UNIVERSITY OF CALGARY

Empirical Analyses of Executable Acceptance Test Driven Development

by

Grigori Igorovych Melnik

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JULY, 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 "Empirical Analyses of Executable Acceptance Test Driven Development" submitted by Grigori Igorovych Melnik in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

_____

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

_____

Dr. Guenther Ruhe, Department of Computer Science

_____

Dr. Barrie R. Nault, Haskayne School of Business

_____

Dr. Tak Shing Fung, Information Technologies

_____

External Examiner, Dr. Philippe Bertrand Pierre Kruchten,

University of British Columbia

_____

Date

Abstract

This research investigates the process of Executable Acceptance Test-Driven Development (EATDD) in the context of specifying functional requirements using the FIT framework, when developing line-of-business applications.

It is guided by three key research questions: 1) how business and technology experts utilize EATDD in the software development lifecycle; 2) what kind of benefits and limitations EATDD manifests, and 3) to what extent improvements in software quality (if any) are associated with EATDD?

The research employs methods of quantitative and qualitative inquiries. Based on the findings of two academic observational studies, one academic quasi-experiment, and three industrial multi-case studies, the following main conclusions are drawn: 1) the use of Executable Acceptance Test-Driven Development is correlated with the enhanced communication in software teams; 2) executable acceptance tests are suitable for specifying functional requirements and are in fact unambiguous, consistent, verifiable, and usable (from both the business experts' and technology experts' perspectives); 3) EATDD provides sufficient evidence of requirements traceability in regulated environments; and 4) current state of tool support negatively impacts the maintainability and scalability of the artifacts produced in the course of EATDD.

In addition, our contribution includes emerged conceptualizations and the socio-technical model of the EATDD process. These not only help to explain the ways how EATDD is used in practice, but also form a base for future investigations.

Acknowledgements

I express my wholehearted *gratitude*, *спасибо*, and *merci* to:

- my Family, who are precious, for their hearts and enduring support;
- my Friends, who are few but true;
- my supervisor (Dr. Frank Maurer), who allowed me a long leash, but was always at the other end when I needed him;
- my supervisory committee (Dr. Günther Ruhe, Dr. Barrie R. Nault) for their encouragement and direction, as well as external members of the examination committee (Dr. Philippe Kruchten, Dr. Tak Shing Fung) for their constructive criticism;
- my collaborators (Thomas Chau, Dr. Mike Chiasson, Ron Jeffries, Robert C. Martin, Dr. Frank Maurer, Kris Read, Dr. Michael Richter, and Dr. Carmen Zannier) for their expertise, care, and enthusiasm;
- my numerous colleagues all around the world, who, through our debates, continue to challenge, stimulate, and inspire;
- my students and industry participants of the studies for providing data for this research, particularly those who illustrated the depth of thought and passion;
- my teaching assistants for their time and hard work;

- reviewers of my papers and reports for their critiques and suggestions.

This research was supported by:

- The Natural Sciences and Engineering Research Council of Canada (NSERC) - Le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG),
- iCore – Informatics Circle of Research Excellence,
- the Killam Trusts,
- and the Agile Alliance.

Thank you!

To my family

for their encouragement,

support and relentless understanding.

Table of Contents

List of Publications

1. Jeffries, R., Melnik, G. ”Test-Driven Development – The Art of Fearless Programming”. *IEEE Software*, 24(3): 24-30, 2007.
2. Martin, R., Melnik, G. “Tests and Requirements, Requirements and Tests: A Moebius loop”. *IEEE Software*, 24(6), 2007.
3. Maurer, F., Melnik, G. “Driving Software Development with Executable Acceptance Tests”, *The Cutter Consortium Report*, 7(11): 1–30, 2006.
4. Melnik, G. “Test-Infecting Future Software Engineers”. *Proc. 5th Annual Workshop on Teaching Software Testing (WTST 2006),* online: www.testingeducation.org/wtst5/ WTST5%20GMelnik%20submission%20final.pdf
5. Melnik, G. “Teaching Acceptance Testing in Contexts of Web Systems Development and Game Programming”. *Proc. 4th Annual Workshop on Teaching Software Testing (WTST 2005),* online: www.testingeducation.org/conference/wtst4/GMelnik%20 Teaching%20Acceptance%20Testing%20final.pdf
6. Melnik, G., Maurer, F. "Multiple Perspectives on Executable Acceptance Test-Driven Development", *Proc. XP2007*, Lecture Notes in Computer Science, Vol. 4536, Springer Verlag: 245–249, 2007.
7. Melnik, G., Maurer, F. “A Cross-Program Investigation of Students' Perceptions of Agile Methods”. *Proc. 27th International Conference on Software Engineering* (*ICSE 2005)*, ACM Press: 481–489, 2005.
8. Melnik, G., Maurer, F. “Direct Verbal Communication as a Catalyst of Agile Knowledge Sharing”. *Proc. Agile Software Development Conference 2004*, IEEE Press: 21–31, 2004.

9. Melnik, G., Maurer, F. “Introducing Agile Methods in Learning Environments: Lessons Learnt”. *Proc. eXtreme Programming/Agile Universe 2003 Conference*, Lecture Notes in Computer Science, Vol. 2753, Springer Verlag: 172–184, 2003.

10. Melnik, G., Maurer, F., Chiasson, M. “Executable Acceptance Tests for Communicating Business Requirements: Customer Perspective”. *Proc. Agile 2006 Conference*, IEEE Computer Press: 35–46, 2006.

11. Melnik, G., Maurer, F. “The Practice of Specifying Requirements Using Executable Acceptance Tests in Computer Science Courses”. *Proc. 20th International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005)*, ACM Press: 365–370, 2005.

12. Melnik, G., Read, K., Maurer, F. “Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective”. *Proc. XP/Agile Universe 2004*, Lecture Notes in Computer Science, Vol. 3134, Springer Verlag: 60–72, 2004.

13. Read, K., Melnik, G., Maurer, F. “Examining Usage Patters of the FIT Acceptance Testing Framework.” *Proc. 6th International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2005)*, Lecture Notes in Computer Science, Vol. 3556, Springer Verlag: 127-136, 2005.

14. Read, K., Melnik, G., Maurer, F. “Student Experiences with Executable Acceptance Testing”. *Proc. Agile 2005 Conference*, IEEE Press: 312-317, 2005 .

15. Zannier, C., Melnik, G., Maurer, F. “On the Successes of Empirical Studies in the International Conference on Software Engineering”. *Proc. 28th International Conference on Software Engineering (ICSE2006)*, ACM Press: 341–350, 2006.

List of Tables

List of Figures

Chapter I Introduction

I.1 Definitions

This dissertation examines a collaborative methodology of developing software called *Executable Acceptance Test-Driven Development*. It is based on a symbiotic relationship of software tests and software requirements.

As is often the case, there is a lack of uniformity in the terminology associated with software testing, requirements engineering, and software engineering at large. Therefore, before discussing the dissertation's scope and significance of its findings, it is important to introduce the following key terms as they are used and applied in the course of this dissertation:

acceptance test = a test conducted to determine whether or not a software system has satisfied a subset of its acceptance criteria;

customer = a person or an organization who is responsible for contracting and paying for the software (development), and, therefore, who is responsible for acceptance of the software produced (could be an internal department, an external customer, or the general public);

(end-)user = a person who ultimately operates and makes use of the software (could be the same as the customer);

software requirement = *a* capability that must be met or possessed by the software in order to satisfy some customer or user needs, desires, or expectations;

functional requirement = a required computational functionality, a feature;

para-functional requirement = an aspect of software beyond functionality (such as reliability, usability, scalability, security, performance, installability, compatibility, portability, etc.);

business expert = a person who possesses skill and knowledge in some business domain and provides to other stakeholders insight into the business problem that the software is meant to address (could be the customer, a potential user, or a business analyst);

technology expert = a person who provides technical expertise on various aspects of the engineering of the software (including design, architecture, coding, testing, administration, maintenance etc.);

software project stakeholders = individuals who contribute to the outcome of the software development project, including customers, business experts, and technology experts;

software testing = a technical investigation done to expose quality-related information about the software to stakeholders [68];

requirements engineering = an investigation done to discover, prune, reconcile, and document stakeholders' requirements about the software being built;

software quality = "value to some person" (as per [138]), where value is the perceived degree of some stakeholder's satisfaction of a software product and its set of attributes.

socio-technical system = any system that is made up of individuals, technologies, processes, and information, and that requires successful integration of all these elements for its proper functioning.

ubiquitous language = a language structured around the domain model and used by all team members (business experts and technology experts) to connect all of the activities of the team with the software (as per [36]) (also known as *domain language*).

I.2 Context of Research

Ambiguous and incomplete software requirements along with insufficient testing are key causes of software projects' failures today [142]. A report by the Workshop on Strategic Directions in Software Quality indicates that software quality becomes the dominant success criterion in the software industry [109].

Despite this, software testing activities are still often overlooked by project teams. Eighty-three percent of organizations' software developers do not like to test code [23]. One of the reasons is simply a lack of time to perform diligent and proper testing, which is frequently the result of inadequate planning and time overruns in other activities. And testing, as the last stage in a waterfall process, is then cut short when delivery deadlines are fixed due to external constraints (e.g. an upcoming tradeshow).

When software testing is performed, often it is done at the level of unit and integration tests by technology experts. However, the goals and mentality of the technology experts may not entirely correspond with those of the customer and business experts. *Acceptance tests* evaluate a software system's functionality from the business perspective. Automated acceptance tests allow to do this more efficiently and also serve as regression tests, to ensure that previously working functionality continues to behave as expected. These tests are often created based on a requirement specification, and serve to verify that contractual obligations are met. Traditionally, authoring and execution of acceptance tests is left till very late in the development lifecycle.

An evolutionary approach to programming, Test-first or Test-driven development (TDD), is gaining popularity in the industry [89]. Essentially, in TDD the developer proceeds by writing a single test case, implementing just enough code to make that test work, and then proceeding forward with another bit of test and the corresponding code. TDD uses a series of small tests to guide the process of detailed design, development and testing.

The TDD paradigm can be extrapolated: to add a feature, there must be an acceptance test for it first. This process is called *Executable Acceptance Test-*

Driven Development. As demonstrated in the body of this dissertation, this process makes it possible to formalize the expectation of the business into an executable and readable specification that programmers follow in order to produce and finalize a working system. This is supposed to establish "a clear context for customers and developers to have a conversation and weed out misunderstandings" [67]. Consequently, it is expected that the risk of building the wrong system is reduced.

Executable acceptance tests can be accessed, revised and run by anyone on the team. This includes a manager or the customer, who may be interested in seeing the progress of the development, or exercising some additional "what-if" scenarios to gain even more confidence that the system is working properly.

I.3 Problem Statement

Executable Acceptance Test-Driven Development is a new approach which is becoming increasingly popular among software engineering teams (especially, agile teams). However, the extant literature lacks systematic empirical evaluation of this practice from the perspectives of all stakeholders. There are claims and anecdotes on the effect of the practice with regard to the improved communication about requirements among members of the software teams. But those are not substantiated by independent research. Furthermore, it is largely unknown how exactly software teams utilize executable acceptance tests in their software development processes, what challenges they face and how they resolve them. This lack of a empirical evidence may hinder the adoption of the practice beyond the industry innovators. It also makes it difficult to compare the practice to other approaches.

I.4 Dissertation scope

The dissertation methodically examines **the suitability of automated acceptance tests and the use of evolutionary process of Executable Acceptance Test-Driven Development (EATDD) for communicating functional requirements and driving software development**. In the context of this dissertation, "suitability" is defined as a degree to which the functional requirements are found to be unambiguous, verifiable, consistent, and usable by all project stakeholders – business and technology experts – for understanding the software system. Through a series of quantitative and qualitative studies (Chapters V and VI), this dissertation explores how EATDD improves communication and helps mitigate the "sins" of software requirement specifications (discussed in detail in §II.1).

In a socio-technical system, equal emphasis is placed on both the technical and the social (human) aspects of the system. Hence, this research also addresses the social and cognitive aspects of applying EATDD.

I.5 Research Questions

The key research questions are:

- how business and technology experts utilize EATDD in the software development lifecycle;
- what kinds of benefits and challenges EATDD manifests; and
- to what extent improvements in software quality (if any) are associated with EATDD?

I.6 Significance

It is estimated that 85 percent of the defects in developed software originate in the requirements [142]. "Irrespective of the format chosen for representing

requirements, the success of a product strongly depends upon the degree to which the desired system is properly described" [58]. This research shows how combining discovery and specification of requirements with the discipline and automation of testing can have a significant impact on the overall system quality and, as a result, on the economics success of software engineering projects. Empirical analyses of the cognitive, social and technical processes of EATDD can provide a solid foundation for teams and decision makers responsible for software process improvement.

I.7 Contributions to the Academic Body of Knowledge

This research contributes to the current software engineering body of knowledge by providing:

1. a substantial multi-perspective empirical analysis of the use of EATDD in both industrial and academic settings and of the ways business experts and technology experts communicate with executable acceptance tests;
2. a synthesis of benefits and limitations of EATDD;
3. an added understanding of the cognitive aspects of authoring and interpreting requirements in the form of acceptance tests;
4. an added understanding of the formation of a common domain (ubiquitous) language;
5. insights into organizational aspects of using EATDD in software teams;

Secondary contributions include:

6. an experience base of examples, cases and lessons learnt from using executable acceptance tests for specifying programming assignments in computer science courses, which can be used by other academics – both for pedagogical purposes and for research investigations (e.g., replication of the quasi-experiment);

I.8 Basic Assumptions

The following assumptions and constraints are made:

- only **line of business (LOB) applications** are considered; these may include mission-critical systems, but no safety-critical systems;
- research focuses on specifying and communicating **functional business** requirements; para-functional requirements and qualities of services are not of primary focus;
- EATDD was studied in the context of **utilizing the following open-source tools only: FIT, FitNesse, and FitLibrary**; no commercial tools for functional testing (such as HP/Mercury WinRunner, HP/Mercury QuickTest Pro IBM/Rational Robot, Microsoft Visual Studio 2005 Team Edition for Software Testers, Borland SilkTest, Empirix eTester,) were evaluated since they do not easily render themselves to the application of the Test-Driven Development paradigm;
- respondents answer the questions **truthfully**.

In conclusion, we have no doubts about the veracity and willingness of the respondents, and feel privileged that respondents shared their thinking and their work with us.

Chapter II Research Foundations and Literature Review

II.1 Requirements Articulation

Most software requirements (under the stated assumptions) are not specified using formal languages, but instead are written as some form of a business requirement document (commonly called the "*functional spec*"). Normally such documents are written using natural languages (prose), diagrams and pictures.

There are several "sins" to avoid when specifying requirements, some of which are listed by Meyer[1] [100]. We have expanded this list to the following thirteen risks:

1. **Noise.** Noise manifests itself as information not relevant to the problem, or a repetition of existing information phrased in different ways. Noise may also be the reversal or shading of previously specified statements. Such inconsistencies between requirements constitute 13% of requirements problems [58].
2. **Silence.** A second risk is silence, in which important aspects of the problem are simply not mentioned or overlooked. Omitted requirements account for 29% of all requirements errors [46].
3. **Overspecification.** Overspecification can happen, for example, when aspects of the solution are mentioned as part of the problem description.

