UNIVERSITY OF CALGARY

FitClipse: a Testing Tool for Supporting Executable Acceptance Test Driven Development

by

Chengyao Deng

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

September, 2007

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "FitClipse: a Testing Tool for Supporting Executable Acceptance Test Driven Development" submitted by Chengyao Deng in partial fulfilment of the requirements of the degree of Master of Science.

Supervisor, Dr. Frank Oliver Maurer, Department of Computer Science

Dr. Victoria Mitchell, Haskayne School of Business

Dr. Behrouz Homayoun Far, Department of Electrical and Computer Engineering

Date

Abstract

Limited data has shown how Executable Acceptance Test Driven Development has been conducted in the industry environment. Therefore, the question of what kind of tool support for acceptance testing should be provided is still unknown. We conducted a survey on Executable Acceptance Test Driven Development. The results show that there is often a substantial delay between defining an acceptance test and its first successful pass. Therefore, it becomes important for teams to easily be able to distinguish between tasks that were never tackled before and tasks that were already completed but whose tests are failing again. In this thesis I designed and implemented a tool, called FitClipse, which extends Fit by maintaining a history of acceptance test results to generate reports that show when an acceptance test is suddenly failing again. The pilot evaluation of FitClipse demonstrates positive prospects of the tool concept as well as its shortcomings.

Publications

Some content, ideas and figures from this thesis have appeared previously in the following peer-reviewed publications:

Chengyao Deng, Patrick Wilson and Frank Maurer: "**FitClipse: A Fit-based Eclipse Plug-in For Executable Acceptance Test Driven Development**", *Proceedings of the 8th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2007),* Como, Italy, Jun. 2007 (Springer)

C. Deng, P. Wilson, F. Maurer: FitClipse: "**An Eclipse Plug-in For Execuatble Acceptance Test Driven Development"**, *Procedings on Eclipse Technology eXchange (ETX 2006),* Interactive Poster, Oct. 2006

Acknowledgements

I would like to thank my supervisor, Dr. Frank Maurer: you are always giving me support and guidance whenever needed. It is you who give me the chance to start up my new life in North America and help me grow up to a mature man.

I would like to thank my family: thank you for bring me to the wonderful world and trying your best to take good care of me, support me and encourage me. I love you forever, no matter where I am.

I would like to thank Patrick: thank you for all your help for building the fundament of the tool and all your precious suggestions on my papers. I am always lighted up by your smart ideas.

I would like to thank Shelly and David: thank you very much for the final grammar revision of this thesis. You save me and my thesis from my limited English.

I would like to thank Grigori, Carman, Beth, Ruth, Robert and all the people in the lab: thank you for always standing beside me and give me a hand when I am weak.

I would like to thank all my friends: thank you for your care for me when I scratched my face fell in badly ill, broke my ankle and whenever I am alone.

Dedication

To my family, who are always supporting me, and **my beloved wife, Nuo Lu**, who accompanies me through the darkness, sickness and poor.

Table of Contents

List of Tables

List of Figures and Illustrations

Chapter One: Introduction

This thesis presents a novel tool – FitClipse – for supporting Executable Acceptance Test Driven Development and the pilot study for the usefulness and usability of the tool. I begin this chapter with an introduction of software testing and Agile Methods. Then, I present the advantage of using acceptance testing in Agile Methods. I mention the survey I conducted which reveals the limitation of existing acceptance testing tools. Further, I present the motivation and the specific goals of my research. Finally, I conclude the chapter with an overview of this thesis.

1.1 Testing in Software Engineering

Software testing is the process of using input combinations and defined states to reveal unexpected behavior of software systems [Binder 2000]. Its purpose is to ensure the correctness, completeness, security and quality of computer software. Software testing can be divided into several categories:

- Unit testing: To validate that each unit of a system is working well independently.
- Integration testing: To combine functional units into groups and validate the transactions between them.
- System testing: To detect the defects in the functionalities on the system level.
- Acceptance testing: The process of a customer testing the system functionality to determine whether the system meets the requirements.

1.2 Agile Software Development and Agile Methods

Agile Methods are a group of methodologies that specify conceptual framework for software development. They include a family of different methodologies that share four core values: individuals and iterations over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; and responding to change over following a plan [Agile Manifesto 2007].

Agile software development methods are focused on trying to produce a working solution and to respond to changes from the customer requirements [Hunt 2006]. Extreme Programming, which is also called XP, is the most widely (38%) used Agile Methodology [Pancur 2003]. To address the constraints of the previous software development processes, XP is a lightweight methodology for teams of any size developing software in the face of vague or rapidly changing requirements [Beck 1999][Beck 2004].

XP emphasizes five values with which it drives the development process: Communication, Simplicity, Feedback, Courage, and Respect. By practicing the key values, XP desires to achieve the goal of creating the most value from the project.

1.3 Testing in XP

XP uses Test Driven Development (TDD). TDD means writing tests before any code is written down. At first the tests are made to fail. After the implementation is completed, the tests assert each value by calling the code and eventually to get the tests to pass which means the implementation is working properly. Two test driven approaches are used in XP, one is Unit Testing and the other is Acceptance Testing or Story Testing.

Unit Test Driven Development (UTDD) [1] is originally called Test Driven Development and first introduced by Kent Beck in [Beck 2003]. In UTDD, unit tests are created and maintained in a test suite, which includes a group of tests that assert the functional requirements of the software. Once the tests are created, all the tests in the test suite are run initially without the actual application code to make sure the tests are failing. This step is to negatively test the test itself, which makes sure the new feature fails as expected without the new code being implemented. Then the developers make changes to the system code to make the tests pass which means the new function is working properly. At last all the tests in the test suite should run again to make sure the newly added code does not break other existing functionalities of the system. TDD helps the development teams build loosely coupled, highly cohesive systems with low defect rates and low maintenance cost [Beck 2003].

Automated acceptance tests are also used in TDD which is called Executable Acceptance Test Driven Development (EATDD). It is also known as Story Test-Driven Development or Customer Test-Driven Development. Acceptance tests for a feature should be written first by the customer with the help of the development team, before the application code is implemented [Cohn 2004]. The tests represent the system requirements and specifications. Then the development team will work on the implementation with the guidance of the acceptance tests. The implementation can be seen as completed when all the corresponding acceptance tests are passing. Through an

[1] I would like to use UTDD as short for Test Driven Development using unit tests in this thesis to separate it from EATDD which uses acceptance tests.

initial empirical evaluation of TDD, Geras suggested that tool support is really needed in both academia and industry for the use of customer/acceptance testing [Geras 2004].

Recent evidence suggests that the major reason for project failures is improper management of user requirements [Davis 2004]. It has been reported in Standish Group [Chaos Report] that the top three reasons of failed projects are: the lack of user input, incomplete requirements and specifications, and changing requirements and specifications. All these reasons had to do with requirement management practices.

EATDD is a possible solution to these problems. From the customer's perspective, EATDD provides the customer with an "executable and readable contract that the programmers have to obey" if they want to declare that the system meets the given requirements [Tracy 2004]. Observing acceptance tests also gives the customers more confidence that the correct functionality is being developed for the system. From the perspective of the programmers, EATDD helps to make sure that they are delivering what the customers want. In addition, the results help the team to understand if they are on the right track with the development progress. Further, as EATDD propagates automated acceptance tests, these tests can play the role of regression tests in later development to make sure that the newly added functionalities does not break the existing features.

1.4 Thesis Motivation

A major difference between UTDD and EATDD is the timeframe between the definition of a test and its first successful pass. Usually, in UTDD the expectation is that all unit tests pass all the time and that it only takes a few minutes between defining a new test and making it pass [Beck 2003]. As a result, any failed test is seen as a problem that needs to be resolved immediately. Unit tests cover very fine grained details which make this expectation reasonable in a TDD context.

Acceptance tests, on the other hand, cover larger pieces of system functionality. Therefore, we expected that it may take the developers several hours or days, sometimes even more than one iteration, to make them pass.

If our expectation is true, due to the substantial delay between the definition and the first successful pass of an acceptance test, a development team can NOT expect that all acceptance tests pass all the time. A failing acceptance test can actually mean two things:

- Unimplemented Feature: The development team has not yet finished working on the story with the failing acceptance test (including the developer has not even started working on it).
- Regression Failure: The test has passed in the past and is suddenly failing – i.e. a change to the system has triggered unwanted side effects and the team has lost some of the existing functionalities.

The first case is simply a part of the normal test-driven development process. It is expected that a test that has never passed before should fail if no change has been made to the system code. However, the later case should be raising flags and should be highlighted in the progress reports to the team. Otherwise the users have to rely on their recollection of the past test results or reorganizations of the test suites to determine the meaning of the failing test. For anything but very small projects, this recollection will not be reliable. In my research I have found no tool that is designed for identifying these two kinds of failures in EATDD.

Therefore, my research is motivated, first of all, to investigate how EATDD is conducted in the industry and analyze what kind of tool support is needed. Secondly, based on the findings of support for EATDD in the industry context, a tool should be developed to provide a novel support for EATDD. Further, since EATDD is a novel technology for XP and this tool provides a new support to EATDD, it would be interesting to find out how people think the tool can help in EATDD.

1.5 Thesis Problems

In this thesis I am going to address the following problems:

1. **It is unknown how Executable Acceptance Test Driven Development is conducted in industry, especially the time frame of Acceptance Testing.**
2. **Based on the finds of Problem 1, novel tool support should be built for doing EATDD in Agile environment.**
3. **Once such a tool support is built, an evaluation is needed to determine the usefulness and usability of the tool in Agile environment.**

1.6 Research Goals

In this thesis, I will address the problems mentioned above with the following goals:

1. I will investigate how EATDD is being conducted in industry environment on the aspect of acceptance testing time frame.

2. I will perform analysis on the investigation data from the industry, find out whether it is helpful to persist the test result information and identify the necessary support for EATDD.
3. I will design and implement a tool to facilitate EATDD.
4. I will conduct a pilot study to evaluate the effectiveness and usability of the tool.

Figure 1.1 describes the context and scope of this research. Software Engineering is the outmost context of the research. This research focuses on a sub area of Agile Methods from software engineering overlapping with Software Testing. Testing tools support for Agile Methods, which has always been in need of more research and it is a core issue that can help maintain software quality. My research is focused on investigating the acceptance testing issues in an Agile environment, the related tool support and its evaluation.

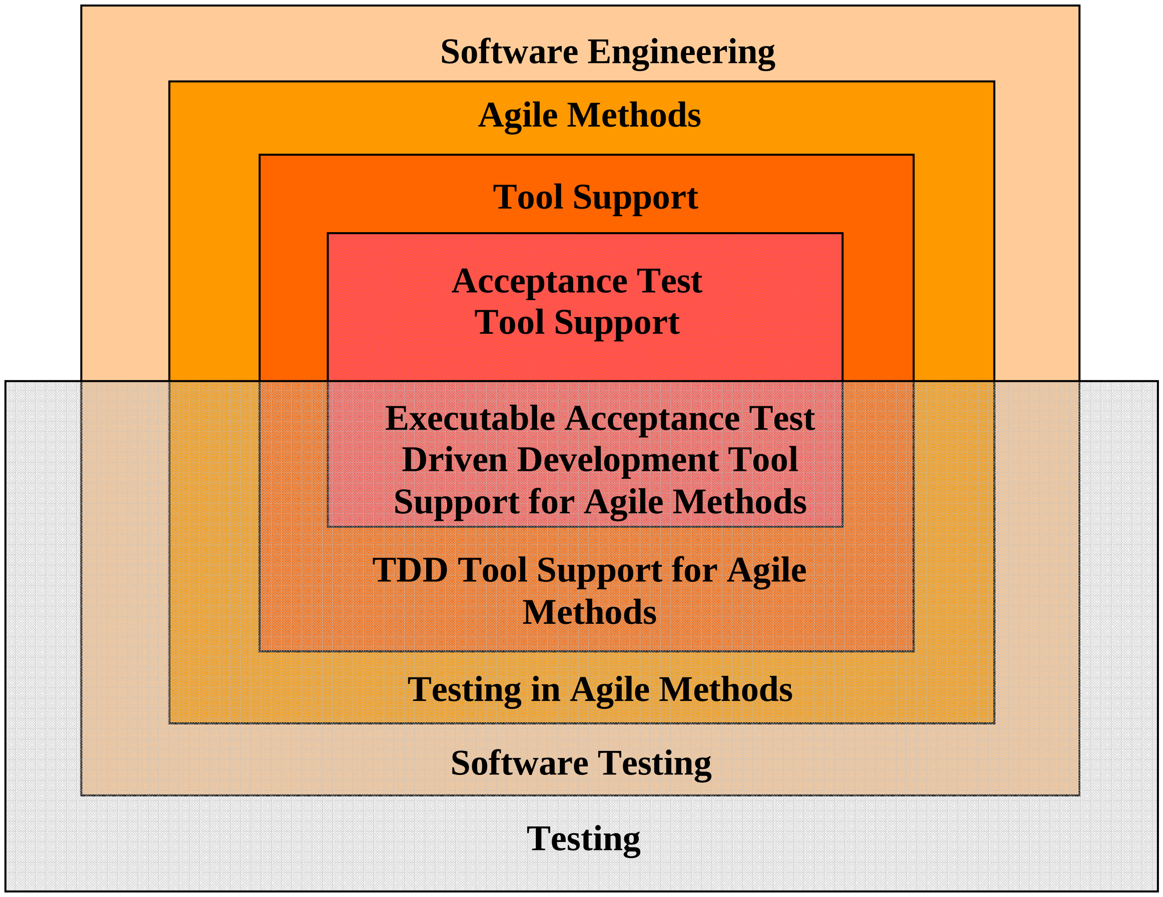

Software Engineering
Agile Methods
Tool Support
Acceptance Test Tool Support
Executable Acceptance Test Driven Development Tool Support for Agile Methods
TDD Tool Support for Agile Methods
Testing in Agile Methods
Software Testing
Testing

Figure 1.1: The research context and scope of the thesis

1.7 Thesis Structure

The thesis is divided into seven chapters:

In Chapter 2, I introduce Agile Methods and testing techniques that are used in an Agile environment. Then I discuss related tools used for acceptance testing. This chapter also summarizes the empirical studies on using Fit and FitNesse for acceptance testing.

In Chapter 3, I present a web survey, whose purpose is to find the current state of EATDD being conducted in industry and whether there is a need to persist test result information. Further, basing on the findings of the survey, I investigate tool support that may be helpful to the Agile development team for doing EATDD.

In Chapter 4, I demonstrate the user scenario of using FitClipse for supporting EATDD in Agile environment.

In Chapter 5, first the acceptance testing tool requirements is outlined. Then the detailed design and implementation of the testing tool, called FitClipse, is described. Further, the comparison of FitClipse with other acceptance testing tools is discussed.

In Chapter 6, I describe the results of an initial evaluation of FitClipse. The aim of this evaluation is to investigate the usefulness and usability of FitClipse.

In Chapter 7, I conclude the research and suggest possible future work in the research area.

Chapter Two: Background

In this chapter, I give an overview of the background information of Agile software development processes, methods and software testing. Then I introduce the special testing techniques which are used in Agile environment, including Unit Test Driven Development and Executable Acceptance Test Driven Development. In addition, I summarize the existing tools supporting acceptance testing in various ways. Because my proposed tool uses the Fit/FitNesse framework to write and run acceptance tests, the related empirical studies of Fit and FitNesse are also summarized at the end of this chapter.

2.1 Agile Software Development

Agile software development is a conceptual framework for undertaking software engineering projects. The Agile Manifesto [Agile Manifesto 2007] proposed four values for Agile software development: individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation and responding to change over following a plan.

First, Agile software development emphasizes the value of people and their communication and interaction over the whole team. Even though software development processes, methodologies and tools contribute toward the success of a project, the people’s behavior is always the most critical element in software development. It is required that an effective agile team will maintain close team relationships and team working environment arrangements to improve the information sharing and communication flow among team members.

Second, Agile teams take advantage of continuous publishing of fully tested working software over dependence on heavy documentation. The final goal of software development is to produce valuable software and never just the documentation, because documents are used for supporting the process of software development or the software itself. In Agile environment, new releases with added functionalities are published in frequent intervals. Although these releases are small, they are fully tested and have gone through an entire software life cycle. By doing this, the developers have to make the code simple, straightforward and easy to understand, thus reducing the documentation to an appropriate level. On the other hand, the customers are also satisfied in the continuous functional releases, because they can always see the process of the development through newly added working functionalities.

Third, it is ideal if the Agile team can keep an onsite customer who can collaborate with the developers rather than working on the details of contract negotiations. From the Agile team point of view, the onsite customer will clarify the software requirements, which in result will make the team to be on the right track of the software development. On the other hand, from the customer point of view, keeping in close touch with the development team will eventually make sure that the team is delivering the right business value. A close customer collaboration will definitely reduce the risks of producing the wrong software.

Fourth, Agile software development allows changes in the software development and can quickly adapt to the changes. This is not saying the traditional methodologies do not allow changes. In the traditional methodologies, changes are made late in the project and making late changes are much more expensive than making changes in the early stages of the software development. The traditional software development process fails because of the frequent changes and late changes in the product requirement specification. By conducting small releases and other practices, Agile software development can reduce the cost of frequent changes and late changes happening during the development process.

2.2 Agile Methods

Rather than a specific methodology for Software Engineering, Agile Methods are a family of project planning and software development processes. Agile Methodologies focus on trying to produce a working solution and at the same time responding to changes of customer requirements [Hunt 2006]. These methodologies include: Extreme Programming (also known as XP) [Beck 1999], Scrum [Schwaber, 2001], Crystal Clear and other Crystal methodologies [Cockburn 2004], Feature Driven Development (also known as FDD) [Palmer 2002], Dynamic Systems Development Method [Stapleton 1997], Adaptive Software Development [Highsmith 2000] and etc. In the thesis, only Extreme Programming, which is directly related to this thesis, will be discussed.

2.3 Extreme Programming

Among all kinds of Agile Methods, Extreme Programming, or as it is commonly known as XP, is the most widely used technique by Agile teams [Cao 2004] [Maurer 2002]. According to Kent Beck, one of the creators of XP, Extreme programming is “about writing great code that is really good for business” [Beck 2004]. Combining the definition in [Beck 1999] and [Beck 2004], Extreme Programming is a lightweight methodology for teams of any size developing software in the face of vague or rapidly changing requirements.

XP is lightweight. In XP, people do not need to do more than they need to create a value for their customers. It has, although not directly, the implications on the project portfolio management, financial justification of projects, operations marketing, sales and other aspects that can be seen as project constraints. XP can work for teams of any size. Even though the initial purpose of creating XP is for small or medium teams, it has also been scaled up. Through augmenting and altering the practices, the values and principles in XP are applicable for any size of teams. XP adapts to vague or rapidly changing requirements. It can adapt to the rapid changes in the modern business world. However it can also be used where the requirements stay stable.

XP includes five values for guiding the development process. They are:

- Communication: It is the most important issue in software development. It is critical for building the spirit of a team and maintaining effective cooperation.
- Simplicity: Make the system simple enough that it is simple to expand the scope of the system.
- Feedback: It is an important part of communication. In XP, people strive to generate feedbacks early and often.
- Courage: People need the courage to speak the truth. It can strengthen the communication and the trust.

- Respect: In a XP environment, each person in the team needs to be respected.

These values lead to the following key practices:

- Sit together in an informative environment: XP needs people in one team to communicate effectively. All team members should be located in the way they can enable easy communication. In addition, integrated information about the whole project should be provided, so that everybody in the team can easily understand how the project is going. For instance, some teams achieve information centralization by putting story cards on a white board which can be easily accessed by all the team members.
- Pair Programming: The idea is two people working together on the same computer. One person can review the code at the same time the other person is coding. Working in pairs enables continuously code reviews and feedbacks between the two developers.
- Stories: The stories are units of customer understandable functionality with estimations of the development effort. The stories are created in XP planning with customers involved in the development team. Estimations are made early in the stories' life cycle.
- Short Iterations: Iterations should be relatively short to allow quick and frequent feedback and changes. Small working software releases are produced at the end of the iteration.

- Continuous Integration: A team programming is a process of divide, conquer and integrate. In XP, integrating and testing changes are performed within a couple of hours.
- Test First Programming: Write a test and make it fail before making changes to the existing code. The automated tests made are not only for quality assurance, but they also help design and refactor the software system.
- Make code work for today: Unlike in traditional software engineering, people in XP do not create detailed design anticipating future needs. Instead, they just make the software meet today's requirement. If the requirement changes later, they are confident that they can adapt the design. In other words, the XP team invests into the design in proportion for the established requirements of the current system.

2.3.1 XP Project Lifecycle

Figure 2.1 shows a typical lifecycle of a project in Extreme Programming [Don Wells 2000].

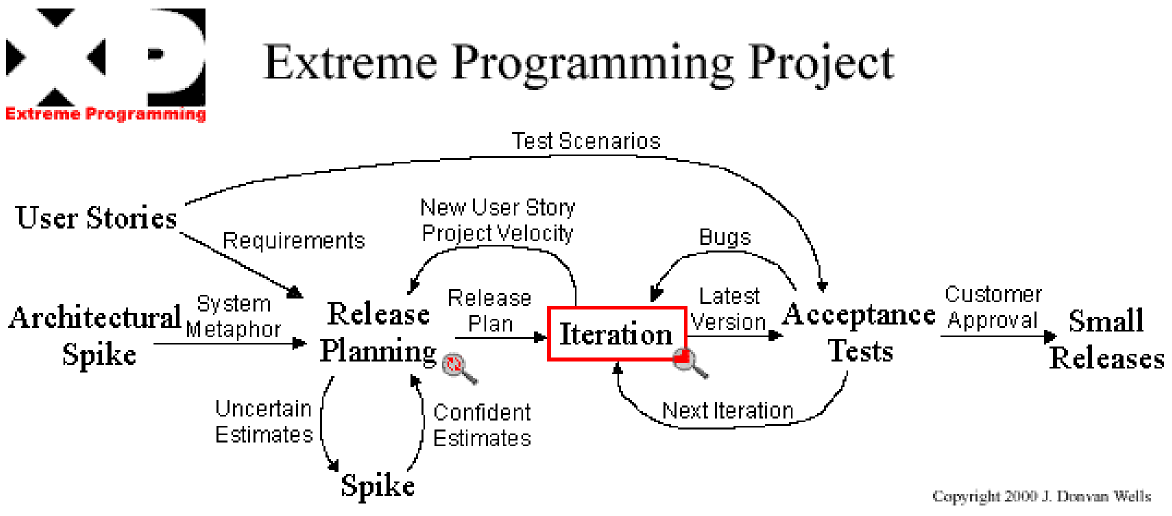


Figure 2.1: Life cycle of a project in Extreme Programming [Don Wells 2006]

Architectural Spike: An architectural spike is used for reducing the risk of the project. During the spike, uncertainty for the system, technology and application domain are investigated in the form of research and evaluation. The most important spike is on the overall system architecture. Spike is also performed later in the development process.

Release Planning: Release planning happens in a release planning meeting. This planning meeting will determine all the stories (requirements) which will be implemented in this release and put them into a repository called “backlog”. The customers assign priorities to stories. The technical people roughly estimate the development efforts and resource cost on each story. The estimates can be refined later. Basing on the estimate, the development team will figure out the stories and their delivery time in this release, resulting in a release plan.

Iterations: Iterations provide the development process with the adaptability to changes. At the beginning of the iteration, changes can be made to the tasks that is planned to be implemented in the iteration. An iteration meeting is held before the iteration starts in order to determine which stories are going to be included in this iteration. The users or end customers will pick up the stories with the highest priority in the backlog. The customers can change the priorities of stories as well.

Acceptance Tests: At the end of each iteration, users or customers run all the acceptance tests, including tests from the previous iterations and those from the last iteration, to determine whether the newly added functionalities are acceptable and prevent new changes from causing side effects to previous working functionalities. If the customers are satisfied with all the acceptance tests, which means they obtained all the expected functionalities, the development team can check all the code and tests into a repository and move on to another iteration or release. On the other hand, if the customers do not accept the system, the development team will go back to fix all the bugs or make changes according to the customers' requirements. This will end up with another acceptance testing stage.

Release: The most distinct features of XP releases are small and often. Each release contains implemented system functionalities that provide business values. These functions should be provided to the customers when they are available. This also enables quick feedbacks from the customer side to the development side.

2.4 Software Testing

It is commonly known that more than two-thirds of software projects today fail [Chaos Report]. They are either terminated, exceed time and budget restriction, have reduced functionalities or provide insufficient reliability. The lack of testing is one of the most important reasons of such failures.

Software testing is the process of using input combinations and selected states to reveal unexpected behavior of the software systems [Binder 2000]. Its purpose is to ensure the correctness, completeness, security and quality of computer software.

2.4.1 Software Testing Process

The testing process can vary between different organizations. However there is a typical cycle of testing:

- *Test Planning*: determining how to generate and organize the test cases by making Test Strategy and Test Plans.
- *Test Development: developing Test Cases, Test Scenarios and Test Scripts for the software under test.*
- *Test Execution: The testers execute the planned test cases against the product under a certain environment and report any unexpected system behavior to the development team.*
- *Test Reporting: Based on the outcome of test execution, the testers generate reports on their test effort and whether the software under test can be released.*
- *Retesting*: retesting the defects that are supposed to be fixed by the developers.

2.4.2 Different Levels of Testing

According to the different testing levels, software testing can be divided into several categories.

2.4.2.1 Unit Testing

The goal of unit testing is to validate that each unit of a whole system is working well independently. A unit is the smallest testable part in the system. Unit test tests all the interfaces of one unit to the others. Therefore, unit tests cover fine grained system details. Unit testing is typically done by the developers.

Automated unit tests provide support for continuous changes in the source code. Passing unit tests mean successful implementation of small system functionalities. Later changes in the system functions should not break the previous working units. Running unit tests continuously can expose the changes that break other parts of the system.

In addition, unit tests can work as documentation for the developers. Developers create unit tests based on the interface of a unit. Therefore, the unit tests include all the information of what the interface of each unit means. Other developers can read the tests to understand how to use the interface of a unit.

2.4.2.2 Integration Testing

The purpose of integration testing is to combine the functional units into groups and validate the transactions between the groups of units. Integration testing always comes after unit testing and before the system testing. There are two ways of integration testing: the Big Bang approach and the Bottom Up approach.

In Big Bang approach, all or most of the testing units are put together to form the whole system. Then tests cases are created to run test on the units working in a whole system. The Big Bang approach is supposed to be an effective way to save time in testing. However if the test cases and results are not organized and recorded properly, the testing process will be very complicated and cannot achieve the goal of integration testing.

In Bottom Up approach, system functional units are organized into different levels to be tested. Test cases are generated from a lower to a higher level. There are also different methods to organize the functional units into the different levels.

2.4.2.3 System Testing

System testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements [IEEE 1990]. System testing tries to detect the defects on the system level, thus it does not require any knowledge of inner design or logic of the code.

2.4.2.4 Acceptance Testing

Acceptance testing is the process of the customers testing the functionality of a system in order to determine whether the system meets the requirements. Acceptance tests are concrete examples of the system features. They help the customers to know whether the system is doing the right things and decide whether they want to accept the system [Miller 2001] [Acceptance Test 2006].

Ideally, the acceptance tests are defined by the customers. It is ideal that the customers can write the acceptance tests themselves, which is not always the fact. However the acceptance test should at least be understood by the customers. Because in the end it is the customers who will run the acceptance tests and determine whether the software can be deployed. In practice it is often the developers who will translate the customer stories into testing code [Erickson 2003]. With some tool support, such as Fit or FitNesse, customers can write their own acceptance tests. Sometimes they may also need the testers to organize or refine their tests to maintain a better test suite.

Acceptance tests are designed to test the system functionalities on a system level. The major purpose of Acceptance testing is not finding bugs (even though bugs can be found by acceptance tests) of the system but demonstrate the system's functionalities. In Extreme Programming, recent literature suggests that the executable acceptance tests should be created for all stories (which are the system functionalities) and that a story should not be considered to be completed until all the acceptance tests are passing successfully [Miller 2001].

There are many synonyms for acceptance testing. Table 2.1 is from [Maurer 2006] which includes the synonyms.

Table 2.1: Acceptance Testing Synonyms [Maurer 2006]

Term	**Introduced/Used by**
Functional tests	[Beck 1999]
Customer tests	[Jeffries 2001]
Customer-inspired tests	Beck
Story tests and story-test-driven development	[Joshua 2005]
Specification by example	[Fowler 2006].
Coaching tests	[Marick 2002]
Examples, Business facing example, example-driven development	Marick
Conditions of satisfaction	[Cohn 2005]
Scenario tests	[Kaner 2003]
Keyword-driven test	[Kaner 2002]
Soap opera tests	[Buwalda 2004]
Formal qualification tests	[USDD]
System tests	[IEEE 1996] [Erickson 2003]

Using Acceptance Testing offers the following advantages.

