDD. At one point, the agile community considered adding an item to the agile manifesto which "forbade" traceability. It is of the utmost importance that any merging of TDD and traceability be as imperceptible and seamless as possible.

3.4 Advantages of Traceable Test-Driven Development

The challenges presented above suggest directions of research to be undertaken before integration of traceability and TDD can take place. ***If the challenges discussed above can be overcome***, a number of benefits can be realized through the merging of traceability and TDD. In particular, a TDD development environment integrating the notion of co-changing artifacts can help to mitigate the challenges outlined above. The notion of co-changing artifacts has been successfully applied in software evolution and in mining of software repositories [13,14,15]. In [15], co-changing artifacts are limited to files and defined as files that were modified almost at the same time[3] (around the same moments in time). In a TDD process, one can assume that the same time variability occurs - test scenarios and test cases can be written long before the code or immediately before coding. Also, the time frame for refactoring actions can be spread fairly evenly. As data is collected by the development environment, there is the possibility to define rich data formats, specifically targeted for traceability documentation.

Moreover, the co-change notion can be easily applied at the function and method level without the accuracy problem of [14] since in TDD the environment knows exactly the instant in time and the modified functions, methods, or classes. For example, the environment can report who changed what, what else was changed by the same developer, what test case and user story caused this affect, and so on.

In mining software repositories [13,14,15], co-changes neither necessarily imply a causal relationship among the individual changes of the different co-changing files nor logical dependencies among the files. For example, two files may change at the same time because developers decided to adopt a new license. However, in a TDD process, a non-intrusive development environment, as stated above, can record events together with other information such as the developers' IDs, thus producing more accurate co-changing information. Causality will always be questionable. However, in TDD, the co-changing relation of test cases, code, or code modifications due to refactoring actions will be a much stronger indication that a causal relation may indeed exist.

In essence, a TDD process will produce, almost for free, a network of co-changed and co-changing relations helpful to document traceability relation refactoring and modification as well as to more easily identify co-changing artifacts. These co-changing and co-changed relations can be used to evolve and refactor traceability relations, keeping them up to date with a minimum of manual intervention (limited to those cases where contradicting facts are discovered).

In addition to these benefits, we know that new test cases will be written at some point in the future when new user stories are provided and must be implemented. When writing a new test case, it is certainly possible that code already exists that might partially trace to the new test cases. One must have traceability information in order to determine if such code exists. It is also possible that when new code is written for a new test, that new code might also trace to formerly existing test cases. Again, traceability information can assist with this determination.

4. What to study and how to study it?

As detailed above, our experience suggests that we need to study (a) the data that can be collected, analyzed, and provided back to the developer throughout the TDD process; and (b) the methodology for collecting, maintaining, and modifying traceability information throughout the TDD process. We address each below.

The following types of data need to be studied:

- the data that is currently captured by TDD developers as they work,
- data that could easily be captured in the background as TDD proceeds (such as executing a dynamic tracer as TDD developers run test cases and automatically capturing the method names that are executed when a test case is run; tools such as calgrind, valgrind, Hackystat, and Zorro [18] might prove helpful),
- data that could be offered to the TDD developer as they do their work (for example, if a test case is added, run IR methods in the background and suggest code that may already satisfy part of the test case), and
- data could be offered to the TDD developers to suggest what tests need to be re-run based on the

[3] The meaning of *same time* is left intentionally vague since in [15] the goal was mining software repositories and thus time could be measured in minutes or hours, depending on the data in a given CVS repository.

methods touched as part of refactoring (see the second item in the list above).

On the methodology side, we essentially need to study the means of obtaining, maintaining, and modifying traceability information and the means of reporting it back to the developer. In particular, we need to study:

- the means of creating (candidate) RTMs from artifact co-change information,
- the means of establishing links between new and old portions of artifacts,
- methodology for (re)-capturing traceability information after refactoring operations, and
- visualization and presentation of the on-the-fly traceability information to the developer.

The next step after addressing these issues is design of methodology for improving TDD by leveraging traceability information. In particular, we need to study:

- in-process guides that identify requirements remaining to be implemented,
- in-process identification of inconsistencies between requirements and unit tests,
- traceability-aided automated test generation, and
- generation of change-impact information when refactoring (tools such as Chianti [17] might prove helpful here).

5. Acknowledgements

This work is funded in part by the National Science Foundation under NSF grant CCF-0811140. We thank Giulio Antoniol for his ideas on co-changing.

6. References

[1] Grand Challenges in Traceability, *Center of Excellence of Traceability Technical Report*, COET-GCT-06-01-0.9, September 10, 2006.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, Volume 28, No. 10, October 2002, 970-983.

[3] Jane Cleland-Huang, Raffaella Settimi, Oussama Ben Khadra, Eugenia Berezhanskaya, Selvia Christina: Goal-centric traceability for managing non-functional requirements. *ICSE* 2005: 362-371.

[4] Andrian Marcus, Jonathan I. Maletic: Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. *ICSE* 2003, pp. 125-137.

[5] Jane Huffman Hayes, Alex Dekhtyar, Senthil Karthikeyan Sundaram: Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. *IEEE Trans. Software Eng*. 32(1): 4-19 (2006).

[6] Jane Cleland-Huang, Carl K. Chang, Mark Christensen, "Event-Based Traceability for Managing Evolutionary Change," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 796-810, September, 2003.

[7] Jazz, www.ibm.com/software/rational/jazz.

[8] Kent Beck. *Test Driven Development: By Example*, Addison-Wesley, 2003.

[9] Rick Mugridge, Managing Agile Project Requirements with Storytest-Driven Development, *IEEE Software*, 25(1): 68-75 (2008).

[10] Borge Haugset, Geir Kjetil Hanssen, Automated Acceptance Testing: A Literature Review and an Industrial Case Study, *Agile* 2008, pp.27-38.

[11] David S. Janzen, Hossein Saiedian, Does Test-Driven Development Really Improve Software Design Quality?, *IEEE Software*, 25(2): 77-84 (2008).

[12] Robert C. Martin, "The Test Bus Imperative: Architectures That Support Automated Acceptance Testing, *IEEE Software*, 22(4): 65-67 (2005).

[13] G. Antoniol, F. Rollo, G. Venturi. Detecting groups of co-changing files in cvs repositories. In *International Workshop on Principles of Software Evolution*, pages 23--32, Lisbona, Portugal, Sept 2005.

[14] Zimmermann, P. Weissgerber, S. Diehl, A Zeller Mining version histories to guide software changes. In *ICSE*, pages 563{572, 2004

[15] S. Bouktif and Yann Gael and G. Antoniol Extracting Change-patterns from CVS Repositories Proceedings of *IEEE Working Conference on Reverse Engineering* , Benevento, Itably, Oct 23-27 2006, pages:221-230;

[16] Jeffries, Ron Melnik, Grigori, Guest Editors' Introduction: TDD--The Art of Fearless Programming, *IEEE Software*, Vol 24, Issue 3, May-June 2007, pp. 24—30.

[17] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for practical change impact analysis of Java programs. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems and Applications (OOPSLA)*, pages pp 432–448, October 2004.

[18] Johnson, P. M. and Kou, H. 2007. Automated Recognition of Test-Driven Development with Zorro. In *Proceedings of the AGILE 2007* (August 13 - 17, 2007). AGILE. IEEE Computer Society, Washington, DC, 15-25.

[19] Egyed, A. A Scenario-Driven Approach to Traceability. In *IEEE Proc. Int. Conf. on Software Engineering (ICSE2001*), pp. 123-132, May 2001.

Formatted: English (U.S.)