[1] Meyer's classification is frequently referenced (see pp.232-233 in [112] for example); we have added some additional deficiencies to the traditional "seven sins".

Requirements should describe what is to be done, not how it is implemented [29].

4. **Wishful thinking.** This hazard occurs when prose describes a problem to which a realistic solution would be difficult or impossible to find or simply too costly.

5. **Ambiguity**. This is common when natural languages allow for more than one meaning for a given word or phrase. Ambiguity is problematic when jargon includes terms that are familiar to the other party in different ways [100].

6. **Reader subjectivity**. Prose is also prone to reader subjectivity, since each person has a unique perspective (based on cultural background, language, personal experience, and so on).

7. **Forward references.** These mention aspects of a problem not yet stated and cause confusion in larger documents.

8. **Oversized documents.** Lengthy documents are difficult to understand, use, and maintain.

9. **Customer uncertainty.** When an inability to express specific needs results in an inclusion of vague descriptions, customer uncertainty may arise. This, in turn, leads to developers making assumptions about "fuzzy" requirements; it has been estimated that incorrect assumptions account for 49% of requirements problems [58].

10. **Multiple representations.** Making requirements understandable to the customer and verifiable by the developer might lead to the creation of multiple representations of the same requirements. Preserving more than one document can then lead to maintenance, translation, and synchronization problems.

11. **Tools for requirements capture.** Requirements are sometimes lost, especially para-functional requirements (like scalability, security,

maintainability, and performance), when the use of tools for requirements capture only support a strictly defined format or template.

12. **Little to no user involvement.** Requirements documents are often poor when written with little or no user involvement, instead being compiled by requirements solicitors, business analysts, domain experts, or even developers.
13. **Gold plating**. Gold plating of requirements can be an issue, particularly when the requirements are supposed to define the complete scope of a project (as common in waterfall processes). At the beginning of a project, the business experts and the technology experts know less about actual needs than at any later stage in the project. Even if the customer is not yet sure about a requirement, he or she needs to include it in the specification since adding it later would be problematic or costly.

II.2 Dimensions of Software Testing

Software testing is defined as a technical investigation done to expose quality-related information about the software to stakeholders [68]. When speaking about software testing, it is useful to distinguish the types of tests; those that are *business-facing* from those that are *technology-facing*. Business-facing tests speak of the problem and are specified in the context of the application using the language of the domain. The language must be understandable by a business expert. In fact, it is desirable that these tests are written by or with the business expert. For example the following statement can be seen as a business facing test:

> "*The senior consultant John Deltoid retrieves the Net Present Value for the SuperNet project in Canadian dollars. The result is CDN$2,030,820.55*".

Business facing tests should be powerful enough to describe, communicate and clarify requirements and also to provide the appropriate contexts.

Technology-facing tests speak of the implementation and are specified in terms of technical artifacts, functionality, and relationships. They talk about things that are relevant to technology experts but are often too technical for business experts. An example of a technology-facing test is:

> "*Adding a consultant John Smith from Calgary, Canada, results in a non-redundant creation of a row in the database table Corporation.Consultant with the auto-generated userid, name 'John Smith', city 'Calgary', country 'Canada', timestamp, and status 'unverified'. A temporary random 8-ASCII-characters long password is generated, SHA-1 encoded, stored in the table Sys.Login, and emailed to "john.smith@detloid.ca" via the Mailing service*".

The boundaries between business facing and technology facing tests can be somewhat fuzzy. For example, para-functional testing includes primarily technology-facing areas such as performance, scalability, interoperability, security etc., but also spans onto usability, which is, obviously, more business-facing.

While in waterfall-like Tayloristic projects, testing is just a phase at the end of the development lifecycle when the coding is completed. In agile projects, testing moves to a more prominent role. Testing is seen as an all-encompassing activity performed at different levels by all stakeholders – technology experts (programmer's unit testing and test-driven development, acceptance testing, exploratory testing, specialized para-functional testing) and business experts (acceptance testing, usability testing, compliance testing).

II.3 Acceptance Testing

An *acceptance test* (which is known under several other terms, see Table 1) is a (formal) test conducted to determine whether or not a system has satisfied its acceptance criteria. It must be defined from the business perspective. An acceptance test should enable the customer to determine whether or not to accept

the system (as defined in [1] and [112]). The objective is to provide confidence that the delivered system meets the business requirements.

Table 1. Alternative Terms for "Acceptance Tests".

Term	Introduced/Used by
– *"functional tests"*[2]	Beck, Extreme Programming Explained, 1/e [16]
– *"customer tests"*	Jeffries [63], Beck, Extreme Programming Explained, 2/e
– *"customer-inspired tests"*	Beck, Extreme Programming Explained, 1/e [16]
– *"story-tests"* and *"story-test-driven development"*	Kerievsky [73]
– *"specification by example"*	Fowler [47]
– *"coaching tests"*	Marick [82]
– *"examples", "business-facing example",* and *"example-driven-development"*	Marick [81]
– *"conditions of satisfaction"*	Cohn [26]
– *"scenario tests"*	Kaner [70]
– *"keyword-driven test"*	Kaner, Bach, Pettichord [69]
– *"soap opera tests"*	Buwalda [20]
– *"formal qualification tests"*	e.g. DOD [30]
– *"user acceptance tests" (UAT)*	e.g. IEEE [59]
– *"client acceptance tests"*	e.g. IEEE [59]
– *"system tests"*	e.g. IEEE [59]

These tests are often created based on a *requirement specification* (or a "*functional spec*", as it is often called in the industry). This creates a dependency between the requirements specification and acceptance test suite, a dependency that may involve a great deal of overhead and excessive costs. Changes to one side

[2] Generally, functional tests and acceptance tests are not synonymical (a para-functional test may be an acceptance test, and, conversely, not all functional tests are acceptance tests). This dissertation, however, focuses on functional acceptance tests only.

necessitate changes to the other, and effort is needed to ensure that the written requirements correspond precisely to the expected test results (and vice versa). This dependency means that problems in the requirements specification will directly impact quality. Moreover, this necessitates translation between the requirements specification and acceptance tests. Such translation is not only costly but it also can increase the risk of misunderstanding.

Business-facing acceptance tests are meant to eliminate some of these deficiencies by complementing traditional high-level abstract requirements specifications with tangible, concrete examples.

Acceptance tests must test the system as a whole (as opposed to unit testing, which tests internal units and technical details). The primary motivation for acceptance testing is to demonstrate the working functionality rather than to find bugs (although bugs may be found as a result of acceptance testing). They are traditionally specified using scenarios or rule sets, and performed by quality assurance teams together with the business experts.

II.4 Scenarios

Jarke et al. define a scenario as "*a description of a possible set of events that might reasonably take place"* [61]. The main purpose of developing scenarios is *"to stimulate thinking about possible occurrences, assumptions relating these occurrences, possible opportunities and risks, and courses of action"* [61].

Alexander argues that *"scenarios are a powerful antidote to the complexity of systems and analysis. Telling stories about systems helps ensure that people – stakeholders – share a sufficiently wide view to avoid missing vital aspects of problems. Scenarios vary from brief stories to richly structured analyses, but are always based on the idea of a sequence of actions carried out by intelligent agents. People are very good at reasoning from even quite terse stories, for example detecting inconsistencies, omissions, and threats with little effort. These innate human capabilities give scenarios their power"* [3]. Scenarios are

applicable to systems of all types. Importantly, scenarios are not just abstract artifacts, but a "critical representation of the realities as seen by those who create them." [61]

Rolland et al. [118] provide a good survey which distinguishes between the purpose or intended use of a scenario, the knowledge content contained within a scenario, how a scenario is represented, and how it can be changed or manipulated. Another taxonomy by Carroll [22] classifies scenarios according to their use in systems development.

Scenarios are also used as a good mechanism for understanding software systems and validating software architectures [72]. For example, in the 4+1 View Model of architecture proposed by Kruchten, the scenario view consists of a small set of critical use case instances, which illustrate how the elements of the other four views (logical, process, development and physical) work together seamlessly. *"The architecture is partially evolved from these scenarios."* [74]

Sutcliffe suggests considering scenarios along a continuum from the real world descriptions and stories to models and specifications. *"At one end of this dimension, scenarios are examples of real world experience, expressed in natural language, pictures, or other media. At the specification end are scenarios which are arguably models such as use cases, threads through use cases and other event sequence descriptions."* Sutcliffe continues: *"Within the space of scenarios are representations that vary in the formality of expression in terms of language and media on one dimension and the phenomena they refer to on the other; ranging from real world experience, to invented experience, to behavioural specifications of designed artefacts."* [132]

In the context of this research, we consider the most common form of scenarios – examples or stories grounded in real world experience. As will be shown, the key element of the EATDD process is the executable acceptance test, which is a form of a scenario, which is also executable.

II.5 Early Test Design

In [48], Gause and Weinberg wrote *"Surprisingly, to some people, one of the most effective ways of testing requirements is with test cases very much like those for testing the completed system."* By this statement they were asserting that the act of writing tests is an effective way to test the completeness and accuracy of the requirements. Their suggestion was that these tests should be written as part of the process of gathering, analyzing, and verifying requirements; long before those requirements are coded. In the same reference they go on to say: *"We can use the black box concept during requirements definition because the design solution is, at this stage, a truly black box. What could be more opaque than a box that does not yet exist?"* [48] Clearly, the authors put a very high value on developing early test cases as a requirements analysis technique.

Testing expert Graham agrees and also emphasizes the importance of performing test design activities early – "*as soon as there is something to design tests against - usually during the requirements analysis"* [55]. Graham regards the act of test design as highlighting what the users really want the system to do. If tests are designed early and with users' involvement, problems will be discovered before they are built into the system.

This recommendation of Gause & Weinberg and Graham to write acceptance tests *early* has also been promoted by the testing community [57] but remains at odds with much current practice. Most development organizations do not write acceptance tests at all. The first tests they write are often manual scripts written after the application starts executing. These regression tests are based on the behavior of the *executing* system as opposed to the original requirements. Instead of manual tests, some organizations use record & playback tools as a way to automate their tests. These tools record the tester's strategic decisions by watching that tester operate the *current* system, and remembering what the system does in response. Later the tool can repeat the sequence and report any deviation. While such record-playback tools can be valuable (see, for example, various strategies in [98]), it is also clear that they are written far later

than Gause & Weinberg and Graham suggest, and that their connection to the original requirements is indirect at best[3].

II.6 Test-Driven Development

Test-First Design or Test-Driven Development (TDD), as it is also called, is a discipline of design and development where every line of new code written is in response to a test. A TDD practitioner thinks of what small step in capability would be a good next addition to the program. She then writes a short test showing that that capability is not already present. She implements the code that makes the test pass, and verify that all the tests are still passing. She reviews the code as it now stands, improving the design as she goes (this activity is known as refactoring). Finally the process is repeated, devising another test for another small addition to the program.

As the practitioner follows this simple cycle, shown in Figure 1, the program grows into being. She thinks of one small additional thing it needs to do; she writes a test specifying just how that thing should be invoked and what its result should be. She implements the code needed to make it work, and finally she improves the code, folding it smoothly into the existing design, evolving the design as needed to keep it clear.

[3] Several specialized requirements management tools (such as Telelogic DOORS and Rational RequisitePro) provide test traceability.

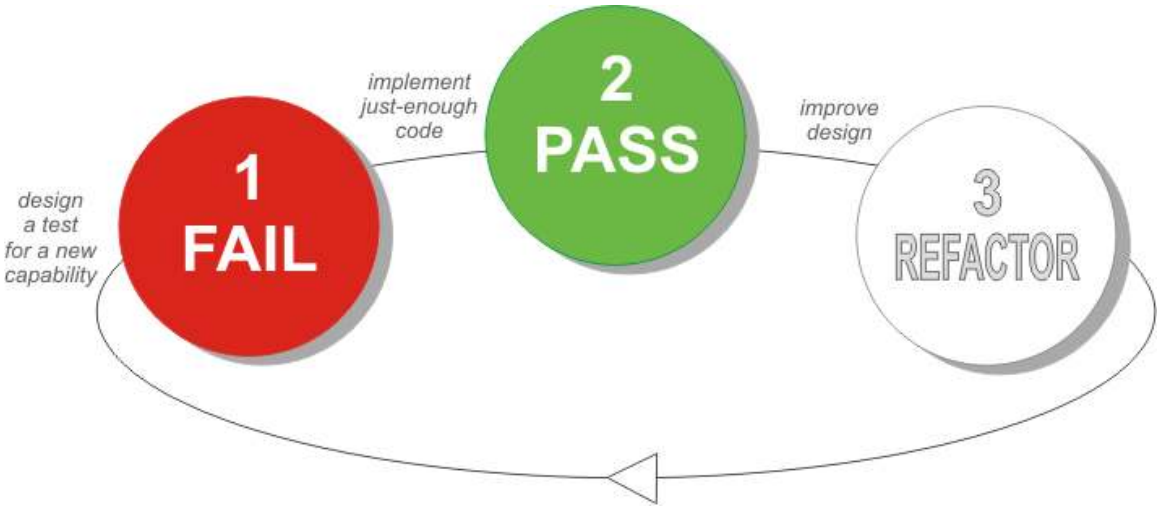


Figure 1. TDD Step Cycle

At all times, the intention is that all tests pass except for one new one that is "driving" the development of new code. In practice, of course, even the best programmers make mistakes. The growing collection of comprehensive tests (the regression suite) tends to detect these problems.

With Test-Driven Development, each functional bit of the program is specified and constrained by automated tests. These tests tend to prevent errors, and they tend to detect errors when they do occur. The best response to a discovered error when practicing TDD is to write the test that was missing — the test that would have prevented the defect.

Test-Driven Development approach extends Boris Beizer's original assertion of 1983 that *"the act of designing tests is one of the most effective bug preventers known"* [17]. Test-Driven Development as a practice appeared as part of the Extreme Programming discipline, as described in 1999, in Beck's "Extreme Programming Explained" [16]. TDD tools have come into being for almost every computer language you can imagine, from C++ through Visual Basic, all the major scripting languages, and even some of today's and yesterday's more exotic languages.

Notice, TDD is a design and programming activity, not a testing activity per se. Because of the possible confusion, the new terms such as Behavior-Driven

Development [9] and Example-Driven Development [79] have been recently introduced. The testing aspect of TDD is largely confirmatory (through the regression suite produced). The investigative testing still needs to be performed by professional testers.

TDD has caught the attention of a large software development community: they find it to be a good and rapid way to develop reliable code, and many practitioners find it to be a very enjoyable way to work. TDD embodies elements of design, testing, and coding, in a cyclical, rhythmic style. In short cycles, the programmer writes a test, makes it work, and improves the code each time around the loop. The fundamental rule of TDD is never to write a line of code except for those necessary to make the current test pass.

The current state of research on TDD is reflected by Table 2 and Table 3. We summarize on the productivity and quality impacts. The results are controversial (more so in academic studies). This is to no surprise. Controversy is partially due to a difficulty in isolating the effects of solely TDD when lots of context variables are playing out and due to incomparable measurements. In addition, many studies do not have statistical power to allow for generalizations. One thing all researchers seem to agree on is that, as minimum, TDD encourages better task focus and better test coverage. A mere fact of more tests does not necessarily mean that the software will be of better quality, but, nevertheless, the increased attention of programmers to test-design thinking is encouraging. In the view of testing as a sampling process (of a very large population of potential behaviors), *"to the extent that each test is capable of finding an important problem that none of the other tests can find, then as a group, more tests means a more thorough sample"* [11]. This is useful especially if you can run them cheaply.