First, acceptance tests can improve the communication between the customers and the development group, thus improving the specification of requirements. According to [Chaos Report], user Involvement is ranking the first in the ten most important factors of successful projects. 85% of the defects in developed software originate in the software requirement [Ralph 2001]. One of the most important reasons is insufficient customer involvement. Acceptance tests build a bridge between the customer and the development team by providing support of specifying detailed functional requirements. With the help of acceptance tests, the development team can reduce the risk of misunderstanding the customers’ expectation and make sure that the system is working as the customer expected.

Second, a suite of acceptance tests (a set of tests which are organized together and run together) helps the development team understand the development progress [Erickson 2003]. Normally, it is hard to decide when a task for one requirement can be tagged as finished. The customer may reject the implemented requirements which the developers think they have already completed. By using acceptance tests, when all the acceptance tests that belong to one requirement pass and the customer accepts the functionality, the developer can tag the requirement as finished. In addition, as the development goes on, more and more acceptance tests are made pass with the number of functionalities increases. From the changing number of passing acceptance tests, the development team can understand how their project is progressing. By analyzing the details of the test suite, the team can also make changes to their current development strategy. Acceptance tests help both the customers and the development team to make sure that the project is progressing in the right direction [Crispin 2001 B].

Third, in the later development the previously passing acceptance tests can play the role of regression tests on a regular basis [Holmes 2006] [Rogers 2004]. Acceptance tests represent the system requirements. Once they are passing, it means the development team has already created the required functions. However, the development team will keep changing the code to implement new functions and perform refactoring to the existing code, which may cause side effects to other parts of the system. This side effect may break already-working functions, resulting in regression failures. By maintaining a suite of acceptance tests and keeping track of passing and failing acceptance tests, the developers can detect the regression failures on the system level easily. [Kitiyakara 2002] mentioned their experience of using acceptance tests as regression tests in their project.

Fourth, writing acceptance tests before implementing the system can improve the accuracy of the development effort estimation. Acceptance tests are defined by the customers as detailed system requirements. The tests expose all the functions that are expected to be released. There are no hidden functions or misunderstanding of the functionalities. Therefore, based on the acceptance tests it is easier for the developers to accurately estimate the required development time and effort.

At last, acceptance tests can be used as documentation in Agile environment. The acceptance tests document the correct behavior of the system [Andersson 2003]. By reading acceptance tests, people can understand what the system is supposed to do and by running the automated acceptance tests, people can see what the system is really doing.

Common acceptance testing workflow has three components [Mugridge 2005 A]:

- *Setup:* put the system into a specific condition.
- *Change or make transaction:* make something happen to the system.
- *Check:* compare the outcome from the system to the expected result.

2.4.3 Comparison of Testing

Testing is conducted differently in Traditional Software Development and Agile Methods such as Extreme Programming. In this section, the comparison of testing methodology in both software engineering processes is discussed. Extreme Programming is taken as an example of Agile Methods.

2.4.3.1 Time of Testing

In the traditional software engineering, testing is performed at the end of the process, after the code has been implemented and right before the product is delivered. Test at this time does not ensure quality effectively, because the testers will make mistakes or omit tests due to the time pressure before releasing the new system.

In Extreme Programming, testing is conducted before the implementation of any code, which is called Test Driven Development (TDD). By conducting TDD, a well designed system with fewer defects and low maintenance cost is implemented. Advantages of TDD will be discussed in section 2.5.

Early testing means a better chance of removing defects early. Removing defect later is much more expensive and threatening to the project's success. This effect is especially obvious in the context of requirement errors. It has been estimated by Boehm and Papaccio that it costs from 50 to 200 times more to remove requirement errors in the maintenance time than to correct them in the early project phase [Boehm 1988].

2.4.3.2 Frequency of Testing

Traditional software engineering does testing only in one phase at the end of implementation, while Extreme Programming does testing frequently. Tests are run tens or hundreds times on a daily basis. Figure 2.2 and Figure 2.3 compare the differences of testing in both software engineering processes on the frequency basis.

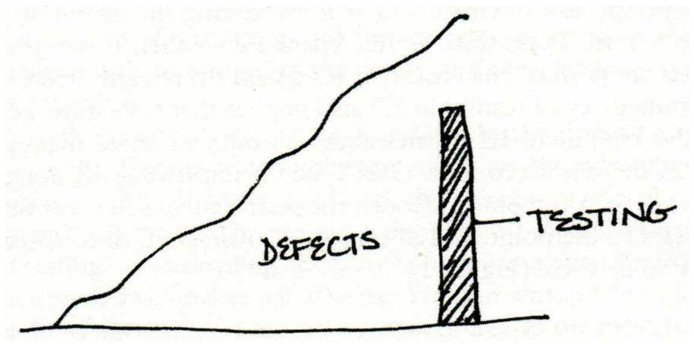


Figure 2.2: Traditional software engineering: late, expensive testing leaves many defects [Beck 2004]

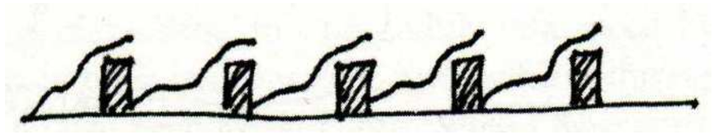


Figure 2.3: Agile software engineering: frequent testing reduces costs and defects [Beck 2004]

Defects grow faster and faster with time going on and based on more and more defects. The cost of fixing bugs depends on the number of defects and the effort to find and fix the defects. It is harder to locate the origin of a defect and fix it with larger number of defects in the system. As shown in the above two figures, the traditional process of the testing is more expensive and leaves more defects than in the Agile process. Frequent testing reduces the number of defects, thus limiting the cost for fixing them.

2.5 Test Driven Techniques

Test Driven Development (TDD) means writing tests before implementing the application code. In this section, two kinds of test driven development techniques are discussed: one using the unit tests and the other using the acceptance tests.

2.5.1 Unit Test Driven Development

Unit Test Driven Development (UTDD) is originally called Test Driven Development and was introduced by [Beck 2003].

Generally speaking, UTDD means writing unit tests before you write the code. No application code is being written before it has the associated tests [Jeffries 2003]. A suite of unit tests are maintained which will guide the development all through the implementation process. It may take the developers several minutes to create unit tests and make them pass. Thus, at any time in the development process, all of the tests should be passing successfully.

UTDD helps the development teams to build loosely coupled, highly cohesive systems with low defect rates and low maintenance cost [Beck 2003]. There are two reasons for this effect. First, UTDD helps to reduce the defects. The sooner the defects are found, the less the cost will be. UTDD maintains a suite of detailed unit tests which is run hundreds of times a day to make sure that a bug is captured immediately before it gets propagated. By reading the failing tests, the defect is located easily and correctly.

There is overwhelming anecdotal evidence which indicates that UTDD is helpful in reducing defects. No opposite effect has been found yet. Second, UTDD shortens the timeframe of feedback on design decision. Unit tests are actually testing the interface of each method. This interface can easily be changed to the real implementation of application programming interface (API). Rather than implementing the design and waiting weeks or months for someone else to feel the pain of improper designed API, TDD enables the feedbacks in second or minutes from self testing.

The general UTDD cycle is: [Beck 2003]

a. *Quickly add a test to the suite*: based on the imagination of how the application would work, invent the interface.
b. *Run all the tests*: run all the tests to see the newly added test is failing. At this time red bar[2] should be seen.
c. *Make the test pass*: change the system code to make the newly added test pass, also run the whole suite of the tests to make sure all tests pass in order to prevent breaking other parts of the system. If some other tests fail, find the reason and make them pass again until the green bar[3] is shown for all the tests.
d. Make the system right: now the system is working, but in an improper way. Refactoring needs to be done by removing duplicates that have been introduced to the system.

---

[2] Red bar means the test is failing.
[3] Green bar means the test is passing.

After working on one test, people continue adding another one to make the progress to the system.

2.5.2 Executable Acceptance Test Driven Development

Executable Acceptance Test Driven Development (EATDD) is another Test Driven Technique. It uses acceptance tests instead of using unit tests.

The concept of Acceptance Test Driven Development is described in [Beck 2003] (called Application Test-Driven Development). Before implementation begins, it is the customers' responsibility (with the help of the development team) to write acceptance tests for driving the development process.

Acceptance test should be written first, before the application code is implemented [Cohn 2004]. Acceptance tests represent the system requirements. It works as the guideline to tell the developers the targeted system functionalities they strive for. The proper process is to: first, the customers write the acceptance tests with the help of the development team; then the developers implement the functional code to make the acceptance tests pass; and at last the customer runs all the acceptance tests to make sure all the requirements have been met.

Acceptance tests should be lightweight. Lightweight means the tests should be easy to modify. The customer requirements may change frequently and Agile Methods adapt to these frequent requirement changes. Therefore the acceptance tests, which represent the customer requirements, should also be easy to be modified in order to be in sync with the customer requirements.

Acceptance test must be executable [Crispin 2001 A]. Manual testing does not ensure quality if done under a schedule pressure before releasing new functionalities. Manual testing is tedious. When testers are encountered by lots of detailed tests and under a time constraint, they are prone to make mistakes, cut corners, omit tests and miss problems [Martin 2005]. In addition manual testing is too time-consuming. It is the bottleneck before the product delivery [Talby 2005]. This fact prevents manual testing to be applied in Agile environment which emphasizes continues testing with fast feed back and small continuous releases.

In XP, the cycle of acceptance testing is shown in Figure 2.4:

a. *Customers define acceptance tests*: the acceptance tests are written by the customers with the help of the development team. The acceptance tests are based on the stories which are the customer requirement specifications in XP.

b. *Developers create fixtures*: the developers create fixtures for automating the acceptance tests.

c. *Make the acceptance tests fail*: the developers run the whole test suite and see all newly added acceptance tests fail.

d. *Make the acceptance tests pass*: Use UTDD to implement the system functionalities. Make the acceptance tests pass one by one after the corresponding system functions are working.

e. *Refactor*: delete the duplicates and make the necessary structure changes in the system code.

f. *Run all tests*: the customers run all acceptance tests to see whether the expected functionalities are working. The customers will accept the system if they are satisfied with the acceptance tests. If they do not accept, the development team will go over the above steps again.

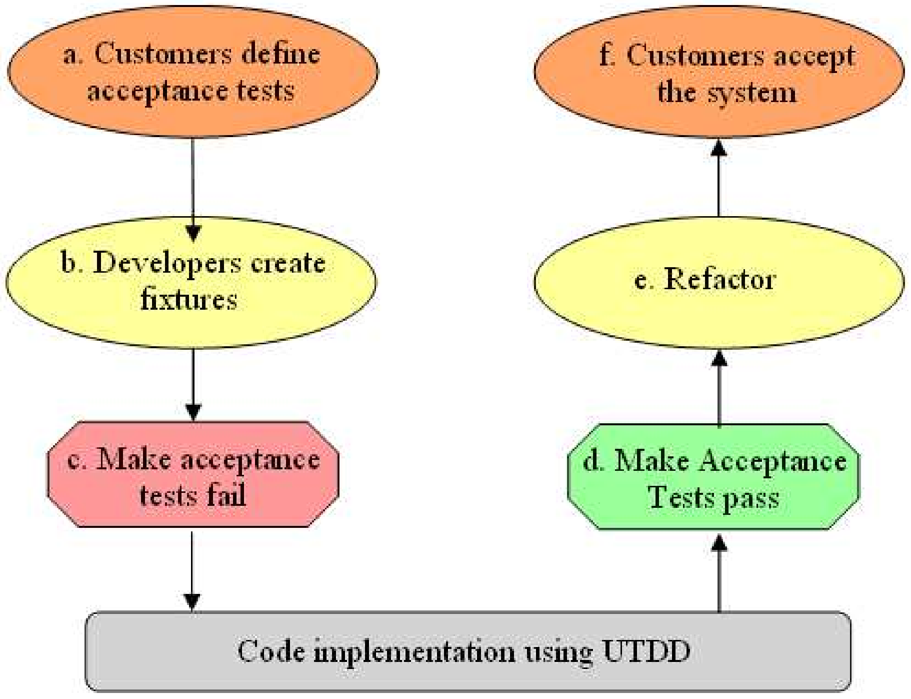

a. Customers define acceptance tests
b. Developers create fixtures
c. Make acceptance tests fail
Code implementation using UTDD
d. Make Acceptance Tests pass
e. Refactor
f. Customers accept the system

Figure 2.4: The cycle of acceptance testing

After the customers accept the system release, the whole development team and the customers enter a new development phase. The customers will create new stories and a new EATDD process start.

2.6 Tools and Frameworks for Acceptance Testing

Tools and Frameworks for Acceptance Testing are introduced in this section. First I discuss open source tools and frameworks.[4] Then I introduce several Eclipse plug-ins that use Fit/FitNesse framework for writing and running acceptance tests.

2.6.1 Open Source Frameworks & Tools

2.6.1.1 Fit

Fit is a Framework for Integrated Testing and is probably the most popular acceptance testing tool today. “It is well suited to testing from a business perspective, using tables to represent tests and automatically reporting the results of those tests.” [Mugridge 2005 B] Fit is designed for ordinary people to be able to understand or even to write acceptance tests. Fit tests are in the form of tables with assertions in the table cells. Fixtures, made by the programmers to execute the business logic tables, map the table contents to call into the software system. Test results are displayed using three different colors for different states of the test: green for passing tests; yellow for tests that cannot be executed and red for the failing tests. The fixture layer is hidden from the customers.

Fit test tables can be created with common business tools such as spreadsheets and word processors. The tables can also be integrated into other types of documents, including Microsoft Word, Microsoft Excel, HTML and etc. Various languages can be used for writing fixtures, including Java, C#, Ruby, C++, Python, Objective C, Perl and small talk [Cunningham 2007].

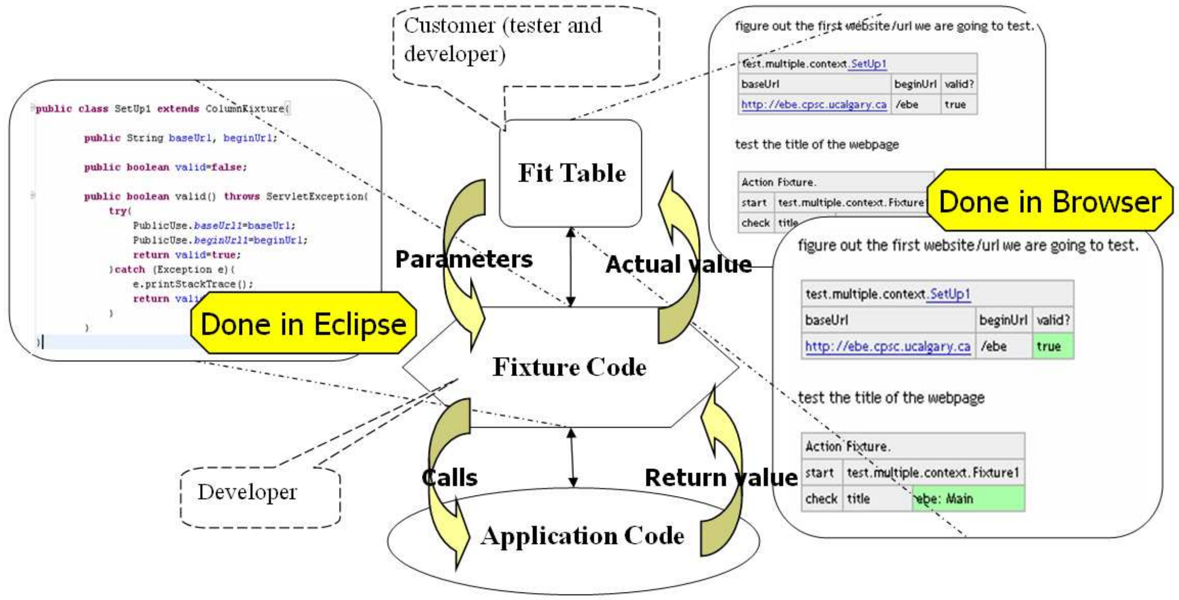
[4] To my knowledge, no commercial tools are providing substantial novel support for Acceptance Testing.

2.6.1.2 FitNesse

FitNesse is a Wiki front-end testing tool which supports team collaboration for creating and editing acceptance tests [FitNesse 2007]. FitNesse uses the Fit framework to enable running acceptance tests via a web browser.

FitNesse enables a development team to collaboratively create tests and run the tests on a Wiki website. Users are able to compose and organize the tests using simple Wiki syntax without any knowledge of programming and HTML. In addition, FitNesse is a cross platform standalone Wiki server which does not need any other server to be running or other configuration, thus it is very easy to set up.

Figure 2.5 shows the framework of FitNesse to be used as an acceptance testing tool.



```
public class SetUp1 extends ColumnFixture{

        public String baseUrl, beginUrl;

        public boolean valid=false;

        public boolean valid() throws ServletException{
            try{
                PublicUse.baseUrl1=baseUrl;
                PublicUse.beginUrl1=beginUrl;
                return valid=true;
            }catch (Exception e){
                e.printStackTrace();
                return vali
            }
        }
}
```

figure out the first website/url we are going to test.

test.multiple.context.SetUp1		
baseUrl	beginUrl	valid?
http://ebe.cpsc.ucalgary.ca	/ebe	true

test the title of the webpage

Action Fixture.	
start	test.multiple.context.Fixture1
check	title

figure out the first website/url we are going to test.

test.multiple.context.SetUp1		
baseUrl	beginUrl	valid?
http://ebe.cpsc.ucalgary.ca	/ebe	true

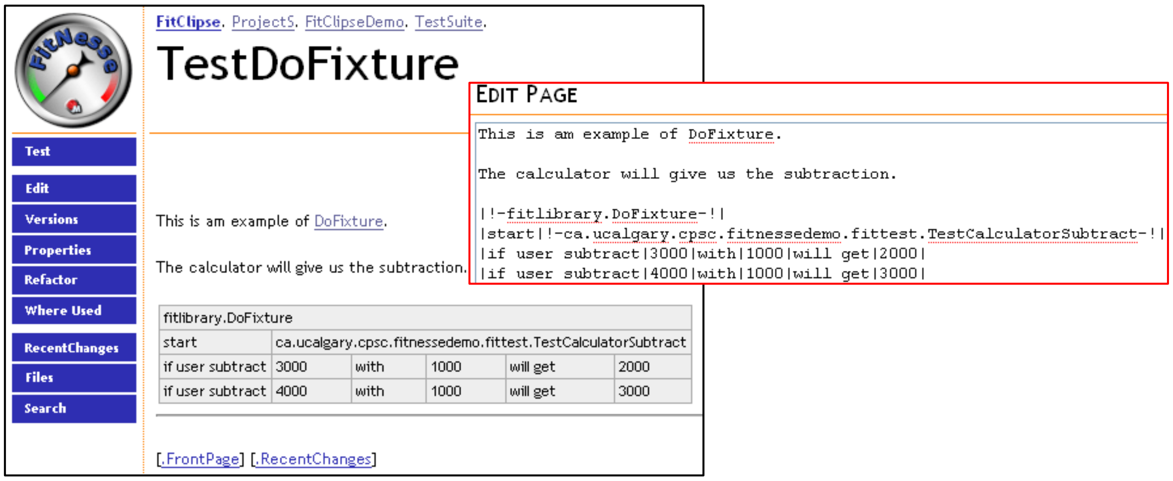
test the title of the webpage

Action Fixture.		
start	test.multiple.context.Fixture1	
check	title	ebe: Main

Figure 2.5: Processes of using Fit/FitNesse framework for EATDD

In this figure, the middle section shows the workflow of the Fit/FitNesse framework between the different layers; the left side shows the Java code and the right side shows the Fit table style acceptance tests. The steps of acceptance testing using Fit/FitNesse include:

- *Write acceptance tests*: the customers write acceptance tests using a browser with the help of developers and the testers. The tests are written in a form of executable tables with Wiki syntax. Figure 2.6 shows a sample Acceptance test within FitNesse environment. The tests are then saved directly on the FitNesse server. The upper right part of Figure 2.5 shows how the Fit tests look like. In the run time, the tests are executed by the FitNesse test runner inside the FitNesse server.



FitClipse. ProjectS. FitClipseDemo. TestSuite.

TestDoFixture

This is am example of DoFixture.

The calculator will give us the subtraction.

fitlibrary.DoFixture					
start	ca.ucalgary.cpsc.fitnessedemo.fittest.TestCalculatorSubtract				
if user subtract	3000	with	1000	will get	2000
if user subtract	4000	with	1000	will get	3000

[.FrontPage] [.RecentChanges]

EDIT PAGE

```
This is am example of DoFixture.

The calculator will give us the subtraction.

|!-fitlibrary.DoFixture-!|
|start|!-ca.ucalgary.cpsc.fitnessedemo.fittest.TestCalculatorSubtract-!|
|if user subtract|3000|with|1000|will get|2000|
|if user subtract|4000|with|1000|will get|3000|
```

Figure 2.6: FitNesse acceptance test sample with Wiki syntax

- *Write fixture code*: After the customers define the acceptance tests, the developers work on implementing the application code. They will develop the fixture code for the acceptance test to make the tests pass. The fixture code dispatches the call from the acceptance test runner into a real application code. It also captures returned values from the application code, modifies the values and passes them back to the acceptance test runner.

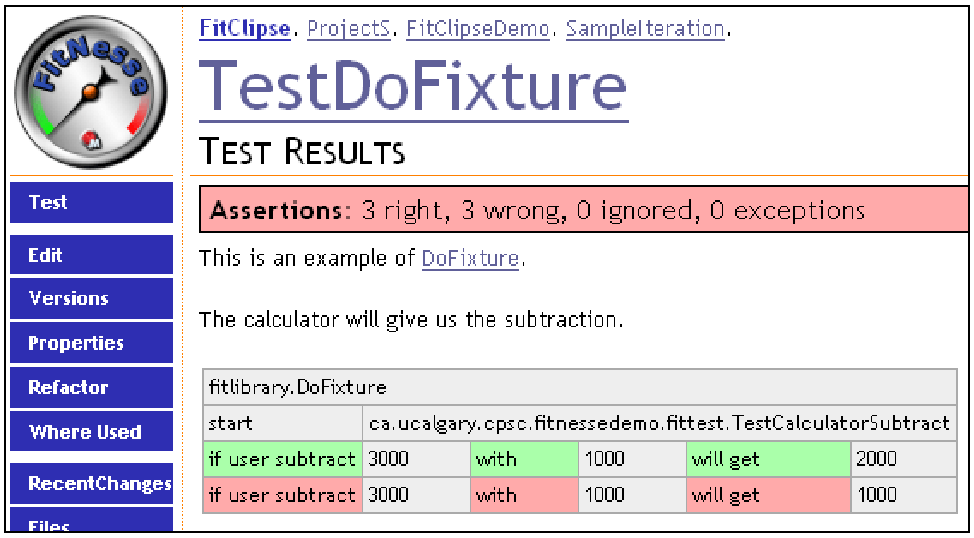
- *Run acceptance tests*: Acceptance tests are executed through the FitNesse test runner. The test runner parses the test tables to find the key words and link them together to construct the class name of the fixture code. With the constructed class name, the test runner calls the underlining fixture code developed by the development team. It also passes the values as the parameters of fixture class call. After the fixture class processes the application with the given parameters, the test runner captures the returned values from the fixture code and makes assertions which show as different colors in test table cells.
- *View test results*: After running the acceptance tests on the FitNesse server, the results are shown in the form of the original test table with the appropriate value cells colors that represent the acceptance testing results. The meanings of the different colors are described in Table 2.2.

Table 2.2: Test Result Schema of FitNesse

Colors	**Meaning**
🟩	The test is passing.
🟨	There are exceptions in the test.
🟥	This test is failing.

Figure 2.7 shows a sample FitNesse test result of a DoFixture test. The test result is shown in detail which provides enough information for the customer. However, the

FitNesse test result can only be viewed after running the test, which makes it inconvenient when there are lots of tests that need longer time to run. FitNesse does not persist the test results.



FitClipse. ProjectS. FitClipseDemo. SampleIteration.

TestDoFixture

TEST RESULTS

Assertions: 3 right, 3 wrong, 0 ignored, 0 exceptions

This is an example of DoFixture.

The calculator will give us the subtraction.

fitlibrary.DoFixture					
start	ca.ucalgary.cpsc.fitnessedemo.fittest.TestCalculatorSubtract				
if user subtract	3000	with	1000	will get	2000
if user subtract	3000	with	1000	will get	1000

Figure 2.7: Sample FitNesse test result for DoFixture

2.6.1.3 FitLibrary

FitLibrary provides general-purpose fixtures and runners for acceptance tests with Fit and FitNesse [FitLibrary 2007]. It includes a collection of extensions for the fixtures in Fit for business processes (workflow), calculations, constraints and lists. Other than the test styles that Fit provides, it also supports testing grids and images, which makes it convenient to test expected layouts.

2.6.1.4 GreenPepper

GreenPepper Open is a framework for creating executable specification documents [GreenPepper 2007]. It is similar to Fit. It ensures that the documentation is easy to read by minimizing the formatting rules and code references. Instead of fixing the tabular structure of acceptance tests (like Fit), GreenPepper provides multiple structures of test cases, including tables of rules, sequences of actions and collections of values.

GreenPepper Server is the commercial version of GreenPepper. It supports multiple client applications by centralizing all coherent data. The GreenPepper server enables running specifications both locally and remotely. It also maintains a history of executions[5] and it supports several versions of front end application as extensions.

One of the extensions is an Eclipse plug-in which is the most relevant to my work. The GreenPepper server Eclipse plug-in provides support for Story Test Driven Development or Acceptance Test Driven Development. Acceptance tests are created and saved on GreenPepper server which works as a test repository. The Eclipse plug-in is working as a front end application for running the acceptance tests. The test results are shown as HTML pages and the test results are reported in the console.

2.6.1.5 Exactor

Exactor is another framework for writing acceptance tests. It uses plain ASCII text scripts written by the customers as acceptance tests [Exactor 2007]. Exactor interprets the scripts to trigger the Java class created by the programmers for making calls to the real application code. The commands are defined one for each line starting at the left hand side with parameters supplied next, separated by white spaces.

[5] I investigated this feature. But I did not see the tool was providing features that are similar to FitClipse's novel contribution.

2.6.1.6 TextTest

TextTest organizes acceptance tests using plain text files. Instead of making assertions against application code, TextTest works by comparing the current version of a plain text logged by programs with a previous standard version of that text [TextTest 2007]. A user interface is also implemented for generating TextTest scripts using record and replay approach[6] [Andersson 2004].

2.6.1.7 EasyAccept

EasyAccept is a script interpreter and runner that uses text files to create acceptance tests using the user-created commands that is close to natural language [Sauve 2006]. Comparing to other acceptance testing tools, it has two main advantages. First, it supports the creation of acceptance tests as sequential commands as well as in tabular format. Second, EasyAccept makes it easier for the developers to implement the test fixture, for which it uses a single Façade which may even already exist for the software under construction.

2.6.1.8 Selenium

Web applications are often tested manually because UI testing is brittle and complex. In previous years, automated testing of web applications was often conducted with a simulated browser, such as HttpUnit, JWebUnit, or CanooWebTest. However, it is still a problem to test heavy client side JavaScript. Selenium is a web application testing tool

[6] Record and replay means the support of recording a set of transactions into scripts automatically. This support enables the transactions to be re-executed later.

which runs acceptance tests in a wide range of browsers, such as Internet Explorer, Firefox and Safari. (A detailed list of supported browsers can be found on [Selenium 2006].) It embeds a test automation engine into the browser with JavaScript and Iframes. Tests are written in the form of tables which have different commands for talking to the browser and asserting expected return values.

There are also two additional tools for selenium: Selenium IDE and Selenium Remote Control. Selenium IDE is an integrated development environment that works as a Firefox extension. It supports recording events as Selenium tests and playback, editing test scripts and debugging tests. Selenium Remote Control allows writing automated web application UI tests in any programming language against any HTTP website [Selenium Remote Control 2006].

2.6.1.9 WATIR & WATIJ

The Web Application Testing in Ruby (WATIR) and its sibling, the Web Application Testing in Java (WATIJ), also provide automated acceptance testing against web applications through real web browsers [WATIR 2007] [WATIJ 2007]. The test scripts are written in Ruby or Java and communicate with the browser in the same way as people do, such as filling in some text and clicking a button. Even though their syntax are simple for making acceptance tests, WATIR/WATIJ test scripts are still similar to programs, thus requiring prior IT or even programming experience to understand the test cases.

2.6.2 Eclipse Plug-ins Based on Fit

There are also Eclipse plug-ins that use Fit or FitNesse for acceptance testing including FitRunner [FitRunner 2007], conFIT [conFIT 2007] and AutAT [Schwarz 2005].

