A Test-Driven Approach to Establishing & Managing Agile Product Lines

Yaser Ghanam, Shelly Park, Frank Maurer
University of Calgary
{yghanam, parksh, maurer}@cpsc.ucalgary.ca

Abstract

Test Driven Development (TDD) is an agile method that emphasizes writing tests before writing code as a means of 1) assuring the satisfaction of customer requirements, and 2) reinforcing good design habits. While the first objective is usually accomplished by acceptance tests, the second objective is achieved by unit tests. The tests also serve as a multilevel cohesive reference of the system specifications. We propose the use of this referencing mechanism – test artifacts – to establish and manage agile product lines. In this paper, we delve into some of the issues that need to be tackled before test artifacts are relied on as a driving force for reuse in product lines. These issues include establishing a framework for reuse, tests comparability, test traceability, test refactoring and test versioning. We also discuss the suitability of acceptance tests and unit tests as reusable artifacts, and we present a preliminary study to analyze their utilization.

1. Introduction

1.1. Background

As defined by Clement et al. [1], a product line is a group of software systems that share a common set of features. Because of this commonality, a high level of reuse occurs in software product lines allowing for a shorter delivery time and a considerable improvement in quality. What is also appealing about developing software families when compared to developing single systems is the opportunity for mass customization at a relatively low cost To efficiently handle the commonalities and the variations in the customer specifications, an intensive domain analysis takes place upfront to determine the scope of the product line, the points of variations, and the expected variants. Once the domain has been well understood, a flexible architecture (along with potentially reusable assets) is designed that is capable of accommodating potential members of the family and proactively address their variability. Then an application engineering process takes place to start instantiating members of the software family. This strict "domain-then-application" model, although has been deemed successful [2], is resource intensive and risky. It can be financially and managerially exhausting to small-to-medium scale organizations. Therefore, efforts to make software product lines more agile are increasingly important as the hope is that less upfront effort will be required and risk will be reduced by utilizing agile approaches in the development of product lines. An agile model to establishing and managing product lines may seem to be in conflict with the traditional practiced approaches, but resolving this conflict has a strong potential to cut down the upfront investment in domain analysis and architectural design. Blending agile methods and software product line engineering also has the advantage of introducing a minimalistic design philosophy that reduces or eliminates unnecessary investment in requirements that are too hard to predict in the long run. Finding which agile methods work best to address the different issues of product line engineering is a broad research area that has been gaining special attention in the past few years.

1.2. Motivation

TDD [3] is amongst the most commonly practiced agile methods. Contrary to traditional software development models that delegate testing to a late stage of the project life cycle, agile software development adopts a "test-first" philosophy that uses tests to aid with software design. That is, before implementing any feature, tests are to be written to determine what defines the completion of a feature. Once the tests are written, they are expected to fail because no implementation exists to satisfy them. As the development goes on, test cases start to gradually pass. Development stops as soon as all tests related to a given feature are passing and the customer representatives confirm that the feature is done. Generally, tests occur at two different levels. At a high level, where the customer is concerned, acceptance tests (AT) are pro-

duced as a well-defined, concrete agreement on what features are to be delivered. Acceptance tests are black box tests at the feature level and, hence, need not worry about the details of the implementation. At a lower level is unit testing (UT). Developers write unit tests as a means of ensuring the quality, modularity and neatness of the design, and therefore they are highly dependent on the specifics of the implementation. Unit tests can occur at different granularities such as modules, classes and methods.

In agile contexts, ATs are thought of as a way of keeping a cohesive record of the specifications of the system under development. Since they are usually written in a format accessible to both technical and non-technical audiences, they can always be a reference of what was requested by the customer, what has been done so far and what is to be done next. Furthermore, with the assistance of tools like Fit [4], ATs can be automatically executed providing what is now known as "executable specifications".

