Updating Requirements from Tests during Maintenance and Evolution*

Eya Ben Charrada
Department of Informatics
University of Zurich, Switzerland
charrada@ifi.uzh.ch

ABSTRACT

Keeping requirements specification up-to-date during the evolution of a software system is an expensive task. Consequently, specifications are usually not updated and rapidly become obsolete and unreliable. The goal of our research is to preserve the alignment between requirements and the implementation by supporting the maintenance of the specification. In this proposal, we explore the idea of using tests to automatically generate hints about the evolution of requirements. We discuss the main research questions that need to be addressed, and propose ideas to approach them.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Documentation

Keywords

Requirements maintenance, Software evolution, Change propagation

1. MOTIVATION AND PROBLEM STATEMENT

1.1 Importance of Requirements Documents During Software Maintenance

Program comprehension is a crucial part of software maintenance. Before applying changes, it is important to correctly understand how the current software works and what is the rationale behind its design and implementation. Program comprehension is known to be difficult and time consuming. The presence of a dependable requirements specification considerably influences the effort needed for this comprehension. In fact, requirements give a high-level view of the functioning of the system, which is easier to understand than code. It also provides the rationale behind the implementation. Understanding the function of a system from the structure of its parts only is very complex because the purpose or intent is omitted in the implementation [12]. Moreover, if the rationale is missing, important design and implementation decisions can be undone during maintenance. Therefore, losing the software knowledge contained in the requirements specification makes the software less *evolvable* and leads it to enter the *servicing* stage where only minor changes can be applied to it [3].

Another interesting feature of requirements specifications is that they are accessible to all stakeholders as they do not require a technical background to be understood. Thus they can be used to negotiate and discuss changes with stakeholders.

1.2 Limitations of Current Practice for Requirements Maintenance

Updating the requirements specification is usually a manual task. Ideally, when a change request arrives, maintainers should go through the requirements document, find all the requirements that are affected by the change and update them. Then they propagate these changes from the specification to all other software artifacts. Applying changes to requirements first and then to code means that maintainers have to do change impact analysis twice: once at the requirements level where they have to hypothesize about the impact of change and then at the code level. This task is both expensive and error prone, therefore maintainers usually choose to apply changes to code directly, and leave the requirements document unchanged [3][17].

Another way to update requirements is to propagate changes from implementation backward to specification based on the traceability links between these artifacts. The fundamental limitation of such a technique is the difference in structure between code and requirements [6]: requirements represent high-level customer needs, while source code reflects many implementation and design decisions. Thus relating requirements to code is usually complex. Moreover, the number of links between requirements and code tends to be very high due to scattering (the implementation of a requirement is distributed over many classes) and tangling (one class contributes to the implementation of many requirements). Therefore, propagating change from code to specification is very difficult. *Consequently, the requirements document is usually not maintained and rapidly becomes obsolete and unreliable.*

*PhD Proposal Paper (Advisor: Prof. Martin Glinz, University of Zurich, Switzerland, glinz@ifi.uzh.ch).

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

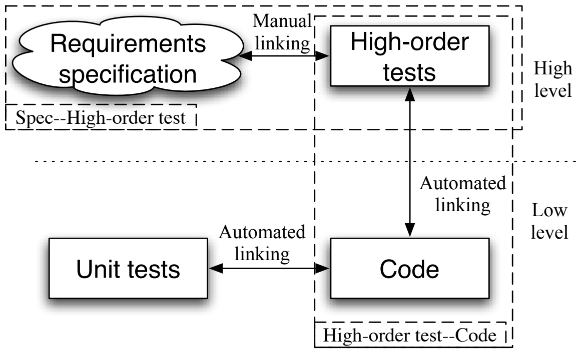

Figure 1: Relating requirements to code using high-order tests

2. RESEARCH GOAL

Our goal is to support engineers in keeping requirements aligned with the code during the evolution of software systems. Our envisaged solution automatically generates hints about requirements to be changed from an analysis of changes in the code and its test suite.