2.6.2.1 FitRunner

FitRunner is an Eclipse plug-in for Fit. It enables running Fit tests inside Eclipse environment. FitRunner contributes to Eclipse a new Runner configuration for running Fit tests. It can run both single Fit test and a folder containing multiple Fit tests.

2.6.2.2 conFIT

conFIT is also an Eclipse plug-in which supports running FitNesse and using a Fit testing framework inside Eclipse easily. It maintains a local FitNesse Server which can be controlled from within Eclipse. Eclipse browsers, which are configured to explore a remote FitNesse server, can be used to create and run acceptance tests.

conFIT is integrated with Eclipse whereby the user can run a wizard to load a FitNesse project with a sample Java source code and wiki pages. It contains tool bar buttons to start and stop local servers. The toolbar buttons for opening a remote server browser and class variables for assessing Fit and FitNesse libraries are also available.

2.6.2.3 AutAT

AutAT enables non-technical users to write and execute automated acceptance tests for web applications using a user-friendly graphical editor [Schwarz 2005]. In AutAT, people draw pages and their relationships in the form of “boxes” and “arrows”.

Executable acceptance tests are automatically created based on these drawings. The tests can be executed by an underling test engine. This runner is interchangeable and currently implemented by Fit plus jWebUnit.

2.7 Empirical studies of Fit and FitNesse

In this section, I summarize empirical studies of Fit and FitNesse to demonstrate that Fit and FitNesse are easy to use and helpful for acceptance testing.

2.7.1 Customer Specifying Requirements

A study was conducted to find out whether customers can specify requirements with FitNesse in [Melnik 2006]. The finding is: that an average customer will have difficulties in learning the Fit framework; however, once the learning curve has been surpassed, the customers find Fit and FitNesse easy to use and they can specify functional business requirements from the positive perspective [7]in the form of executable acceptance tests clearly.

2.7.2 Developer Understanding and Implementing Requirements

The suitability of Fit for communicating functional requirements to the developers is examined in [Melnik 2004]. The result includes: first, Fit tests which describe customer requirements can be easily understood and implemented by developers with a little background of the Fit framework; second, the learning curve for reading and

[7] In this study, the customers specified dominantly positive test cases. The negative test cases only accounted for 6% of all the tests.

implementing Fit test is not prohibitively steep. The former finding is also supported by [Read 2005 A] and [Read 2005 B] that students of information technology are able to read and understand acceptance tests written using Fit framework independent of outside information sources.

2.7.3 Fit Acceptance Test in Industry Environment

Using Fit as the single source of behavioral requirements for the stories in the industry is evaluated in [Gandhi 2005]. The major finding is that using Fit documents to define story details is very helpful in Agile project and it is a successful approach for reducing ambiguity and avoiding duplications. Fit documents created as a single source of system behavior specifications are effective testing technique and can be achieved through effective collaborations. Therefore, [Gandhi 2005] recommends this approach to develop software in an Agile environment.

According to Geras's qualitative analysis of acceptance testing tools [Geras 2005], Fit is one of the tools that can optimize the effectiveness of EATDD in software development process.

Mugridge and Tempero retrofitted their test framework with Fit for acceptance testing in socket-based servers with multiple clients [Mugridge 2003], They found that the Fit framework was straightforward to use. With the help of Fit, the customers find it much easier to write and understand the sequence of communications.

2.7.4 Fit Acceptance Test in Academic Environment

Fit framework is also used in academic environment for specifying course requirements. [Steinberg 2003] and [Melnik 2005] describe their successful experience with the Fit framework in the introductory to programming course and how the students did acceptance testing for their courses. The practices they conducted by using the Fit framework was straightforward to implement regardless of the course context. These findings indirectly prove that the Fit framework can be used for specifying requirements and people can understand the specifications of Fit acceptance tests.

2.8 Summary

The background information on acceptance testing is provided in this chapter. Acceptance testing is a testing technique which has been used in test driven pattern for Agile software development. The tools and frameworks for acceptance testing, including Eclipse plug-ins using Fit/FitNesse, are introduced and analyzed. Empirical studies on Fit/FitNesse are included in order to show the Fit/FitNesse acceptance testing framework as a useful tool for EATDD. In the next chapter, a survey is presented that investigated the state of using EATDD in the industry environment. The findings of the survey directly contribute to the main motivation for providing a novel support for EATDD in Agile software development.

Chapter Three: Survey & Thesis Motivation

In my work, I seek to provide useful support for EATDD which meets the needs of industry. Therefore I conducted a survey to determine the state of people using EATDD in industry. In this chapter, first I describe the objective and design of the survey. Then I summarize the survey participants and the findings. And finally, basing on the survey findings, I introduce the motivation of FitClipse as a testing tool that provides enhanced support for EATDD in industry.

3.1 Objective of the Survey

The overall objective of the survey is to find out how Executable Acceptance Test Driven Development is conducted in industry. Based on the findings of the survey, my aim is to develop a tool to support EATDD which works in an industry environment.

A special motivation of the survey is to find out the time frame between creating an acceptance test and making it pass successfully for the first time. The acceptance testing time frame issue is first proposed in [Beck 2003] that it will take the developers longer to pass an acceptance test than to pass a unit test. However there is a lack of industry evidence that supports this idea. In the survey, I tried to gather the time frame information of acceptance testing to verify that passing one acceptance test will take substantial longer time than passing a unit test.

3.2 Survey Design

Ethics approval was obtained from the University of Calgary (Appendex A.1) before conducting the study.

The survey was designed to ask specific questions about the participants' experience on EATDD in real industry projects. A questionnaire was used to this end. As in Appendix B.2 the survey questionnaire included questions on the subjects' information and knowledge of testing techniques, their experience of running acceptance tests and open ended questions about their expectation of tool support for acceptance testing.

Questionnaires were sent out both in paper and emails format. The paper format questionnaires were given to the Calgary Agile Method User Group [CAMUG 2007]. Emails containing links to the online questionnaire were sent to mailing lists of Agile communities all over the world. Finally, the questionnaire reached 16 user groups. (Please refer to Appendix B.1 for detailed lists of the user groups) The online questionnaire was designed as a website that could be reached publicly. Responses were collected from February – April of 2007.

3.3 Response Rate

The questionnaire was provided to 16 Agile user groups all over the world. Responses were received from 40 subjects. Eleven out of the 40 subjects are excluded because of incomplete answer or not using EATDD in their process. Therefore, 29 subjects are eventually analyzed. However, the number of participants reported in following studies may vary. This is because some participants did not answer all the questions in the questionnaire. They might skip some of the question which ended up with different numbers in the summarization. Responses were subjective in nature and analyzed by myself.

3.4 Participant Demographics

All the participants are working in industry (identified from their position) and following EATDD which ensures that all findings are based on the industry data. Their positions range widely in industry, which include: Customer, Developer, Tester, Business Analyst, Software Designer, System Architect, Interaction Designer, End User Representative, Consultant, Project Manager and Agile Coach.

The source data is based on the experience of the subjects. Therefore it is critical to understand their experiences and knowledge of acceptance testing. In order to make the data accurate, subjects whose experience of acceptance testing is less than half a year are excluded. Figure 3.1 shows their acceptance testing experience:

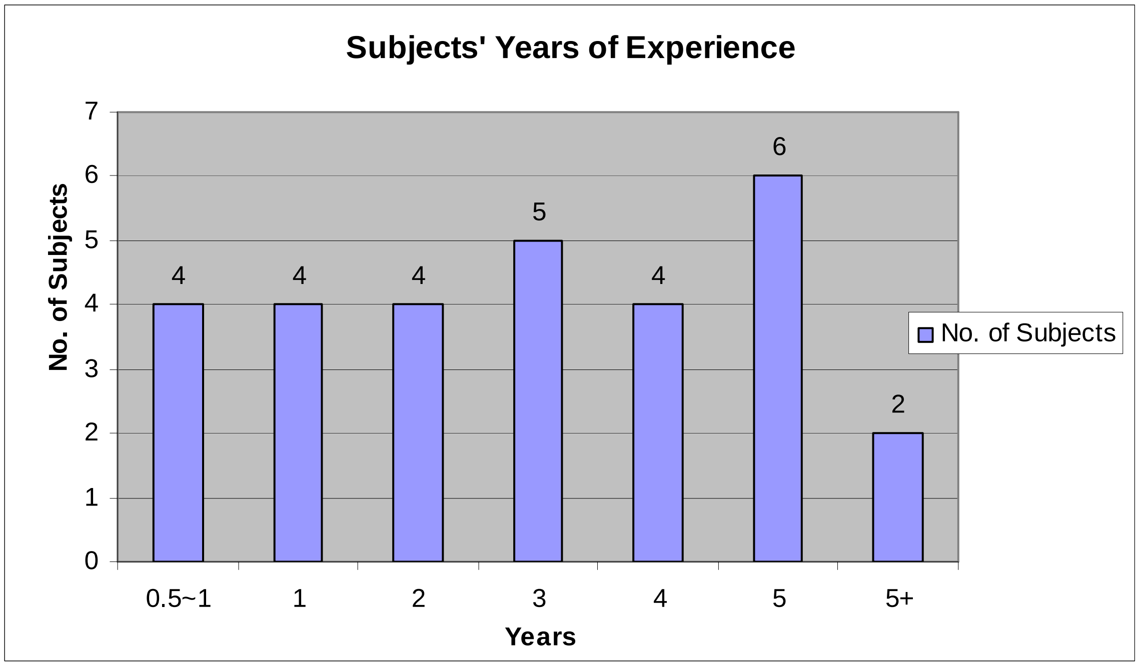


Figure 3.1: The subjects' experience of acceptance testing

In the above figure, only 14% (4/29) of the subjects have limited experience of 0.5 – 1 year. However, considering the learning curve of acceptance testing, this time frame is enough for people to thoroughly understand this technique. A total of 72% of the subjects had more than two years experience on acceptance testing. Therefore we can say that the set of subjects has satisfactory experience of doing acceptance testing.

The tools used by the participants with the number of users are shown in Figure 3.2:

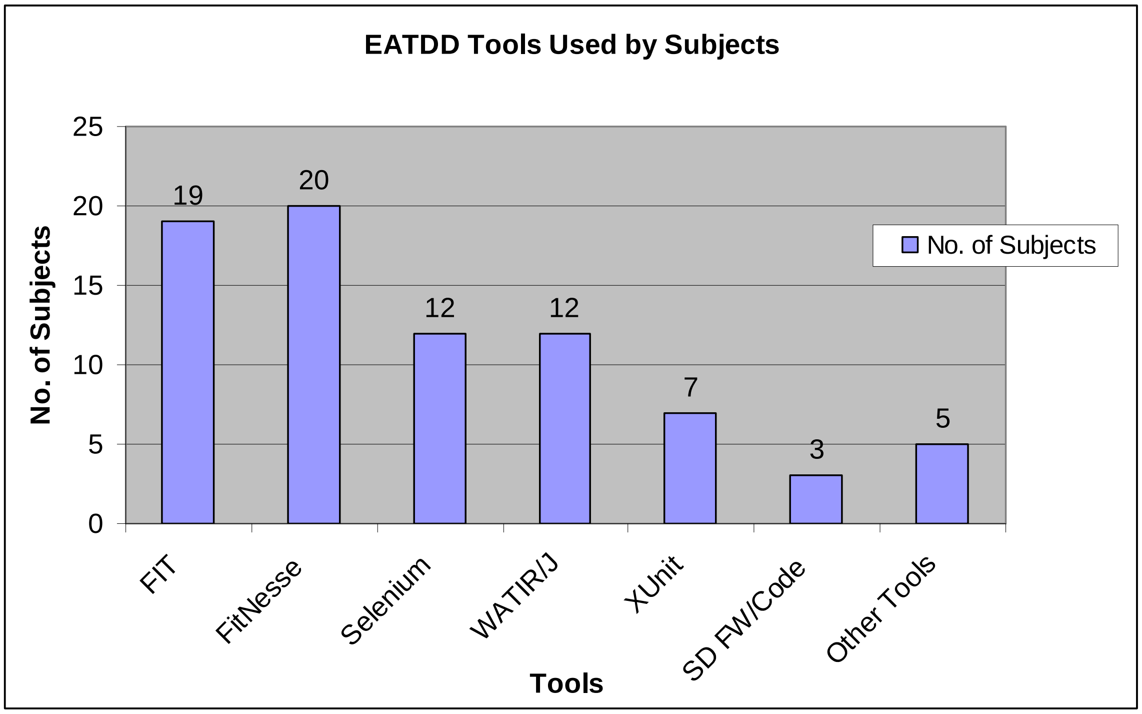


Figure 3.2: Tools used by the subjects for EATDD

As shown in the above figure, FitNesse, Fit, Selenium and WATIR/WATIJ are widely used by the industry[8]. XUnit tools are also used for acceptance testing besides their original purpose of unit testing. Some of the subjects also developed their own frame work and tools to do acceptance testing (shown in the figure as “SD FW/Code”, which means Self Developed Framework or Code). Besides the tools included in the above figure, other tools are also used in industry for acceptance testing, including Canoo, WebTest, JWebTest, Rational Robot, Avignon and Mercury QuickTest Professional™.

3.5 Survey Findings

In this section, six main findings are shown, including the number of acceptance test and the project scale, the frequency of adding or changing acceptance tests, the frequency of running acceptance tests, the test running pattern, the average and maximum time frame of acceptance testing.

3.5.1 Number of Acceptance Tests and Project Scale

In the questionnaire, the subjects are asked about the number of acceptance tests they have for the current or latest project and the scale of the project. The number includes all the acceptance tests created for the whole project. The answer is shown in Figure 3.3:

[8] The sum of the number of users exceeds the number of subjects, because people may have used multiple tools for acceptance testing.

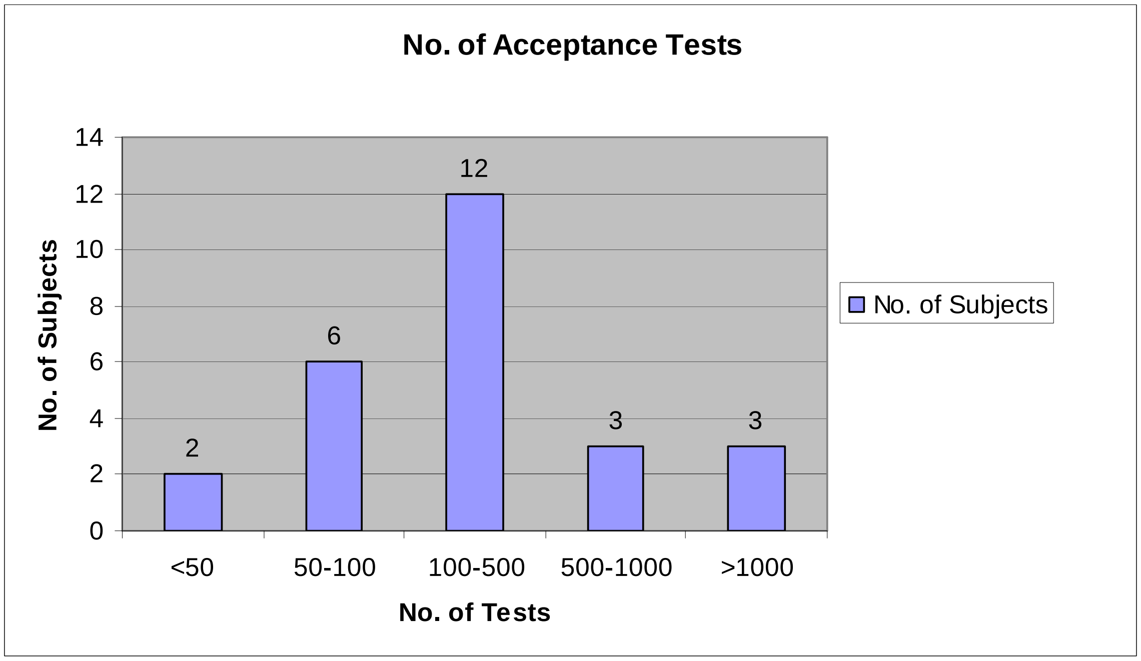


Figure 3.3: Number of acceptance tests the subjects have for their current project or latest project with the project scale

There are 26 valid answers for this question. Only two of them (8%) have less than 50 acceptance tests. However, one of the two answers is 30 acceptance tests which are only the ones automated by the developers. It can be expected that there are also tests that are run manually. 23% (6/26) reported the number to be 50–100. 46% (12/26) has tests of 100–500, which is the most of all categories. The numbers of 500 – 1000 and above 1000 are both 12% (3/26).

Finding 3.1**: the number of acceptance tests for a project is commonly (69%, 18/26): from 50 to 500.**

3.5.2 Frequency of Adding or Changing Acceptance Tests

Subjects reported their frequency of making changes to the pre-defined acceptance tests, among which 26 are valid. Figure 3.4 shows the results to this question:

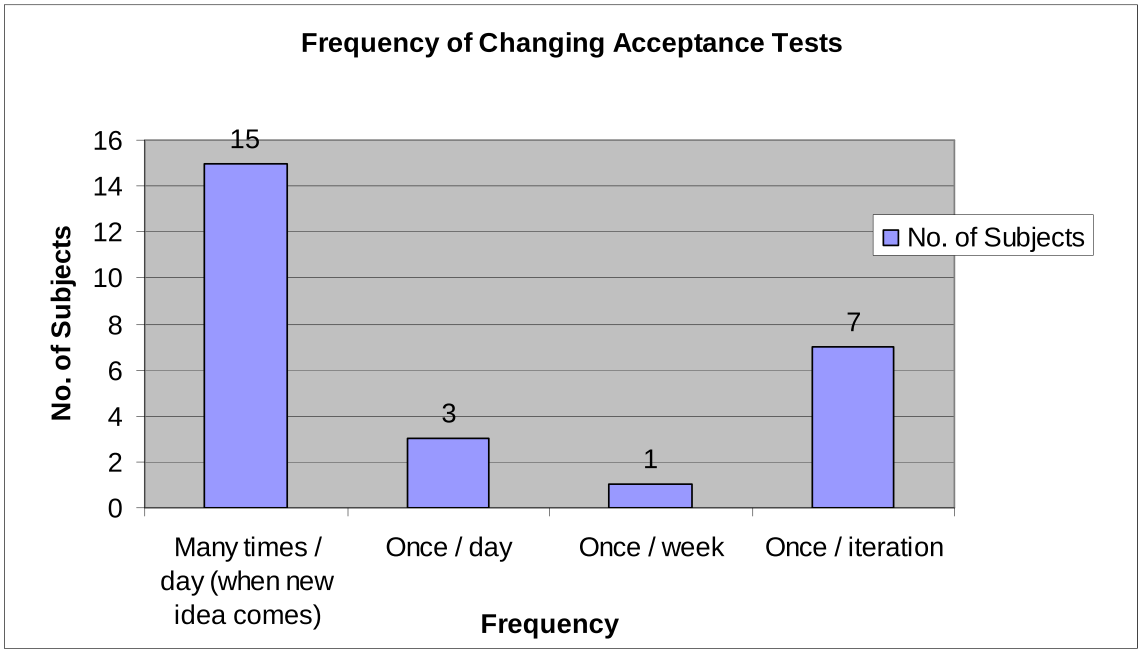


Figure 3.4: Frequency of adding or changing acceptance tests basing on customer requirement changes

In this chart, most subjects (58% 15/26) change the acceptance tests very frequently, several times a day, when they come up with new ideas. This result may come from the fact that in the implementation phase the developers often need to refactor the detailed requirement, at which time they will discuss these issues with the customer and change the predefined acceptance tests or adding new acceptance tests. 12% (3/26) of the subjects change the tests once a day. 27% (7/26) of the subjects make changes to the acceptance tests once per iteration. This may be because acceptance tests are made to guide the development for each iteration. At the beginning of an iteration, the customers join the development team to pick stories for this iteration according to the reviewed prioritization. At the same time the customers define the acceptance tests for each story.

***Finding 3.2*: in most cases (58%, 15/26), acceptance tests are modified frequently for many times a day.**

3.5.3 Frequency of Running Acceptance Tests

29 valid answers are collected for the question on the frequency of running acceptance tests. The detailed results are shown in Figure 3.5.

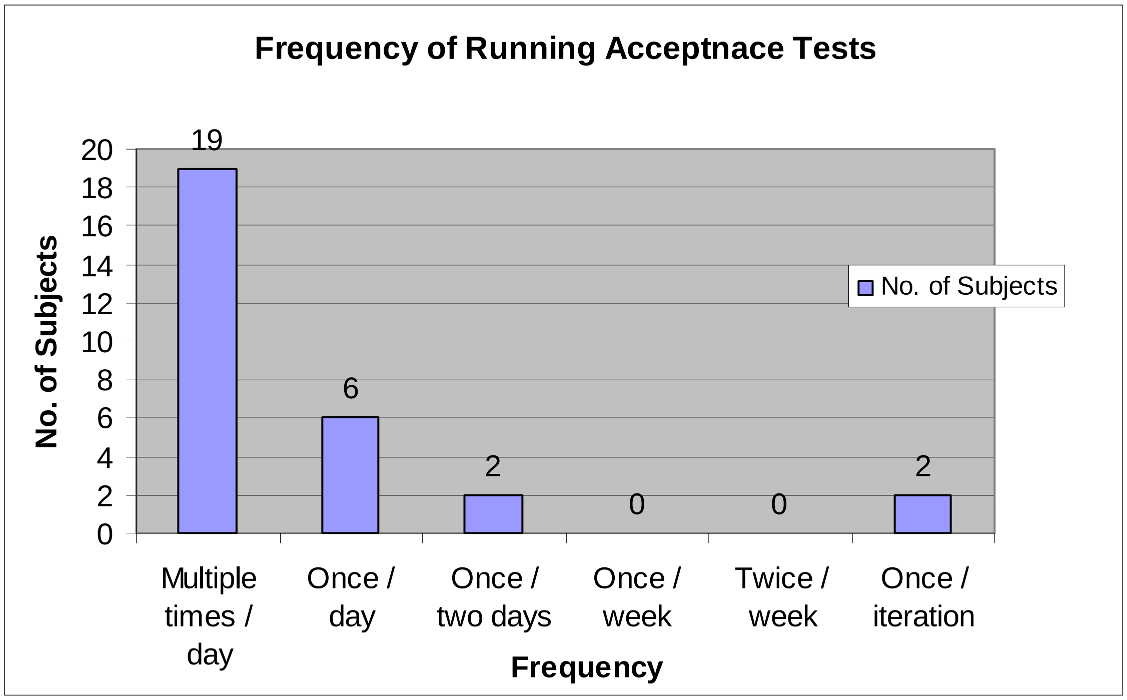


Figure 3.5: Frequency of running acceptance tests.

In this figure, we can see that most of the subjects (66% 19/29) run acceptance tests multiple times per day. This result may be because they run acceptance tests as regression tests to ensure no loss functionality. 21% (6/29) of the subjects run acceptance tests once per day. This may be because of daily integration: they run acceptance tests before they check in the code everyday to make sure the newly added functionalities are working compatibly with the previous system. Only 7% (2/29) of the subjects run acceptance tests every two days and no body is running acceptance tests on a weekly basis. 7% (2/29) of the subjects run acceptance tests once per iteration. They may only run all the acceptance tests right after the integration at the end of the iteration.

***Finding 3.3*: in most cases (65.52%, 19/29), acceptance tests are running often and multiple times a day.**

3.5.4 Test Running Pattern

In the questionnaire, we asked the subjects about their pattern of running acceptance tests, including only run tests for the current story, always run all existing tests or run in both patterns from time to time. Figure 3.6 shows the result with 27 valid answers.

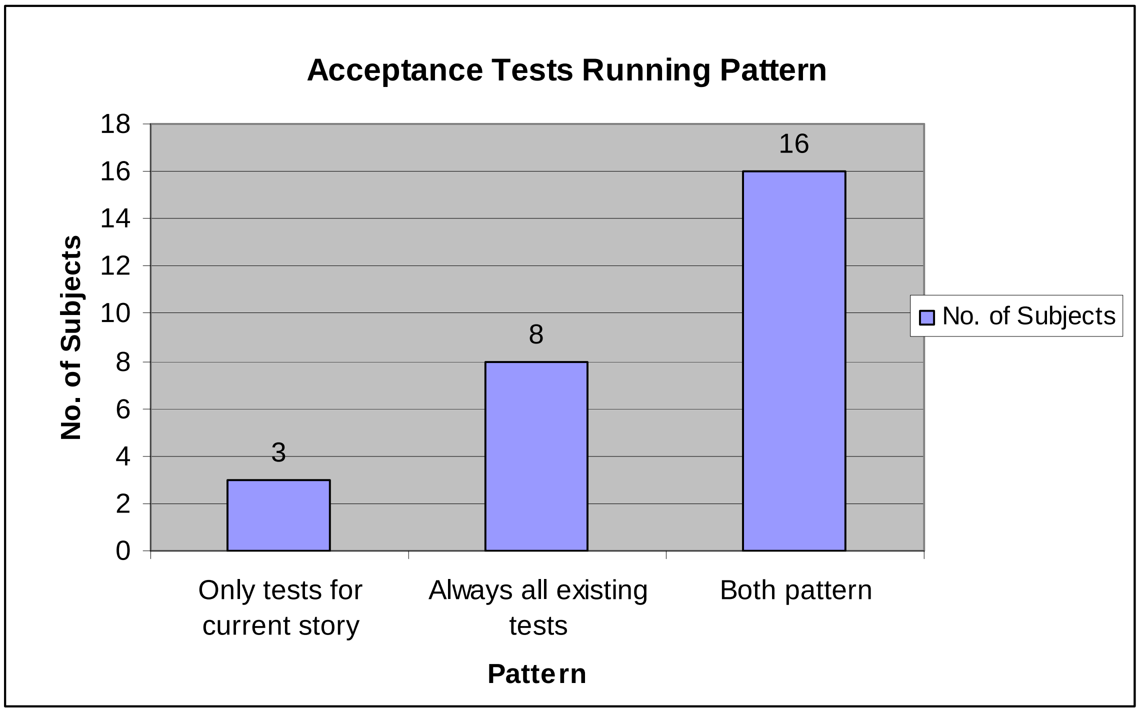


Figure 3.6: Test running pattern

When running acceptance tests, 11% (3/27) of the subjects only run the acceptance tests that belong to the current story they are working on and 30% (8/27) of the subjects always run all the acceptance tests. The majority (59% 16/27) run acceptance tests in both patterns. One of the subjects said his team "*run test related to the story*" and "*continuous integration system runs them (acceptance tests) all on each 'check-in'*". This means the acceptance tests of a single story are run at development time, while all the tests are run as regression tests after system integration.

***Finding 3.4*: in most cases (59%, 16/27), acceptance tests for one story are run for verifying whether the story is finished and acceptance tests for all existing stories are run for continuous integration.**

3.5.5 Time Frame of Acceptance Testing

We define the time frame of EATDD to be the time between creating a new acceptance test and the first time of its successful pass. In the questionnaire, we asked about both the AVERAGE and the MAXIMUM EATDD time frame. The results are shown in Figure 3.7 for both average and maximum acceptance testing time frame.

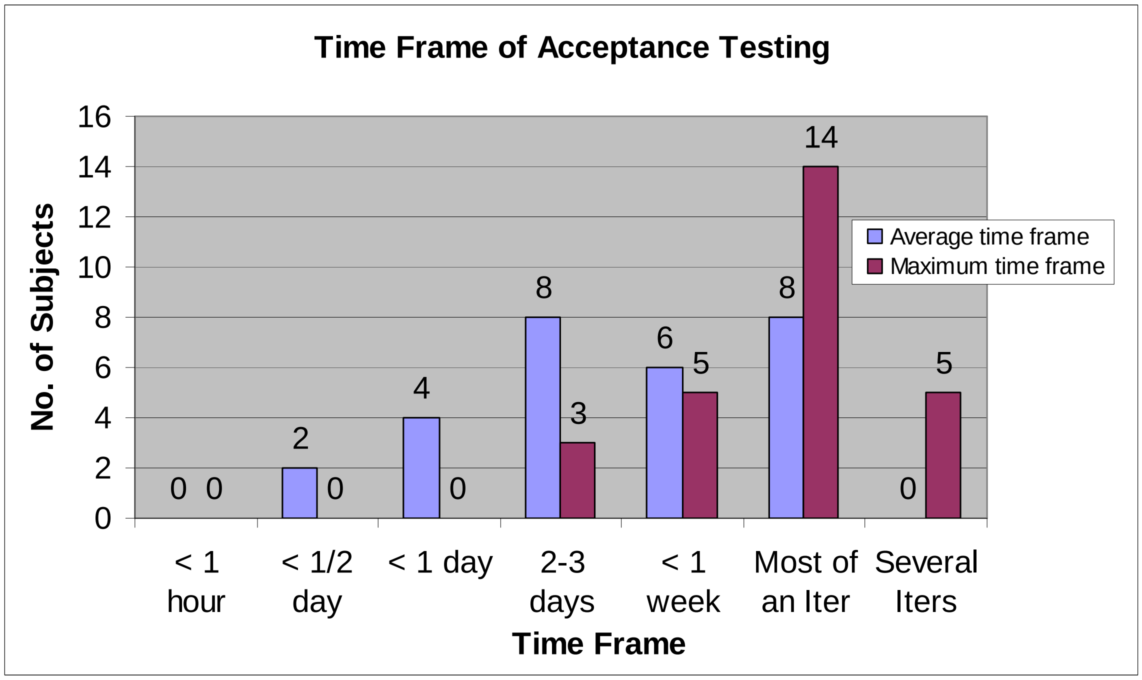


Figure 3.7: Average and Maximum EATDD time frame

28 valid answers are analyzed for the average time frame. In Figure 3.7, we can see that a total of 93% (26/28) of the participants reported the average timeframe to be more than half a day (4 hours) for defining an acceptance test and making it pass and the number increased to 100% when they reported the maximum time. Most subjects (8, 6 and 8) reported the average time frame to be between 2-3 days and an iteration. None of

the subjects has average time frame of several iterations, because it is expected that all requirements be implemented within one iteration.[9]

27 valid answers are analyzed for the maximum acceptance testing time frame. In Figure 3.7, we can observe that 100% the subjects reported the maximum acceptance testing time frame to be more than one day. Most subjects (52%, 14/27) reported the maximum time frame to be most of an iteration. Even 19% (5/27) subjects reported the time frame to be several iterations, including one subject reporting "*several months*" and another subject reporting "*2-3 iterations*".