This dependency on tests as a means of communicating the system requirements at different levels brings the idea that tests might be a good driving force for reuse in an agile software product line. That is, ATs and UTs can be the reusable artifacts that steer reuse of other artifacts in an agile product line. However, in order for tests to qualify for this responsibility, a raft of issues is to be considered such as tests reusability, comparability, traceability, refactoring, and versioning. *This paper discusses the different aspects of the proposed test-driven approach to shed light on what is to be done before we can establish a test-based agile product line*.

The remaining of this paper is structured as follows. The next section is a literature review of related efforts. Section 3 talks about the different issues that need to be addressed in order to establish test-driven agile product lines. Section 4 is a discussion about identifying features in product lines, furnished with an initial study of existing systems developed using a test-driven approach. Finally, we briefly go over some of our future plans to investigate the matter further.

2. Related Work

Unit testing is a mature field that has been discussed well in literature and successfully practiced in industry. In comparison, executable acceptance testing is a new agile technique that, supported by tools like FitClipse [5], is becoming increasingly popular. Melnik et al. [6] and Ricca et al. [7] studied the effectiveness of Fit acceptance tests in conveying requirement specifications and agreed that these tests do improve requirement understanding. Moreover, Park et al. [8] studied the benefits of executable acceptance tests. They assert that the tie between these tests and the source code allows for better progress tracking, enhances communication amongst stakeholders, and consequently leads to better software quality.

Integrating agile methods and SPLE practices is a new research area that is starting to gain attention in academia and industry. A handful of efforts proposed different ways to blend the two software paradigms. For example, Hanssen et al. [9] suggested the use of SPLE at the strategic level and the use of agile methods at the tactical level. Moreover, Paige et al. [10] proposed the use of Feature Driven Development (FDD) as a way to build product lines under special considerations for the product line architectural and component design.

3. Open Questions

Traditionally, domain analysis is the most crucial step in establishing software product lines. Not only is it a prerequisite to the application engineering step that is completed before applications are being developed, but is also what defines commonality, variability and reuse in the product line. To suggest an agile approach to build product families is to take the responsibility of defining an alternative model that satisfies the requirements of a successful product line practice. Our proposal is to use acceptance or/and unit tests to support this alternative. In this section, we discuss *some* of the major issues that need to be addressed in order for tests to qualify for this task.

3.1. A framework for reuse

For a test to be capable of driving reuse in the product line, it needs to be reusable *itself*. Tests that have previously been produced to drive development in applications of the same domain are to be refactored and reused whenever applicable. However, determining this applicability is not straightforward. We need first to put required features in the new system within the context of previously developed ones (the family) to search for a match from existing assets. This implies that we also need to know how to *compare* these assets. The other implications of wanting to establish a framework for reuse in an agile product line is to have a record of every instance of reuse for future maintenance, extension and quality improvement. This leads us to ask many questions about the *traceability* of

reusable assets. One more issue that needs to be delved into is how to do the *refactoring* when a match is found. This topic is particularly significant considering the risks that arise from incorporating changes that might regressively have influence on a number of products in the family at the same time. Also, performing the refactoring process several times will result in different versions of the test artifact existing at different points of time. So how do we mange and keep track of these *versions*, and ensure their consistency with the source code? The following subsections will talk about these issues in more detail.

3.2. Tests comparability

In the collaborative activity of writing acceptance tests, the customer and the developers produce tests that describe the features in the system under development. Later, when a new system (in the same domain) is being developed, a new set of acceptance tests are written for the same purpose. In order to crunch knowledge about whether the newly requested features correspond to the previously developed ones, there needs to be a mechanism to compare tests to realize how similar they are. This is especially tricky because we need first to clearly define what it means for two tests to be similar, what metrics should be used to determine similarity, and what degree of similarity is satisfactory to make an economical reuse. We may need to rely on mathematical and economical models to answer these questions and make it feasible to automate the comparison process.