Notably, a Cutter Consortium report authored by Khaled El-Emam, based on a survey of companies on various software process improvement practices, identified TDD as the practice with the second highest impact on project success (after code inspections) [34].

TDD is also making its way to university and college curricula (IEEE/ACM SE2004 Guidelines for Software Engineering Undergraduate Programs list test-first as a desirable skill [66]). Educators report success stories when using TDD in computer science programming assignments.

Test-Driven Development is becoming a popular approach across all sizes and kinds of software development projects. Example of its use in diverse and non-trivial contexts include: control system design [32], GUI development [120], and database development [6]. In addition, Johnson et al inspect the aspect of incorporating performance testing in TDD [64].

Table 2. Summary of Selected Empirical Studies on TDD. Industry Participants.

Family of Studies	Type	Development time analyzed	Legacy project?	Organization studied	Software Built	Software Size	# participants	Language	Productivity effect	Quality effect
Sanchez et al 2007 [121]	Case study	5 years	Yes	IBM	Point of sale device driver	medium	9-17	Java	Increased effort 19%	40% (A)
Bhat/Nagappan Microsoft Research 2006 [18]	Case study	4 months	No	Microsoft	Windows Networking common library	small	6	C/C++	Increased effort 25-35%	62% (A)
	Case study	≈7 months	No	Microsoft	MSN Web services	medium	5-8	C++/C#	Increased effort 15%	76% (A)
Canfora et al 2006 [21]	Controlled-experiment	5 hrs	No	Soluziona Software Factory	Text analyzer	very small	28	Java	Increased effort by 65%	Inconclusive based on quality of tests
Damm/Lundberg 2006 [28]	Multi-case study	1-1.5 year	Yes	Ericsson	Components for a mobile network operator application	medium	100	C++/Java	Total project cost increased by 5-6%	5-30% decrease in Fault-Slip-Through Rate; 55% decrease in Avoidable Fault Costs
Melis et al 2006 [87]	Simulation	49 days (simulated)	No	Calibrated using KlondikeTeam & Quinary data	M@rket info project	medium	4 (simulated in 200 runs)	Smalltalk	Increased effort 17%	36% reduction in residual defect density
Mann 2005 [76]	Case study	8 months	Yes	PetroSleuth	Windows-based oil&gas project management with elements of statistical modeling	medium	4-7	C#	n/a	81% (C); customer & developers' perception of improved quality
Geras et al 2004 [51]	Quasi-experiment	≈ 3 hrs	No	Various companies	simple database-backed business information system	small	14	Java	No effect	Inconclusive based on the failure rates; Improved based on # tests & frequency of execution
George/Williams 2003 [50]	Quasi-experiment	4 ¾ hrs	No	John Deer, Role Model Software, Ericsson	Bowling Game	very small	24	Java	Increased effort 16%	18% (B)
Ynchausti 2001 [141]	Case study	8.5 hrs	No	Monster Consulting	Coding exercises	small	5	n/a	Increased effort 60-100%	38-267% (A)

Notes: (A) Reduction in the internal defect density; (B) Increase in % of functional black-box tests passed (external quality); (C) Reduction in external defect ratio (cannot be solely contributed to TDD, but to a set of practices);
green background = improvement; red background = deterioration.

Table 3. Summary of Selected Academic Empirical Studies on TDD. Academic Participants.

Family of Studies	Type	Development time analyzed	Legacy project?	Organization studied	Software Built	Software Size	# participants	Language	Productivity effect	Quality effect
Flohr/Schneider 2006 [43]	Quasi-experiment	40 hrs	Yes	University of Hannover	Graphical workflow library	small	18	Java	Improved productivity by 27%	Inconclusive
Abrahamsson et al 2005 [2]	Case study	30 days	No	VTT	Mobile application for global markets	small	4	Java	Increased effort by 0% (iteration 5) - 30% (iteration 1)	No value perceived by developers
Erdogmus et al 2005 [35]	Controlled-experiment	13 hrs	No	Politecnico di Torino	Bowling game	very small	24	Java	Improved normalized productivity by 22%	No difference
Madeyski 2005 [75]	Quasi-experiment	12 hrs	No	Wroclaw University of Technology	Accounting application	small	188	Java	n/a	-25-45% (B)
Melnik/Maurer 2005 [Error! Reference source not found.**]**	Multi-case study	4 months projects over 3 years	No	University of Calgary/SAIT Polytechnic	Various web-based systems (surveying, event scheduling, price consolidation, travel mapping)	small	240	Java	n/a	73% of respondents perceive TDD improves quality
Edwards 2004 [33]	Artifact Analysis	2-3 weeks	No	Virginia Tech	CS1 programming assignment	very small	118	Java	Increased effort 90%	45% (B)
Pančur et al 2003 [110]	Controlled experiment	4.5 months	No	University of Ljubljana	4 programming assignments	very small	38	Java	n/a	No difference
George 2002 [49]	Quasi-experiment	1 ¼ hr	No	North Carolina State University	Bowling Game	very small	138	Java	Increased effort 16%	16% (B)
Müller/Hagner 2002 [103]	Quasi-experiment	≈10 hrs	No	University of Karlsruhe	Graph library	very small	19	Java	No effect	No effect, but better reuse & improved program understanding

Notes: (B) Increase in % of functional black-box tests passed (external quality).
green background = improvement; red background = deterioration.

II.7 Acceptance Test Automation

Manual acceptance testing, where a tester trivially follows a test script written by somebody else (scripted testing)[4], by triggering system functionality via the user interface, is time consuming, especially when the tests need to be executed repeatedly for multiple releases or multiple configurations. Manual tests are also prone to human errors. To address these concerns, it is recommended to automate acceptance test execution[5].

Additionally, the nature of any iterative process (but especially of those with short and frequent release cycles) will dictate such automation of acceptance tests (i.e. producing ***executable acceptance tests***). If it takes long time to execute regression tests, the chances are they will not be run frequently. As a result, their power (of gathering and providing feedback to the stakeholders) about the "health" and stability of the system will be drastically reduced. As attested by Kaner, Bach and Pettichord, *"the most successful companies automate testing to enhance their development flexibility"* and not to eliminate testers [69, p.94].

It is important for automated regression suites to remain in sync with the product. Automated regression suites that drive the application via its user interface (produced by many popular capture-replay tools) tend to decay quickly even with slightest user interface (UI) changes. To shield the regression tests from this dreadful fortune, many experts today agree that automation should be done at the level just beneath the UI [119, 78, 52, 125]. Of course, the UI needs to be tested as well, but if the core principle of the separation of concerns is applied rigorously, the UI would be thin with the bulk of processing (the business logic)

---

4 Note this is different (and significantly less powerful) from exploratory testing, in which even though manual execution of test cases takes place it is interweaved with continuous test design and learning about the system. For more on exploratory testing, we refer the reader to the first-rate explanation by Bach [10].

5 In fact, the recommendation is to automate testing before the existence of required business functionality. Usability testing should be conducted on top of that before the system is formally accepted.

beneath it – and that bulk is what should get well-tested by the automated acceptance test suite.

Importantly, placing an emphasis on test automation by no means rules out manual exploratory testing performed by skilled human beings. We regard both approaches – automated acceptance testing and manual exploratory testing – as plausible and complimentary.

II.8 Executable Acceptance Test-Driven Development

Extreme Programming (XP) and Industrial XP apply Test-Driven Development at a higher level of acceptance tests and advocate writing executable acceptance tests at the beginning of the development iteration (in the test-first fashion). As a result, ***Executable Acceptance Test-Driven Development*** (hereby referred to as "***EATDD"***) makes it possible to formalize the expectation of the business into an executable and readable specification that programmers follow in order to produce and finalize a working system [63]. Extrapolating the standard Test-Driven Development paradigm, to add a feature, there must be an acceptance test for it first. This helps establish *"a clear context for customers and developers to have a conversation and weed out misunderstandings"* [67]. Consequently, it is claimed that the risk of building the wrong system is reduced.

Executable acceptance tests can be accessed, revised and run by anyone on the team. This includes a manager or the customer, who may be interested in seeing the progress of the development, or exercising some additional "what-if" scenarios to gain even more confidence that the system is working properly.

It is claimed that EATDD helps with requirements discovery, clarification, and communication. Such tests are specified by the customer, domain expert or analyst, prior to implementing features, and serve as executable acceptance criteria. Once the code is written, these tests are used for automated system acceptance testing.

Few industrial testimonials of the use of EATDD in the real world were documented in the form of experience reports. Reppert, for example, describes a way executable acceptance test-driven development is changing the way business and technology experts of the Nielsen Media Research work [116]. The perceptions of the members of the team that adopted this process on a major data warehousing project were very positive. These include opinions of a senior project manager, two senior SQA analysts, and a product manager. The product manager emphasized that after a few months of absorbing the practice, *"now everyone on the team really sees its value"*. The project manager agrees – *"It was difficult to trust the process in the beginning, but it's so much better than what we used to do"*.

Nielsen and McMunn report on several projects in a large financial services organization, in which automated acceptance testing was routinely performed at the end of each iteration [108]. However, it is unclear who wrote the tests (business experts or technology experts).

Andrea discussed an approach involving generating code from acceptance tests specified in a declarative tabular format within Excel spreadsheets [8].

While the reports provided in these papers are valuable, their limitation is that the evidence provided is mainly anecdotal and that no systematic and rigorous evaluation was used.

Within the research community, little attention is being drawn to executable acceptance testing and EATDD. Steinberg has looked into how acceptance tests can be used by instructors to clarify programming assignments and by students to check their progress in introductory courses [130]. There is an ongoing debate about who should write acceptance tests [122] and the differences between acceptance testing and unit testing has been examined by Rogers [117]. He provides practical advice on defining a common domain language for requirements, helping customer to write acceptance tests, and integrating the acceptance tests into the build process. Watt and Leigh-Fellows described an adaptation of XP style planning that makes acceptance tests central not only to

the definition of a story but also central to the process itself. They showed how acceptance testing can be used to drive the entire development process using industrial case [135]. Mugridge and Tempero discussed evolution of acceptance tests to improve their clarity for the customer. The approach using tables for acceptance test specification was found to be easier to use than previously developed formats [104].

As tutorials and peer-to-peer workshops on acceptance testing frameworks and practices become more prominent at agile software engineering conferences and more empirical evidence becomes available, it is envisioned that the practice of EATDD will receive a wider adoption. A recent book by Mugridge and Cunningham dedicated to EATDD is another step in the direction of EATDD crossing the chasm. This book is a definitive guide that is full of examples and rationalizations intended to introduce the practice to both business experts and technology experts [**Error! Reference source not found.**].

II.9 Tabular Representations and the FIT Framework

Since the business perspective is the most important when specifying acceptance tests, it is logical to think of ways , in which business experts would be comfortable in doing so. A tabular representation is one such method.

Parnas recognized the value of tabular specification as early as 1977 when he was working on the A-7 project for the U.S. Naval Research Lab. In 1996 he wrote: *"Tabular notations are of great help in situations like this. One first determines the structure of the table, making sure that the headers cover all possible cases, then turns one's attention to completing the individual entries in the table. The task may extend over weeks or months; the use of the tabular format helps to make sure that no cases get forgotten."* [60]

Cunningham also used tables to create the FIT Framework [38], which today is the most popular framework supporting EATDD. Its name is derived from the thesaurus entry for “acceptable.” The goal of FIT is to express an acceptance test

in a way that an ordinary person can read or even write it. To this end, FIT *tests* come in two parts: tests defined using ordinary tables and usually written by business experts, and later FIT *fixtures*, which are written to map the data from table cells onto calls into the system (this process is known as "*fixturizing* acceptance tests"). Fixtures are implemented by the technology experts and usually are not visible to business experts. By abstracting the definition of the test from the logic that runs it, FIT opens up authorship of new tests to anyone who has knowledge of the business domain.

ca.easytix.fixtures.CalculateDiscount				
tickets	senior	student	employee	discount?
9	no	no	no	0.00
10	no	no	no	0.05
20	no	no	no	0.10
1	no	no	yes	0.10
10	no	no	yes	0.10
20	no	no	yes	0.10
1	yes	no	yes	0.15
1	no	yes	yes	0.15
1	yes	yes	yes	error
1	yes	no	no	0.15
1	no	yes	no	0.15
1000	no	no	no	0.40 *expected* 0.1 *actual*

```
public class   CalculateDiscount
           extends ColumnFixture {

  public int tickets;

  public boolean senior;
  public boolean student;
  public boolean employee;


  public float discount ()
     throws DiscountException {
     return
     ca.easytix.core.DiscountRule.
     getDiscount(tickets,
        senior, student, employee);
  }

}
```

Figure 2. Sample FIT table and ColumnFixture in Java.

Figure 2 demonstrates one popular style of specifying acceptance tests via calculation rules. The first row in the table is the reference to the fixture (that links the test to the real system). The first four columns of the second row are the labels of the test attributes and the last column denotes the calculated value. The rest of the rows represent the acceptance test cases with test inputs in the first four columns and the expected values in the last one. When executed, the FIT engine (the underlying test runner) delegates the execution of the business logic to the fixture and highlights the assertion cells in green (if the test passes) or red

(in case if it fails). The third option possible is an exception in the business logic that has not been caught and handled gracefully. In this case the engine will highlight it in yellow and will optionally include the error message and the stack trace. Thus, each colored cell represents a test case. A test page may comprise from multiple test tables, which can also interact.

Figure 3 shows an example of another style — expressing business workflows or transactions. FIT tables can be created using common business tools including, spreadsheets or word processors, and can be included in many types of documents (HTML, MS Word, and MS Excel). Fixtures that call into the application can be written in a variety of languages, including Java, Ruby, C#, C++, Python, Objective C, Perl, and Smalltalk.

fit.ActionFixture		
start	samples.Browser	
enter	select	1
check	title	Akila
check	artist	Toure Kunda
enter	select	2
check	title	American Tango
check	artist	Weather Report
check	album	Mysterious Traveller
check	year	1974
check	time	3.70
check	track	2 of 7

```
public class   Browser
           extends ActionFixture {
  ...
  public void select(int i) {
    MusicLibrary.
    select(MusicLibrary.library[i-1]);
  }

  public String title() {
    return MusicLibrary.looking.title;
  }

  public String artist() {
    return MusicLibrary.looking.artist;
  }
  ...
}
```

Figure 3. Simple FIT table and ActionFixture in Java.

II.10 FitNesse

This idea of enabling anyone to author FIT tests is taken one step further by FitNesse [40], a Web-based collaborative testing and documentation tool designed around FIT. FitNesse provides a very simple way for teams to collaboratively create documents, specify tests, and even run those tests through a Wiki Web site. A Wiki Web is and editable web site whose contents can easily be

changed and extended using standard web browsers. FitNesse is a self-contained standalone cross-platform Wiki server that does not require any additional servers or applications to be installed and, therefore, is very easy to set up. The FitNesse Wiki[6] allows anyone to contribute content to the website without knowledge of HTML or programming technologies. FitNesse tests can also be run in the command-line mode, which allows them to be easily integrated into auto-build scripts.