Finding 3.5**: the average timeframe between defining one acceptance test and making it pass successfully, following EATDD, is more than half a day (4 hours).**

Finding 3.6**: the maximum timeframe between defining one acceptance test and making it pass successfully, following EATDD, may be most of an iteration or even more than one iteration.**

3.6 Motivation of FitClipse

The findings from section 3.5 demonstrate how acceptance tests are being used in industry. These findings, especially ***Finding 3.5*** and ***Finding 3.6***, directly motivate my work.

[9] In Agile Method, requirements are in the form of stories. Customers pick up the stories for each iteration basing on the developer's effort estimation, and the stories' priority. The developer's estimation on the stories indicates that the development team will finished all the stories, which are picked up by the customers, within the iteration.

3.6.1 Identifying Two Test Failure States

A major difference between UTDD using unit tests and EATDD is the timeframe between the definition of a test and its first successful pass. Usually, in UTDD the expectation is that all unit tests pass all the time and that it only takes a few minutes between defining a new test and making it pass [Beck 2003]. Unit tests cover very finely grained details, which makes this expectation reasonable in a UTDD context. As a result, any failed test is seen as a problem that needs to be resolved immediately.

Acceptance tests, on the other hand, cover larger pieces of functionality. Therefore, we expected that it often may take developers several hours or days, sometimes even more than one iteration, to make them pass.

Finding 3.5 and ***Finding 3.6*** strongly support our expectation. Therefore we can draw the conclusion that the time frame between the definition of an acceptance test and its first successful pass is significantly longer than that of a unit test.

Due to the substantial delay between the definition and the first successful pass of an acceptance test, a development team can *NOT* expect that all acceptance tests pass all the time. A failing acceptance test can actually mean two things:

- *Unimplemented Feature*: The development team has not yet finished working on the story with the failing acceptance test (including the developer has not even started working on it).
- *Regression Failure*: The test has passed in the past and is suddenly failing – i.e. a change to the system has triggered unwanted side effects and the team has lost some of the existing functionality.

The first case is simply a part of the normal test-driven development process: It is expected that a test that has never passed before should fail if no change has been made to the system code.

The later case should be raising flags and needs to be highlighted to the development team, because the number of tests is relative large, which is proved by ***Finding 3.1***. It is hard for the developers to remember the test failure states and identify them. We can easily imagine that people will have difficulty remembering the states for more than 100 tests. In addition, taking the long time frame of EATDD and other aspects, such as distributed development and frequent change of acceptance tests (supported by ***Finding 3.2***), into consideration, it is also difficult for people to manage a small project with 50 – 100 acceptance tests. If no support for automatically identifying the two kinds of failures[10] is provided, the users have to rely on their recollection of the past test results or reorganization of the test suites to determine the meaning of a failing test. For anything but very small projects, this recollection will not be reliable and the reorganization will be either difficult or time consuming.

FitClipse raises a flag for a test which is failing but was passing in the past. It identifies this condition by storing the results of previous test executions and is, thus, able to distinguish these two cases and splits up the "failed" state of Fit into two states: Unimplemented Feature, which is "still failing state" and Regression Failure "now failing after passing earlier".

Separating the additional failed state has the following advantages:

[10] Two failures: failing tests without implementation and failing tests which have been passing before.

- *The distinction enables better understanding of the current state of the project*: At some time of the project, many tests may fail. Some of the test failures mean the problems have not yet been solved by the developers, while the other failures are caused by changes of the system.

- *The distinction enables better progress reporting*: In the progress report, different flags of failing tests represent unfinished system features or previously completed features which are broken by changes of the system. This detailed information can help the project manager make better decisions for later iteration planning.

- *This feature helps the developers to accurately identify the regression failures*: With acceptance tests functioning as regression tests, whenever the changes of the system in one place cause side effects on other parts of the existing system, the flag is raised - instead of totally relying on the memory of the developers - to inform the team that they are breaking the system. In this case, steps can be undertaken to fix the broken functionalities.

- *It keeps the developers from being overwhelmed by many failures at the beginning of an iteration*: At this time point, the test failures are a deliberate result of EATDD. The content of the tests has not been addressed by the developers.

3.6.2 Test Result History Report

FitClipse has the functionality of showing the test result history. The motivations of building such a feature are:

First, keeping the history of the number of passing and failing acceptance tests of a project can help the development team understand the development progress. From the statistics, the development team can grasp the speed of their development and where they are in the development process. For instance, if an iteration runs a suite of acceptance tests (a suite is a list of tests which are related or communicate with each other), which represent system functionalities, from the test history we can monitor the number of passing tests and thus gauge the progress of the iteration. The change in the number of tests passing can help the development team to keep track of how fast they are developing (if the number of functional features are increasing), and better estimate how many features can be undertaken in a single iteration.

Second, changes are often made to acceptance tests. ***Finding 3.2*** illustrates the fact that most people make changes to acceptance tests many times a day when they come up with new ideas. It can be understood that acceptance tests which were changed before might need to be reversed back to a previous version. Of course version control has been implemented by tools such as FitNesse or in other forms of test repository. However, only keeping the version information is not sufficient enough. Sometimes the developers or tests make improper changes and keep adding changes to the tests for a period of time. After some time when people discover the mistake, provided only a version number and a date, it is very hard for them to decide which version of the test is useful. It will be very helpful if the test result information can be kept with the corresponding versions of the test. By viewing the test results, people can easily identify the test that is performing as expected. FitClipse achieves this goal by keeping test result record after each test run.

In addition, identifying the regression failure of acceptance tests requires maintaining the history of the tests to look up whether the tests were passing before. Further, the frequency of running acceptance tests and the test running pattern enables the availability of history data. From survey ***Finding 3.4*** we can see that acceptance tests are running multiple times a day and both tests for single story and all existing tests are run. Single test run of a story will ensure the history information being available for later lookup to identify regression failure. Running of all existing tests will make sure the project progress information is maintained for generating project development reports. In addition, survey ***Finding 3.3*** shows the acceptance tests are running frequently, which provides the rich source for test result history data.

3.7 Summary

In this chapter, a survey is summarized in order to find out what kind of support should be provided for EATDD. According to the findings of the survey, the motivation of building an acceptance testing tool, FitClipse, is also introduced: there is a lack of support for the substantial long time frame of EATDD. Tool support for separating regression failures from unimplemented features, and persisting test result history information needs to be provided for acceptance testing. In the next chapter, I will present the user scenarios of using FitClipse, the tool we developed for supporting EATDD in Agile environment, in order to help the readers get familiar with the tool and its feature.

Chapter Four: FitClipse User Scenario

Before introducing the design of FitClipse, I would like to present what FitClipse can do and how it works by describing the user scenarios. In Chapter 2, I have talked about how FitNesse server alone works as an acceptance testing tool. In this chapter, I will present user scenarios of using FitClipse with FitNesse server as the backend Wiki repository for EATDD.

4.1 FitClipse Workflow

In section 2.6.1.2, I introduced workflow of FitNesse when it is used as an acceptance testing tool. To use FitNesse, the users need to edit and run the acceptance tests inside a web browser and implement the fixture code in another development environment. However, FitClipse, instead of using a browser, enables writing and running acceptance tests inside Eclipse environment while cooperating with other useful Eclipse plug-ins for development purpose. Figure 4.1 shows the workflow of FitClipse working as an Eclipse plug-in.

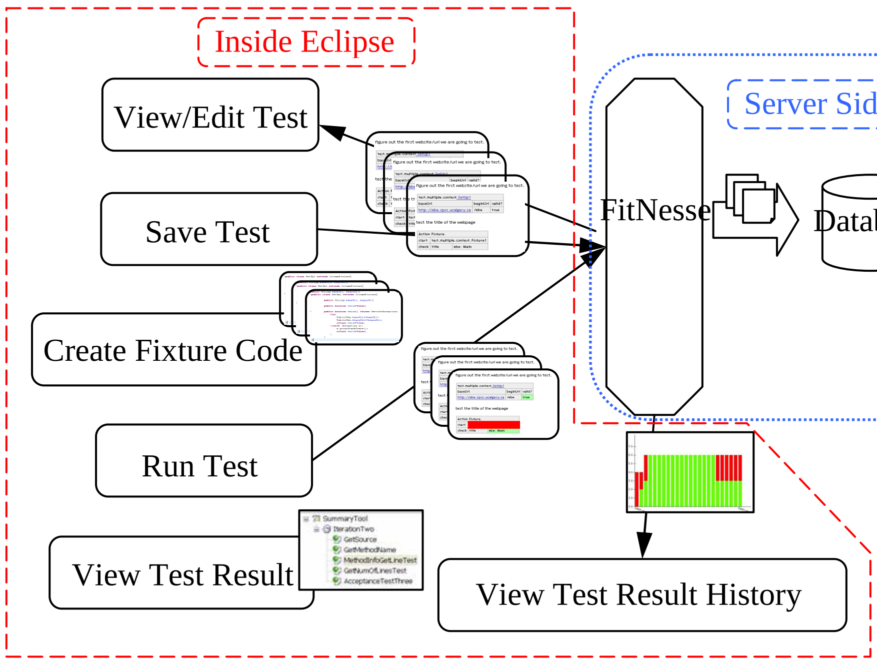
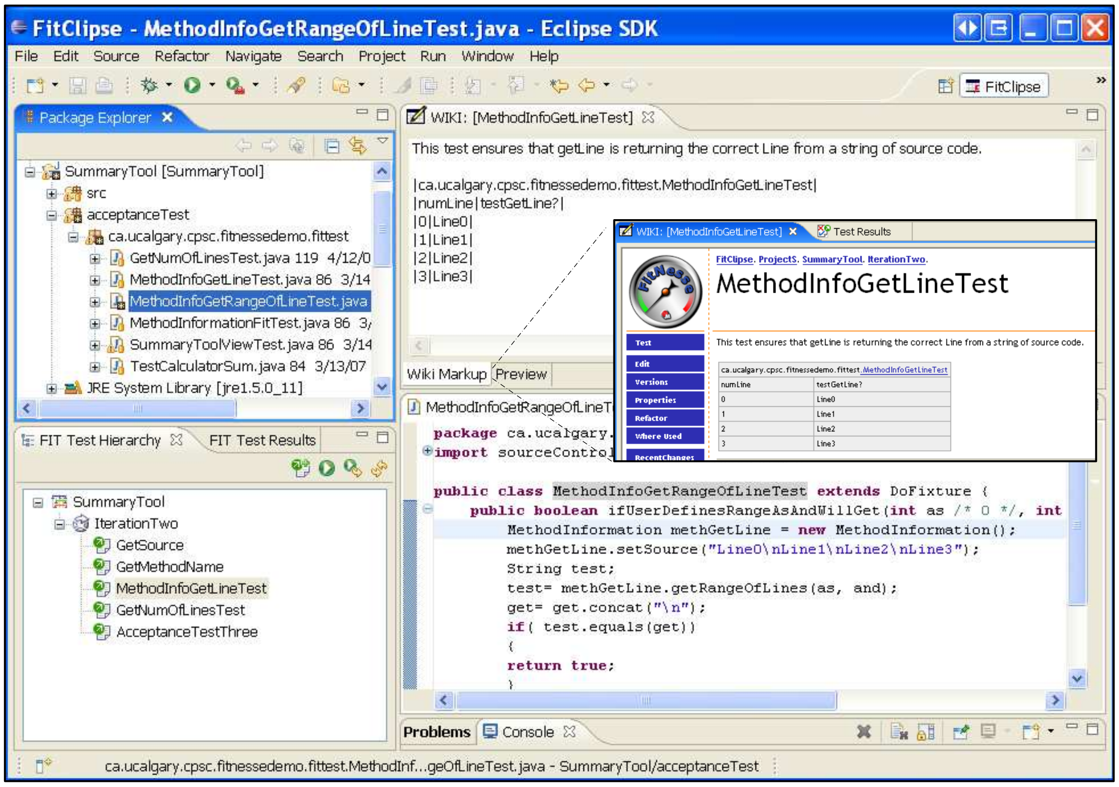


Figure 4.1: FitClipse workflow

The following is the user scenario for the customers and the development team to use FitClipse for EATDD:

- *Create and modify Acceptance Tests*: acceptance tests are detailed system specifications. In EATDD, composing acceptance tests happens at the beginning of each iteration or before the work on a story is started. The customers first pick up the predefined stories from the project backlog. The selected stories are the tasks they expect the development team to work on for the on-going iteration. Then the customers with the development team define the acceptance tests for each story. These stories specify how the system should behave and help the whole development team to understand this information. Acceptance tests are created in the Fit Test Hierarchy View. As shown in Figure 4.2 (upper right), acceptance tests are edited in the Fit Test Editor in the form of Wiki syntax and can be previewed in Html format in the preview tab. The acceptance tests created in the editor are directly saved on the server.


FitClipse - MethodInfoGetRangeOfLineTest.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
FitClipse
Package Explorer
SummaryTool [SummaryTool]
src
acceptanceTest
ca.ucalgary.cpsc.fitnessedemo.fittest
GetNumOfLinesTest.java 119 4/12/0
MethodInfoGetLineTest.java 86 3/14
MethodInfoGetRangeOfLineTest.java
MethodInformationFitTest.java 86 3/
SummaryToolViewTest.java 86 3/14
TestCalculatorSum.java 84 3/13/07
JRE System Library [jre1.5.0_11]
FIT Test Hierarchy
FIT Test Results
SummaryTool
IterationTwo
GetSource
GetMethodName
MethodInfoGetLineTest
GetNumOfLinesTest
AcceptanceTestThree
WIKI: [MethodInfoGetLineTest]
Wiki Markup
Preview
Test Results
FitClipse. ProjectS. SummaryTool. IterationTwo.
MethodInfoGetLineTest
Test
Edit
Versions
Properties
Refactor
Where Used
RecentChanges
MethodInfoGetRangeOfLineTest.java
Problems
Console
ca.ucalgary.cpsc.fitnessedemo.fittest.MethodInf...geOfLineTest.java - SummaryTool/acceptanceTest

```
This test ensures that getLine is returning the correct Line from a string of source code.

|ca.ucalgary.cpsc.fitnessedemo.fittest.MethodInfoGetLineTest|
|numLine|testGetLine?|
|0|Line0|
|1|Line1|
|2|Line2|
|3|Line3|
```

This test ensures that getLine is returning the correct Line from a string of source code.

ca.ucalgary.cpsc.fitnessedemo.fittest.MethodInfoGetLineTest	
numLine	testGetLine?
0	Line0
1	Line1
2	Line2
3	Line3

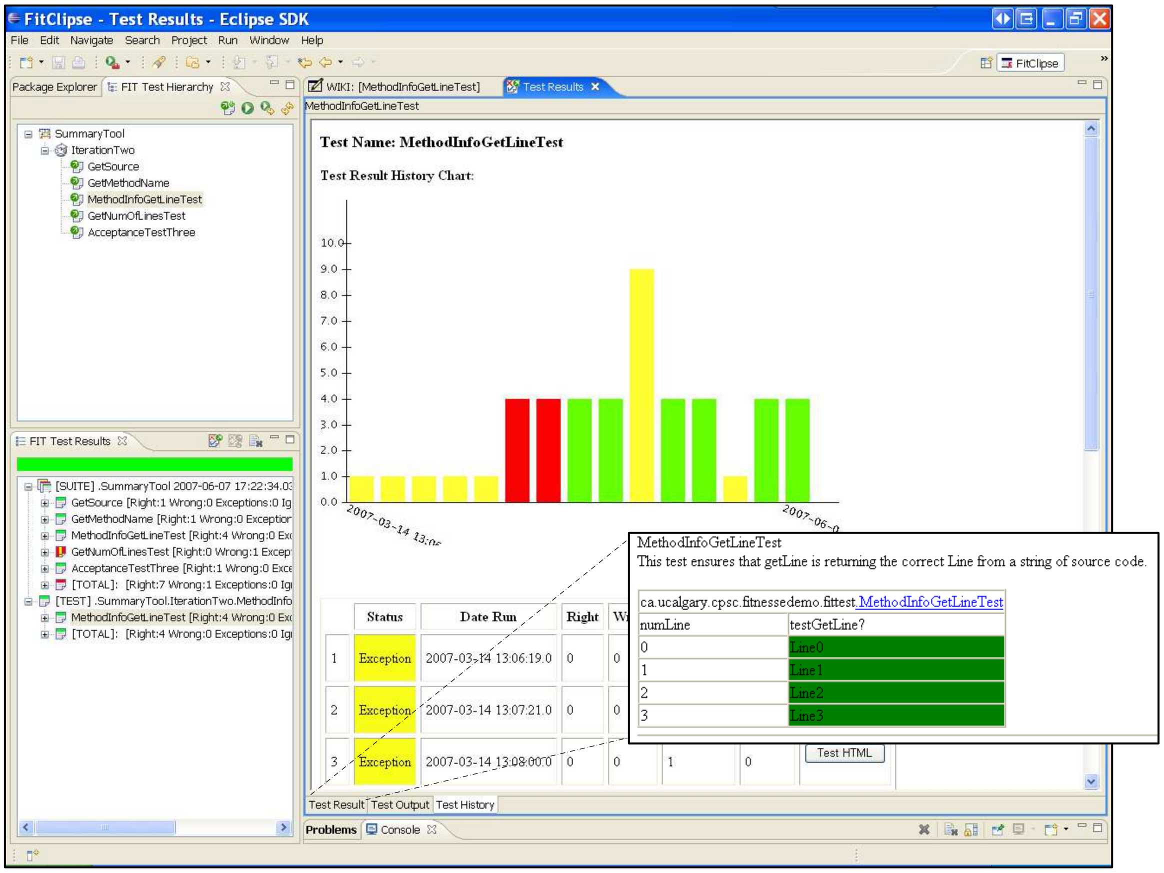
```
package ca.ucalgary.
import sourceControl

public class MethodInfoGetRangeOfLineTest extends DoFixture {
    public boolean ifUserDefinesRangeAsAndWillGet(int as /* 0 */, int
        MethodInformation methGetLine = new MethodInformation();
        methGetLine.setSource("Line0\nLine1\nLine2\nLine3");
        String test;
        test= methGetLine.getRangeOfLines(as, and);
        get= get.concat("\n");
        if( test.equals(get))
        {
        return true;
        }
```

Figure 4.2: FitClipse environment for writing acceptance tests and fixture code

- *Create Test Fixtures*: the fixture classes are implemented with Java code. In EATDD, fixtures are created after the application functions are implemented following UTDD. The fixtures are created by the developers in the ordinary Eclipse Java environment, which is shown in Figure 4.2 as the upper left area and lower right area. Based on a given acceptance test, FitClipse uses a wizard to generate the fixture code stubs automatically.

- *Implementation*: unit test-driven development is utilized in conjunction with EATDD. Developers follow UTDD to implement the features of the system.
- *Running Acceptance Tests and Viewing the Test Results*: Acceptance tests are run frequently all through the implementation of the system. FitClipse provides two kinds of test failure states and maintains the test result history for the development team to keep track of their development process. Figure 4.3 shows all test result states and a sample test result history in FitClipse. In the test result view on the lower left, the test results are summarized in a tree structure. In the right editor, first the test history chart is provided for a single test. The red and green bars mean the number of the failing and passing assertions with yellow meaning exceptions. Down the chart, detailed test result history information is provided, from which the developers can also view the result at a particular time.



[SUITE] .SummaryTool 2007-06-07 17:22:34.03
GetSource [Right:1 Wrong:0 Exceptions:0 Ig
GetMethodName [Right:1 Wrong:0 Exceptior
MethodInfoGetLineTest [Right:4 Wrong:0 Exc
GetNumOfLinesTest [Right:0 Wrong:1 Excep
AcceptanceTestThree [Right:1 Wrong:0 Exce
[TOTAL]: [Right:7 Wrong:1 Exceptions:0 Ig
[TEST] .SummaryTool.IterationTwo.MethodInfo
MethodInfoGetLineTest [Right:4 Wrong:0 Exc
[TOTAL]: [Right:4 Wrong:0 Exceptions:0 Ig

Test Name: MethodInfoGetLineTest

Test Result History Chart:

	Status	Date Run	Right	Wi			
1	Exception	2007-03-14 13:06:19.0	0	0			
2	Exception	2007-03-14 13:07:21.0	0	0			
3	Exception	2007-03-14 13:08:00.0	0	0	1	0	Test HTML

MethodInfoGetLineTest
This test ensures that getLine is returning the correct Line from a string of source code.

ca.ucalgary.cpsc.fitnessedemo.fittest.MethodInfoGetLineTest	
numLine	testGetLine?
0	Line0
1	Line1
2	Line2
3	Line3

Figure 4.3: FitClipse environment for viewing the acceptance test results

4.2 Summary

In this chapter, I introduced the user scenarios using FitClipse for EATDD. FitClipse enables the whole cycle of EATDD to be inside Eclipse IDE as well as provides supports for EATDD. In the next chapter, the tool requirements and detailed design of FitClipse is demonstrated. The tools aims to 1) provide automatic support for identifying two different acceptance test failure states and 2) maintains test result history to enable progress report for running acceptance tests.

Chapter Five: FitClipse Requirements and Design

In this chapter, I first analyze the requirements for building the tool which provides support for acceptance testing. Then I explain the overall structure of FitClipse. In addition, I present the tool design from different aspects: FitClipse client side and FitNesse server side.

5.1 Requirements Analysis

As analyzed in Chapter 3, special support should be provided for recording and displaying the acceptance test results. In addition, other basic functionalities for running acceptance tests should also be available. As a result, the requirements of the acceptance testing tool which uses Fit/FitNesse framework should include:

1. *Viewing and editing acceptance tests*: the acceptance tests are originally saved on the FitNesse server and organized in certain hierarchy. A tree view should be provided to view all the acceptance tests in a tree hierarchy and an editor should be build to view and edit the test and save the content back on the server.
2. *Creating and deleting acceptance tests*: inside Eclipse users should be able to cerate and delete acceptance tests. These tasks will be performed inside the tree viewer.
3. *Running acceptance test and test suite*: acceptance tests are executed by Fit test runner inside the tool. Tests should also be executed as a suite if they are organized into a suite in the tree hierarchy on the server.

4. *Reporting the test result*: test report summery should be shown after each test run. The information should include the number of passing, failing, exception and ignored test.

5. *Separating regression failure from unimplemented feature*: it has been analyzed in Chapter 3 that it will be helpful to separate regression failure from unimplemented feature automatically. In the tool, regression failure check will be performed after each test run and the regression failure will be tagged differently from the unimplemented feature.

6. *Test result history chart*: test result history should be persisted in the database. To show the test result history, a chart showing the test running date and result details should be provided.

7. *Supporting distributed development environment*: Nowadays, it is not uncommon that team members in a development team work at different times and/or different places while implementing the same project. Therefore a shared repository is needed for the team to check out or in the acceptance tests and view the test results.

8. *Fit test server configuration*: FitNesse server is very convenient for doing acceptance testing. A configuration page should be provided to configure the server information such as the server host, the port number and the classpath.

9. *Generating Fit fixture code*: fixture code needs to be developed for running Fit acceptance test. However people must be very careful to write working fixture

code that exactly matches the Fit tests. So support for automatically generating fixture code will be helpful to the developers.

10. *Integrating with Eclipse environment*: In as much as Fit is a useful tool for running acceptance tests, it lacks convenience when used for development work. For instance, when a typical developer uses Fit to run acceptance tests, it is necessary to keep switching between several windows, such as an IDE, a command line window for running the tests, and a browser to view the test results. In order to avoid switching between IDEs while coding, the tool should be integrated with the Eclipse development environment in the form of an Eclipse plug-in. Additional effort should be made on making the UI of the tool to be consistent with the Eclipse environment.

The next section presents the design of FitClipse. The design considerations are based on the ten requirements that have been proposed above.

5.2 Overall Structure

FitClipse [FitClipse 2007] is an Eclipse plug-in supporting the creation, modification and execution of acceptance tests using the FIT/FitNesse framework. The FitClipse tool consists of (multiple) FitClipse clients for editing and running acceptance tests, the Wiki repository for storing acceptance test definitions and the Database for storing the test execution history (Figure 5.1).

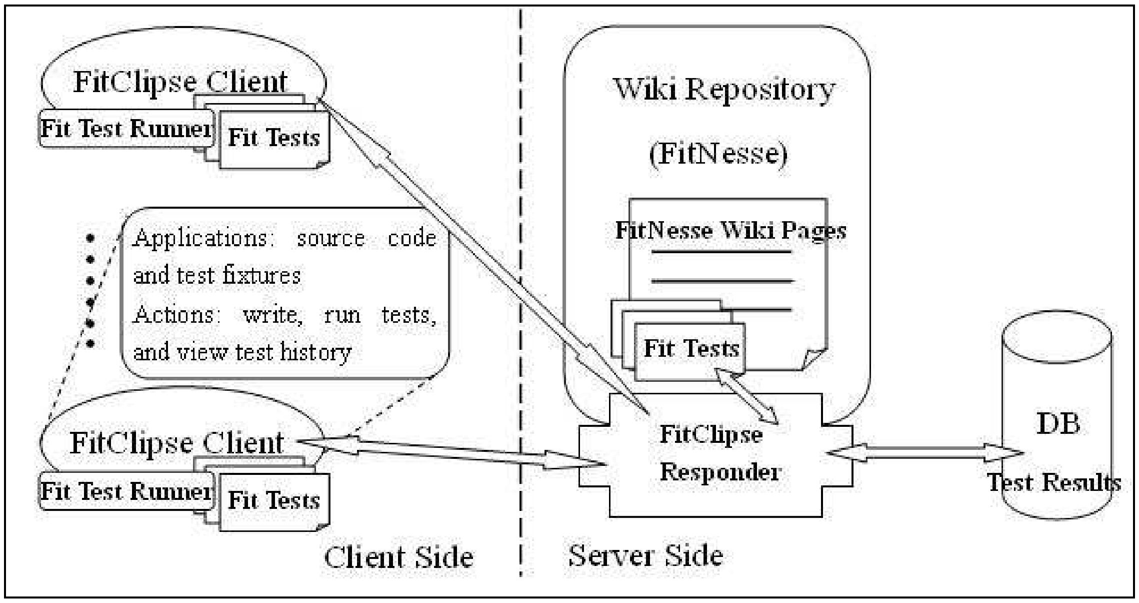


Figure 5.1: FitClipse overall structure of three components

As shown in Figure 5.1, FitClipse works as the client side application in the Client-Server structure. FitClipse clients talk to the FitNesse server, which is used as the Wiki repository, through HTTP calls to save the acceptance tests as Wiki pages on the server and save the test result into the database. On the server side, there are several responders implemented for processing different HTTP calls and deal with these tasks.

Working as a client with a Wiki repository server, FitClipse helps a distributed development team to share test definitions and test history between the developers. FitClipse uses a synchronized server to save the acceptance tests and a shared database for managing acceptance test results, which enables all the team members to synchronize on the latest changes in the process of EATDD. After the developers run all the acceptance tests and make sure all the tests are passing, they can safely check in their code to the code repository.

In FitClipse, we use the Fit framework for composing and running acceptance tests. The major motivation is the empirical studies introduced in section 2.7 which shows that Fit acceptance testing frame work is easy to use and helpful for conducting acceptance testing. In addition, Fit framework is widely used in industry. This was also indicated by the survey in Figure 3.2. There is no doubt that creating a tool using a widely used framework will enlarge the number of potential users.