3.3. Test traceability

It is clear that reuse without traceability is impractical. For an asset to be reused across different products, there has to be a mechanism to trace this reuse so that future reference is feasible. Say after reusing a specific module in a number of different applications, a bug is reported and needs to be fixed. With the unavailability of comprehensive documents stating where reuse has occurred in the product line, it would be very time consuming to go over all the products in the family and look deep at the module level to figure out which members of the family has used this specific module. The other alternative, and a more practically appealing one, is to employ tests to trace code reuse across different products. That is, a database stores information about all reuse instances that have occurred in the product line. How sophisticated this database needs to be is a function of the intricacy of the domain, and how much the product line has matured in the evolutionary process. Furthermore, tracing tests in a product line provides the advantage of being able to run regression tests across all products in the family when any of the reusable features experiences modification to make sure the modification has no side effects.

3.4. Test refactoring

The vision of the stakeholders becomes clearer as the project evolves over time resulting in change requests being initiated. Assuming a healthy practice of a test-driven approach, development occurs only with accordance to written tests. Therefore, when a change is to occur in the source code, all corresponding tests have to be updated. The process of updating these tests is time-consuming and can be risky, especially for acceptance tests. Because, according to Ordelt et al. [11], acceptance tests "lack the regression safety net that production code has." This issue becomes more incisive when considering a family of systems where change is more likely to occur and affect a wider range of source code. That is, as the product line architecture evolves over time, the separation between the commonality layer and the variability layer of reusable assets ossifies as a result of the continuous refactoring of core assets. This refactoring process needs to be reflected in associated tests to ensure consistency. Ordelt et al. [11] has studied this matter, and extended FitClipse [5] to support acceptance test refactoring using the Eclipse refactoring framework.

3.5. Test versioning

Continuous integration is a vital practice in agile software development. Developers continuously commit the changes they make to a repository and wait for a confirmation that the project has incorporated their changes successfully. If anything goes wrong in the new build, it is always possible to go back to an older error-free state. The rule of thumb is that developers should never check out broken code. Being able to go back and forth between different versions of the system is a necessity not a luxury for a successful continues integration practice. Unit tests are usually imbedded within the same project containers, and therefore their version need not be addressed separately (they already are versioned together with the code). This is not the case with acceptance tests, however. For they are built and executed separately in tools like Fit, associating versions to them is not common. This might

not be an issue when thinking about single systems where a simple one-to-many relationship exists between the system and its acceptance tests. However, the issue becomes a quagmire in product families due to the complex many-to-many relationships that occur between applications and reusable assets.

An interesting approach would be to develop a framework that incorporates the different versions of test artifacts within the versioning system in the repository. A mechanism to support traceability of artifacts in these repositories, as discussed earlier, can also be integrated to make the versioning process of test artifacts more meaningful in the product line context.

4. Identifying Features in Product Lines

Tests hold some important information about the design and quality requirements and allow the developers to extract the common code based on functional features. Therefore, we want to find out if tests can be used to determine the commonalities in software systems of the same domain. In test-driven development, there are two types of tests. Executable acceptance tests are specified by the customer and they communicate the system requirements in testable format. These tests are defined in terms of functional features. Unit tests are written by the developers and they communicate the structural quality requirements. Unit tests are defined in terms of structural features. By writing the tests first, the tests become essential artifacts that communicate functional and structural requirements of the software. Instead of heavy up-front design of the software product line, which can be costly, we want to investigate whether these tests can support the establishment of an agile software product line. Our research question is to find out which type of tests provides richer information about the code.

Let's suppose that these tests carry very important information about the product line and that we can use these tests to extract the code for the common layer easily. As part of the research, we decided to measure the code coverage using two different types of tests in order to determine the traceability of the tests to the code. Our assumption is that Fit tests provide functional features, but may not necessarily trace well to the code as acceptance tests are based on black-box testing concept. Unit tests provide better traceability of test to code, but it may be too fine grained. Additionally, we are interested in finding out the intersecting code coverage by these two types of tests in order to determine the traceability of code from acceptance tests to unit tests code and ultimately to the code. If this relationship does exist, then upgrading one Fit test to the common layer would tell the developers which part of the code or unit tests must also be upgraded to the common product line layer.