II.11 FitLibrary

FIT provides a set of basic test styles/fixtures that support workflows and business calculations. It also enables the team to extend the framework by adding its own test table shapes (fixtures). Mugridge extended the standard FIT with several useful fixtures and assembled them in the FitLibrary [39]. It is becoming more and more popular today (in fact, many of the new fixtures are now part of FitNesse). In particular, we should introduce the reader to the DoFixture. It is analogous to the ActionFixture (Figure 3) in the way that it allows to define business workflows and transactions. However, if ActionFixture assembles those workflows through operations that resemble user interface controls (like press, check, enter), the DoFixture leverages the semantic power of English sentence composition. The aim is to make tests even more easily readable. Consider a fragment of a workflow test in Figure 4. The row that starts with "user posts a new trip…", reads as a normal English sentence. There is no special jargon or order of elements that looks like a function call. It is a natural way most people would tell stories. Notice that the first, third, fifth, and seventh cells contain keywords, which provide information about the role of the data that's in the alternating cells highlighted in bold – the second, fourth, and sixth ("Vancouver", "attending CADE Conference", "05-01-2005", "05-03-2005"). The keywords are coloured when the test is executed. The keywords all joined together to give the name of the action that a developer would implement in the fixture class. If a

[6] http://wiki.org/wiki.cgi?WhatIsWiki

negative test case needs to be specified, DoFixture allows that with a special prefix keyword “reject”, that checks that the action fails, as expected.

ca.ucalgary.seng.myworld.TripFixture							
user posts a new trip to	Vancouver	for the purpose of	attending CADE Conference	from	05-01-2005	to	05-03-2005

```
public boolean userPostsANewTripToForThePurposeOfFromTo (
      String place, String purpose, Date from, Date to) {
      // call the business object that supports this transaction
      // ...
}
```

Figure 4. DoFixture-style test fragment and the corresponding fixture code.

Other useful test table styles include ArrayFixture (for ordered lists), SetFixture (for unordered lists), FileCompareFixture (for comparing files and directories, etc). FitLibrary also provides support for grids and images (with GridFixture and ImageFixture) which makes it easy to define tests that require specific layouts (particularly useful when a feature is supposed to generate a report in which the layout matters, e.g. an invoice).

II.12 Ubiquitous Language

Domain-Driven Design is a philosophy has developed as an undercurrent in the object-oriented analysis and design (OOAD) community over the last two decades. The premise of Domain-Driven Design is two-fold:

- For most software projects, the primary focus should be on the domain and domain logic; and
- Complex domain designs should be based on a model.

According to its creator, Evans, Domain-Driven Design is not a technology or a methodology. *“It is a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains”* [36].

The term “Ubiquitous Language” is central to Domain-Driven Design. It means a *“language structured around the domain model and used by all team members (including the business representatives) to connect all the activities of the team with the software”*.

If business experts and technology experts use different terms for the same ideas, then it is almost impossible for the two to communicate effectively. Building a ubiquitous language means committing for all team members to use a common set vocabulary; if a concept is required that is not in that language, then the concept should be named and the language extended.

Chapter III Research Approach.

III.1 Research Goal

Research goals are designed to address the research questions articulated in §I.5. The goals evolved from the related research in the areas of requirements engineering and software testing, as well as from the preliminary analysis of agile practices conducted using a grounded theory approach.

The **Main Goal** is to determine how business and technology experts use EATDD for discovering, articulating and validating functional software requirements. On the deeper level, the following sub-goals emerged:

- **Sub-Goal 1**: capture and conceptualize experiences of the teams following EATDD;
- **Sub-Goal 2**: evaluate the detective, communicative, inciting, and creative powers of EATDD;
- **Sub-Goal 3**: determine the challenges business and technology experts encounter when using EATDD;
- **Sub-Goal 4**: investigate the effectiveness of the FIT framework and the FitNesse tool for authoring, managing and executing acceptance tests.

III.2 Research Design

Research design is a strategy of inquiry that includes various research methods of observation and analysis of the observed data. Figure 5 outlines the overall research design and the research process flow. Individual phases with summarized objectives, used methods, subjects, and outcomes are presented in Table 4. Based on this design, research methods were selected to obtain the relevant data in accordance with the research goals and questions. This was an

emergent design – each new study and findings led to refinement of initial questions and formulation of new ones. The initial design did not include the qualitative fieldwork "in the wild". However, as we proceeded with the first three stages it became apparent that such an investigation would be necessary.

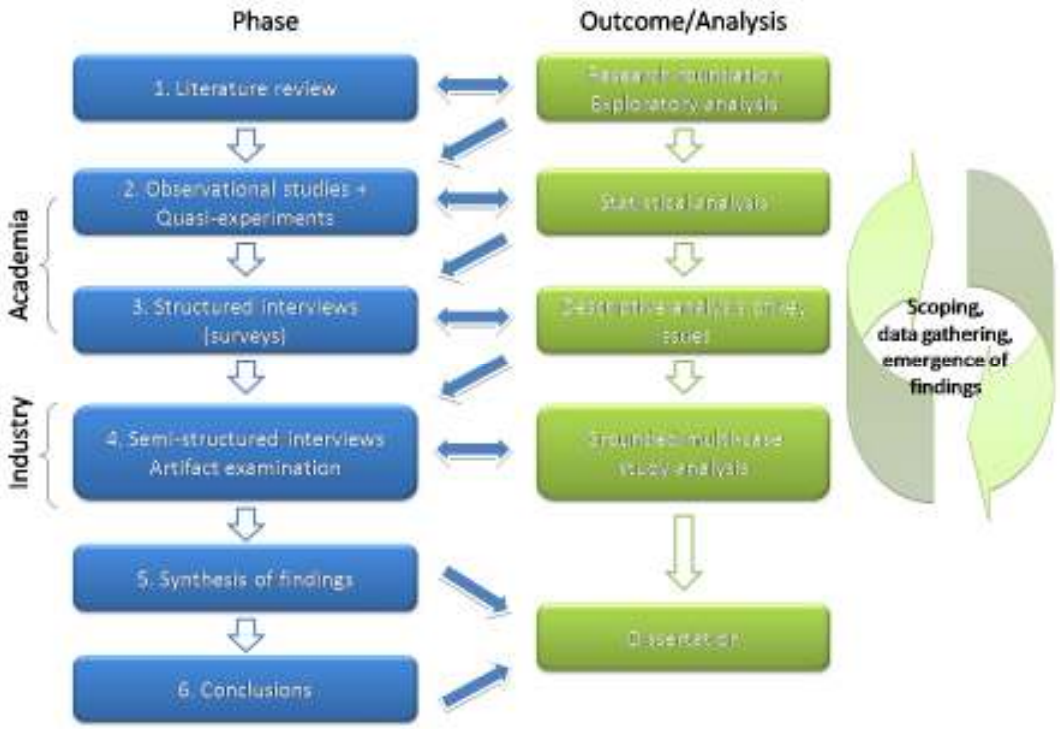


Figure 5. Research Design

Individual rounds of data gathering were distinctive, following different objectives and using different methods. However, they all built up towards a coherent holistic research goal. The research began with the *first* round (Table 4, phase 1) by reviewing the existing body of knowledge and generating an initial set of research questions. The *second* round (Table 4, phases 2-4) dealt primarily with analyzing whether technology experts are capable of interpreting and authoring requirements in the form of executable acceptance tests and implementing code to satisfy those requirements. We also identified various usage patterns. One of the limitations of the second round was that teams of students (though cross-assigned) had to play dual roles of business and

technology experts by both specifying and implementing functional requirements. Therefore, in the *third* round (Table 4, phase 5) we specifically design the quasi-experiment in such way that subjects only had to play a single role: either business expert or technology expert. Furthermore, we invited graduate students from the business school with less programming experience to participate (to model real world business experts as close as possible). This was a major improvement which led to a series of useful findings in terms of the authoring ability of business experts to communicate with acceptance tests. The *fourth* round of investigation dealt primarily with aggregating results from the previous three rounds, comparative analysis of EATDD to other types of testing techniques, the survey of the existing EATDD tools (Table 4, phase 6). Throughout the first four rounds of studies, we continued informal discussions with industry professionals (at the conferences and through the mailing lists). Our preliminary results with business school graduates were overly optimistic in comparison with the anecdotal evidence from the field. A deeper investigation was necessary. The *fifth* and the longest round (Table 4, phases 8-9) focused on the qualitative evidence "from the wild". Through a multi-case study analysis which included multiple iterations of semi-structured interviews, and analyses of testing and coding artifacts, we have deepened our understanding of the process of EATDD, its cognitive, technological and social aspects.

Table 4. Research Process Flow and Summary of Outcomes

Phase	Published study	Objective	Perspective	Method used	Subjects	Outcome
1	–	Foundation building; study of the existing body of knowledge	TE/BE	Literature review	AG/AU/I	– Problem statement – Initial set of research questions – Narrowed scope of research
2	*Melnik, Read, Maurer 2004* [95]	Investigate suitability of FIT acceptance tests for specifying functional requirements	TE	Observational study	AU	– Interpretation ● – Learnability ● – Implementation ◖ – Authoring ○
3	*Read, Melnik, Maurer 2005a* [114]	Identify usage patterns	TE	Observational study	AU	– Incremental implementation ● – Regression ● – Fixture refactoring ○ – Maintainability ○
4	*Read, Melnik, Maurer 2005b* [115]	Study technology experts' perceptions	TE	Survey	AU	– Collaborative interpretation ● – Independent interpretation ● – Authoring ◖
5	*Melnik, Maurer, Chiasson 2006* [94]	Investigate communicative powers	BE	Quasi-experiment	AG/AU	– Collaborative authoring ● – Learnability ◖
6	*Maurer, Melnik 2006 (Cutter Report)* [85]	Aggregate info on various usages and tools; comparative analysis to various testing techniques	BE/TE	Literature & tool review	AG/AU/I	– Aggregate of tools – Strong inciting power – Strong communicative power – Weak detective power
7	*Martin, Melnik 2007 (IEEE Software)* [84]	Formulate equivalence hypothesis that concrete requirements blend with acceptance tests	BE/TE	Exploratory descriptive study	I	– Equivalence hypothesis – Business functionality examples – Performance example – Concurrency example
8	*Melnik, Maurer 2007 (XP 2007)* [90]	Investigate communicative, inciting and detective power	BE/TE	Multi-case study	I	– Communicative power ● – Learnability ● – Ease of use ● – Sufficiency ○
9	*Melnik 2007 (Dissertation)*	Aggregately investigate suitability of acceptance tests for specifying functional requirements; develop theoretical frameworks	BE/TE	Grounded theory	I/AG/AU	– Theoretical framework ● – Improved communication ● – Collaborative interpretation ● – Independent interpretation ● – Collaborative authoring ● – Independent authoring ○ – Learnability ● – Regression ● – Traceability ● – Maintainability ○

Legend: ● positive results ◖ inconclusive ○ negative results
AU academic undergraduate AG academic graduate I industry
TE technology experts BE business experts

III.3 Research Methods Summary

Without going into the quantitative-qualitative argument, we firmly believe that both types of studies can be used to develop analysis, elaborate on it and provide rich details. Therefore, we employed a combination of quantitative and qualitative methods for our investigation of various aspects of the EATDD in the context of both academic programming assignments and industrial projects. As a result, our quantitative studies, on one hand, are intended to "*persuade the reader through de-emphasizing individual judgment"* and stressing the use of established statistical procedures, leading to generalizable results; while our qualitative research, on the other hand, *"persuades through rich depiction and strategic comparison across cases, thereby overcoming the abstraction inherent in quantitative studies."* [37]

The detailed descriptions of the research methods employed are included in the corresponding chapters (quantitative methods in Chapter IV, and qualitative methods in Chapter V).

III.4 Evaluation criteria

Inspired by Marick [80], we have identified the following set of evaluation criteria for EATDD:

- *the communicative power* of clarifying requirements, and improving conversations between the technology experts and the business experts (the primary criterion);
- *the inciting power* of provoking the technology experts to focus on the right code;
- *the creative power* of inspiring the business experts and helping them more quickly realize the possibilities inherent in the product;
- *the detective power* of helping to find bugs in the product.

III.5 Cognitive framework:

We have identified four levels of comprehension within the cognitive domain of executable acceptance testing – from simple recognition of facts and "understanding of scenarios with assistance", as the lowest level, through increasingly more complex and abstract mental levels, to the highest order which is classified as "authoring independently". These levels are inspired by the Bloom's taxonomy [19].

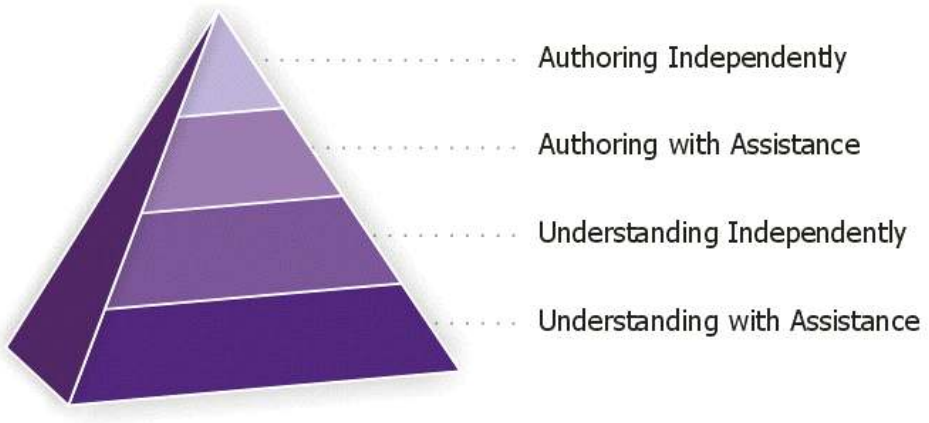


Figure 6. Four levels of Executable Acceptance Testing comprehension.[7]

The first (lowest) level of understanding is characterized by being able to read and understand executable acceptance tests with the assistance of a trained expert. This level of understanding is at least what is expected from the business experts who do not have a technical background. The second level of understanding is being able to read and understand acceptance tests independent of outside information sources. This level of understanding requires knowledge of acceptance testing, the notation and framework used, and often the ability to interpret, understand and articulate the functional requirements found in the

[7] The pyramid depicts the gradual increase the level of comprehension and skill. In order to move to a higher level, one must master the activities at the lower levels first. Also, the metaphor emphasizes the fact that there are likely fewer people capable of achieving higher levels.

underlying test cases. This level of understanding must be achieved by the technology experts in order to implement the requirements depicted in the acceptance tests. The third level of understanding is required in order to specify new test cases with assistance of a trained expert (who could be a tester, a developer, or a business analyst). Authoring acceptance tests is more difficult than reading and understanding them. Tools may make organizing and inputting tests easier. However, tools cannot give one the cognitive ability to make inferences, to come up with good examples, and to judge the quality of an acceptance test. The fourth and the greatest level is when a business expert is able to author the acceptance tests independently.

While authoring and understanding both heavily involve the use of scenarios and examples, it is important to emphasize how different these processes are. In case of authoring, the person not only invents concrete, illustrative examples/scenarios of a certain feature but discovers new features along the way. The goal of understanding is different – it is to get enough context and details about the feature in order to implement it, to improve it, to extend the set of acceptance tests, or to simply learn more about the underlying model of the system.