3. RESEARCH IDEA

The main idea behind our approach is to use *high-order tests* [15] (acceptance tests, function tests and system tests) as a link between requirements and implementation (Figure 1). While unit tests are concerned with the design and implementation details of the system, high-order tests only deal with the visible behavior of the system. They usually derive from requirements directly, as they are meant to check that the deployed system satisfies the specified requirements. Inputs and scenarios used for these tests can even be used to complement specifications, as they are a concrete illustration of abstract requirements [10]. Thus, defining traceability links between high-order tests and requirements (Figure 1: *Spec—High-order Test*) is straightforward.

An interesting characteristic of tests is that they are executed against code. Thus it is possible to obtain relations between tests and source code automatically (Figure 1: *High-order test—Code*) by analyzing execution traces (also called footprints). Moreover, if there is an inconsistency between tests and code, it should be easily detected as it results in a test failure. Finally, if some parts of the code are not covered by tests they can be automatically detected based on code coverage.

The ability to easily link requirements and high-order tests from one side, and automatically link tests and code from the other side makes the idea of using tests to relate specifications and implementation and propagate change between them worth exploring.

4. PROPOSED APPROACH

In this section we sketch our approach for generating hints about changes in requirements based on an analysis of changes applied to tests. The generated hints will guide the maintainer during the update of the requirements specification.

Different types of maintenance can be applied to a software system (addition of function, refactoring, bug fix, etc.). We need to differentiate between these types of change to get a suitable hint for each case. In the rest of the paper we use the terminology proposed by Chapin et al. in [5] to classify types of software maintenance. In our work we focus on the four following types: reductive maintenance, corrective maintenance, enhancive maintenance and performance maintenance. These types usually imply changes in the test suite. In the rest of this section we briefly present each of these types and discuss the expected hint and how to generate it. As an example, we use a simple library system (LS).

4.1 Enhancive Maintenance

Among the maintenance types we are considering, enhancive maintenance is the most common one [5]. We differentiate two types of enhancive maintenance: 1) extension and addition of new functionalities, 2) replacement and modification of functionalities.

Extensions and Additions

This type of change is usually followed by the addition of new tests. We propose to use the new tests and their execution traces to generate hints about the added functionality. We illustrate our idea using the following LS example: a new functionality is added for sending reminders to customers that should return a book in the following ten days. An envisioned hint to provide maintainers with is:
A new functionality has been added to the system. Related requirements: "loan period", "return book". Related keywords: send, Reminder.
We can obtain related keywords from tests' names. As developers usually use meaningful method and class names, we expect the name of the added test to look like "testSendReminder". If we analyze the execution traces of the test, we are likely to find a method named "sendReminder". The documentation related to that method might also contain relevant information about the functionality. The relation between the new functionality and the existing requirement can be obtained based on footprint overlaps [8]. If footprints of test A and test B overlap then the requirements associated to these tests are probably related to each other. It is also possible to get additional relations based on information retrieval methods: we look for similarities between the terms used in the tests and the terms in the requirements document.

Replacement and Modification

This type of change is usually followed by a change in the test suite. We propose to use traceability links between requirements and tests to identify the requirements that need to be updated in the specification. If we change the number of books that a customer is allowed to borrow, we expect the following hint: *The system has been modified, the following requirements are likely to be affected by the change: "borrow book", "number of borrowed books"*

4.2 Reductive Maintenance

Reductive maintenance removes or limits existing system functionalities. It is usually followed by the deletion of the tests related to the suppressed functionalities. In such case, we propose to use traceability links to identify the requirements impacted by the change.

4.3 Performance Maintenance

This type of maintenance affects the performance of the software system. It usually results in changes in the performance tests. Here too we propose to identify requirements affected by change based on traceability links.

4.4 Corrective Maintenance

A maintenance task is corrective when it fixes a functionality or makes it more precise (e.g., handle new exceptions). Corrective maintenance does not imply changes in requirements, because it improves the conformity of the system with the specified requirements. However, it might result in the addition of new tests to check that the fix works. Thus it might be confused with the addition of new functionalities (section 4.1). We need to be able to differentiate between these two types of maintenance. This can be done based on test coverage: if the added test has coverage that is very similar to old tests, then it is probably related to a bug fix. However, if the new test covers newly added code that is not covered by other tests then it is more likely to be related to a new functionality.