Although code traceability or code coverage is irrelevant for acceptance tests, all functional features that are defined in the acceptance tests should be traceable to some parts of the code. We want to find out how much of the code can be traceable via acceptance tests in order to easily identify the code that needs to be upgraded to the common layer of the software product line.

4.1. Study Setup

We have taken two software projects and analyzed the code coverage. The two software products used for the experiment are FitClipse [5] and Flickr Desktop [12]. FitClipse is built for academic research purpose. It is a support tool for Fit tests and runs as an Eclipse plug-in. FitClipse is developed by numerous graduate and undergraduate students over the years. It has about 5500 lines of code. Flickr Desktop allows the user to browse pictures on Flickr website [13] and displays the selected pictures as desktop background in a slide show format. The Flickr Desktop project was a product of a course work by about a dozen students. It has about 1500 lines of code. Both software products used Fit tests for acceptance tests. We decided to measure the code coverage based on statement coverage criteria and measured the intersecting code coverage by both Fit tests and unit tests. Intersecting code coverage refers the lines of code that were executed by both Fit tests and Unit tests. The code coverage was measured manually due to the lack of code coverage tool for Fit tests.

4.2. Data Analysis

Table 1 summarizes the results obtained from the analysis. FitClipse have 20 Fit tests and 119 unit tests. Fit tests had 13% code coverage and unit tests had 14% code coverage. Only 46 lines of code were actually used by both Fit tests and unit tests. Flicker Desktop had 26 Fit tests and 59 unit tests. Fit tests and unit tests both covered 48% of the code. However, only 36% of the code was executed by both Fit tests and unit tests.

Table 1: Code Coverage Result for Fit and Unit Tests

	Fit	Unit
FitClipse		
Number of Tests	20	119
Code Coverage	692/5469 (13%)	747/5469 (14%)
Intersecting Code Coverage	46/5469 (0.8%)	
Flickr Desktop		
Number of Tests	26	59
Code Coverage	725/1509 (48%)	730/1509 (48%)
Intersecting Code Coverage	549/1509 (36%)	

The analysis shows an interesting initial result. Fit tests can cover comparable amount of code compared to unit tests with fewer number of tests. For FitClipse, 20 Fit tests were able to achieve similar code coverage as that of 119 unit tests. Likewise, 26 Fit tests were able to achieve the same code coverage as 59 unit tests in Flickr Desktop. Acceptance tests define a feature when unit tests define much more refined structural testing specifications. The reason for more code coverage by Fit tests is due to the setup that is needed in order to execute the functional specification. Unit tests often did not require extensive data or code setup to execute.

The two software products used for the analysis have relatively low test coverage. Due to the lack of time and money, achieving full test coverage is often very difficult. Given these constraint, it is interesting how Fit tests and unit tests do not necessarily test for the same things or even similar part of the code when the coverage is limited. For example, FitClipse has only 0.8% of the code that is executed by both Fit tests and unit tests. In Flickr Desktop, only 36% of the code is tested by both types of tests. The result suggests that the developers and customers do not necessarily think alike when it comes to testing and they view importance of the testing features differently given limited time to write the complete set of tests.

Given that most of industrial software development projects also do not achieve full test coverage, the early experiment results points that we need to look at both acceptance tests and unit tests for the purpose of product line engineering. Unit tests may be too fine grained, as seen in the results for FitClipse that 119 unit tests had similar code coverage as 20 Fit tests. Unit tests seem to test too fine grained structure of the code for extracting common layer code. Fit tests seem to provide higher level of system abstraction information, because each test mapped to larger set of code. However, it is often not possible to trace the fit tests to the corresponding unit tests.