Chapter IV Quantitative Analyses

IV.1 Academic Study One: Technology Experts' Perspective

IV.1.1 Impetus

As discussed in §II.9, FIT tests are a tabular representation of customer expectations. If the expectations themselves adequately explain the requirements for a feature, can be defined by the business expert, and can be read by the technology expert, there may be some redundancy between the expression of those expectations and the written system requirements. Consequently, it may be possible to eliminate or reduce the size of prose requirements definitions. An added advantage to increased reliance on acceptance tests may be an increase in test coverage, since acceptance testing would both be mandatory and defined early in the project life cycle. To this end an observational study has been designed to evaluate the understandability of FIT acceptance tests for functional requirements specification, primarily from the perspective of the technology experts.

IV.1.2 Instrument

A project was conceived to develop an online document review system (DRS). This system allows users to submit, edit, review and manage professional documents (articles, reports, code, graphics artifacts etc.) called submission objects (*so*). These features are selectively available to three types of users: Authors, Reviewers and Administrators. More specifically, administrators can create repositories with properties such as: title of the repository, location of the repository, allowed file formats, time intervals, submission categories, review criteria and designated reviewers for each item. Administrators can also create

new repositories based on existing ones. Authors have the ability to submit and update multiple documents with data including title, authors, affiliations, category, keywords, abstract, contact information and bios, file format, and access permissions. Reviewers can list submissions assigned to them, and refine these results based on document properties. Individual documents can be reviewed and ranked, with recommendations (accept, accept with changes, reject, etc) and comments. Forms can be submitted incomplete (as drafts) and finished at a later time.

For the present, subjects were required to work on only a partial implementation concentrating on the submission and review tasks (Figure 7). The only information provided in terms of project requirements was:

1. An outline of the system no more detailed than that given in this section.
2. A subset of functional requirements to be implemented (Figure 7).
3. A suite of FIT tests (Figure 8)

Specification

1. Design a data model (as a DTD or an XML Schema, or, likely, a set of DTDs/XML Schemas) for the artifacts to be used by the DocumentReviewSystem. Concentrate on "Document submission/update" and "Document review" tasks for now.
2. Build XSLT sheet(s) that when applied to an instance of so's repository will produce a subset of so's. As a minimum, queries and three query modes specified in DrsAssignmentOneAcceptanceTests must be supported by your model and XSLT sheets.
3. Create additional FIT tests to completely cover functionality of the queries.

Setup files

drs_master.xml - a sample repository against which the FIT tests were written

DrsAssignmentOneAcceptanceTests.zip - FIT tests, unzip them into FITNESSE_HOME\FitNesseRoot\ directory.

Figure 7. Assignment specification snapshot[8]

DRS Assignment One Acceptance Test Suite Startswith Author Search

DrsAssignmentOneAcceptanceTests.FindByAuthorUnsorted
DrsAssignmentOneAcceptanceTests.FindByAuthorSortByTitle
DrsAssignmentOneAcceptanceTests.FindByAuthorSortByTitleDescending
DrsAssignmentOneAcceptanceTests.FindByAuthorSortByType
DrsAssignmentOneAcceptanceTests.FindByAuthorSortByDate
DrsAssignmentOneAcceptanceTests.FindByAuthorSortByDateDescending

Contains Author Search

DrsAssignmentOneAcceptanceTests.FindByAuthorContainsUnsorted
DrsAssignmentOneAcceptanceTests.FindByAuthorContainsSortByTitle
DrsAssignmentOneAcceptanceTests.FindByAuthorContainsSortByTitleDescending
DrsAssignmentOneAcceptanceTests.FindByAuthorContainsSortByType
DrsAssignmentOneAcceptanceTests.FindByAuthorContainsSortByDate

Figure 8. Partial FIT Test Suite. The suite contains test cases and can be executed. For example, the test FindByAuthorUnsorted results in an unsorted list of items matching an author name

Requirements in the FIT Test Suite of our experiment can be described generally as sorting and filtering tasks for a sample XML repository. Our provided suite initially consisted of 39 test cases and 657 assertions. In addition to developing the code necessary to pass these acceptance tests, participants were required to extend the existing suite to cover any additional sorting or filtering features associated with their model. An example FIT Test finding a document by exact match of author name, with results sorted by title in descending order is shown in Figure 9.

Participants were given two weeks (unsupervised) to implement these features using XML, XSLT, Java and the Java API for XML Processing (JAXP). A common online experience base[9] was set up and all students could utilize and contribute to

[8] http://mase.cpsc.ucalgary.ca/EB/Wiki.jsp?page=SENG513w04AssignmentOne

[9] http://mase.cpsc.ucalgary.ca/EB/

this knowledge repository. An iteration planning tool and source code management system were available to all teams if desired.

IV.1.3 Hypotheses

Based on the literature survey and our initial theory of acceptance tests as functional requirements, we formulated two deep hypotheses:

A. FIT acceptance tests describe a customer requirement such that a technology expert can implement the feature(s) for that requirement.

B. Technology experts with no previous FIT experience will be able to learn how to use FIT given the time provided.

In addition, from the exploratory perspective, we hypothesized that:

C. 100% of technology experts will create code that passes 100% of customer provided tests.

D. More than 50% of the requirements for which no tests were given will be implemented and tested.

E. 100% of implemented requirements will have corresponding FIT tests.

fit.ActionFixture		
start	seng513.w04.drs.fixtures.FindActionFixture	
enter	repository	drs_master.xml
enter	querytype	findbyauthor
enter	querymode	isexact
enter	sortorder	title
enter	sortdirection	descending
enter	query	Maurer, Frank
press	find	
enter	select	3
check	author	Read, Kristopher
check	author	Maurer, Frank
check	author	Melnik, Grigori
check	author	Liu, Lawrence
check	title	Advantages of Tablet Usage by Software Development Teams
check	type	tex
check	dateSubmitted	2003-12-25
enter	select	2
check	author	Melnik, Grigori
check	author	Read, Kristopher
check	author	Maurer, Frank
check	title	Distributed Extreme Programming
check	type	rtf
check	dateSubmitted	2003-11-21
enter	select	1
check	author	Maurer, Frank
check	author	Chau, Thomas
check	title	Knowledge Management in Scrum
check	type	pdf
check	dateSubmitted	1999-03-15

Figure 9. A sample FIT test (after execution)

IV.1.4 Sampling

Students of computer science programs from the University of Calgary and the SAIT Polytechnic participated in the experiment (recruited in accordance with the Ethics Board norms – see Appendix A). All individuals were knowledgeable about programming and testing, however, no individuals had any advance knowledge of FIT or FitNesse (based on a verbal poll).

Twenty five (25) senior undergraduate University of Calgary students were enrolled in the course *Web-Based Systems*[10], which introduces the concepts and techniques of building Web-based enterprise solutions and includes comprehensive hands-on software development assignments. Seventeen (17) students from the Bachelor of Applied Information Systems program were enrolled in a similar course, *Internet Software Techniques*[11], at SAIT Polytechnic. The material from both courses was presented consistently by the same instructor in approximately the same time frame. This experiment spans only the first of six assignments involving the construction of a document review system.

Students were encouraged to work on programming assignments following the principles and the practices of extreme programming, including test-first design, collective code ownership, short iterations, continuous integration, and pair programming.

The University of Calgary teams consisted of 4 to 5 members, and additional help was available twice a week from two teaching assistants. SAIT Polytechnic teams had 3 members[12] each; however they did not have access to additional help outside of classroom lectures. In total, there were 12 teams and a total of 42 students.

IV.1.5 Observations

Our first hypothesis was that FIT acceptance tests describe a customer requirement such that a technology expert can implement the feature(s) for that requirement. Our experiment provided strong evidence that customer requirements provided using good acceptance tests can in fact be fulfilled successfully. On average (mean) 82% of customer-provided tests passed in the submitted assignments (SD=35%), and that number increases to 90% if we only

[10] http://mase.cpsc.ucalgary.ca/seng513/W2004/

[11] http://mase.cpsc.ucalgary.ca/apse504/W2004/

[12] SAIT Polytechnic teams had fewer members so that we would have an equal number of teams at each location.

consider the 10 teams who actually made attempts to implement the required FIT tests (SD=24%)[13] (Figure 10). Informal student feedback about the practicality of FIT acceptance tests to define functional requirements also supports our first and second hypotheses. Students generally commented that the FIT tests were an acceptable form of assignment specification[14]. Teams had between 1 and 1.5 weeks to master FIT in addition to implementing the necessary functionality (depending on if they were from SAIT or the University of Calgary).

	University of Calgary						SAIT				
Team	1	2	3	4	5	6	1	2	4	5	6
Customer Tests Pass Ratio	100%	100%	0%	100%	100%	100%	79%	26%	100%	100%	100%

Figure 10. Customer test statistics by teams

Seventy-three percent (73%) of all groups managed to satisfy 100% of customer requirements. Although this refutes our second hypothesis, our overall statistics are nonetheless encouraging. Those teams who did not manage to satisfy all acceptance tests also fell well below the average (46%) for the number of requirements attempted in their delivered product (Fig. 7).

	University of Calgary						SAIT				
Team	1	2	3	4	5	6	1	2	4	5	6
% of Requirements Attempted	87%	55%	42%	77%	42%	68%	32%	10%	59%	32%	35%

Figure 11. Percentage of attempted requirements. An attempt is any code delivered that we evaluate as contributing to the implementation of desired functionality.

[13] One team's data was removed from analysis because of a lack of participation from team members. One other team (included) delivered code but did not provide FIT fixtures.

[14] It should be noted that an academic assignment is not the same as a real-world requirements specification.

Unfortunately, no teams were able to implement and test at least 50% of the additional requirements we had expected. Those requirements defined loosely in prose but given no initial FIT tests were largely neglected both in terms of implementation and test coverage (Figure 12). This disproves our hypothesis that 100% of implemented requirements would have corresponding FIT tests. Although many teams implemented requirements for which we had provided no customer acceptance tests, on average only 13% of those new features were tested (SD=13%). Those teams who did deliver larger test suites (for example, team 2 returned 403% more tests than we provided) mostly opted to expand existing tests rather than creatively testing their new features.

Team	Number New Tests	New Test Pass Ratio	Number New Assertions	New Assertions Pass Ratio	% Additional Tests	% Additional Assertions	% New Features Tested	% Attempted Features Tested	
1	19	100%	208	100%	49%	32%	32%	67%	
2	157	100%	5225	100%	403%	795%	26%	100%	
3	0	0%	0	0%	0%	0%	0%	0%	
4	116	100%	2218	100%	297%	338%	32%	75%	
5	9	100%	99	100%	23%	15%	16%	100%	
6	41	93%	616	95%	105%	94%	37%	100%	University
1	0	0%	0	0%	0%	0%	0%	80%	
2	0	0%	0	0%	0%	0%	0%	100%	
4	56	100%	1085	100%	144%	165%	11%	66%	
5	0	0%	0	0%	0%	0%	0%	100%	
6	5	100%	64	100%	13%	10%	5%	100%	SAIT

Figure 12. Additional features and tests statistics

Customers do not always consider exceptional and deviant cases when designing acceptance tests, and therefore acceptance tests must be evaluated for completeness. Even in our own scenario, all tests specified were positive tests; tests confirmed what the system should do with valid input, but did not explore what the system should do with invalid entries. For example, one test specified in our suite verified the results of a search by file type (.doc, .pdf, etc.). This test was written using lowercase file types, and nowhere was it explicitly indicated that uppercase or capitalized types be permitted (.DOC, .Pdf, etc). As a result, 100% of

teams wrote code that was case sensitive, and 100% of tests failed when given uppercase input.

IV.1.6 Findings

Our hypotheses (A and B) that FIT tests describing customer requirements can be easily understood and implemented by a technology expert with little background on this framework were substantiated by the evidence gathered in this experiment. Considering the short period of time allotted, we can conclude from the high rate of teams who delivered FIT tests (90%) that the learning curve for reading and implementing FIT tests is not prohibitively steep, even for relatively inexperienced developers.

Conversely, our hypotheses that 100% of participants would create code that passed 100% of customer provided tests (C), that more than 50% of the requirements for which no tests were given would be tested (D), and that 100% of implemented requirements would have corresponding FIT tests (E) were not supported. In our opinion, the fact that more SAIT teams failed to deliver 100% of customer tests can be attributed to the slightly shorter time frame and the lack of practical guidance from TA's. The lack of tests for new features added by teams may, in our opinion, be credited to the time limitations placed on students, the lack of motivation to deliver additional tests, and the lower emphasis given to testing in the past academic experiences of these students[15]. At the very least, our observation that feature areas with fewer provided FIT tests were more likely to be incomplete supports the idea that FIT format functional requirements are of some benefit.

The fact that a well defined test suite was provided by the customer up front may have instilled a false sense of security in terms of test coverage. The moment the provided test suite passed, it is possible that students assumed the assignment

[15] Despite the fact that the importance of testing was repeatedly emphasized, students are not accustomed to writing test code. Students were aware that the majority of marks were not being assigned based on new tests.

was complete. This may be extrapolated to industry projects: development teams could be prone to assuming their code is well tested if it passes all customer tests. It should be noted that writing FIT tests is simplified but not simple; to write a comprehensive suite of tests, some knowledge and experience in both testing and software engineering is desirable (for example, a QA engineer could work closely with the customer). It is vital that supplementary testing be performed, both through unit testing and additional acceptance testing. The role of quality assurance specialists will be significant even on teams with strong customer and developer testing participation. Often diabolical thinking and knowledge of specific testing techniques such as equivalence partitioning and boundary value analysis are required to design a comprehensive test suite.

From the outcome of our five hypotheses, along with our own observations and feedback from the subjects, we can suggest how FIT acceptance tests perform as a specification of functional requirements in relation to the criteria stated in our introduction. We believe that *noise* is greatly reduced when using FIT tests to represent requirements. Irrelevant information is more difficult to include in well structured tables than in prose documents. Also, tests which shade or contradict previous tests are easily uncovered at the time of execution (although there is no automatic process to do so). Acceptance tests can be used as regression tests after they have passed in order to prevent problems associated with possible noise. We discovered that *silence* is not well addressed by the FIT framework, and may even become a more serious problem. This was well demonstrated by the failure of our teams to test at least 50% of the requirements for which no tests were given. Our example of case-sensitive document types also clearly demonstrates how a lack of explicit tests can lead to assumptions and a lack of clarifications. Prose documents may be obviously vague, and by this obviousness incite additional communication. *Overspecification* is not a problem since FIT tests do not allow any room for embedded solutions in the tests themselves. FIT tables are only representations of customer expectations, and the fixtures become the agents of the solutions. Although it can be argued that specifying an ActionFixture describes a sequence of actions (and therefore a solution), when writing FIT

tables these actions should be based on business operations and not code-level events. *Wishful thinking* is largely eliminated by FIT, since defining tests requires that the customer think about the problem and make very specific decisions about expectations.