Refactoring

Although refactoring is frequent in software projects, we do not consider it in our approach. Refactoring changes the internal code without changing the external behavior of the system, thus it does not affect requirements. High-order tests should not be affected by refactorings either, because they are black-box tests and should depend only on the external behavior of the system.

5. RESEARCH QUESTIONS

Various research questions emerge from the proposed idea. In this section we present the main questions and discuss them.

R.Q. 1: Can changes in high-order tests be a reliable indicator about changes in requirements?
Changes in tests can be due to different factors: changes in requirements, refactorings, bug fixes, new quality requirements, etc. Is it possible to differentiate between these different types of change and get relevant information about changes in requirements only? We think that this can be feasible in the case of high-order tests. These tests should not depend on design and implementation details, thus changes in these tests usually result from changes in requirements. We need to confirm this hypothesis by exploring the factors influencing changes in different types of tests.

R.Q. 2: How to manage non-functional requirements?
Although non-functional requirements [9] constitute an important portion of a system's requirements, they are not always testable automatically. Therefore, changes in this type of requirements might be hard to detect based on tests. We need to explore the applicability of our approach to different types of non-functional requirements. Even if we might not be able to cover all of them, we should be able to cover some interesting ones such as performance.

R.Q. 3: What effect has the size and complexity of change on the efficiency of the approach?
When many modifications are applied to a system at the same time, they might interfere with each other and make it difficult to identify the impact of the change on requirements. Thus we need to analyze the effect of the size and complexity of change on the generated hints.

6. RESEARCH METHODOLOGY AND VALIDATION

The first research question will be addressed based on an exploratory case study [7]. We will explore the effect of different types of change (change in requirements, refactoring, bug fixes, etc.) on the test suite. We expect to find relations between changes in requirements and changes in tests. We will also explore the effect of changes in non-functional requirements on tests (R.Q. 2).

Based on the observations made when performing the case study, we will build an approach for analyzing changes in tests and deriving hints about changes in requirements. A first evaluation of the approach will be done by measuring the precision and recall of generated hints. The usefulness of these hints for requirements update will be evaluated using a controlled experiment: students will be asked to update a requirements specification *with* and *without* the help of the hints. Then we compare the time spent for updating the requirements, the correctness of the update as well as the confidence of the maintainers about the updates they made. We will estimate the effect of change size on the efficiency of the approach (R.Q. 3) using a controlled experiment: we evaluate the correctness and completeness of generated hints in the case of applying minor and major changes to the system.

7. CONTRIBUTION

Our contribution consists in a novel approach to keeping requirements up-to-date when a software system evolves. The approach uses changes in the test suite to generate hints about changes in requirements. These hints will guide maintainers during the task of maintaining the requirements specification. We expect the approach to reduce both the time and effort needed for requirements update.

8. STATE OF WORK

We developed rules for estimating the likelihood of a requirement to be affected by change in the case of enhancive maintenance [2]. The rules help maintainers detecting requirements that need to be updated in the specification. However, the approach is not yet decisive enough: in many cases we cannot decide whether a requirement is affected by the change or not. The cause is that a test might check many requirements at the same time.
We are currently working on improving the decisiveness of our approach by taking other factors (e.g., test coverage, type of change) into account. We intend to validate our approach by evaluating the correctness of the estimated likelihoods for a representative case study.

We started exploring the relation between changes in requirements and changes in code based on an open source system for processing bar code images. The system, named Zxing [19] is meant for decoding bar code images on mobile phones. It is implemented in Java and has a test suite developed with JUnit. We have chosen this project because it has a set of black-box tests that seems to be maintained regularly. The project is also very active and has a large number of users. Most changes applied to the system are additions and extensions of functions, refactorings and bug fixes. Initial results show that all additions of new tests (in the case of Zxing) are related to additions and extensions of functions. This result is very interesting, as it can be the starting point for using added tests to get hints about new requirements.