The early result provides an interesting look into the relationship between acceptance tests and unit tests when the software project does not have enough time to achieve full test coverage. However, is there a point when having more Fit tests does not necessarily increase the code coverage? Although the main purpose of acceptance tests is not about achieving full code coverage, as the concept of code coverage is from structural testing practice, a functional feature from acceptance tests should be traceable to some parts of the code. In addition, it would be also interesting to look at the reasons for the intersecting lines of code coverage. We need to find out the contributing causes that influence why unit tests and fit tests do not necessarily test for the same parts of the code. Do developers and customer really think differently? Or do they have different testing priorities? Although code coverage is not the aim of acceptance tests, the result shows an interesting phenomenon that developers and customer do not necessarily test the same feature. The results show that they ended up testing the different parts of the code. Even looking at the result from redundancy point of view, these tests serve different purposes and thus redundancy does not mean waste of effort: acceptance test is for supporting customer collaboration and unit tests are for improving software design. The test coverage is simply a side effect. However, it would be interesting to know why the developers did not write unit tests for the portions that were covered by the Fit tests as the developers clearly did encounter these code while writing the fixtures code that attached the Fit test specifications to the code. The preliminary result proposes the interesting question of whether the developers and customers view the system differently and thus give different emphasis on the features that require testing. We need to take a look at more industrial projects to see if the phenomenon persists.

6. Conclusions

Developing software families is a successful paradigm that promises considerable cuts in cost and development efforts accompanied with enhancements in quality. Introducing agile software development practices to software product line engineering aims at magnifying these advantages even further. In this paper, we proposed the use of test artifacts to establish

and manage agile product lines. We highlighted some open questions that need to be researched before tests can be utilized as a driving artifact for reuse. A preliminary analysis of two systems was also discussed to shed light on the roles that acceptance tests and unit tests could play in the context of a test-driven product line. The initial results suggest that because developers and customers view the system differently, they give different emphasis on what is to be tested. Therefore, more is to be done to understand the multimodality of test artifacts and their appropriateness to steer reuse. Currently, we are preparing a controlled experiment to better understand the factors that contribute to how stakeholders write tests, and how developers translate these tests into design decisions. We also aim to conduct a series of experiments to investigate issues related to test-driven reusability and refactoring.

7. References

[1] Clements, P., and Northrop, L., *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.

[2] Pohl, K., Böckle, G., and Linden, F., *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, Germany, 2005.

[3] Beck, K., *Test Driven Development- by Example*. Boston:
Addison-Wesley, 2003.

[4] FIT: Framework for Integrated Test, http://fit.c2.com, last accessed on May 29th, 2008.

[5] FitClipse, http://ase.cpsc.ucalgary.ca/index.php/FitClipse, last accessed on May 29th, 2008.

[6] Melnik, G., Read, K., and Maurer, F., Suitability of fit user acceptance tests for specifying functional requirements: Developer perspective. In *Extreme programming and agile methods - XP/Agile Universe 2004*, pp. 60–72.

[7] Ricca, F., Torchiano, M., Ceccato, M., and Tonella, P., Talking tests: an empirical assessment of the role of fit acceptance tests in clarifying requirements. *The 9th International Workshop on Principles of Software Evolution 2007, Croatia.*

[8] Park, S., and Maurer, F., The benefits and challenges of executable acceptance testing. *The APSO Workshop - collocated with the 30th International Conference on Software Engineering 2008,Germany.*

[9] Hanssen, G., and Fægri, T., "Process Fusion: An Industrial Case Study on Agile Software Product Line Engineering", accepted for special Issue of *Journal of Systems and Software (JSS)*, 2008.

[10] Paige, R., Xiaochen, W., Stephenson, Z., and Phillip J., "Towards an Agile Process for Building Software Product Lines", *LNCS: XP 2006*, pp. 198 – 199.

[11] Ordelt, H., and Maurer, F., Acceptance test refactoring. *Proceedings of the 9th International Conference on Agile Processes and eXtreme Programming*, Springer, Ireland, 2008.

[12] Flickr Desktop, http://sourceforge.net/projects/flickr-desktop, last accessed on May 20th, 2008.

[13] Flickr, www.flickr.com, last accessed on May 30th, 2008.