Ambiguity may still be a problem when defining requirements using FIT tests if keywords or fields are defined in multiple places or if these identifiers are open to multiple interpretations. However, FIT diminishes ambiguity simply because it uses fewer words to define each requirement. *Forward references* and *oversized documents* may still be an issue if large numbers of tests are present and not organized into meaningful test suites. In our experiment, the majority of groups categorized their own tests without any instruction to do so. *Reader subjectivity* is greatly reduced by FIT tests. Tables are specified using a format defined by the framework (*ActionFixture*, *ColumFixture*, etc). As long as tests return their expected results when executed, the technology expert or business expert knows that the corresponding requirement was correctly interpreted regardless of the terminology used. *Customer uncertainty* may manifest as the previously mentioned problem of silence, but it is impossible for a defined FIT test not to have a certain outcome. FIT tests are executable, verifiable and easily readable by the business expert and technology expert, and therefore there is no need for *multiple representations* of requirements. All necessary representations have effectively merged into a suite of tables. Requirements gathering *tools* can be problematic when they limit the types of requirements that can be captured. FIT is no exception; it can be difficult to write some requirements as FIT tests, and it is often necessary to extend the existing set of fixtures, or to utilize prose for defining non-functional requirements and making clarifications. However, FIT tests can be embedded in prose documents or defined through a collaborative wiki such as FitNesse, and this may help overcome the limitations of FIT tables.

In addressing the characteristics of suitability (as defined in Introduction), our findings demonstrate that FIT tests as functional requirements specifications are in fact unambiguous, verifiable, and usable (from the technology expert's

perspective). However, insufficient evidence was gathered to infer consistency between FIT tests.

Although our results did not match all of our expectations, valuable lessons were learned from the data gathered. When requirements are specified as tests, there is still no guarantee that the requirements will be completed on-time and on-budget. Time constraints, unexpected problems, lack of motivation and poor planning can still result in only some requirements being delivered. As with any type of requirements elicitation, it is vital that the customer is closely involved in the process. FIT tests can be executed by the customer or in front of the customer, and customers can quickly evaluate project progress based on a green (pass) or red (fail) condition. In conclusion, our study provides only initial evidence of the suitability of FIT tests for specifying functional requirements. This evidence directly supports the understandability of this type of functional requirements specification by technology experts. There are both advantages and disadvantages to adopting FIT for this purpose, and the best solution is probably some combination of both prose-based and FIT-based specifications.

IV.1.7 Validity

There are several possible threats to the validity of this experiment that should be reduced through future experiments. One such threat is the limitation of our experiment to a purely academic environment. Although we spanned two different academic institutions, industry participants would be more relevant. Another threat is our small sample size, which can be increased through repeated experiments in future semesters. Moreover, all of the FIT tests provided in this experiment were written by expert researchers, which would not be the case in an industrial setting. Although this was an academic assignment, it was not conducted in a controlled environment. Students worked in teams on their own time without proper invigilation.

IV.2 Academic Study Two: Patterns of Authoring and Organizing Executable Acceptance Tests

IV.2.1 Objectives

In this study we expand on the results from the first academic study (§IV.1) and investigate the ways in which technology experts use executable acceptance tests. We seek to identify usage patterns and gather information that may lead us to better understand the strengths and weaknesses of acceptance tests when used for both quality control and requirements representation. Further, examining and identifying patterns may allow us to provide recommendations on how acceptance tests can best be used in practice, as well as for future development of tools and related technologies. Here we report on results of observations in an academic setting. This exploratory study allowed us to refine hypotheses and polish the design for future industrial studies.

IV.2.2 Context of Study

Data was gathered from two different projects in two different educational institutions over four months. The natures of the two projects were somewhat different; one was an interactive game, and another a Web-based enterprise information system. The development of each project was performed in several two to three week long iterations. In each project, FIT was introduced as a mandatory requirement specification tool. In one project, FIT was introduced immediately, and in the other FIT was introduced in the third iteration (half way through the semester). After FIT was introduced, technology experts were required to interpret the FIT-specified requirements supplied by the instructor. They then implemented the functionality to make all tests pass, and were asked to extend the existing suite of tests with additional scenarios.

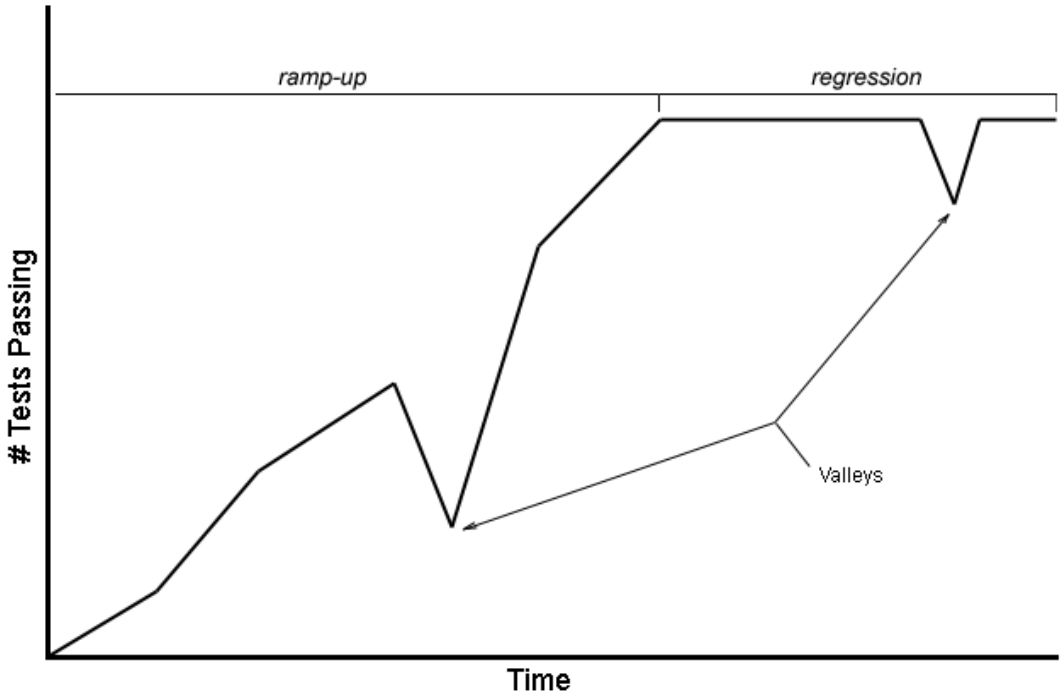


Figure 13. Typical iteration life-cycle

The timeline of both projects can be split into two sections (see Figure 13). The first time period begins when students received their FIT tests, and ends when they implemented fixtures to make all tests pass. Henceforth this first time period will be called the "ramp up" period. Subjects may have used different strategies during ramp up in order to make all tests pass, including (but not limited to) implementing business logic within the test fixtures themselves, delegating calls to business logic classes from test fixtures, or simply mocking the results within the fixture methods (Table 5).

The second part of the timeline begins after the ramp up and runs until the end of the project. This additional testing, which begins after all tests are already passing, is the use of FIT for regression testing. By executing tests repeatedly, technology experts can stay alert for new bugs or problems which may become manifest as they make changes to the code. It is unknown what types of changes our subjects might make, but possibilities range from refactoring to adding new functionality.

Table 5. Samples of Fixture Implementations

Example: In-fixture implementation

```
public class Division extends ColumnFixture {
  public double numerator, denominator;
  public double quotient() {
      return numerator/denominator;  }
}
```

Example: Delegate implementation

```
public class Division extends ColumnFixture {
  public double numerator, denominator;
  public double quotient() {
    DivisionTool dt = new DivistionTool();
    return dt.divide(numerator, denominator);
  }
}
```

Example: Mock implementation

```
public class Division extends ColumnFixture {
  public double numerator, denominator;
  public double quotient() { return 8; }
}
```

IV.2.3 Subjects and Sampling

Students of computer science programs from the University of Calgary (UofC) and the SAIT Polytechnic (SAIT) participated in the study. All individuals were knowledgeable about programming, however, no individuals had any knowledge of FIT or FitNesse (based on a verbal poll). Senior undergraduate UofC students (20) who were enrolled in the Web-Based Systems[16] course and students from the Bachelor of Applied Information Systems program at SAIT (25) who enrolled the Software Testing and Maintenance course, took part in the study. In total, 10 teams with 4-6 members were formed.

IV.2.4 Hypotheses

The following hypotheses were formulated prior to beginning our observations:

16 http://mase.cpsc.ucalgary.ca/seng513/F2004

A) No common patterns of ramp up or regression would be found between teams working on different projects in different contexts.

B) Teams will be unable to identify and correct "bugs" in the test data or create new tests to overcome those bugs (with or without client involvement).

C) When no external motivation is offered, teams will not refactor fixtures to properly delegate operations to business logic classes.

D) Students will not use both suites and individual tests to organize/run their tests.

IV.2.5 Data Gathering

A variety of data gathering techniques were employed in order to verify hypotheses and to provide further insight into the usage of executable acceptance testing. Subjects used FitNesse for defining and executing their tests. For the purposes of this study, we provided a binary of FitNesse that was modified to track and record a history of FIT test executions, both successful and unsuccessful. Specifically, we recorded:

- Timestamp;
- Fully-qualified test name (with test suite name if present);
- Team;
- Result: number right, number wrong, number ignored, number exceptions.

The test results are in the format produced by the FIT engine. *Number right* is the number of passed assertions, or more specifically the number of "green" table cells in the result. *Number wrong* is the number of failed assertions, which are those assertions whose output was different from the expected result. In FIT this is displayed in the output as "red" table cells. *Ignored* cells were for some reason skipped by the FIT engine (for example due to a formatting error). *Number exceptions* records exceptions that did not allow a proper pass or fail of an

assertion. It should be noted that a single exception if not properly handled could halt the execution of subsequent assertions. In FIT exceptions are highlighted as "yellow" cells and recorded in an error log. We collected 25,119 different data points about FIT usage.

Additional information was gathered by inspecting the source code of the test fixtures. Code analysis was restricted to determining the type of fixture used, the non-commented lines of code in each fixture, the number of fields in each fixture, the number of methods in each fixture, and a subjective rating from 0 to 10 of the "fatness" of the fixture methods: 0 indicating that all business logic was delegated outside the fixture (desirable), and 10 indicating that all business logic was performed in the fixture method itself (see Table 5 for examples of fixture implementations).

Analysis of all raw data was performed subsequent to course evaluation by an impartial party with no knowledge of subject names (all source code was sanitized). Data analysis had no bearing or effect on the final grades.

IV.2.6 Analysis

This section is presented in four parts, each corresponding to a pattern observed in the use of FIT. *Strategies of test fixture design* looks at how subjects construct FIT tables and fixtures; *Strategies for using test-suites vs. single tests* examines organization of FIT tests; *Development approaches* identifies subject actions during development; and *Robustness of test specification* analyzes how subjects deal with exceptional cases.

IV.2.6.1 Strategies of Test Fixture Design

It is obvious that there are multitudes of ways to develop a *fixture* (a simple interpreter of the table) such that it satisfies the conditions specified in the table (test case). Moreover, there are different strategies that could be used to write the same fixture. One choice that needs to be made for each test case is what type of FIT fixture best suits the purpose. In particular, subjects were introduced to RowFixtures and ActionFixtures in advance, but other types were also used at

discretion of the teams (see Table 6). Some tests involved a combination of more than one fixture type, and subjects ended up developing means to communicate between these fixtures.

Table 6. Common FIT Fixtures Used by Subjects

Fixture Type	Description	Frequency of Use
RowFixture	Examines an order-independent set of values from a query.	12
ColumnFixture	Represents inputs and outputs in a series of rows and columns.	0
ActionFixture	Emulates a series of actions or events in a state-specific machine and checks to ensure the desired state is reached.	19
RowEntryFixture	Special case of ColumnFixture that provides a hook to add data to a dataset.	2
TableFixture	Base fixture type allowing users to create custom table formats.	30

Another design decision made by teams was whether to develop "fat", "thin" or "mock" methods within their fixtures (Table 7). "Fat" methods implement all of the business logic to make the test pass. These methods are often very long and messy, and likely to be difficult to maintain. "Thin" methods delegate the responsibility of the logic to other classes and are often short, lightweight, and easier to maintain. Thin methods show a better grasp on concepts such as good design and refactoring, and facilitate code re-use. Finally, "mock" methods do not implement the business logic or functionality desired, but instead return the expected values explicitly. These methods are sometimes useful during the development process but should not be delivered in the final product. The degree to which teams implemented fat or thin fixtures was ranked on a subjective scale of 0 (entirely thin) to 10 (entirely fat).

The most significant observation that can be made from Table 7 is that the UofC teams by and large had a much higher fatness when compared to the SAIT teams. This could possibly be explained by commonalities between strategies used at each location. At UofC, teams implemented the test fixtures in advance of any

other business logic code (more or less following Test-Driven Development philosophy [133]). Students may not have considered the code written for their fixtures as something which needed to be encapsulated for re-use. This code from the fixtures was further required elsewhere in their project design, but may have been "copy-and-pasted". No refactoring was done on the fixtures in these cases. This can in our opinion be explained by a lack of external motivation for refactoring (such as additional grade points or explicit requirements). Only one team at the UofC took it upon themselves to refactor code without any prompting. Conversely, at SAIT students had already implemented business logic in two previous iterations, and were applying FIT to existing code as it was under development. Therefore, the strategy for refactoring and maintaining code re-use was likely different for SAIT teams. In summary, acceptance test driven development failed to produce reusable code in this context. Moreover, in general, teams seem to follow a consistent style of development – either tests are all fat or tests are all thin. There was only one exception in which a single team did refactor some tests but not all (see Table 7, UofC T2).

IV.2.6.2 Strategies for Using Test Suites vs. Single Tests

Regression testing is undoubtedly a valuable practice. The more often tests are executed, the more likely problems are to be found. Executing tests in suites ensures that all test cases are run, rather than just a single test case. This approach implicitly forces technology experts to do regression testing frequently. Also, running tests as a suite ensures that tests are compatible with each other – it is possible that a test passes on its own but will not pass in combination with others.

Table 7. Statistics on Fixture Fatness and Size

Team	Fatness (subjective:0-10)		NCSS[17]	
	Min	Max	Min	Max
UofC T1	7	10	28	145
UofC T2	0	9	8	87
UofC T3	8	10	40	109
UofC T4	9	10	34	234
SAIT T1	0	1	7	57
SAIT T2	0	2	22	138
SAIT T3	0	0	24	57
SAIT T4	0	0	15	75
SAIT T5	1	2	45	91
SAIT T6	0	1	13	59

In this experiment data on the frequency of test suite vs. single test case executions was gathered. Teams used their own discretion to decide which approach to follow (suites or single tests or both). Several strategies were identified (see Table 8).

Table 8. Possible Ramp-Up Strategies

Strategy	Pros	Cons
(*) Exclusively using single tests	- fast execution - enforces baby steps development	- very high risk of breaking other code - lack of test organization
(**) Predominantly using single tests	- fast most of the time execution - occasional use of suites for regression testing	- moderate risk of breaking other code
(***) Relatively equal use of suites and single tests	- low risk of breaking other code - immediate feedback on the quality of the code base - good organization of tests	- slow execution when the suites are large

[17] NCSS is Non-Comment Source Lines of Code, as computed by the JavaNCSS tool: http://www.kclee.de/clemens/java/javancss/

Exclusively using single tests may render faster execution; however, it does not ensure that other test cases are passing when the specified test passes. Also, it indicates that no test organization took place which may make it harder to manage the test base effectively in the future. Two teams (one from UofC and one from SAIT) followed this approach of single test execution (Table 9). Another two teams used both suites and single tests during the ramp up. A possible advantage of this strategy may be a more rapid feedback on the quality of the entire code base under test. Five out of nine teams followed the strategy of predominantly using single test, but occasionally using suites. This approach provides both organization and infrequent regression testing. Regression testing using suites would conceivably reduce the risk of breaking other code. However, the correlation analysis of our data finds no significant evidence that any one strategy produces fewer failures over the course of the ramp up. The ratio of peaks and valleys (in which failures occurred and then were repaired) over the cumulative test executions fell in the range of 1-8% for all teams. Moreover, even the number of test runs is not correlated to strategy chosen.