9. RELATED WORK

Propagating changes between software artifacts during maintenance is usually based on traceability links. When an artifact is modified, the developer uses the links to navigate to all related artifacts and update them if necessary. Much research has been done in the field of traceability. However, most of this research is concerned with generating traceability links and updating them. Antoniol et al. [1] and Hayes et al. [11] developed methods to generate traceability links between code and documentations using information retrieval models. Egyed [8] uses trace analysis to semi-automatically find dependencies between requirements and generate links. Mäder and Gotel [13] propose a method to update traceability links for UML models during maintenance. Our research is different as we do not focus on generating links. We suppose that traceability links between requirements and tests are already defined, whether manually or automatically, and we focus on how to use these links to maintain requirements document.

A different way to get a requirements document that is aligned with the implementation is to reverse engineer the specification from code. Yu et al. [18] succeeded to generate goal models from legacy code. The main limitation of reverse engineering methods is that the generated artifacts are either imprecise or incomplete [4]. It is also not possible to derive requirements strictly from code because the intent is missing in the implementation [12]. Thus, updating the specification with our approach should give better results than regenerating a completely new specification from code.

The goal of our approach is to support artifacts co-evolution. This problem has also been addressed by Mens et al. [14] and Reiss [16]. However, the main difference is that their work focuses on the co-evolution of design and implementation, while we are addressing the co-evolution of requirements and implementation.

10. CONCLUSION

In this article we propose a technique to update requirements specifications based on high-order tests. Changes applied to tests during system evolution are analyzed to identify affected requirements and provide developers with hints about how to change these requirements. We expect this technique to reduce the effort (and consequently the costs) needed for maintaining specifications and help preserve the valuable software knowledge contained in the requirements document.

11. REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[2] E. Ben Charrada and M. Glinz. An automated hint generation approach for supporting the evolution of requirements specifications. In *Proc. Joint ERCIM Workshop on Software Evolution and Int'l Workshop on Principles of Software Evolution (IWPSE-EVOL 2010). To appear*, 2010.

[3] K. H. Bennett and V.T. Rajlich. Software maintenance and evolution: a roadmap. In A. Finkelstein and J. Kramer, editors, *The Future of Software Engineering*, pages 73–87, 2000.

[4] G. Canfora and M. Di Penta. New frontiers of reverse engineering. In L. Briand and A. Wolf, editors, *The Future of Software Engineering*, pages 326–341, 2007.

[5] N. Chapin, J.E. Hale, K.M. Khan, J.F. Ramil, and W.G. Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[6] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. In *Proc. Conf. Object-oriented programming, systems, languages, and applications (OOPSLA 1999)*, pages 325–339, 1999.

[7] S. Easterbrook, J. Singer, M.A. Storey, and D. Damian. Selecting empirical methods for software engineering research. *Guide to Advanced Empirical Software Engineering*, pages 285–311, 2007.

[8] A. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Trans. Softw. Eng.*, 29(2):116–132, 2003.

[9] M. Glinz. On non-functional requirements. In *Proc. IEEE Int'l. Conf. Requirements Engineering (RE'07)*, pages 21 –26, oct. 2007.

[10] Dorothy Graham. Requirements and testing: Seven missing-link myths. *IEEE Software*, 19:15–17, 2002.

[11] J.H. Hayes, A. Dekhtyar, and S.K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Trans. Softw. Eng.*, 32(1):4–19, Jan. 2006.

[12] N. G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Trans. Softw. Eng.*, 26(1):15–35, 2000.

[13] P. Mader, O. Gotel, and I. Philippow. Enabling automated traceability maintenance by recognizing development activities applied to models. *Proc. Int'l. Conf. Automated Software Engineering (ASE 2008)*, pages 49–58, 2008.

[14] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views: A case study. *Computer Languages, Systems & Structures*, 32(2-3):140–156, 2006.

[15] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., 1976.

[16] S.P. Reiss. Constraining software evolution. In *Proc. Int'l. Conf. Software Maintenance (ICSM 2002)*, pages 162–171, 2002.

[17] E. Yourdon. *Just enough structured analysis*. http://yourdon.com/strucanalysis/, 2005.

[18] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J.C.S. do Prado Leite. Reverse engineering goal models from legacy code. In *Proc. IEEE Int'l. Conf. Requirements Engineering (RE'05)*, pages 363–372, 2005.

[19] ZXing. http://code.google.com/p/zxing/.