5.3 FitClipse Client Side

Figure 5.2 describes the structure of FitClipse client side structure. Each FitClipse client consists of three main components: FitClipse UI part extended from Eclipse, the Server Connector for connecting and communicating with the server, and the Fit Test Runner for running acceptance tests.

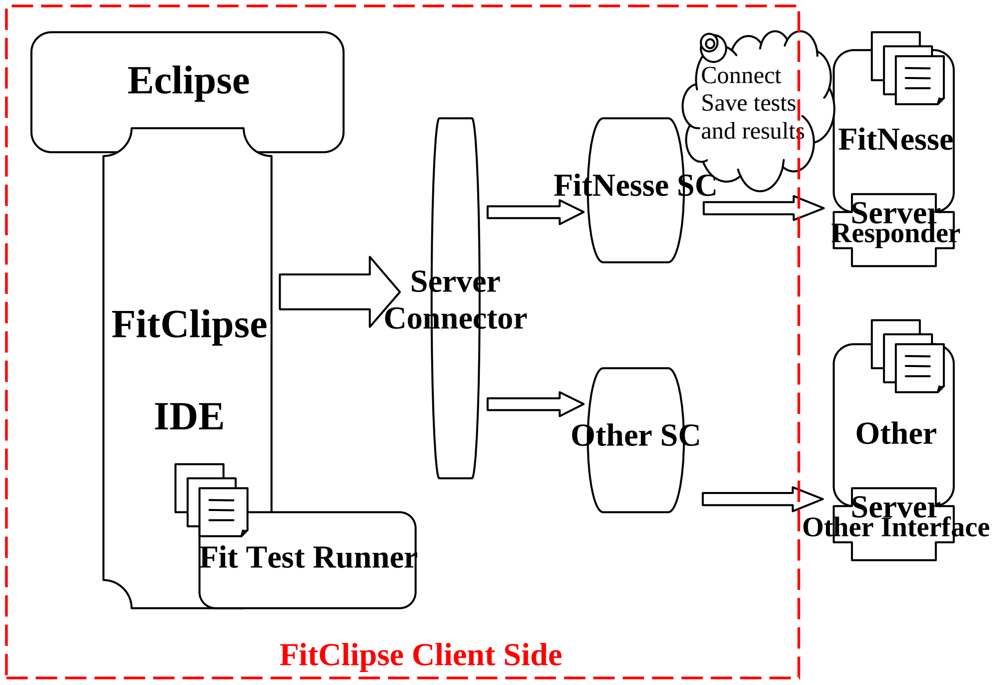
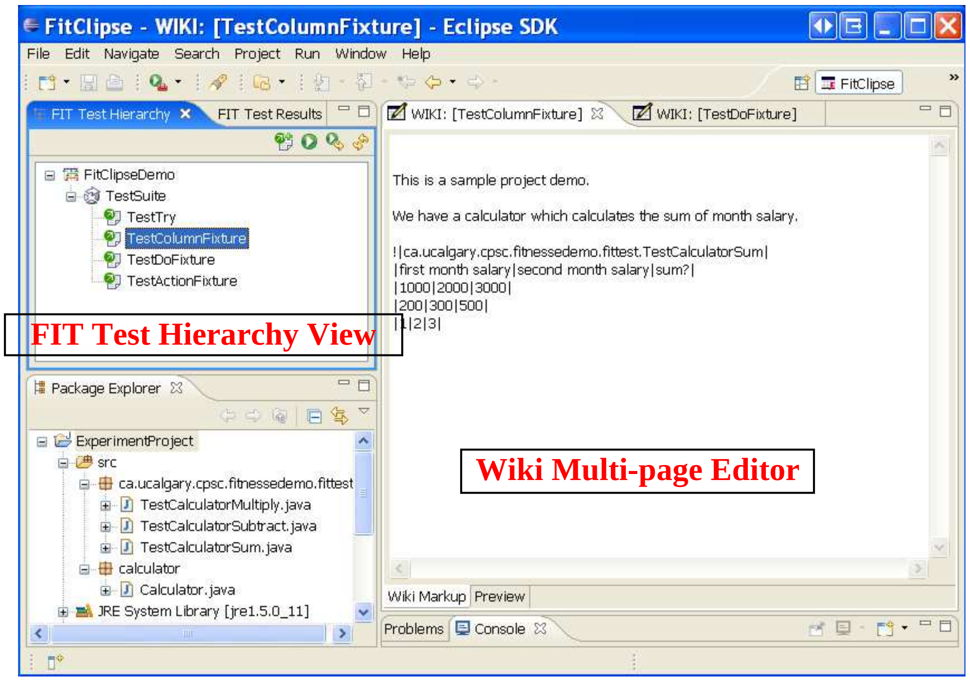


Figure 5.2: FitClipse front-end as an Eclipse plug-in

5.3.1 FitClipse UI

FitClipse user interface is extended from Eclipse working as an Eclipse Plug-in. It contributes to Eclipse the following components:

- Wiki Multi-page Editor: One page of the editor is a wiki editor for composing acceptance tests in Fit style tables. (See Figure 5.3 on the right) The contents are directly saved as wiki files on the server. The other page is for previewing the acceptance tests in HTML format on the server to make sure they appear as the user expected after being rendered by the Wiki engine. (See Figure 5.4 on the right)



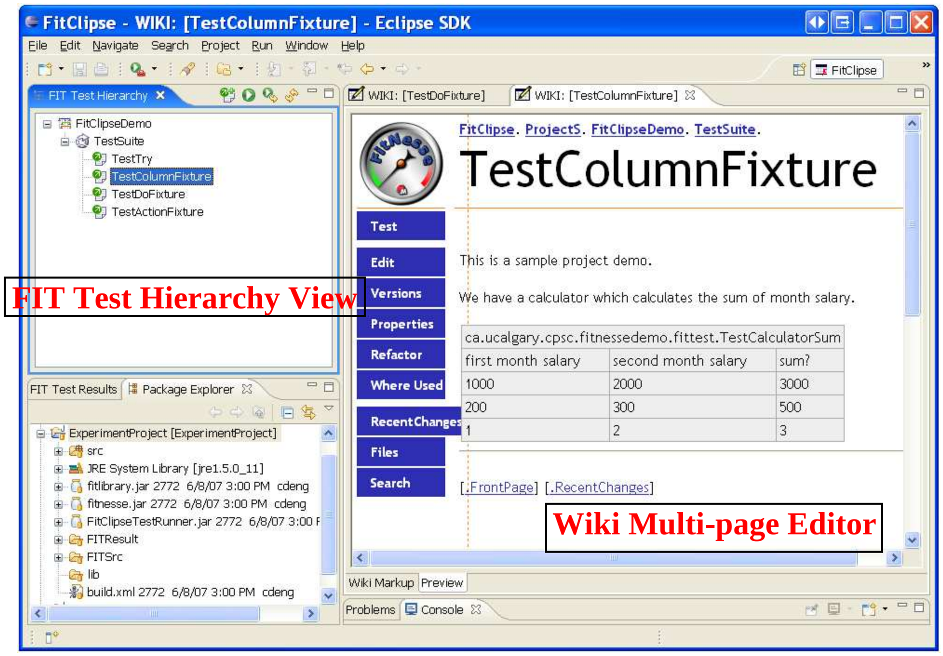
```
This is a sample project demo.

We have a calculator which calculates the sum of month salary.

!|ca.ucalgary.cpsc.fitnessedemo.fittest.TestCalculatorSum|
|first month salary|second month salary|sum?|
|1000|2000|3000|
|200|300|500|
|1|2|3|
```

Figure 5.3: FitClipse environment showing the Test Hierarchy View and the Wiki Markup Editor page

- FIT Test Hierarchy View: this view shows all the acceptance tests on the server for each project in a tree structure. The tree structure is organized according to the tests organization on the server. Tests may be grouped in a project under different iterations. (See Figure 5.3 and Figure 5.4 on the upper left) In this view, we can add, run and delete acceptance tests and generate fixture codes for the acceptance tests.


FitClipse - WIKI: [TestColumnFixture] - Eclipse SDK
File Edit Navigate Search Project Run Window Help
FIT Test Hierarchy
FitClipseDemo
TestSuite
TestTry
TestColumnFixture
TestDoFixture
TestActionFixture
FIT Test Hierarchy View
FIT Test Results
Package Explorer
ExperimentProject [ExperimentProject]
src
JRE System Library [jre1.5.0_11]
fitlibrary.jar 2772 6/8/07 3:00 PM cdeng
fitnesse.jar 2772 6/8/07 3:00 PM cdeng
FitClipseTestRunner.jar 2772 6/8/07 3:00 F
FITResult
FITSrc
lib
build.xml 2772 6/8/07 3:00 PM cdeng
WIKI: [TestDoFixture]
WIKI: [TestColumnFixture]
FitClipse. ProjectS. FitClipseDemo. TestSuite.
TestColumnFixture
Test
Edit
Versions
Properties
Refactor
Where Used
RecentChanges
Files
Search
[.FrontPage] [.RecentChanges]
Wiki Multi-page Editor
Wiki Markup
Preview
Problems
Console

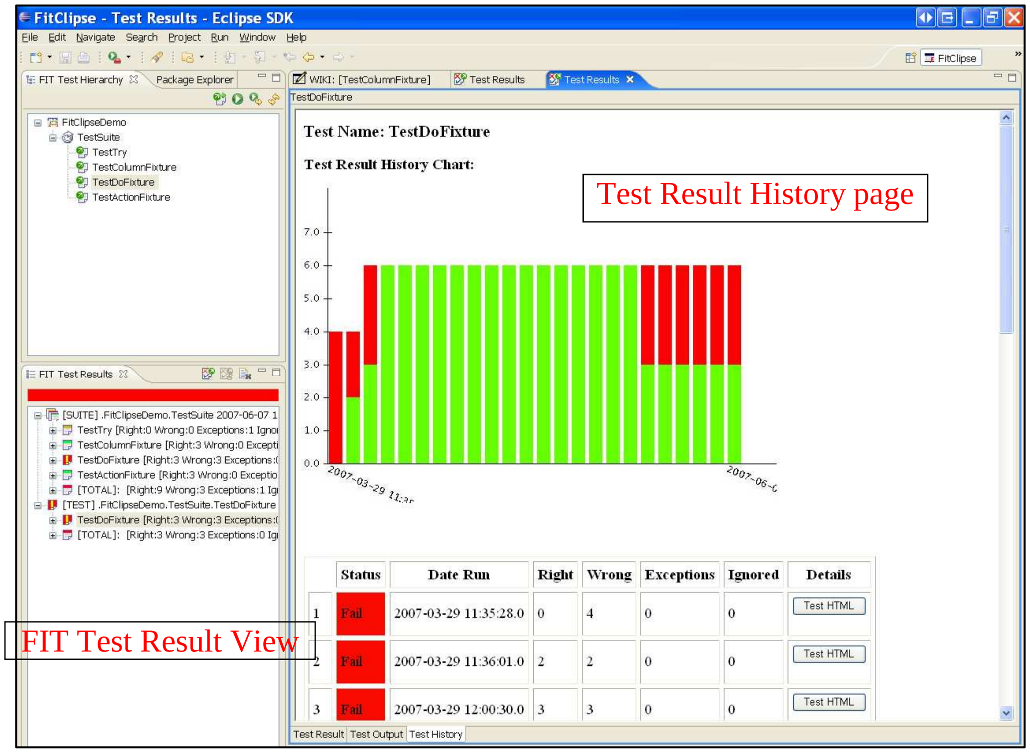
This is a sample project demo.

We have a calculator which calculates the sum of month salary.

ca.ucalgary.cpsc.fitnessedemo.fittest.TestCalculatorSum		
first month salary	second month salary	sum?
1000	2000	3000
200	300	500
1	2	3

Figure 5.4: FitClipse environment showing the Test Hierarchy View and the HTML Preview page

- FIT Test Result View: this view shows the acceptance test results after each run. (See Figure 5.5) As we have discussed before, two different test failures can be identified by different colors. Information about the test result can also be shown, including the time of running the tests, the number of pass, failure and exception assertions for both each test and the whole test suite.



FitClipseDemo
- TestSuite
  - TestTry
  - TestColumnFixture
  - TestDoFixture
  - TestActionFixture

[SUITE] .FitClipseDemo.TestSuite 2007-06-07 1
- TestTry [Right:0 Wrong:0 Exceptions:1 Ignoı
- TestColumnFixture [Right:3 Wrong:0 Excepti
- TestDoFixture [Right:3 Wrong:3 Exceptions:(
- TestActionFixture [Right:3 Wrong:0 Exceptio
- [TOTAL]: [Right:9 Wrong:3 Exceptions:1 Igı

[TEST] .FitClipseDemo.TestSuite.TestDoFixture
- TestDoFixture [Right:3 Wrong:3 Exceptions:(
- [TOTAL]: [Right:3 Wrong:3 Exceptions:0 Igı

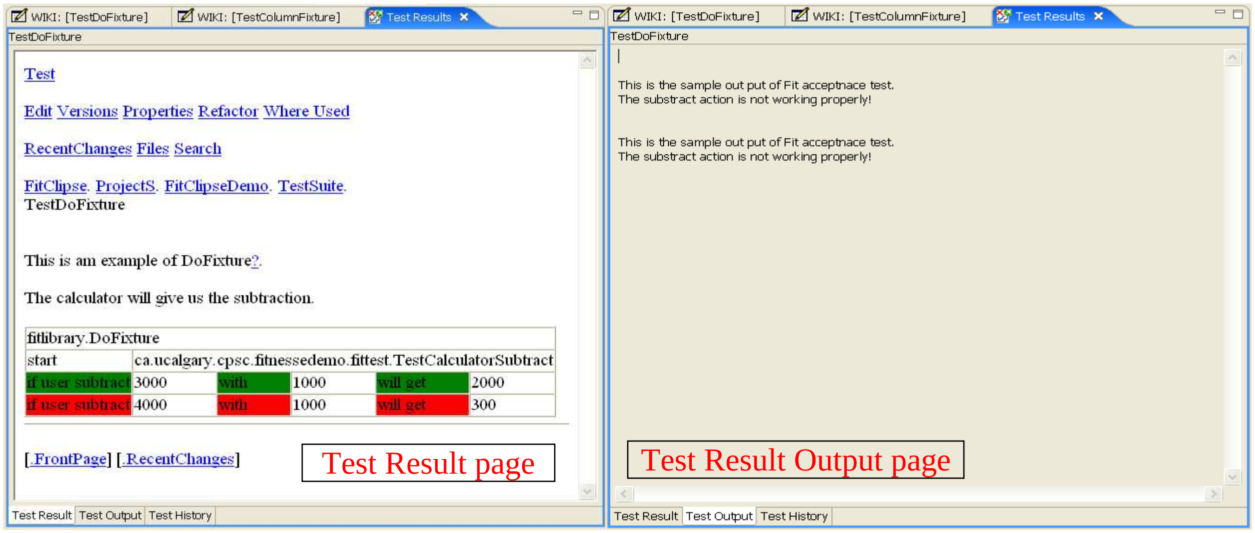
Test Name: TestDoFixture

Test Result History Chart:

	Status	Date Run	Right	Wrong	Exceptions	Ignored	Details
1	Fail	2007-03-29 11:35:28.0	0	4	0	0	Test HTML
2	Fail	2007-03-29 11:36:01.0	2	2	0	0	Test HTML
3	Fail	2007-03-29 12:00:30.0	3	3	0	0	Test HTML

Figure 5.5: FitClipse environment showing the FIT Test Result View and the Test Result History Page

- Test Result Multi-page Editor: this editor is not for editing tests but for showing test results. (See Figure 5.5 and Figure 5.6) The first page shows the test result of the last test run. (See Figure 5.6 on the left) The second page shows the output of the last test run. (See Figure 5.6 on the right) The third page shows the running history of the test. (See Figure 5.5 on the left) The information shown in the third page contains a test result chart and a test result table. The chart is the history visualization of the number of assertions from each test run in three different states: pass, fail and exception. Down the chart there is a table containing the detailed information of each test run. The test result at a specific time can be viewed from the table.



TestDoFixture

Test

Edit Versions Properties Refactor Where Used

RecentChanges Files Search

FitClipse. ProjectS. FitClipseDemo. TestSuite.
TestDoFixture

This is am example of DoFixture?.

The calculator will give us the subtraction.

fitlibrary.DoFixture					
start	ca.ucalgary.cpsc.fitnessedemo.fittest.TestCalculatorSubtract				
if user subtract	3000	with	1000	will get	2000
if user subtract	4000	with	1000	will get	300

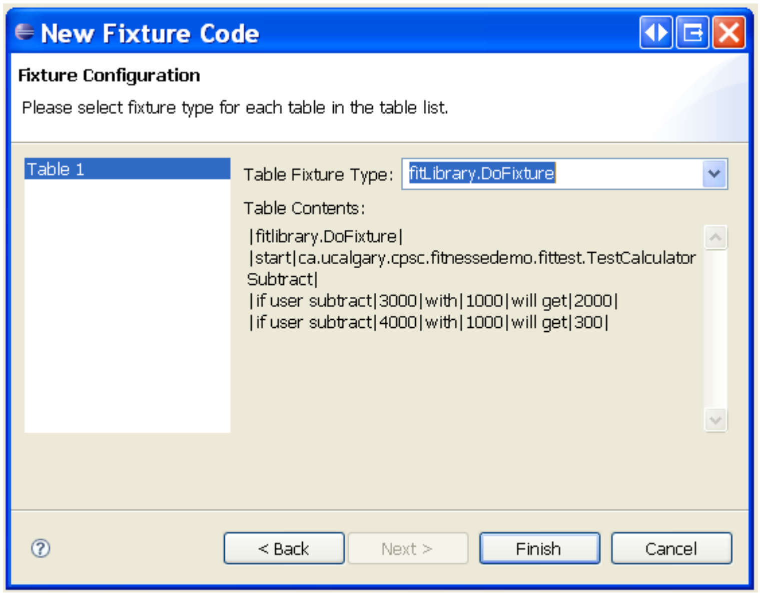
[.FrontPage] [.RecentChanges]

```
This is the sample out put of Fit accepntace test.
The substract action is not working properly!

This is the sample out put of Fit accepntace test.
The substract action is not working properly!
```

Figure 5.6: FitClipse environment showing the Test Result Multi-page Editor

- FitClipse Fixture Generation Wizard: this wizard is for generating the Fit test fixture automatically based on Fit acceptance tests which is written in Wiki syntax. (See Figure 5.7)



Fixture Configuration

Please select fixture type for each table in the table list.

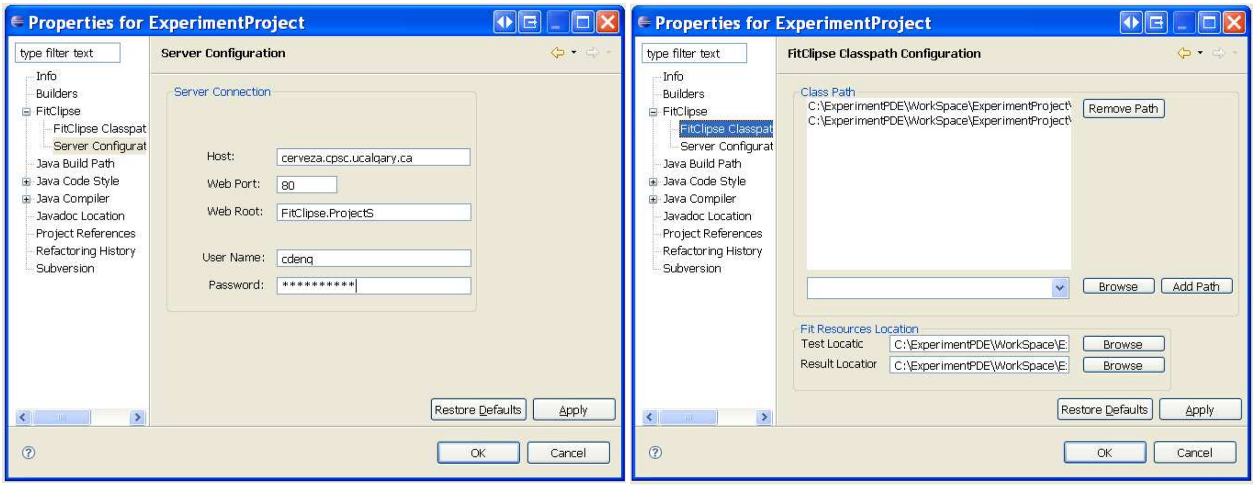
Table Fixture Type: fitLibrary.DoFixture

Table Contents:

```
|fitlibrary.DoFixture|
|start|ca.ucalgary.cpsc.fitnessedemo.fittest.TestCalculator
Subtract|
|if user subtract|3000|with|1000|will get|2000|
|if user subtract|4000|with|1000|will get|300|
```

Figure 5.7: FitClipse environment showing the Fixture Generation Wizard

- FitClipse Property Page: in the property page configurations for FitClipse and connections with FitNesse can be set. (See Figure 5.8)



Server Configuration

Server Connection

Host: cerveza.cpsc.ucalgary.ca

Web Port: 80

Web Root: FitClipse.ProjectS

User Name: cdeng

Password: **********

FitClipse Classpath Configuration

Class Path

C:\ExperimentPDE\WorkSpace\ExperimentProject\

C:\ExperimentPDE\WorkSpace\ExperimentProject\

Fit Resources Location

Test Locatic C:\ExperimentPDE\WorkSpace\E

Result Locatior C:\ExperimentPDE\WorkSpace\E

Figure 5.8: FitClipse environment showing the Property Page

- FitClipse Perspective: FitClipse perspective is for showing the above views and preparing the development environment in Eclipse.

5.3.2 Server Connector

FitClipse uses a server connector for dealing with requests to the server. Its main functions include: connecting to the server, getting all the acceptance tests that belong to one project from the server, saving changes to the tests on the server and persisting the test results after each test run.

The server connector is an interface defining the transaction protocol between the clients and the server. With the server connector, users can connect to different Wiki repository server. The connector hides the implementation details of different type of servers. Currently the connector is implemented for the FitNesse server. The FitNesse connector talks to the FitNesse server via Http calls.

5.3.3 Fit Test Runner

Fit Test Runner is used to run Fit acceptance tests. This part is derived from the FitNesse Test Runner with changes to the input method of the test runner.

In FitClipse, when running an acceptance test the test runner grabs the source test in the form of Html page with the tables which define the acceptance tests. The runner, then, renders the table cells with different colors based on the test results returned from the test fixture. It also generates the test result report and passes it to the connector for saving the result into the database on the server side.

5.3.4 Two Test Failure States

FitClipse splits up the test failure state in Fit or FitNesse into two: Unimplemented Failure and Regression Failure (as has been defined in Section 3.6.1). Table 5.1 shows the four test result states in FitClipse, comparing them to the three states of Fit or FitNesse.

Table 5.1: Test States in FitClipse

Test Result States	**Fit or FitNesse**	**FitClipse**
Failure (the tests fail)	Color Red	🟥 Regression Failure – failure as a result of a recent change losing previously working functionality
		▤ Unimplemented Feature – not really a failure as it might simply mean that the development team hasn't started to work on this feature
Passing (the tests pass)	Color Green	🟩 test page with green bar – no difference to Fit/FitNesse (color green)
Exception (the tests cannot be executed)	Color Yellow	🟨 test page with yellow bar – no difference to Fit/FitNesse (color yellow)

Based on the test result history saved on the server side database, FitClipse is able to distinguish two test failure states. The algorithm for distinguishing different failure states is as follows:

```
for (each test t){
    t.run();
    PersistTestResult (t.result);
    if (t.isFailing){
       getResultHistory(t);
       If (hasPassedBefore(t)){
          displayRegressionFailure();
       }else
          displayUnimplementedFeature(t);
}}}
```

5.4 FitClipse Server Side

The FitClipse server side design includes the FitNesse request responder for processing the HTTP requests from the FitClipse clients and the database for storing test result information.

5.4.1 FitNesse Request Responder

FitClipse are currently using FitNesse as the server for Wiki repository and dealing with the database transactions. It utilizes the FitNesse server's responder structure. In order to process the requests from multiple clients, additional FitNesse Server Responders are designed for these tasks. Table 5.2 lists the major responders and corresponding functions.

Table 5.2 FitClipse Responders and Their Functions

Responder Name	**Function**
FitClipse Responder	Connect to the clients, return the acceptance tests of the project.
DoesTestExist Responder	Assert whether the test with the given name exits on the server side.
SaveTestResult Responder	Save the test result into the database.
GetFitTestFromDB Responder	Return the Fit test result retrieved from the database
TestResultHistory Responder	Return the test result history for single test.
TestResultHistoryGraph Responder	Return the test result history chart for single test.
TestSuiteResultHistory Responder	Return the test result history for a test suite.
TestSuiteResultHistoryGraph Responder	Return the test result history chart for a test suite.

5.4.2 Server Side Database

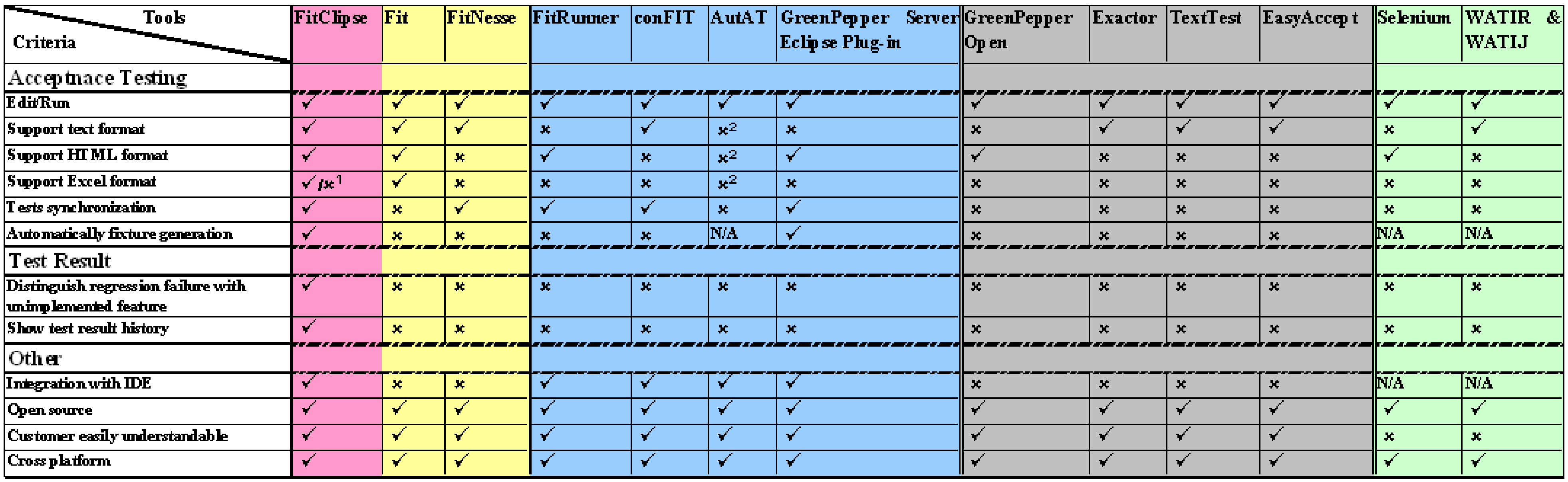
After each test or test suite run, the test result information is persisted in the database maintained on the server side. The information includes:

- Number of passing, failing, exception and ignored tests (as defined by Fit)
- The test result table in html format
- The execution start and end time

Detailed database table design is included in Appendix D:.

5.5 Comparison with Other Tools

Figure 5.9 compares FitClipse with other open source acceptance testing tools.



Tools / Criteria	FitClipse	Fit	FitNesse	FitRunner	conFIT	AutAT	GreenPepper Server Eclipse Plug-in	GreenPepper Open	Exactor	TextTest	EasyAccept	Selenium	WATIR & WATIJ
Acceptnace Testing													
Edit/Run	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Support text format	✓	✓	✓	✗	✓	✗[2]	✗	✗	✓	✓	✓	✗	✓
Support HTML format	✓	✓	✗	✓	✗	✗[2]	✓	✓	✗	✗	✗	✓	✗
Support Excel format	✓/✗[1]	✓	✗	✗	✗	✗[2]	✗	✗	✗	✗	✗	✗	✗
Tests synchronization	✓	✗	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗
Automatically fixture generation	✓	✗	✗	✗	✗	N/A	✓	✗	✗	✗	✗	N/A	N/A
Test Result													
Distinguish regression failure with unimplemented feature	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Show test result history	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Other													
Integration with IDE	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	N/A	N/A
Open source	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Customer easily understandable	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
Cross platform	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

1. FitClipse runs test using Fit test runner which means it is able to support the file format as Fit does. But currently it is working with only wiki text and html format.
2. AutAT usex an graphical based interface to define acceptance tests instead of certain file formats.