Table 9. Frequency of Test Suites vs Single Test Case Executions during Ramp Up

Team	Suite Executions	Single Case Executions	Single/Suite Ratio
UofC T1 (***)[18]	650	454	0.70
UofC T2 (***)	314	253	0.80
UofC T3 (**)	169	459	2.72
UofC T4 (*)	0	597	Exclusively Single Cases
SAIT T1 (**)	258	501	1.94
SAIT T2 (**)	314	735	2.40
SAIT T3 (**)	49	160	3.27
SAIT T4 (*)	8	472	59.00
SAIT T5 (**)	47	286	6.09
SAIT T6 (not included due to too few data points).	8	25	3.13

[18] Using ramp-up strategies as per Table 8.

During the regression testing stage we also measured how often suites versus single test cases were executed (Table 10). For UofC teams, we saw a measured difference in how tests were executed after the ramp up. All teams now executed single test cases more than suites. Team 1 and Team 2 previously had executed suites more than single cases, but have moved increasingly away from executing full test suites. This may be due to troubleshooting a few problematic cases, or may be a result of increased deadline pressure. Team 3 vastly increased how often they were running test suites, from less than half the time to about three-quarters of executions being performed in suites. Team 4 who previously had not run any test suites at all, did begin to run tests in an organized suite during the regression period. For SAIT teams we see a radical difference in regression testing strategy: use single test case executions much more than test suites. In fact, the ratios of single cases to suites are so high as to make the UofC teams in retrospect appear to be using these two types of test execution equally. Obviously, even after getting tests to pass initially, SAIT subjects felt it necessary to individually execute far more individual tests than the UofC students. Besides increased deadline pressure, a slow development environment might have caused this.

Table 10. Frequency of Suites vs Single Test Case Executions during Regression (Post Ramp Up)

Team	Suite Executions	Single Case Executions	Single/Suite Ratio
UofC T1	540	653	1.21
UofC T2	789	1042	1.32
UofC T3	408	441	1.08
UofC T4	72	204	2.83
SAIT T1	250	4105	16.42
SAIT T2	150	3975	26.50
SAIT T3	78	1624	20.82
SAIT T4	81	2477	30.58
SAIT T5	16	795	49.69
SAIT T6	31	754	24.32

IV.2.6.3 Development Approaches

The analysis of ramp up data demonstrates that all teams likely followed a similar development approach. Initially, no tests were passing. As tests are continued to be executed, more and more of the assertions pass. This exhibits the iterative nature of the development. We can infer from this pattern that features were being added incrementally to the system (Figure 14, left). Another approach could have included many assertions initially passing followed by many valleys during refactoring. That would illustrate a mock-up method in which values were faked to get an assertion to pass and then replaced at a later time (Figure 14, right).

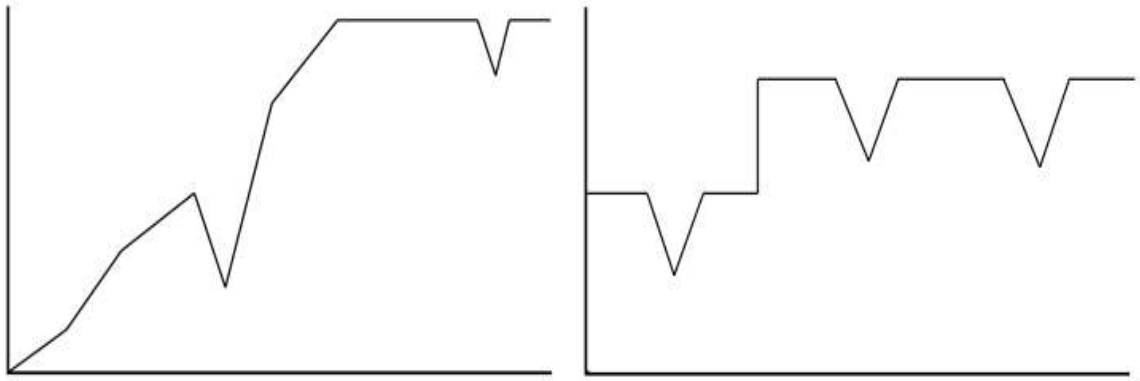


Figure 14. A Pattern of What Incremental Development might Look Like (Left) versus What Mocking and Refactoring might Look Like (Right); (horizontal axis = time, vertical = # passing tests)

Noticeably, there were very few peaks and valleys[19] during development (Table 11). A valley is measured when the number of passing assertions actually goes down from a number previously recorded. Such an event would indicate code has broken or an error has occurred. These results would indicate that in most cases as features and tests were added, they either worked right away or did not break previously passing tests. In our opinion, this is an indication that because the tests were specified upfront, they were driving the design of the project. Because

[19] The number of peaks equals the number of valleys. Henceforth we refer only to valleys.

subjects always had these tests in mind and were able to refer to them frequently, they were more quality conscious and developed code with the passing tests being the main criteria of success.

IV.2.6.4 Robustness of the Test Specification

Several errors and omissions were left in the test suite specification delivered to subjects. Participants were able to discover all such errors during development and immediately requested additional information. For example, one team posted on the experience base the following question: *"The acceptance test listed … is not complete (there's a table entry for "enter" but no data associated with that action). Is this a leftover that was meant to be removed, or are we supposed to discover this and turn it into a full fledged test?"* In fact, this was a typo and we were easily able to clarify the requirement in question. Surprisingly, typos or omissions did not seem to affect subjects' ability to deliver working code. This demonstrates that even with errors in the test specification, FIT adequately describes the requirements and makes said errors immediately obvious to the reader.

Table 11. Ratio of Valleys Found vs Total Assertions Executed

Team	"Valleys" vs. Executions in Ramp Up Phase	"Valleys" vs. Executions in Regression Phase
UofC T1	0.03	0.05
UofC T2	0.07	0.10
UofC T3	0.03	0.10
UofC T4	0.01	0.05
SAIT T1	0.06	0.12
SAIT T2	0.03	0.10
SAIT T3	0.04	0.09
SAIT T4	0.05	0.06
SAIT T5	0.05	0.09
SAIT T6	0.03	0.14

IV.2.7 Academic Study Two Summary

Our observations lead us to the following conclusions. Our hypothesis A that no common patterns of ramp up or regression would be found between teams working on different projects in different contexts was only partly substantiated. We did see several patterns exhibited, such as incremental addition of passing assertions and a common use of preferred FIT fixture types. However, we also saw some clear divisions between contexts, such as the relative "fatness" of the fixtures produced being widely disparate. The fixture types students used were limited to the most basic fixture type (TableFixture) and the two fixture types provided for them in examples. This may indicate that rather than seeing a pattern in what fixture types subjects chose, we may need to acknowledge that the learning curve for other fixture types discouraged their use. Subjects did catch all "bugs" or problems in the provided suite of acceptance tests, refuting our hypothesis B and demonstrating the potential for implementing fixtures despite problems. Hypothesis C, that teams would not refactor fixtures to properly delegate operations to business logic classes, was confirmed. In the majority of cases, when there was no motivation to do so students did not refactor their fixture code but instead had the fixtures themselves perform business operations. Subjects were aware that this was bad practice but only one group took it upon themselves to "do it the right way". Sadly, the part of our subject pool that was doing test-first was most afflicted with "fat" fixtures, while those students who were writing tests for existing code managed by large to reuse that code. In all cases, students used both suites and individual test cases when executing their acceptance tests (refuting our hypothesis D). However, we did see that each of the groups decided for themselves when to run suites more often than single cases and vice versa. It is possible that these differences were the result of strategic decisions on behalf of the group, but also possible that circumstance or level of experience influenced their decisions.

Our study demonstrated that subjects were able to interpret and implement FIT test specifications without major problems. Teams were able to deliver working code to make tests pass and even catch several bugs in the tests themselves.

Given that the projects undertaken are similar to real world business applications, we suggest that lessons learned from this study are likely to be applicable to an industrial setting. Professional developers are more experienced with design tools and testing concepts, and, therefore, would likely overcome minor challenges with as much success as our subjects (if not more).

IV.3 Academic Study Three: Business Experts' Perspective

IV.3.1 Impetus

One of the limitations of the earlier studies (including the one described in §IV.1 and §IV.2) was the use of software engineering undergraduate students to specify acceptance tests. Though some of them may be involved with the requirements specification process in the future, they served as a poor sample of the customer population. A better representation was needed. To address this problem, in this study, we tried to approximate business customers by including both business school graduate students and computer science graduate students as our customer representatives.

IV.3.2 Research questions

Our research questions pertain to both the customer team's capability and the substance of the acceptance tests produced, specifically:

Q_1: Can customers specify functional business requirements in the form of executable acceptance tests clearly when paired with an IT professional?

Q_2: How do customers use FIT for authoring business requirements?

Q_3: What are the trends in customer-authored executable acceptance test-based specifications?

Q_4: Does a software engineering background have an effect on the quality of the executable acceptance test-based specification?

Q_5: Is executable acceptance test-driven development a satisfactory method for customers, based on their satisfaction, their intention on using it in the future, and their intention to recommend it to other colleagues?

IV.3.3 Research design and methodology

IV.3.3.1 Participants

Three groups of University of Calgary students were involved in the study (see Table 12):

- Business school graduate students (further denoted as "*Business-grads*") enrolled in a Master of Business Administration program, taking a course in e-business as one of their elective courses.
- Computer Science graduate students plus one Computer Engineering graduate student ("*Computer-grads*"), typically enrolled in their first year of a Master's degree program, and enrolled in the same course with the Business-grads. Most of them had prior experience in the software industry.
- Senior Computer Science and Computer Engineering undergraduate students ("*Computer-undergrads*") enrolled in a separate course from the other two groups, on enterprise Web-based systems.

Both the graduate and undergraduate courses ran during the same term (Fall 2005).

Table 12. Sample, Programs, and Courses.

Abbreviation	Major	Course	URL	Role	% female	# participants	Team size
Business-grad	Management Information Systems	Enabling E-Business	http://ebe.cpsc.ucalgary.ca/ebe/Wiki.jsp?page=CPSC_601_11_MGIS_797_03_F2005	Business expert	17%	6	2
Computer-grad	Computer Science/ Computer Engineering	Enabling E-Business	http://ebe.cpsc.ucalgary.ca/ebe/Wiki.jsp?page=CPSC_601_11_MGIS_797_03_F2005	Business expert	33%	12	
Computer-undergrad	Computer Science/ Computer Engineering	Web-Based Systems	http://mase.cpsc.ucalgary.ca/seng513/F2005	Technology expert	9%	22	2–3

The graduate students (*Business-grads* and *Computer-grads*) formed customer teams and specified requirements for a Web-based project management system; while the undergrad students formed development teams who were responsible for implementing requirements specified by the customer teams. The system was deliberately chosen such that its requirements would be more accessible to the business students and not as apparent to the undergraduates.

Graduate students self-organized into teams of two, with only one constraint: only one Business-grad was allowed per team. As a result, given differences in business and computer science graduates, nine customer teams were formed, three of which were purely comprised of Computer-grads, while six were a mix of one Business-grad and one Computer-grad. Undergraduate students also self-organized into an equal number of development teams. The total number of teams involved in the research included 9 customer teams and 9 development teams. As a result, there were a total of 18 customer-subjects, and the total number of all participants (including development teams) was 40.

With the exception of one person, all members of the customer teams were mature students and had related industrial experience, with mode being "more than 5 years" and median being "3-5 years". Female/male ratio of subjects was: 1/5 for Business-grads, 4/8 for Computer-grads, and 2/20 for Computer-undergrads.

IV.3.3.2 Method

A quasi-experiment [124, 12] was used as a basis for our research design. The choice was motivated by the use of nonrandom sample (convenience sample of graduate students) and a small size (18 students, 9 teams, out of which one team was disqualified due to their poor participation). The primary source of evidence about the sample's representation to the population of customers is that the majority of subjects had more than 5 years of industrial experience and all were trained in either Management Information Systems or Computer Science. The assignment of development teams to customer teams was also random.

In this research, we primarily sought trends rather than cause-effect relationships.

IV.3.3.3 Hypotheses

Our central hypothesis was that "Customers in partnership with an IT professional would be able to effectively specify functional requirements of the system in the form of executable acceptance tests". We took "effectively" to mean that we would see evidence of "good" tests and thorough coverage of the major system functionality (which would result in a grade of 75% or higher). In order to be "good", acceptance tests had to satisfy the following criteria (these are based on [71]):

- credible (contain realistic and reasonable set of operations to be likely performed by the customer);
- appropriate complexity (involve many features, attributes, workflows, etc.);
- coverage of the major functionality for a Human Resources Intranet system (management of projects, subprojects, time sheets, time-sheet allocation to projects, reporting of cumulative project time, expense claims management; administrative staff features);
- easy to read and informative;

- easy to manage (packaged in meaningfully structured suites, subsuites etc.).

The role of "an IT professional" was performed by the Computer Science graduate students, most of whom had prior work experience in the software industry.

Additionally, we hypothesized that:

A) *Customers with no previous experience with executable acceptance testing or FIT will find it easy to learn how to use FIT given the time provided.* Learnability is to be determined based on individual perceptions (scored on the Likert-scale). Time was provided included to learn the FIT system in a three-hour in-class tutorial by a FIT expert and four-weeks of practice.

B) *Customers will specify predominantly positive test cases.* By "predominantly", we mean 90% or more of all test cases being positive, and 10% or less being negative (i.e. testing various error conditions).

C) *There exists a significant difference in the quality of the specifications produced by customer teams comprised of two members with Computer Science background, vs. customer teams comprised of one Business-graduate and one Computer-graduate.* In this case, the response variable was a subjective grade based on consensus evaluation by two instructors (blinded to the student composition in each group).

D) *The quality of the executable acceptance test specification is strongly and positively correlated with the quality of the implementation produced by the development team.* Correlation is to be determined by calculating Spearman's correlation coefficient. We use the following commonly accepted interpretation ranges: <0.1 – none, 0.1-0.3 – weak, 0.3-0.5 – moderate, 0.5-0.8 – strong, >0.8 – very strong.

IV.3.3.4 Procedure

A project was conceived by the instructors to develop a Human Resources Intranet system to manage projects, consultants' time sheets, and expense claims (see Figure 15). A one-page narrative was given to the customer teams to provide initial ideas about the type of the system to be developed. This was a high-level, generic outline of the vision of what the system was supposed to achieve. No particulars were given and customer teams were free to decide on the business constraints and rules of the system. It was critical to the experiment that the system chosen was more accessible to the business students and not as apparent to the undergraduates to eliminate a potential experience bias.