Figure 5.9: Comparison of FitClipse with other open source acceptance testing tools

In Figure 5.9, tools are separated into four groups:

1. FitClipse, Fit and FitNesse: this group is colored in pink and yellow. These tools are the most relevant to my study.
2. Eclipse plug-ins: this group is colored in blue. These tools are implemented as Eclipse plug-ins that make it easier for the developers to switch between editing acceptance tests and the application code.
3. Other acceptance testing tools: this group is colored in grey. As shown in the table, these tools provided limited support for acceptance testing.
4. Web acceptance testing tools: this group is colored in green. These tools are designed for testing the web applications. One of their differences from the other acceptance testing tools is they do not need code style fixtures for running the test scripts. The test scripts are ready to be executed by the test runner. Therefore it does not make scene to evaluate whether these tools are integrated with an IDE. In addition, by using the scripts these tools require prior IT or even programming experience to understand the test cases.

Compared to other tools, FitClipse has all the common functionalities that most of the other tools have. In addition, FitClipse provides novel support for distinguishing regression test failures with unimplemented features and showing the test result history. To my knowledge, no other tools are able to provide these features to support acceptance testing.

5.6 Summary

In this chapter, I first outlined the requirements that a tool should have for supporting acceptance testing. I summarized ten core requirements that should be achieved by an acceptance tool which integrates Fit framework. Then I presented the overall design of FitClipse which uses Client-Server structure. FitClipse framework is composed of (multiple) Eclipse plug-in front end, a FitNesse Server and a Database. FitClipse enables developers to create, modify and run acceptance tests inside an integrated development environment. It provides novel support for EATDD by identifying acceptance test regression failures and generating acceptance test result history information. In Chapter 6, FitClipse is evaluated in a pilot study for its usefulness and usability.

Chapter Six: Pilot Study

An initial pilot study of FitClipse is demonstrated in this chapter. The objective of the study is to evaluate the usefulness and usability of FitClipse as a tool for supporting EATDD to see whether there is a chance that this tool is useful or usable.

6.1 Objectives

The objectives of this pilot study are:

1. To evaluate the usefulness of FitClipse functionalities:
    a. **Identifying two different kinds of failures**: Regression Failure and Unimplemented Feature.
    b. **Keeping the test result history**: the test history chart and the test history detail table.
2. To evaluate the **usability of FitClipse**.
3. To evaluate the likely **future usage** of FitClipse as a way to support EATDD.

In order to achieve our goals, the main research questions are:

Question 1: To what extent do the participants find FitClipse functionality useful?

Question 2: To what extent do the participants find FitClipse easy to use?

Question 3: How likely are the participants going to use FitClipse in their project?

6.2 Study Methodology

Ethics approval was obtained from the University of Calgary (Appendix A.2) before conducting the study. The participants completed and returned the informed consent forms included in (Appendix C.1).

The study was conducted with 7 participants who were taking a senior computer science course in the University of Calgary. The participants are from 2 development teams. Three participants are from an undergraduate student team with a total of 5 team members and the other four participants are from graduate student (Masters) team with a total of 6 team members. They worked on two different course projects which lasted for 4 months each.

The development teams followed Extreme Programming and used EATDD in their development process. There were four iterations in the two projects. The first iteration was used for the participants to learn the techniques that were going to be used in their projects, such as Acceptance Testing, FitNesse and so forth. EATDD using FitNesse was introduced (through a 30 minutes tutorial) in the second iteration. EATDD with FitClipse support was introduced (through a 30 minutes tutorial) in the third iteration. The participants used FitClipse for EATDD during both the third and fourth iteration. All through the development process, the researcher was with the team to provide support and guidance for EATDD and FitClipse. At the end of the study, an interview and a follow up questionnaire were conducted to gather information.

No customer is involved in this pilot study. In fact, in the real world the customer should also participate in the process of acceptance testing. It is expected that FitClipse can provide full support for both the customer and the developers. However, currently

FitClipse only provides support for the developers. This is why the pilot study only involved the role of developers.

In the study, participants are from two course projects with similar scale. Both of the projects' size is small, because these are course projects and the participants are only working part time on the projects. So the size of the projects does not impact on the result of the study.

6.3 Participants' Related Experience

The related experience of participants is shown in Figure 6.1.

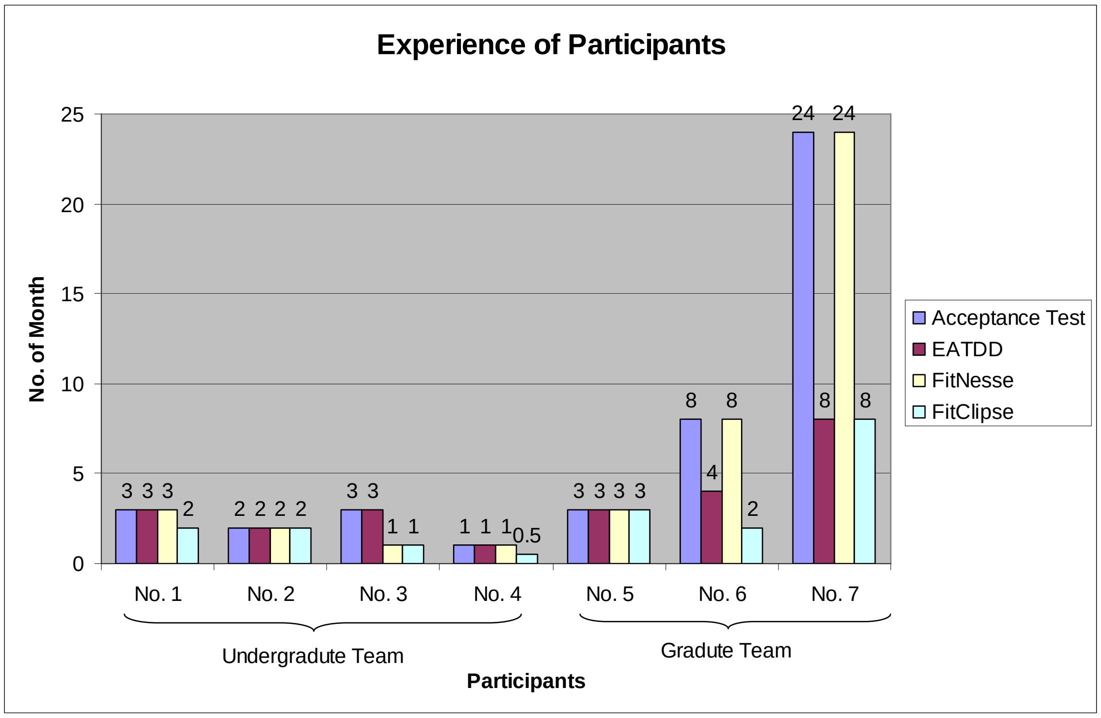


Figure 6.1: Related experience of participants for EATDD

The participants are asked about their experience with acceptance tests, EATDD, FitNesse and FitClipse in number of months at the end of the study. Most participants in both the undergraduate team and graduate team show similar experience from 1 to 3 months. However, in the graduate team two participants have much more experience than other participants. One of the graduate participants has 8 months experience of acceptance test and FitNesse. The other participant has 24 months' acceptance test and FitNesse experience and 8 months' EATDD and FitClipse experience. However, I believe that the experience differences do not have impact on the study results. Because for the participants who are either in the third year of Undergraduate level computer science major or in the first year of Master level computer science major, the learning curve of acceptance testing is quite short. Therefore, when the participants started to use FitClipse they had an appropriate knowledge to evaluate the usefulness and usability of our tool.

The numbers in the chart shows that most of the participants (5/7, 71 %) started to learn acceptance testing and use FitNesse during this study and even more (6/7, 86%) participants started to practice EATDD and use FitClipse during this study.

6.4 Number of Acceptance Tests

The number of acceptance tests is critical for our study in that it can show how much the participants are practicing EATDD. The more acceptance tests the participants created or modified inside the FitClipse environment, the more knowledge they will get from using it.

By the end of the study the undergraduate team had 5 acceptance tests with 8 assertions. The graduate team created 7 acceptance tests which had 43 assertions. Figure 6.2 shows the number of acceptance tests that each participant has created during the study. The sum of individual's number of acceptance tests does not match the total number of the project's tests because they deleted some tests at the end due to customer requirement changes and time pressure.

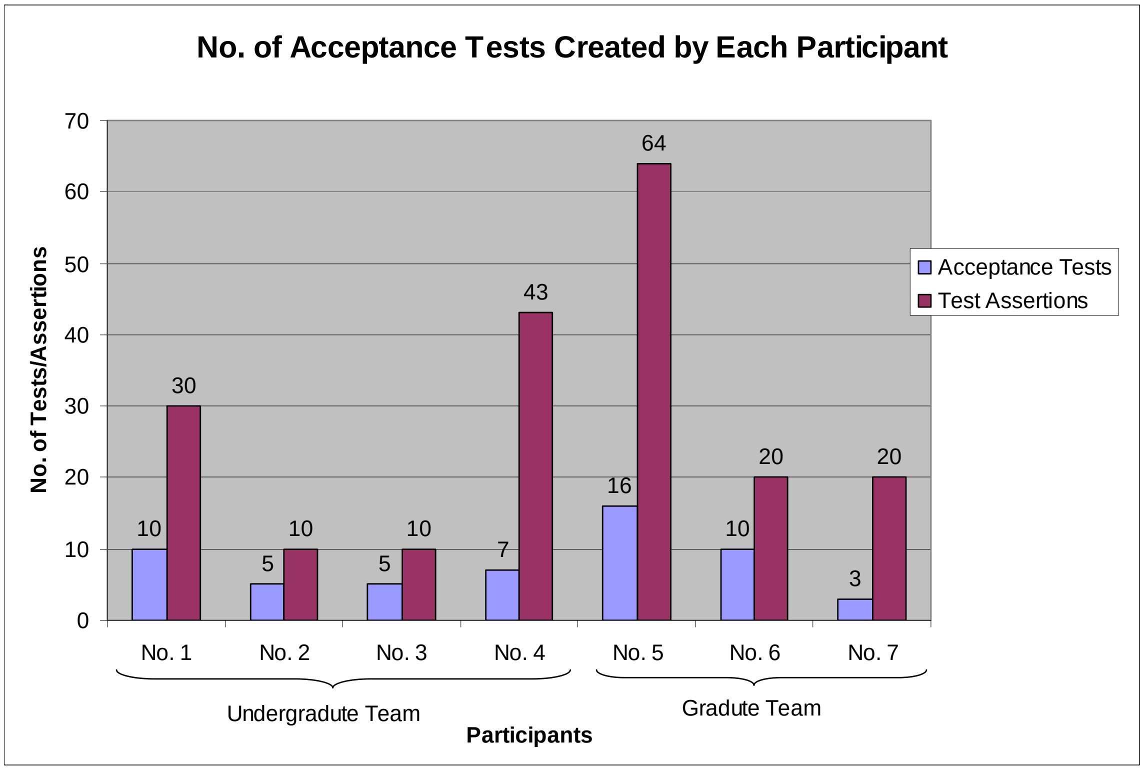


Figure 6.2: No. of acceptance tests created by each participant in the study

The number of assertions is also examined in this study because the participants sometimes created one test that contains several assertions, which made the number of tests small, but in fact rich in functions. Therefore the number of assertions is also included in order to avoid the misleading of big tests with small number of tests. The data was gathered by asking the participants directly in the interview.

In the two projects, the number of acceptance tests is limited. The participants reported two major reasons. First, the two projects are all mainly user interface based applications. Writing acceptance tests for testing the UI is challenging. There is a lack of resources for them to learn how to do acceptance testing for UI. Instead, all the tests are testing the backend business logic. The other reason is the project time scale. Those projects are course projects and the students were developing part time. Under the time pressure their major focus was on implementing the functionalities resulting in insufficient test coverage. Therefore, some of the functionalities lack acceptance tests.

As a result, 71 % (5/7) participants have created more than 20 test assertions for their projects. The other 2 participants have maintained 10 test assertions. Considering the fact that the projects are course projects in small scale, the number of acceptance tests should cover the main functionalities of the projects and should be enough for the participants to understand and practice EATDD.

6.5 Study Results

The study was conducted to determine the usefulness of FitClipse as a tool support of EATDD and the usability of the tool. In order to achieve the goal of the study, four major questions are provided and analyzed.

6.5.1 Usefulness of FitClipse

Question 1: To what extent do the participants find FitClipse functionality useful?

I break down this question into two aspects:

- **Identifying two different kinds of failures**: Regression Failure and Unimplemented Feature.

- **Keeping test result history**: the test history chart and the test history detail table.

6.5.1.1 Usefulness of identifying two different kinds of test failures

The frequency of the participants' meeting the test failure states will affect the usefulness of the function of separating the results. The more often the users cause the regression failure, the more useful it may be to distinguish it from unimplemented failure. For unimplemented features, it is the definition of TDD that requires the participants to make the acceptance tests fail when they are first created. Therefore, the participants all have seen the tests failing as unimplemented features before they implemented the functionalities. However the participants could only see the regression failure when they broke the system and the frequency of seeing the regression failure depended on their individual experience. Table 6.1 shows the frequency of meeting the regression failures reported by the participants.

Table 6.1 Frequency of Regression Failures

Participant #	**Frequency**
No. 1	More than half time of making changes to the system.[11]
No. 2	Every time adding new functionalities.
No. 3	Once per day, when did the daily build.
No. 4	Twice a week.
No. 5	1/4 time of making changes to the system.
No. 6	1/4 time of making changes to the system.
No. 7	N/A

[11] The participants ran all the acceptance tests after they made changes to the system. Making changes to the system also includes adding new functionalities to the existing system.

All the participants except participant No. 7 have met the regression failures quite often in the projects. This means if identifying the difference in acceptance test failure states is useful, the regression failure's frequent appearance rate will amplify its usefulness.

Figure 6.3 shows how useful the participants think the function of identifying two different test result failures is.

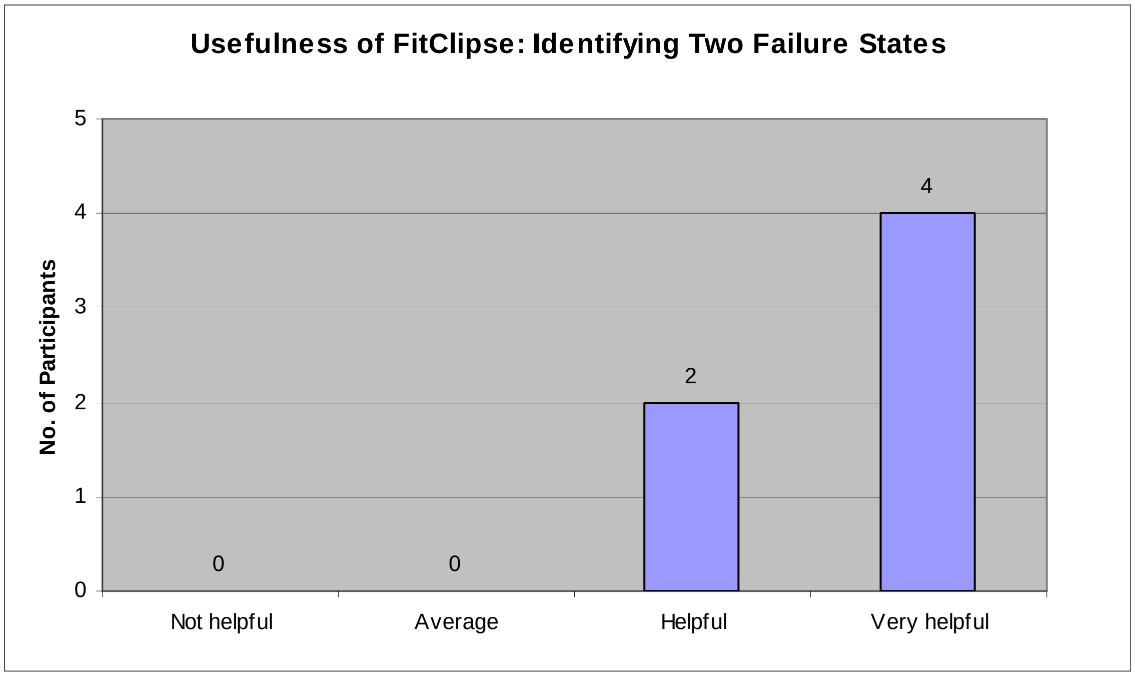


Figure 6.3: Evaluation of the helpfulness of FitClipse for identifying two acceptance test failure states

The data in this figure does not include participant No. 7 because he reported that he never saw regression failures in his project. The figure shows 67% (4/6) of the participants think it is very helpful to identify two different failures, while the other two participants think this function to be helpful. No participant who has used this functionality reported this functionality to be not helpful. One of the participants stated that

“*Identifying regression failure is helpful to me as it tells me at once when I broke something. You can just go in to fix it right away rather than comparing two test results. Further, combined with the test result history it is easier to see what actually got broken.*”

In this section, participants were also asked about how they identified unimplemented features and regression failures if they were not provided with other support. All of them reported that they were only using their memory for reciting the tests that had passed before. One of the participants stated that

“*I use my own memory to memorize which test has passed before instead of taking notes. It works for projects with small number tests. However I think for bigger number tests, my memory will not work.*”

6.5.1.2 Usefulness of keeping test result history

Figure 6.4 shows the participants’ responses in term of how they think the functionality of showing test result history is helpful. 71% (5/7) of the participants think this function to be helpful and the other two of the participants think this function to be very helpful. No participant reported this functionality to be not helpful. As one of the participants reported

“*The result history chart inside FitClipse helps us to see the progress of our project. At first we saw all the tests were failing. Later more and more tests are passing which means more and more functions are completed.*”

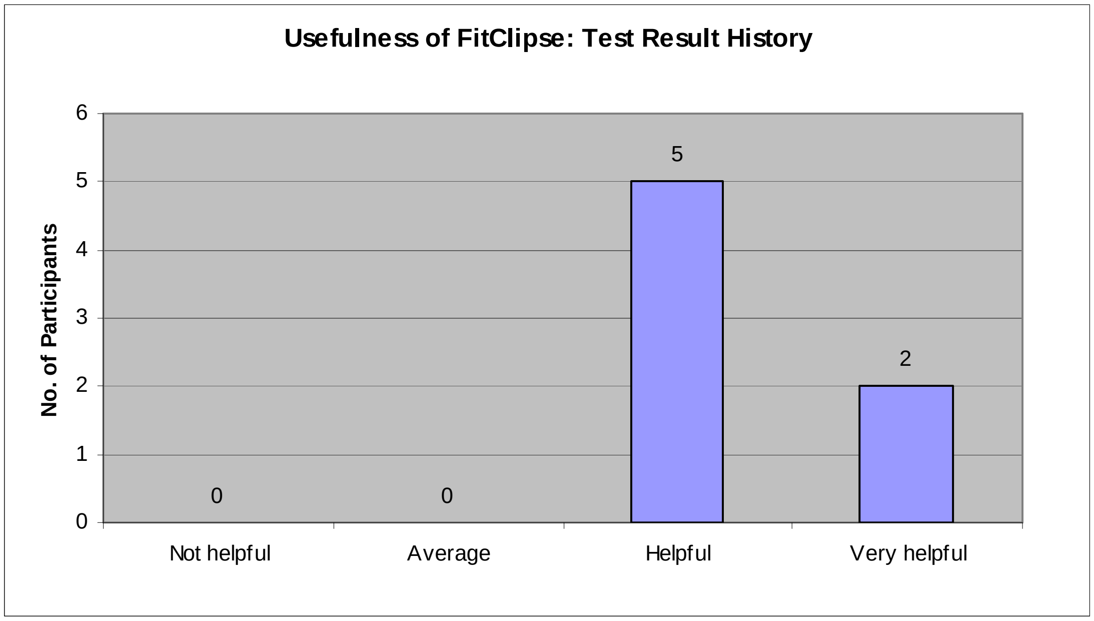


Figure 6.4: Evaluation of the helpfulness of FitClipse for keeping acceptance test result history

In the study, one of the participants has changed a test by accident and he could not remember when he changed it. It means he could not find the right test from the test repository because he did not know which one was the version of the test that had been working. However with the help of the researcher he went to the result history table which showed the test results for each test run. Eventually he found the working version easily form the result history because from the history it is convenient to see which assertion is passing and which is failing in each test.

6.5.2 FitClipse's Ease of Use

Question 2: To what extent do the participants find FitClipse easy to use?

Figure 6.5 shows the participants' answers according to FitClipse's ease of use.

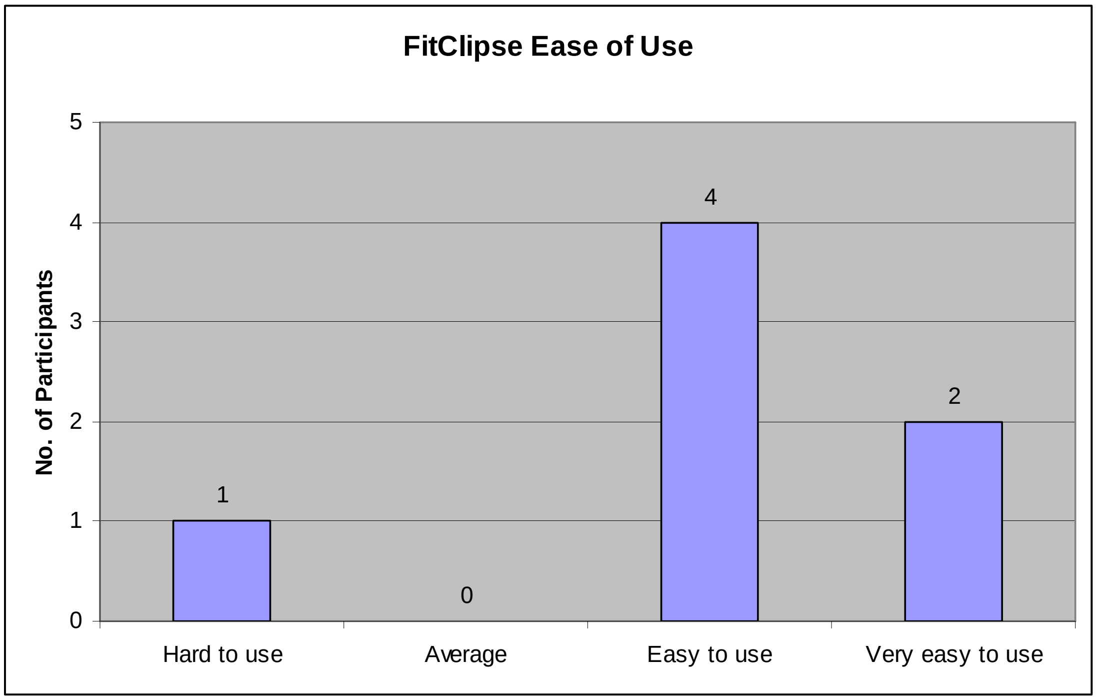


Figure 6.5: Evaluation of FitClipse ease of use

The majority (57% 4/7) of participants find FitClipse easier to use than average tool. Another 2 (29%) of the participants think FitClipse to be very easy to use. One of the participants reported that

"Compared to other tools (Fit/FitNesse), FitClipse is easy to configure and use for running acceptance tests. Its function of automatically generating fixture code makes it much easier to write and run acceptance tests."

Only one participant rates FitClipse to be hard to use. He commented that

"FitClipse separate acceptance tests from the projects into two different views. This makes me hard to understand the tests and uncomfortable when running the tests."

He also suggests that

"In order to improve FitClipse, there should be a way to integrate acceptance tests into the project view. Also the configuration should be more hidden to the developers."

However he also mentioned that

"FitClipse is easy to use in the scene that it avoids the users to switch between several IDEs, which is annoying when the developers are focusing on solving a problem."

6.5.3 Willingness of Using FitClipse

Question 3: How likely are the participants going to use FitClipse in their project?

Figure 6.6 shows the likely usage of FitClipse in the future.

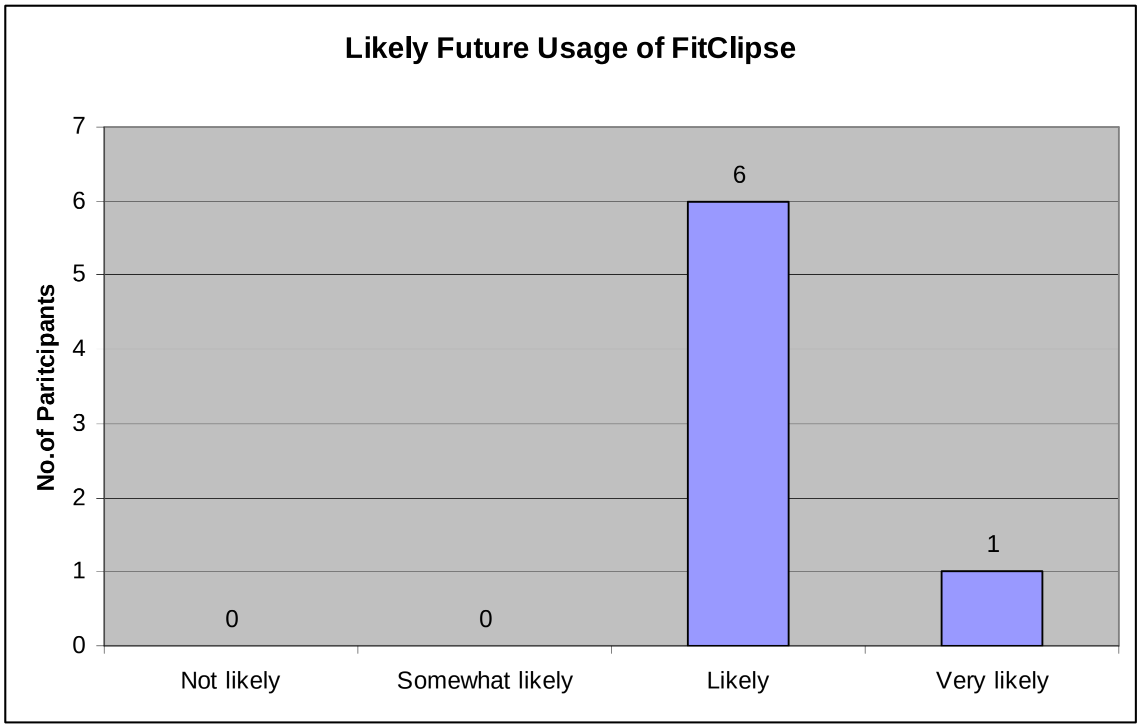


Figure 6.6: Evaluation of FitClipse for likely future usage

All the participants reported the likeness of using FitClipse in the future. One of them reported very likely to use FitClipse in their future projects. One of the participants said

“*We (the participant with other team members) are planning doing a project in which I want to try Extreme Programming. I would like to use FitClipse in the project because it is easy to use and provides useful functionalities.*”

Another participant said

“*I am very likely to use FitClipse in the future because it is easy to use and very helpful to find out problems.*”

6.6 Interpretation of the Participants’ Feedback

The pilot study results reported in section 6.5 indicates that FitClipse as a whole is a useful tool support for EATDD and its usability is considered as easy to use. The positive responses also indicate the participants’ willingness to use FitClipse in the future for projects that follow EATDD. However, for the tool to be used in industry, additional improvements have to be made. The following are the comments from the participants regarding the improvement for FitClipse.

- *Customer perspective*: Now FitClipse is a more like an integrated development tool for the developers to use. However the definition of acceptance tests requires the customers to participate in writing and running the acceptance tests. The customers may not be professionals in the IT field thus may not be familiar with any kind of development environment. Therefore a customer perspective should be provided for the customers to use with the development details totally hidden from them.

- *Mapping of failing assertions to code details*: FitClipse identifies two test failure states for the developer by emphasizing the regression failure with a special flag. Seeing this flag, the developers can understand which test is failing and view the test result to see which assertion is failing. However, it needs the developers to go into the fixture codes to find out which part of source code is being called and then go into the real application code to locate the problems. This is time consuming and may reduce the advantage of using acceptance tests. Therefore, mapping from the failing assertions to corresponding source code positions should be built in to assist the developers to locate the mal-functions.
- *Providing more helpful test result history reports*: FitClipse provides acceptance test result history chart and table to enable project progress monitoring. However, this information report is in the state of prototype. More helpful information on the project level and even statistical analysis, which is based on the information, should also be available to help manage the software project.
- *Organizing acceptance tests into project packages*: Acceptance tests are part of the projects, which has the function of testing the project functionalities and asserting the test results. Therefore, like unit tests, the acceptance tests should

be integrated and organized into the project packages in a way they can be edited and run inside the project.

- *Hiding detailed configurations from the user*: Running acceptance tests needs some pre-configuration which is always time consuming and confusing to developers. For instance, using Fit or FitNesse, a small mistake in the classpath may make all existing acceptance tests throw exceptions even though some of them should be passing. Therefore, support should be provided for making configuration automatically for common usage and the configuration should be hidden from the users.
- *Compatibility with Linux/Unix and Mac systems*: Currently FitClipse has compatible issues with systems other than Windows. As FitClipse is an Eclipse plug-in which is based on Java platform, this should not happen. So future work should also include making FitClipse work on all platforms.

6.7 Validity of the Pilot Study

The purpose of this pilot study shows whether there is a chance that FitClipse is useful and usable for supporting EATDD in order to know whether it worth to do another formal empirical study. There are two major limitations in this study. One limitation is that the participants are all from the academic area instead of from industry and the study is conducted in academic environment. The participants have limited experience and time for development (they were working part-time on the projects). These facts may lead to differences in the effects when the tool is used in a real industry environment.

The other limitation is the scope of the study. There are only 7 participants providing data which threatens the statistical results for the study. In addition, the projects only lasted for 4 months which was a relatively short time. This fact may prevent the participants from producing more acceptance tests and evaluating the long-term effects of the tool.

6.8 Summary

In this chapter, an initial pilot study of FitClipse is introduced. In the evaluation I seek to find out the usefulness of FitClipse, its ease of use and the likeness of future usage. To this end, I conducted experiment with 7 participants from the University of Calgary senior computer science class. This study lasts for 4 months and the results are positive. It is shown in the study that FitClipse is a useful tool support for EATDD and it is easy to use FitClipse in development environment. In addition, the participants all show the willingness of using FitClipse in the future, which again proves its usefulness and good usability. Even though this study has certain limitations, the findings of the study are encouraging to suggest that more research should be taken on this direction to provide more advanced support for EATDD.

Chapter Seven: Conclusion

The conclusion is summarized from the research contributions in the area of acceptance testing. First I present the research problems which are the motivation of FitClipse. Then, I describe my research contributions by showing how the research problems are solved. At last, I suggest aspects of potential future work.

7.1 Research Motivation

In Chapter 1 I presented three research questions on tool support for EATDD in Agile Methods:

1. **It is unknown how Executable Acceptance Test Driven Development is conducted in industry, especially the time frame of Acceptance Testing.** In Agile Methods, acceptance tests are used for driving the process of development and help the customers and the development team to communicate in terms of software requirements. Prior work has been focused on defining acceptance tests so they can be understood by both customers and the developers. However, there is limited research on how acceptance test driven development is used in industry.
2. **Based on the finds of Problem 1, novel tool support should be built for doing EATDD in Agile environment.** After finding the limitations of existing support for EATDD process, useful tool for EATDD in Agile environment needs to be provided.
3. **Once such a tool support is built, an evaluation is needed to determine the usefulness and usability of the tool in Agile environment.** After the tool is built, it is still unknown about the usefulness and usability of the tool.

7.2 Thesis Contributions

1. **For this thesis, a survey is conducted for gathering information on the state of EATDD being used in industry and basing on the survey, support for EATDD is investigated.** The result of the survey reveals the current trend and pattern of using EATDD for Agile development testing in industry. The survey highlights that in EATDD the time frame between creating an acceptance test and making it pass successfully for the first time is much longer than the time frame of unit test in UTDD. This makes it impossible for all the acceptance tests to be passing all the time in EATDD and as a result the developers will see failing tests. These failing tests have different meanings: Unimplemented Failure which means the function of the test has not been worked on and Regression Failure which means the test of the function was passing before but is failing now. When there are more tests and with longer time frame, people will have difficulties to remember the previous test results, thus can not distinguish the two failure states. Therefore it is necessary to provide support for identifying different acceptance test failure states in EATDD.

2. **Based on the results from the survey, a testing tool, FitClipse, is developed.** FitClipse makes use of Fit/FitNesse and Eclipse plug-in, which makes it possible to create acceptance tests in the integrated development environment. It provides support for identifying two test failure states in EATDD and maintains the test result history information to assist project management. Scenarios of using FitClipse to support EATDD are provided.

3. **An exploratory study was conducted with academic participants to evaluate the viability of FitClipse based on three factors.** The first factor evaluated the usefulness. For evaluating the usefulness of FitClipse, the study is focused on FitClipse's functionalities for identifying two test failure states and maintaining the test result history. The second factor assessed the usability of FitClipse. The third factor identified the likely future usage of FitClipse as a tool support for EATDD. The results demonstrated that FitClipse, as a whole, is a useful tool for acceptance testing. Positive responses also indicated FitClipse to be easy to use, although some aspects still needed to be improved. In addition, all the participants indicated a likely use of FitClipse in the future.

7.3 Future Work

The work presented in this thesis is a preliminary step in constructing effective tool for supporting EATDD in Agile software development environment. There is still a lot of room in this research area for future work.

FitClipse is a research prototype tool with system instabilities. It will be beneficial to improve the implementation of the tool in terms of usability in order to better assist EATDD in the industry environment.

As a tool support for Agile Methodology, it will be helpful to integrate this work with other practices in Agile. For instance, acceptance tests can be used in conjunction with story card management to provide more meaningful reports for the customers.

Currently FitClipse works inside Eclipse as a plug-in to enable the developers to create and run acceptance tests inside the integrated development environment. However, support from the customer perspective is missing. An editor with support for editing acceptance tests and a view for better showing the test result report, as the project progress report, will be of great help to the customers.

Lastly, the pilot study has limitations in two aspects: academic based participants and the scope of the study. Participants from the industry working full time on a longer time span project will be more helpful to provide natural user feedback for the long-term effects of the proposed tool.

References

[Acceptance Test 2006] Various Authors, http://c2.com/cgi/wiki?AcceptanceTest.

[Agile Manifesto 2007] Agile Manifesto: http://agilemanifesto.org/, last accessed: April 19, 2007.

[Andersson 2003] J. Andersson, G. Bache, and P. Sutton, *XP with Acceptance-Test Driven Development: A Rewrite Project for a Resource optimization System*, XP 2003, LNCS 2675, pp. 180–188, 2003.

[Andersson 2004] J. Andersson and G. BacheThe, *Video Store Revisited Yet Again: Adventures in GUI Acceptance Testing*, XP 2004, LNCS 3092, p. 1–10, 2004.

[Beck 1999] K. Beck. (1999), *Extreme Programming Explained: Embrace Change*, E1, Addison Wesley, 224.

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

[Beck 2004] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, Second Edition, Addison Wesley 2004.

[Binder 2000] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools (chapter 3)*, Pearson Education, 2000.

[Boehm 1988] B. W. Boehm and P. N. Papaccio, *Understanding and Controlling Software Costs*, IEEE Trans. Software Eng., vol. 14, pp. 1462–1477, Oct. 1988.

[CAMUG 2007] Calgary Agile Method User Group home page: http://www.agilenetwork.ca/camug/, 2007.

[Cao 2004] L. Cao, K. Mohan, P. Xu and B. Ramesh, *How Extreme does Extreme Programming Have to be? Adapting XP Practices to Large-scale Projects*, Proceedings of the 37th Hawaii International Conference on System Sciences, 2004

[Chaos Report] Chaos Report, the Standish Group, West Yarmouth, MA, 1995, 1997, 1999, 2001, 2003.

[Cockburn 2004] A. Cockburn, *Crystal Clear : A Human-Powered Methodology for Small Teams*, Alistair Cockburn, October 2004, Addison-Wesley Professional, ISBN 0-201-69947-8.

[Cohn 2004] M. Cohn, *User Stories Applied for Agile Software Development*, Person Education, Inc., 2004.

[Cohn 2005] M. Cohn, *Do-It-Yourself: A How-to Guide for Fixing a Failing Project*, Better Software, October 2005.

[conFIT 2007] conFIT: *A FitNesse for Eclipse Plugin website*, http://www.bandxi.com/fitnesse, 2007.

[Crispin 2001 A] L. Crispin and T. House, (2001) *Testing in the Fast Lane: Automating Acceptance Testing in an Extreme Programming Environment*, XP Universe Conference.

[Crispin 2001 B] L. Crispin, T. House, and C. Wade, (2001) *The need for speed: automating acceptance testing in an extreme programming environment*, In Second International Conference on eXtreme Programming and Flexible Processes in Software Engineering, pages 96–104.

[Cunningham 2007] W. Cunningham, *Fit: Framework for Integrated Test*, http://fit.c2.com, 2007.

[Davis 2004] F. D. Davis and V. Venkatesh, *Toward Preprototype User Acceptance Testing of New Information Systems: Implications for Software Project Management*, IEEE Transactions on Engineering Management, Vol. 51, NO. 1, 2004.

[Don Wells 2006] Extreme Programming introduction website: http://www.extremeprogramming.org/, Don Wells, 2006.

[Erickson 2003] C. Erickson1, R. Palmer, D. Crosby, *Make Haste, Not Waste: Automated System Testing*, XP/Agile Universe 2003, LNCS 2753, pp. 120–128, 2003.

[Exactor 2007] Exactor homepage: http://exactor.sourceforge.net/index.html, 2007

[FitClipse 2007] FitClipse homepage in EBE website: http://ebe.cpsc.ucalgary.ca/ebe/Wiki.jsp?page=.FitClipse, 2007.

[FitLibrary 2007] FitLibrary project website: http://sourceforge.net/projects/fitlibrary.

[FitNesse 2007] FitNesse FrontPage: http://fitnesse.org/, 2007.

[FitRunner 2007] FitRunner*: an Eclipse plug-in for Fit* website, http://fitrunner.sourceforge.net, 2007.

[Fowler 2006]. M. Fowler, "Specification by Example." www.martinfowler.com/bliki/SpecificationByExample.html, 16 June 2006.

[Gandhi 2005] P. Gandhi, N. C. Haugen, M. Hill and R. Watt, *Creating a Living Specification Using FIT Document*, Proceedings of the Agile Development Conference (ADC'05).

[Geras 2004] A. Geras, M. Smith, and J. Miller. *A Prototype Empirical Evaluation of Test Driven De-Velopment*, 10th International Software Metrics Symposium. 2004. Chicago: IEEE Computer Society.

[Geras 2005] A. Geras, J. Miller, M. Smith, and J. Love, *A Survey of Test Notations and Tools for Customer Testing*, XP 2005, LNCS 3556, pp. 109–117, 2005.

[GreenPepper 2007] GreenPepper software website: http://www.greenpeppersoftware.com/en/products/, 2007.

[Highsmith 2000] J.A. Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, New York: Dorset House, 2000, ISBN 0-932633-40-4.

[Holmes 2006] A. Holmes and M. Kellogg, *Automating functional tests using Selenium*, Agile Conference, 2006, 23-28 July 2006 Page(s):6 pp, Digital Object Identifier 10.1109/AGILE.2006.19.

[Hunt 2006] J. Hunt, *Agile Software Constructionj*, Springer – Verlag London, P19-30, 2006.

[IEEE 1990] Institute of Electrical and Electronics Engineers (IEEE), *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, New York, NY: 1990.

[IEEE 1996] Institute of Electrical and Electronics Engineers (IEEE), *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, IEEE, January 1996.

[Jeffries 2001] R. E. Jeffries, *What is Extreme Programming?*, XProgramming.com, www.XProgramming.com/xpmag/whatisXP.htm.

[Jeffries 2003] R.E. Jeffries, *Test-Driven Development – a Practical Guide*, Pearson Education, Inc, 2003.

[Joshua 2005] J. Joshua, *Industrial XP: Making XP Work in Large Organizations*, Cutter Consortium Agile Project Management Executive Report, Vol. 6, No. 2, February 2005.

[Kaner 2002] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learnt in Software Testing*, Wiley, July 2002.

[Kaner 2003] C. Kaner, *An Introduction to Scenario Testing*, June 2003, www.kaner.com/pdfs/ScenarioIntroVer4.pdf.

[Kitiyakara 2002] N. Kitiyakara, *Acceptance Testing HTML*, XP/Agile Universe 2002, LNCS 2418, pp. 112–121, 2002.

[Marick 2002] B. Marick, *Report from XP/Agile Universe 2002 Conference*, www.pettichord.com/XP_Agile_Universe_trip_report.txt, 14 August 2002.

[Martin 2005] R. C. Martin, *The test bus imperative: Architectures that support automated acceptance Testing*, IEEE Software, 22(4):65–67, September-October 2005.

[Mase 2007] Mase home page at EBE website of University of Calgary, http://ebe.cpsc.ucalgary.ca/ebe/Mase, 2007.

[Maurer 2002] F. Maurer. S. Martel, *Process support for distributed extreme programming teams(info)*, ICSE 2002 Workshop on Global Software Development, http://www.cis.ohio-state.edu/~nsridhar/ICSE02/GSD/, 2002.

[Maurer 2006] F. Maurer, G. Melnik, *Driving Software Development with Executable Acceptance Tests*, Executive Report on Agile Project Management, Vol. 7, No. 11, Cutter Consortium, November 2006.

[Melnik 2004] G. Melnik, K. Read, and F. Maurer, *Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective*, XP/Agile Universe 2004, LNCS 3134, pp. 60–72, 2004.

[Melnik 2005]G. Melnik, F. Maurer, *The Practice of Specifying Requirements Using Executable Acceptance Tests in Computer Science Courses*, OPSLA'05, October 16–20, 2005, San Diego, California, USA.

[Melnik 2006] Grigori Melnik, Frank Maurer and Mike Chiasson, *Executable Acceptance Tests for Communicating Business Requirements: Customer Perspective*, Proceedings of AGILE 2006 Conference (AGILE'06).

[Miller 2001] R. W. Miller and C. T. Collins, *Acceptance Testing*, Proceedings of the XP Universe, July 2001.

[Mugridge 2003] R. Mugridge and E. Tempero, *Retrofitting an Acceptance Test Framework for Clarity*, Proceedings of the Agile Development Conference (ADC'03).

[Mugridge 2005 A] R. Mugridge and W. Cunningham, *Agile Test Composition*, XP 2005, LNCS 3556, pp. 137–144, 2005.

[Mugridge 2005 B] R. Mugridge and W. Cunningham, *Fit for Developing Software: Framework for Integrated Tests*, Prentice Hall, 2005.

[Palmer 2002] S.R. Palmer and J.M. Felsing , *A Practical Guide to Feature-Driven Development*, Prentice Hall, 2002 (ISBN 0-13-067615-2).

[Pancur 2003] M. Pancur, M. Ciglaric, M. Trampus and T. Vidmar, *Towards Empirical Evaluation of Test-Driven Development in a University Environment*, in EUROCON 2003 IEEE Press.

[Ralph 2001] Y., Ralph. *Effective Requirements Practices*, Addison - Wesley Professional, March 2001.

[Read 2005 A] K. Read, G. Melnik, and F. Maurer, *Examining Usage Patterns of the FIT Acceptance Testing Framework*, XP 2005, LNCS 3556, pp. 127.136, 2005.

[Read 2005 B] K. Read, G. Melnik, F. Maurer, *Student Experiences with Executable Acceptance Testing*, Proceedings of the Agile Development Conference (ADC'05).

[Rogers 2004] R. O. Rogers, *Acceptance Testing vs. Unit Testing: A DeveloSper's Perspective*, XP/Agile Universe 2004, LNCS 3134, pp. 22–31, 2004.

[Sauve 2006] J. P. Sauve and etc, *EasyAccept: A Tool to Easily Create, Run and Drive Development with Automated Acceptance Tests*, AST'06, May 23, 2006.

[Schwaber, 2001] K. Schwaber., M. Beedle, *Agile "Software Development with Scrum"*, Prentice Hall, 2001.

[Schwarz 2005] Christian Schwarz, Stein Kåre Skytteren, and Trond Marius Øvstetun, *AutAT – An Eclipse Plug-in for Automatic Acceptance Testing of Web applications*, OOPSLA'05, October 16–20, 2005, San Diego, California, USA. ACM 1-59593-193-7/05/0010. (See also: http://boss.bekk.no/autat/).

[Selenium 2007] Selenium homepage on OpenQA website: http://www.openqa.org/selenium/, 2006.

[Selenium Remote Control 2006] Selenium Remote Control homepage on OpenQA website: http://www.openqa.org/selenium-rc/, 2006.

[Stapleton 1997] J. Stapleton, *Dynamic System Development Method – the Method in Practice*, Addison Wesley, 1997.

[Steinberg 2003] D. H. Steinberg, *Using Instructor Written Acceptance Tests Using the Fit Framework*, XP 2003, LNCS 2675, pp. 378–385, 2003.

[Talby 2005] D. Talby, O. Nakar, N. Shmueli, E. Margolin and A. Keren, *A process-complete automatic acceptance testing framework*, Software - Science, Technology and Engineering, 2005. Proceedings. IEEE International Conference on 22-23 Feb. 2005 Page(s):129 – 138, Digital Object Identifier 10.1109/SWSTE.2005.2.

[TextTest 2007] TextTest: Verifying Application Behaviour with TextTest: http://texttest.carmen.se/TextTest/index.html, 2007.

[Tracy 2004] Tracy Reppert, *Don't Just Break Software, Make Software, Better Software Magazine*, July/August 2004, available on line: http://industriallogic.com/papers/storytest.pdf.

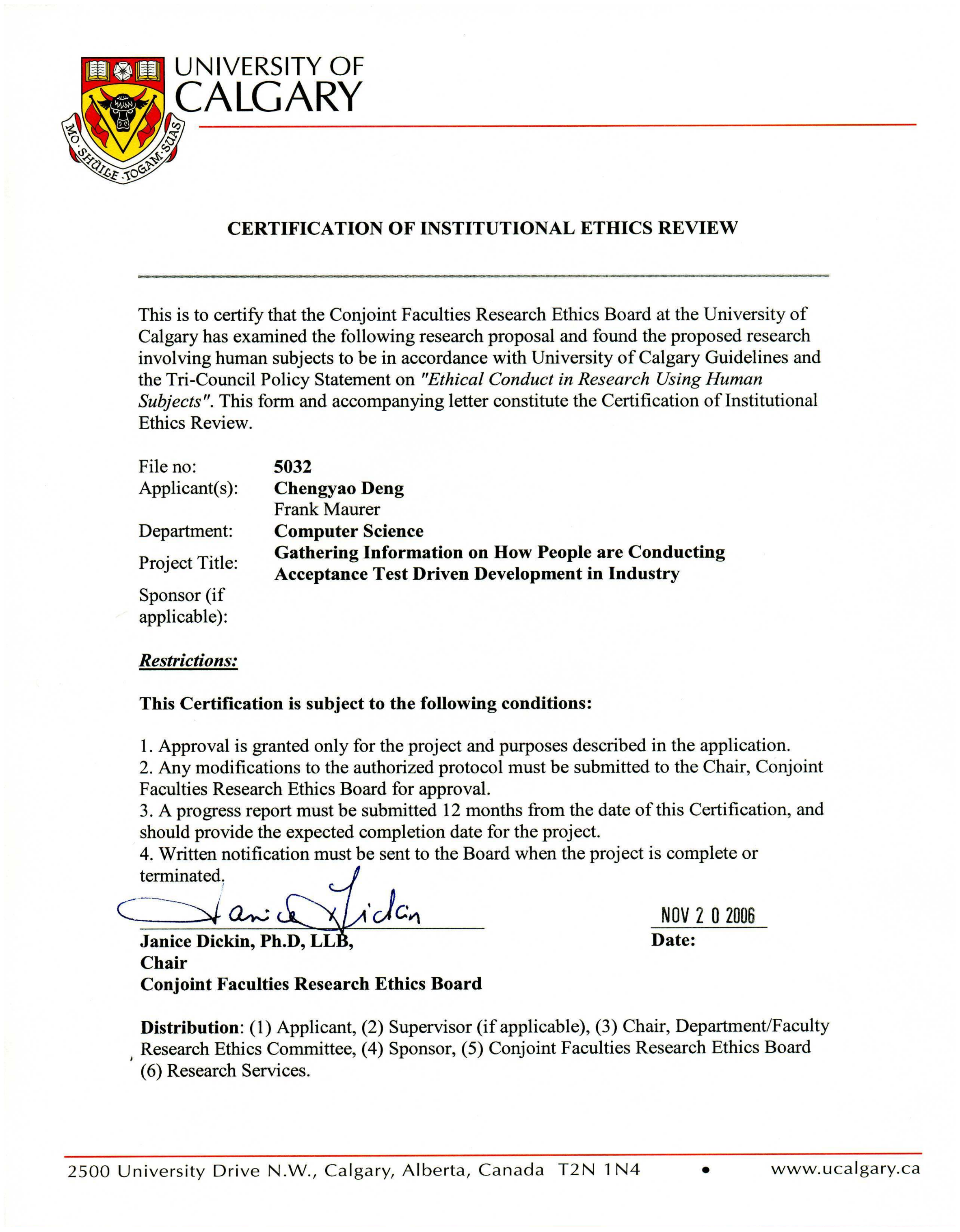
[USDD] US Department of Defense, *Military Standard Defense System Software Development, DODSTD- 2167, Section 5.3.3.* Formal Qualification Testing Distribution Statement A, www2.umassd.edu/SWPI/DOD/MIL-STD-2167A/DOD2167A.html.

[WATIJ 2007] WATIJ: Web Application Testing in Ruby website: http://watij.com/, 2007.

[WATIR 2007] WATIR: Web Application Testing in Ruby website: http://wtr.rubyforge.org/, 2007.

APPENDIX A: ETHICS APPROVAL

A.1. Ethics Approval for the Web Survey



UNIVERSITY OF CALGARY

CERTIFICATION OF INSTITUTIONAL ETHICS REVIEW

This is to certify that the Conjoint Faculties Research Ethics Board at the University of Calgary has examined the following research proposal and found the proposed research involving human subjects to be in accordance with University of Calgary Guidelines and the Tri-Council Policy Statement on *"Ethical Conduct in Research Using Human Subjects"*. This form and accompanying letter constitute the Certification of Institutional Ethics Review.

File no:	**5032**
Applicant(s):	**Chengyao Deng** Frank Maurer
Department:	**Computer Science**
Project Title:	**Gathering Information on How People are Conducting Acceptance Test Driven Development in Industry**
Sponsor (if applicable):	

Restrictions:

This Certification is subject to the following conditions:

1. Approval is granted only for the project and purposes described in the application.
2. Any modifications to the authorized protocol must be submitted to the Chair, Conjoint Faculties Research Ethics Board for approval.
3. A progress report must be submitted 12 months from the date of this Certification, and should provide the expected completion date for the project.
4. Written notification must be sent to the Board when the project is complete or terminated.

Janice Dickin

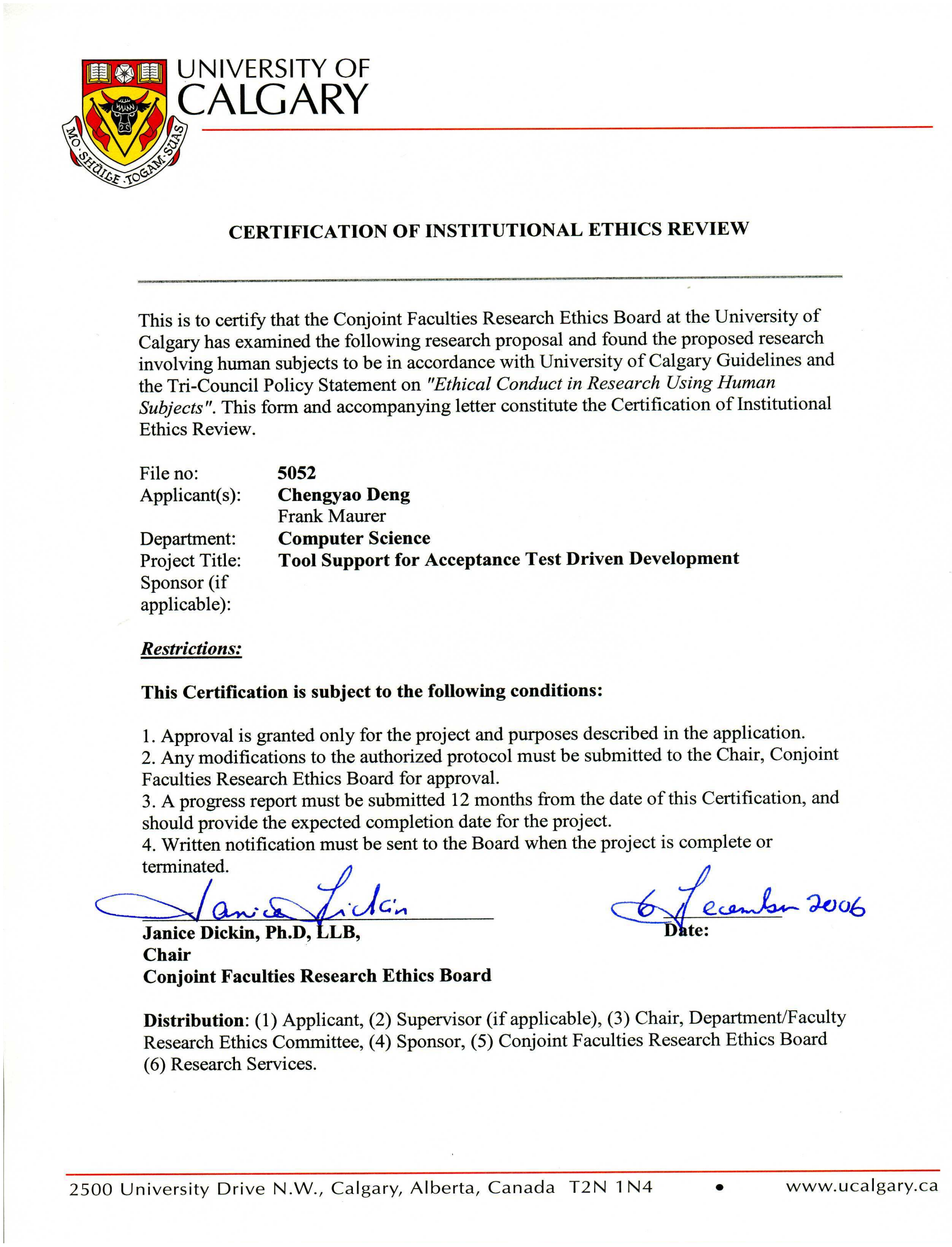
Janice Dickin, Ph.D, LLB,
Chair
Conjoint Faculties Research Ethics Board

NOV 2 0 2006
Date:

Distribution: (1) Applicant, (2) Supervisor (if applicable), (3) Chair, Department/Faculty Research Ethics Committee, (4) Sponsor, (5) Conjoint Faculties Research Ethics Board (6) Research Services.

2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4 • www.ucalgary.ca

A.2. Ethics Approval for the Pilot Study



UNIVERSITY OF CALGARY

CERTIFICATION OF INSTITUTIONAL ETHICS REVIEW

This is to certify that the Conjoint Faculties Research Ethics Board at the University of Calgary has examined the following research proposal and found the proposed research involving human subjects to be in accordance with University of Calgary Guidelines and the Tri-Council Policy Statement on *"Ethical Conduct in Research Using Human Subjects"*. This form and accompanying letter constitute the Certification of Institutional Ethics Review.

File no:	**5052**
Applicant(s):	**Chengyao Deng** Frank Maurer
Department:	**Computer Science**
Project Title:	**Tool Support for Acceptance Test Driven Development**
Sponsor (if applicable):	

Restrictions:

This Certification is subject to the following conditions:

1. Approval is granted only for the project and purposes described in the application.
2. Any modifications to the authorized protocol must be submitted to the Chair, Conjoint Faculties Research Ethics Board for approval.
3. A progress report must be submitted 12 months from the date of this Certification, and should provide the expected completion date for the project.
4. Written notification must be sent to the Board when the project is complete or terminated.

Janice Dickin

Janice Dickin, Ph.D, LLB,
Chair
Conjoint Faculties Research Ethics Board

6 December 2006
Date:

Distribution: (1) Applicant, (2) Supervisor (if applicable), (3) Chair, Department/Faculty Research Ethics Committee, (4) Sponsor, (5) Conjoint Faculties Research Ethics Board (6) Research Services.

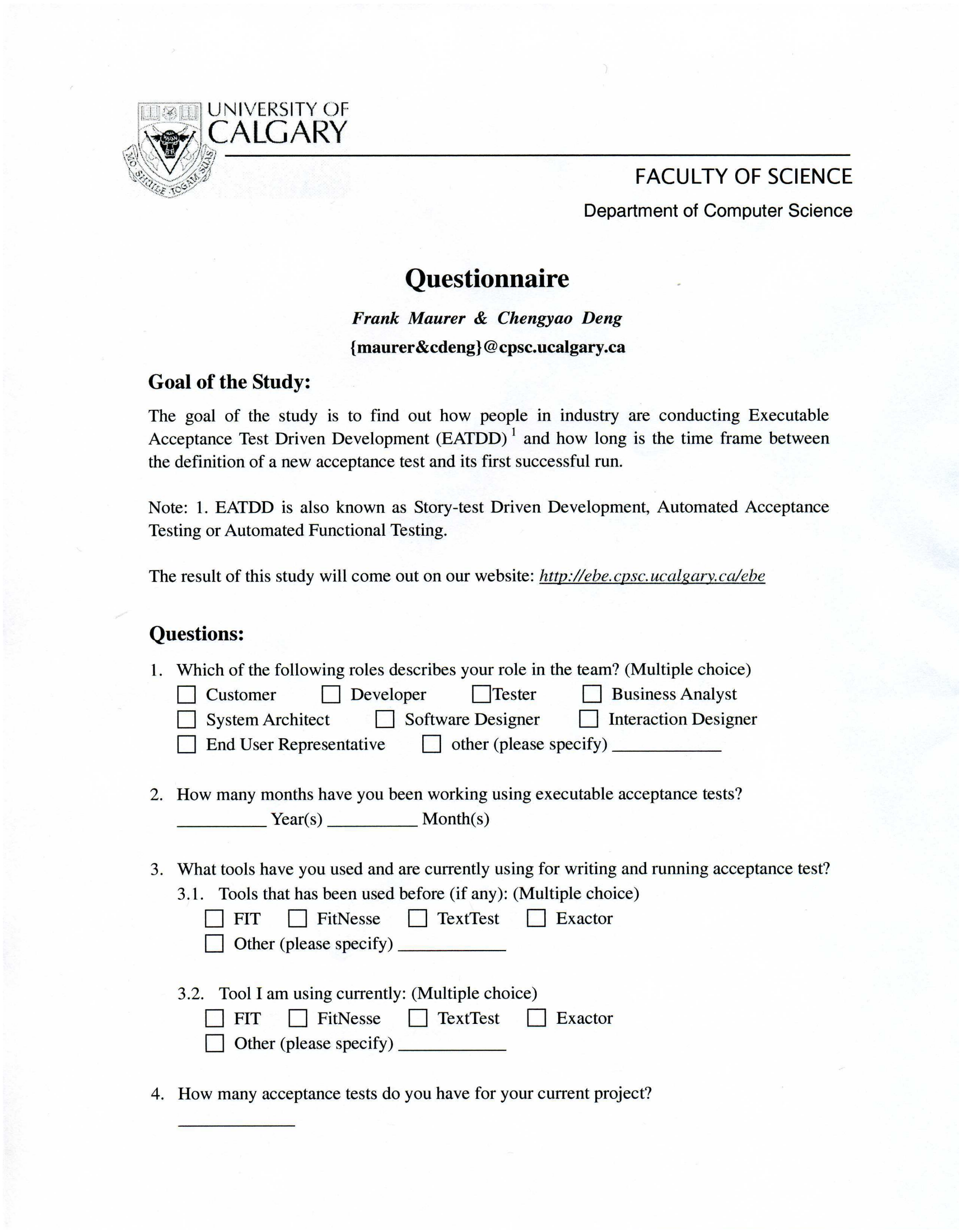
2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4 • www.ucalgary.ca

APPENDIX B: SURVEY MATERIALS

B.1. Mailing Lists of Agile Community

1. Atlanta XP User Group - AXPUG
2. Boston AgileBazaar
3. Brazil, XP Brasil
4. Calgary Agile Method User Group - CAMUG
5. Chicago – CHAD
6. Extreme Programming San Diego – XPSD
7. Ireland, Dublin Agile SIG
8. New York – XpNewYorkCity
9. Seattle XP Group
10. Southern California Agile/XP User Group
11. UK, Manchester/Liverpool England – AgileNorth
12. Vancouver - Agile Vancouver
13. Washington DC XP Users Group
14. Yahoo Agile Testing
15. Yahoo Agile Usability
16. Yahoo Extreme Programming

B.2. Survey Questionnaire



UNIVERSITY OF CALGARY

FACULTY OF SCIENCE

Department of Computer Science

Questionnaire

Frank Maurer & Chengyao Deng

{maurer&cdeng}@cpsc.ucalgary.ca

Goal of the Study:

The goal of the study is to find out how people in industry are conducting Executable Acceptance Test Driven Development (EATDD) [1] and how long is the time frame between the definition of a new acceptance test and its first successful run.

Note: 1. EATDD is also known as Story-test Driven Development, Automated Acceptance Testing or Automated Functional Testing.

The result of this study will come out on our website: *http://ebe.cpsc.ucalgary.ca/ebe*

Questions:

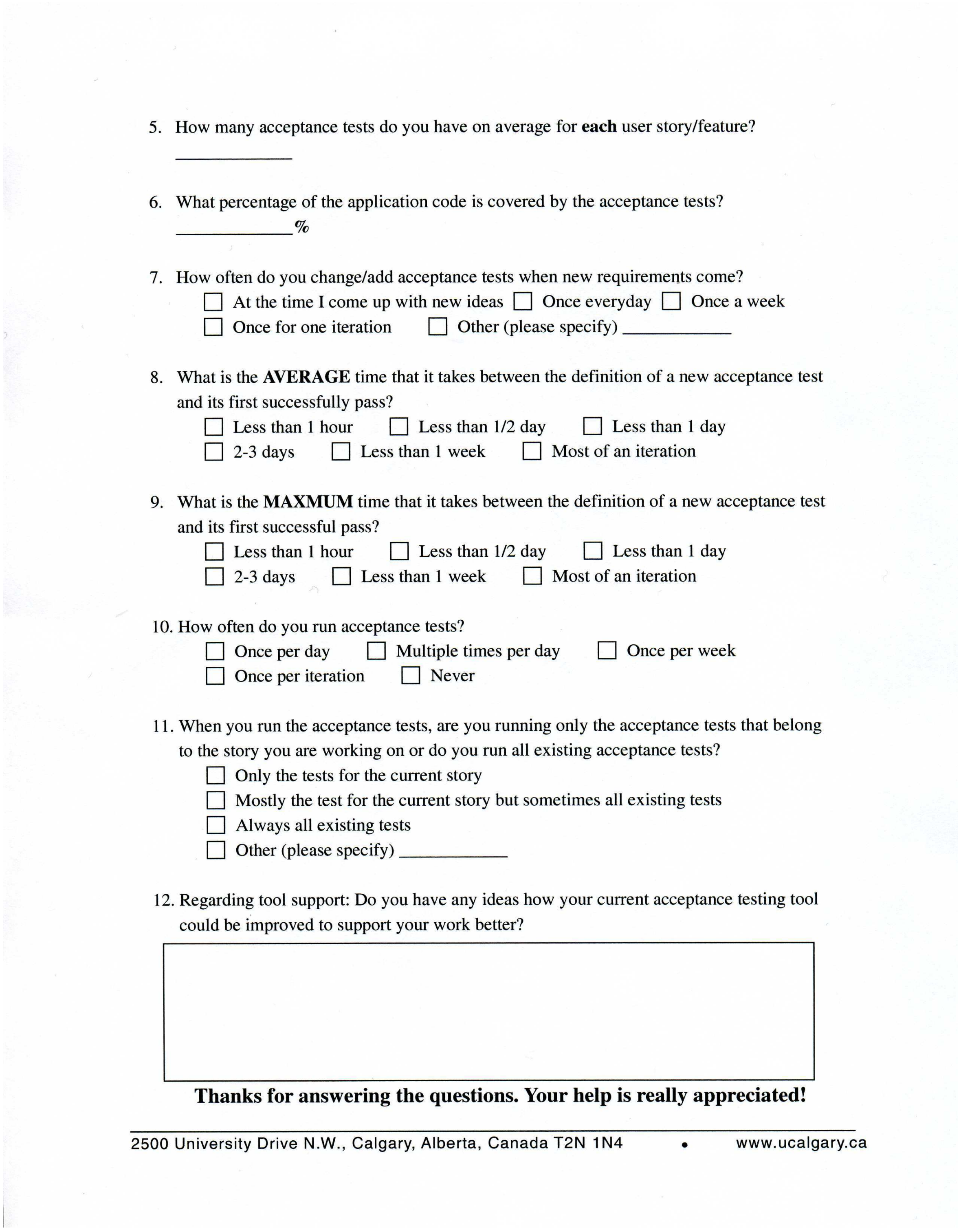
1. Which of the following roles describes your role in the team? (Multiple choice)
   ☐ Customer ☐ Developer ☐Tester ☐ Business Analyst
   ☐ System Architect ☐ Software Designer ☐ Interaction Designer
   ☐ End User Representative ☐ other (please specify) ____________

2. How many months have you been working using executable acceptance tests?
   __________ Year(s) __________ Month(s)

3. What tools have you used and are currently using for writing and running acceptance test?
   3.1. Tools that has been used before (if any): (Multiple choice)
   ☐ FIT ☐ FitNesse ☐ TextTest ☐ Exactor
   ☐ Other (please specify) ____________

   3.2. Tool I am using currently: (Multiple choice)
   ☐ FIT ☐ FitNesse ☐ TextTest ☐ Exactor
   ☐ Other (please specify) ____________

4. How many acceptance tests do you have for your current project?
   ____



5. How many acceptance tests do you have on average for **each** user story/feature?

    ____

6. What percentage of the application code is covered by the acceptance tests?

    ______________%

7. How often do you change/add acceptance tests when new requirements come?

    ☐ At the time I come up with new ideas ☐ Once everyday ☐ Once a week
    ☐ Once for one iteration ☐ Other (please specify) ____________

8. What is the **AVERAGE** time that it takes between the definition of a new acceptance test and its first successfully pass?

    ☐ Less than 1 hour ☐ Less than 1/2 day ☐ Less than 1 day
    ☐ 2-3 days ☐ Less than 1 week ☐ Most of an iteration

9. What is the **MAXMUM** time that it takes between the definition of a new acceptance test and its first successful pass?

    ☐ Less than 1 hour ☐ Less than 1/2 day ☐ Less than 1 day
    ☐ 2-3 days ☐ Less than 1 week ☐ Most of an iteration

10. How often do you run acceptance tests?

    ☐ Once per day ☐ Multiple times per day ☐ Once per week
    ☐ Once per iteration ☐ Never

11. When you run the acceptance tests, are you running only the acceptance tests that belong to the story you are working on or do you run all existing acceptance tests?

    ☐ Only the tests for the current story
    ☐ Mostly the test for the current story but sometimes all existing tests
    ☐ Always all existing tests
    ☐ Other (please specify) ____________

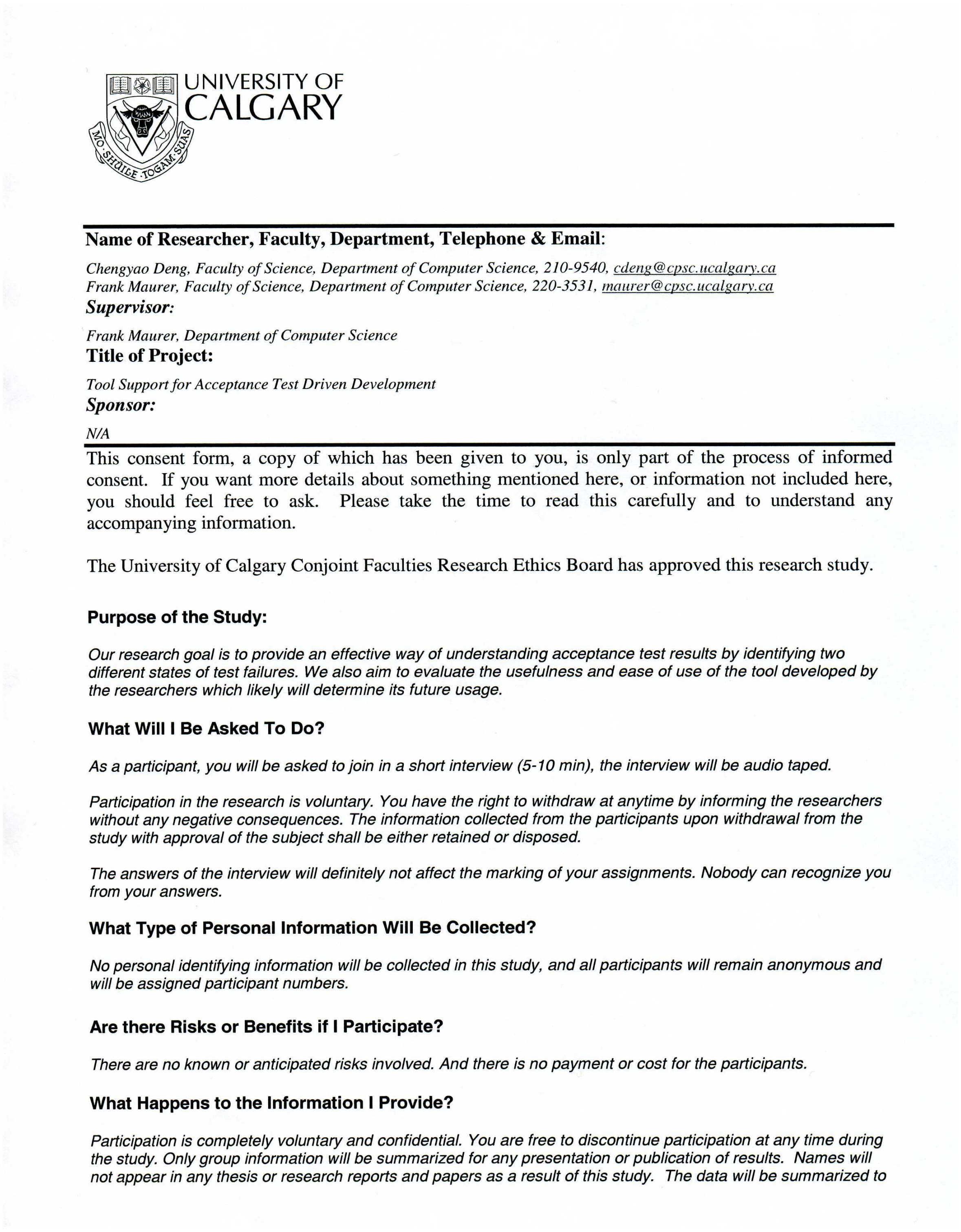
12. Regarding tool support: Do you have any ideas how your current acceptance testing tool could be improved to support your work better?

Thanks for answering the questions. Your help is really appreciated!

2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4 • www.ucalgary.ca

APPENDIX C: PERCEPTIVE STUDY MATERIALS

C.1. Consent Form



UNIVERSITY OF CALGARY

Name of Researcher, Faculty, Department, Telephone & Email:

Chengyao Deng, Faculty of Science, Department of Computer Science, 210-9540, cdeng@cpsc.ucalgary.ca
Frank Maurer, Faculty of Science, Department of Computer Science, 220-3531, maurer@cpsc.ucalgary.ca

Supervisor:

Frank Maurer, Department of Computer Science

Title of Project:

Tool Support for Acceptance Test Driven Development

Sponsor:

N/A

This consent form, a copy of which has been given to you, is only part of the process of informed consent. If you want more details about something mentioned here, or information not included here, you should feel free to ask. Please take the time to read this carefully and to understand any accompanying information.

The University of Calgary Conjoint Faculties Research Ethics Board has approved this research study.

Purpose of the Study:

Our research goal is to provide an effective way of understanding acceptance test results by identifying two different states of test failures. We also aim to evaluate the usefulness and ease of use of the tool developed by the researchers which likely will determine its future usage.

What Will I Be Asked To Do?

As a participant, you will be asked to join in a short interview (5-10 min), the interview will be audio taped.

Participation in the research is voluntary. You have the right to withdraw at anytime by informing the researchers without any negative consequences. The information collected from the participants upon withdrawal from the study with approval of the subject shall be either retained or disposed.

The answers of the interview will definitely not affect the marking of your assignments. Nobody can recognize you from your answers.

What Type of Personal Information Will Be Collected?

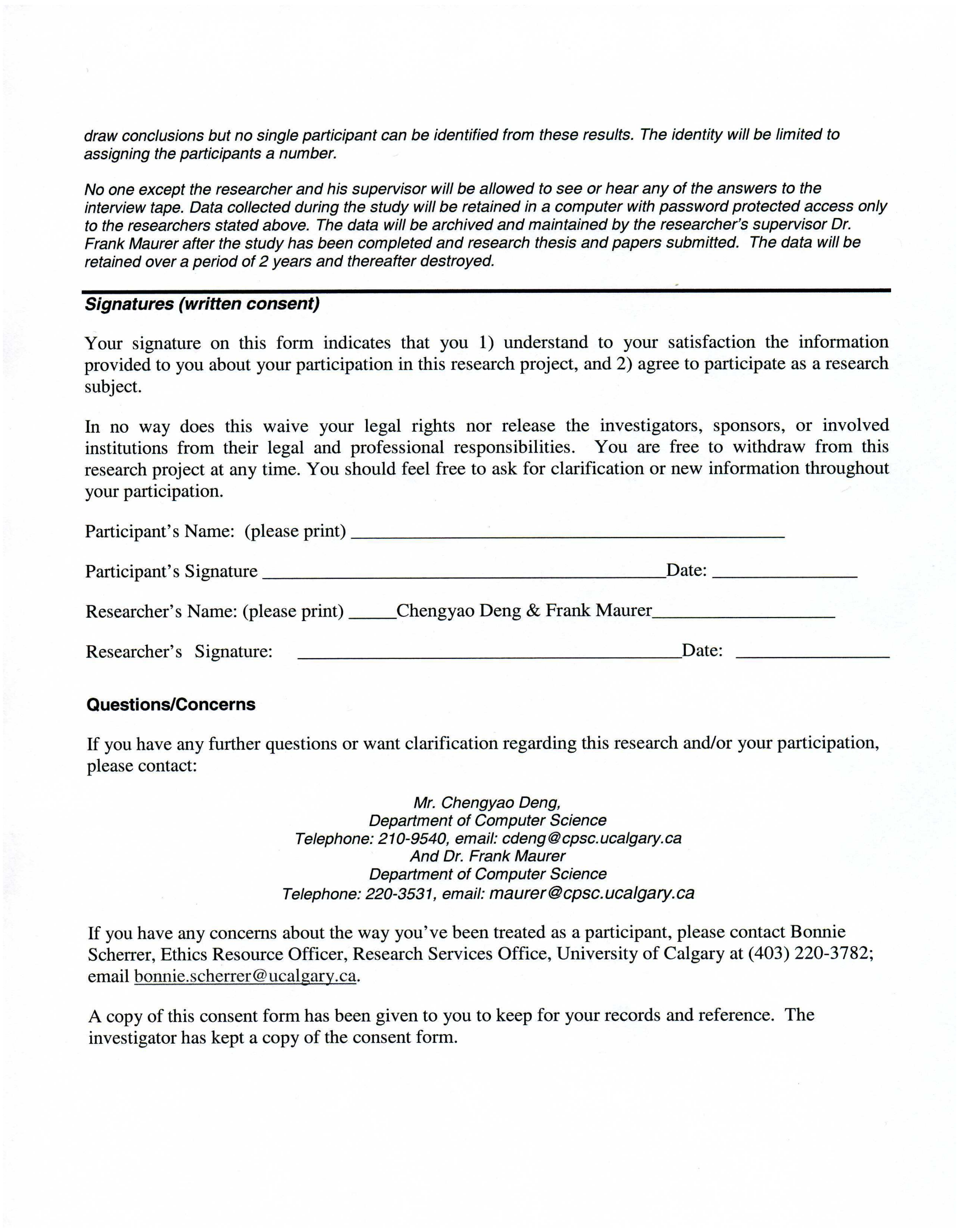
No personal identifying information will be collected in this study, and all participants will remain anonymous and will be assigned participant numbers.

Are there Risks or Benefits if I Participate?

There are no known or anticipated risks involved. And there is no payment or cost for the participants.

What Happens to the Information I Provide?

Participation is completely voluntary and confidential. You are free to discontinue participation at any time during the study. Only group information will be summarized for any presentation or publication of results. Names will not appear in any thesis or research reports and papers as a result of this study. The data will be summarized to



draw conclusions but no single participant can be identified from these results. The identity will be limited to assigning the participants a number.

No one except the researcher and his supervisor will be allowed to see or hear any of the answers to the interview tape. Data collected during the study will be retained in a computer with password protected access only to the researchers stated above. The data will be archived and maintained by the researcher's supervisor Dr. Frank Maurer after the study has been completed and research thesis and papers submitted. The data will be retained over a period of 2 years and thereafter destroyed.

Signatures (written consent)

Your signature on this form indicates that you 1) understand to your satisfaction the information provided to you about your participation in this research project, and 2) agree to participate as a research subject.

In no way does this waive your legal rights nor release the investigators, sponsors, or involved institutions from their legal and professional responsibilities. You are free to withdraw from this research project at any time. You should feel free to ask for clarification or new information throughout your participation.

Participant's Name: (please print) ____

Participant's Signature ____Date: _______________

Researcher's Name: (please print) _____Chengyao Deng & Frank Maurer___________________

Researcher's Signature: ____Date: ________________

Questions/Concerns

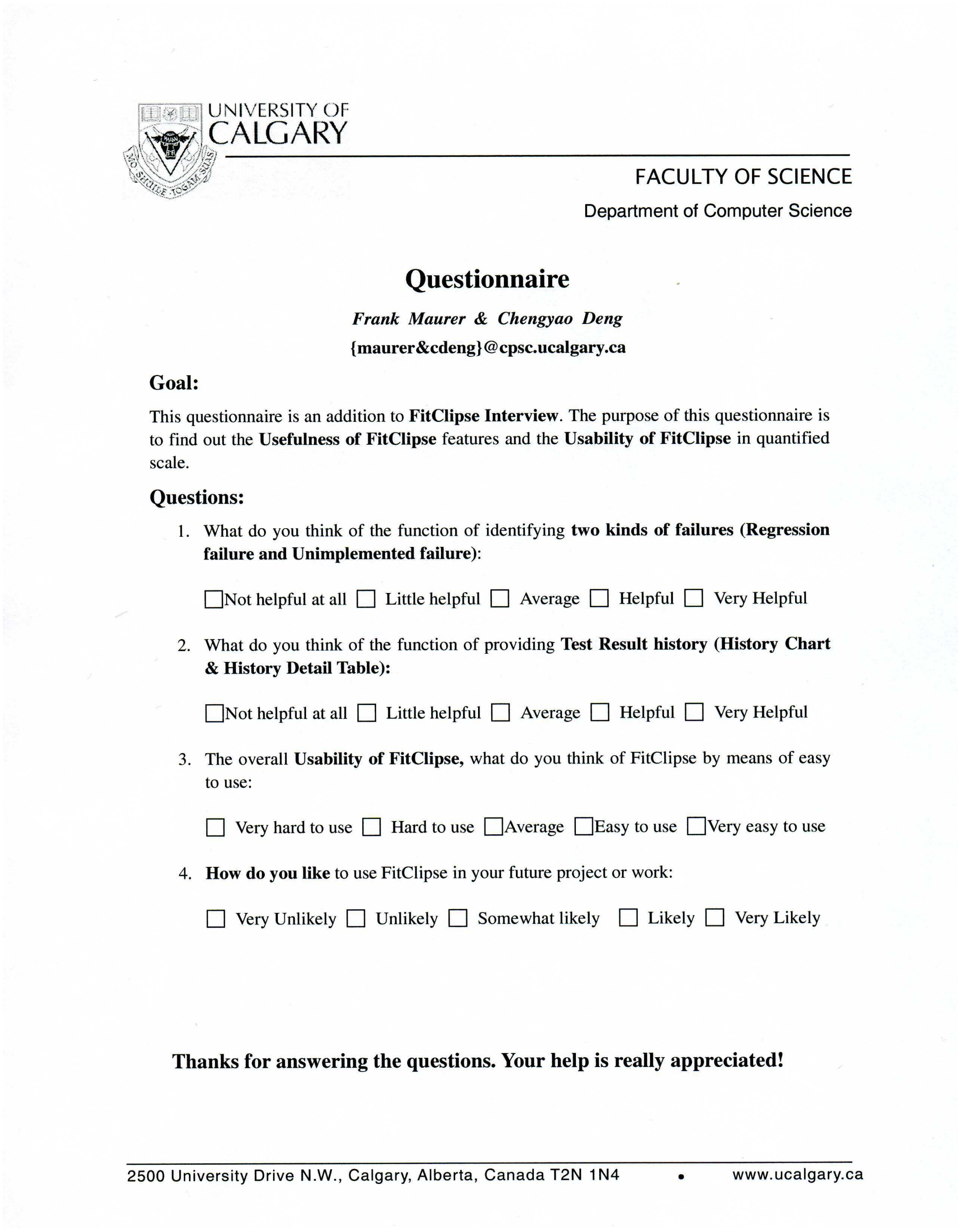
If you have any further questions or want clarification regarding this research and/or your participation, please contact:

Mr. Chengyao Deng,
Department of Computer Science
Telephone: 210-9540, email: cdeng@cpsc.ucalgary.ca
And Dr. Frank Maurer
Department of Computer Science
Telephone: 220-3531, email: maurer@cpsc.ucalgary.ca

If you have any concerns about the way you've been treated as a participant, please contact Bonnie Scherrer, Ethics Resource Officer, Research Services Office, University of Calgary at (403) 220-3782; email bonnie.scherrer@ucalgary.ca.

A copy of this consent form has been given to you to keep for your records and reference. The investigator has kept a copy of the consent form.

C.2. Post Study Questionnaire



UNIVERSITY OF CALGARY

FACULTY OF SCIENCE

Department of Computer Science

Questionnaire

Frank Maurer & Chengyao Deng

{maurer&cdeng}@cpsc.ucalgary.ca

Goal:

This questionnaire is an addition to **FitClipse Interview**. The purpose of this questionnaire is to find out the **Usefulness of FitClipse** features and the **Usability of FitClipse** in quantified scale.

Questions:

1. What do you think of the function of identifying **two kinds of failures (Regression failure and Unimplemented failure)**:

   ☐Not helpful at all ☐ Little helpful ☐ Average ☐ Helpful ☐ Very Helpful

2. What do you think of the function of providing **Test Result history (History Chart & History Detail Table):**

   ☐Not helpful at all ☐ Little helpful ☐ Average ☐ Helpful ☐ Very Helpful

3. The overall **Usability of FitClipse,** what do you think of FitClipse by means of easy to use:

   ☐ Very hard to use ☐ Hard to use ☐Average ☐Easy to use ☐Very easy to use

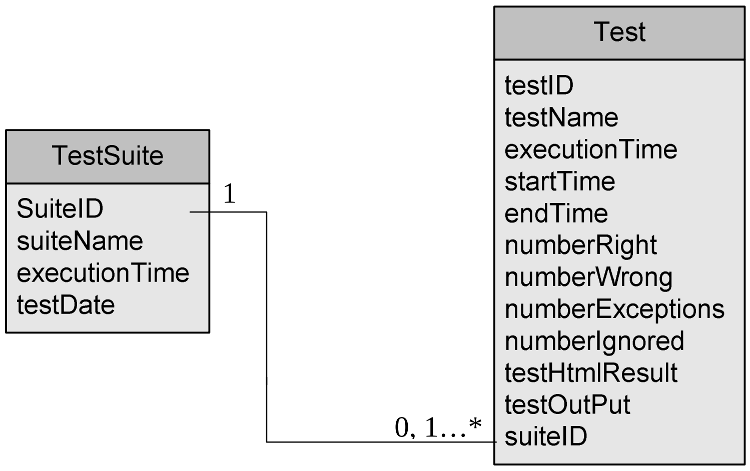
4. **How do you like** to use FitClipse in your future project or work:

   ☐ Very Unlikely ☐ Unlikely ☐ Somewhat likely ☐ Likely ☐ Very Likely

Thanks for answering the questions. Your help is really appreciated!

2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4 • www.ucalgary.ca

APPENDIX D: DATABASE DESIGN



APPENDIX E: MATERIALS AND RAW DATA

1. Survey materials and questionnaires are in the “SurveyMaterials” folder.
    a. Survey charts and raw data are included in the “SurveyData.xls” excel file.
    b. Survey questionnaire is included in this folder as “SurveyQuestionnaire.doc”
2. Pilot study materials and questionnaires are in “PilotAnalysis” folder
    a. Presentation slides of the tutorials to the participants are in this folder.
    b. Pilot study questionnaire is included in the folder as “PilotStudyQuestionnaire.doc”.
    c. Pilot study charts and the raw data are in this folder as “PilotData.xls” excel file.
3. FitClipse source code, FitNesse server source code and the sample project source code containing acceptance tests are in the “FitClipseSrc” folder.
4. FitClipse user manual including the installation guide and other installation required materials are in the “FitClipseManual” Folder. The information is also available online at: http://ebe.cpsc.ucalgary.ca/ebe/Wiki.jsp?page=FitClipse .