A three-hour tutorial on executable acceptance testing, the FIT framework and the FitNesse tool was offered to the customer teams. It was attended by all subjects. The tutorial demonstrated the use of FIT for specifying two types of business rules: a) transactions, workflows and processes (with DoFixture type tables); and b) decision tables and business calculations (with ColumnFixtures type tables). The use of RowFixture for business queries and reporting features was left for self-study using the FIT/FitNesse documentation available online [38, 40]. A case study was used to illustrate the framework during the tutorial. An expert on FIT (Author of the dissertation) was available for consultation both in person and via email.

> Deltoid Consulting is a consulting company, headquartered in Calgary, with over 120 IT and business consultants. They are hired by major corporations in the development of large IT-enabled systems for streamlining business processes. Their engagements with clients take them all over North America. Ironically, Deltoid has trouble managing **time sheets** and **expense claims** for its large consulting staff, and requires an information system to support this task.

Figure 15. Project Mission Statement.

Three of the subjects had prior experience with the framework. The rest of the customers had no experience with the authoring of executable acceptance tests in FIT or any other format. Table 13 depicts level of experience of subjects with other requirement specification techniques.

Customer teams in the graduate class were required to specify suites of executable acceptance tests so that the development teams in the undergraduate course could implement a system. To motivate exploration and to eliminate a fear of potentially damaging the test artifacts, customers were informed that all test suites were version-controlled and that it was possible to revert any changes they made at any time.

Table 13. Summary of Knowledge Levels of Customers' Experiences with Various Requirement Specification Techniques.

	Median	Mode	Min	Max
Narrative/prose	2	2	0	4
Use cases	2	3	0	4
User stories	1	1	0	4
Domain-specific languages	1	0	0	3
Scenarios	2	2	0	4
Stroyboards	1	1	0	4
Prototypes	2	3	0	4
Mind maps	0	0	0	4
Personas	0	0	0	4
System archeology	0	0	0	3
Executable acceptance tests/FIT	0	0	0	4
Other	0	0	0	0

Note: N=17, legend: 0 = unfamiliar, 1 - some knowledge, 2 - average knowledge, 3 - good knowledge, 4 - extensive knowledge

Development teams were given a one-paragraph mission statement (no more detailed than outlined in Figure 15). Detailed requirements were to be given in the form of executable acceptance tests by the customer teams. Development teams were instructed that their designs and implementations should be driven

by those tests. If they were unclear about a requirement, they were encouraged to communicate directly with their customer by any means they deemed useful. Development teams were also warned about the potential change of the requirements and their responsibility was to adapt their code to that change if it occurred.

To allow communication between the customer and development teams, each customer-development team pair had their own dedicated password-protected Wiki-based virtual space that also contained all acceptance tests (9 instances of FitNesse servers were running on different ports of a centralized server machine).

Considering the time limitation of the customer team members (they could only be involved with the project – on a part time basis – for 4 weeks), this study only covers the first iteration of the project. During this iteration, development teams were required to design a content model for the artifacts to be used by the system (in the form of XML Schemas), to build the necessary XSLT sheets for performing queries and renderings on XML raw data as per customer requirements, and to implement the XML processing logic necessary to satisfy customer requirements.

All test and coding artifacts were archived and analyzed after both courses were completed and final grades were assigned.

In addition, two questionnaires were administered (pre- and post-iteration). The objective of the pre-questionnaire was to collect data on the prior experience of the customer teams and their familiarity with various requirement specification techniques. The objective of the post-questionnaire was to gain a better understanding on how customer teams accomplished their task of specifying business requirements with executable acceptance tests and to gather qualitative feedback on various aspects of using the framework, various requirement “sins”, communication with the development team, and their perceptions of the executable acceptance test-based specification technique.

Participation in this research was voluntary. Subjects were permitted to withdraw their data from the study (though nobody did). Knowledge of their participation remained anonymous until after the course grades were submitted.

IV.3.4 Findings

IV.3.4.1 Central hypothesis: Customers in partnership with an IT professional can effectively specify acceptance tests

Our central hypothesis was that customers would be able to effectively describe functional requirements of the system in the form of executable acceptance tests so that a development team could implement the features for those requirements. The evaluation of the quality of executable acceptance test specifications ("spec scores") produced by the customer teams are shown in Table 14 . Notice, one team was excluded from the data analysis because both customers acknowledged their lack of participation in this project due to other heavy commitments and time constraints. Using a one-sample t-test, we wish to test whether the mean of spec score differs significantly from 75% (in accordance with our central hypothesis formulation). Specifically, we test: H_0: $\mu = 75.00$, H_A: $\mu > 75.00$. The results of this one-sided t-test are as follows: t=2.873, df=7, p=0.012, mean=91.56, mean difference=16.56. Since the computed t-value (2.873) is larger than the critical value for t-distribution with 7 degrees of freedom at the 5% significance level (1.895) (as per [128]), we reject the null hypothesis and accept the alternative hypothesis. The mean of the spec score variable for our sample of teams is 91.56, which is statistically significantly different from the test value of 75.00. We conclude that this supports our central hypothesis at the 5% significance level, i.e. the sampled group of teams has a significantly higher mean on the quality of executable acceptance test specifications than 75%.

IV.3.4.2 Learnability and ease-of-use of FIT and FitNesse

To remove prior knowledge bias, we have excluded from analysis three students who have characterized their level of knowledge of executable acceptance testing and FIT as "good" or "extensive". The total number of responses evaluated for this hypothesis was 14.

As can be seen from Figure 16, half of the customer team members found it hard to learn FIT, thus rejecting our Hypothesis A that customers with no previous experience with executable acceptance testing or FIT will find it easy to learn how to use FIT given the time provided. Furthermore, four subjects (29%) found FIT to be easy to use and seven subjects (50%) found FitNesse easy to use as well. This speaks to the usability aspect of FitNesse, which is based on a Wiki and follows very simple syntax rules. FitNesse also allows test specification, management, and execution though a consistent centralized Web interface. Notice, these evaluations are made by the customer teams only, and they do not

Table 14. Evaluation of the Quality of Specification and the Quality of Implementation

Customer Team	Team Type	Spec score, /100	Code score, /100
1	Business+CompSci	100.0	94.0
2	Business+CompSci	75.0	67.0
3	Business+CompSci	57.5	71.0
4	Business+CompSci	100.0	79.0
5*	Business+CompSci	100.0	80.0
6	CompSci+CompSci	100.0	89.0
7	CompSci+CompSc	100.0	70.0
8	CompSci+CompSc	100.0	70.0
*9***	*Business+CompSci*	*42.5*	*89.0**
	Mean	91.56	77.50
	SD	16.31	9.84
	Median	100.00	75.00

Notes: * Team 5 produced an exceptional suite of acceptance tests.

** Team 9 was excluded from the analysis due to lack of participation.

address the issue of usability of FIT/FitNesse from the technology expert's perspective (i.e. those responsible for writing the "glue" in the form of fixtures).

Spearman's rho calculation shows a statistically significant correlation (rho_s=0.549, p=0.021, N=14) between the FIT learnability and the program subjects were enrolled in (coding used: 0 = Business-grads, 1 = Computer-grads). As expected, Computer-grads found it easier to learn FIT than Business-grads. Furthermore, a statistically significant strong positive correlation between learnability and FIT ease-of-use was found at p=0.001 (rho_s=0.789, N=13). In other words, those customers who thought it was easy to learn FIT, found it also easy to use. We arrived to similar conclusions with regard to the ease of use of FitNesse (rho_s=0.834, p=0.001, N=14).

Further analysis of correlations between FIT learnability, FIT ease-of-use, and FitNesse ease-of-use with prior work experience, presents no significant evidence of such relationships.

To sum up, based on the results with highly trained Business-grads and Computer-grads, it seems that an average customer representative may experience similar difficulties in learning the FIT framework. However, once the learning curve has been surpassed, subjects find both FIT and FitNesse easy to use and they produce good-quality specifications (as Table 14 shows).

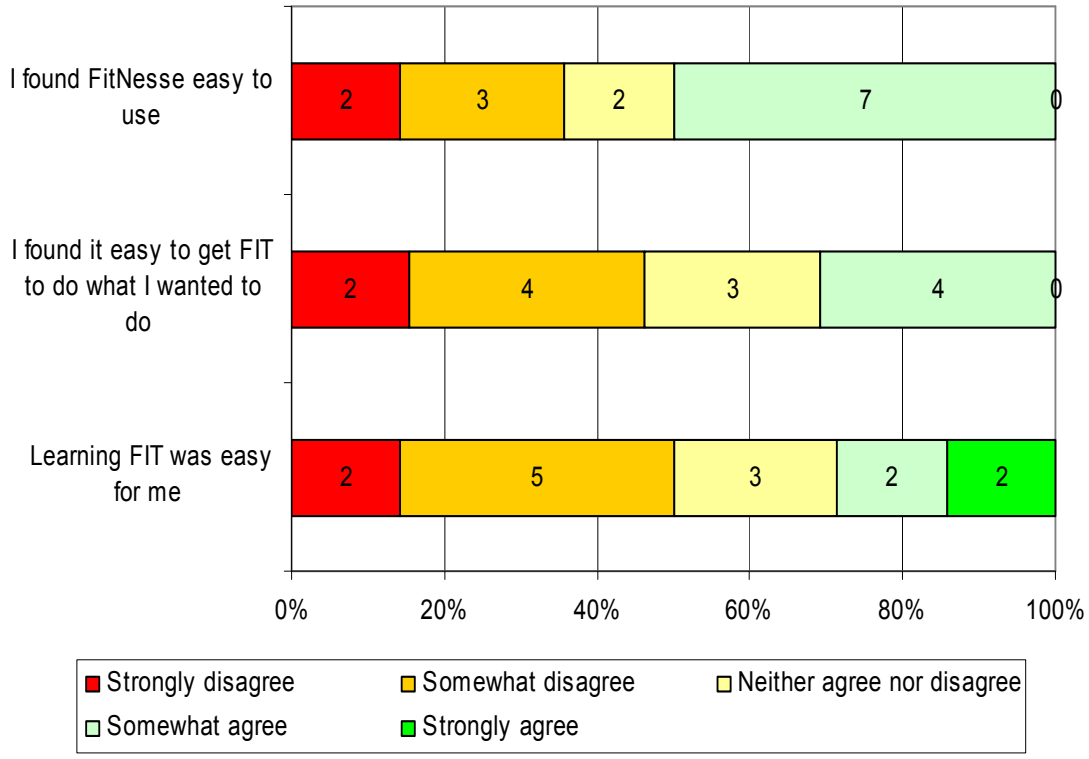


Figure 16. Learnability and Ease-of-Use

IV.3.4.3 Positive vs negative test cases

Negative test cases identify how a system responds to incorrect or inappropriate information or action. Negative testing is performed to ensure that the system is able to handle inconsistent information. Negative acceptance tests (often expressed in the form of negative scenarios) are increasingly recognized as a powerful way of thinking about requirements, possible conflicts, and identifying threats [5, 4].

In our experiment, we hypothesized that positive tests (the normal flows of logic) would dominate. Indeed, negative tests (that would deal with deviance from the course of action or misuse of the system) accounted only for 6% of all tests (as can be evidenced from Table 15). Additionally, Table 15 contains descriptive statistics of the total number of test pages (those would be the test pages in FitNesse) and total number of test/test cases (of which there could be few on a single test page). A one-sample t-test for test value=80% renders the following

results: t=3.366, df=7, p=0.006, mean=90.86. This statistically supports our Hypothesis B that positive tests are prevalent in the executable acceptance test specification written by the customers.

Having examined the patterns of negative vs positive test cases among the pure Computer-grad customer teams and mixed teams (Computer-grad + Business-grad), no evidence was found that pure Computer-grad teams would produce larger number of negative tests; though one would expect that to be the case since Computer-grad students should be familiar with testing techniques and aware of the need for negative test cases.

Table 15. Test Page and Test Case Type Distributions.

Customer team	Total, Test Pages	# Negative tests	% Negative of Total	# Positive tests	% Positive of Total	Total tests
1	31	18	26%	52	74%	70
2	48	14	6%	220	94%	234
3	44	0	0%	199	100%	199
4	11	19	14%	121	86%	140
5	118	1	0%	490	100%	491
6	6	39	18%	181	82%	220
7	21	16	7%	220	93%	236
8	27	9	3%	279	97%	288
Mean	38		6%		94%	

IV.3.4.4 All Computer-grad customers teams vs. mixed customer teams

Recall, that according to the research design (§IV.3.3), two types of customer teams were formed (see Table 14, blue and green sections). In one type, only Computer-graduates were put together (there were 3 teams of this type). The other type mixed one Business-graduate with one Computer-graduate (6 teams). In order to investigate whether there is a significant difference in quality of specifications produced by these treatment groups and bearing in mind that the normality of the data cannot be assumed, we resort to a non-parametric Mann-Whitney U right-tailed test. Specifically, the following hypotheses were tested: H_0: *specification quality of mixed team = specification quality of pure computer science team;* H_A: *specification quality of pure computer science team >*

specification quality of mixed team. The calculated value of U is 4.5 (p=0.357), which is larger than the critical value (1.0, as per [127]), with mean values of 100.0 and 86.5 for computer science teams and mixed teams respectively. Hence, the result of the test indicates no significant difference in the quality of the specification produced between two team types at the 5% significance level, and therefore does not support Hypothesis C. This is an interesting finding because it suggests that the customer teams we equal (perhaps because the technique bootstraps the Business group to the same level as the pure Computer Science students). At a minimum, this deserves further investigation.

IV.3.4.5 Correlation between the quality of acceptance test-based specification and the quality of implementation

In addition to the evaluation of the executable acceptance specifications created by the customer teams, the code produced by the development teams was subjectively evaluated by the author of the dissertation and his teaching assistant (see Table 14). A Spearman's correlation coefficient computation was used. Although there is a moderate and positive correlation between the quality of executable acceptance test specifications created by the customer team and the quality of implementation produced by the development team (rho_s = 0.455), the probability of this correlation occurring by chance is too high (p=.229). Therefore, no significant correlation between these two variables at the 5% significance level can be reported. As a result, Hypothesis D, that the quality of the executable acceptance test specification produced by the customer team will be strongly and positively correlated with the quality of the implementation produced by the development team, is not supported. Considering the importance of the quality of requirement specifications on the success of software development project [142], this issue deserves further attention and experimentation by other researchers.

IV.3.5 Additional observations

Here we discuss additional observations for which no prior hypotheses were made, but which help in answering the research questions (outlined in §IV.3.2).

IV.3.5.1 Types of activities

Subjects were asked to self-assess the amount of time they spent on the following activities as a portion of the total time devoted to the project:

- searching for information;
- inventing an example/scenario;
- translating substantial amounts of information from some other source into the system;
- adding small bits of information to a test that you have previously created;
- reorganizing and restructuring tests that you have previously created;
- playing around with the new ideas, what-if scenarios, without being sure what will result;
- other.

Figure 17 depicts distribution of activities by duration (position on the chart) and frequency (diameter of the bubble). It appears that most of the time is spent on inventing examples/scenarios/test cases (from 10% to 90% of the total project time, with mode = 50%) and reorganizing/restructuring existing tests (from 5% to 50%, with mode = 15%).

The majority of subjects (87%) were involved with extending existing test cases, but this activity, on average, did not consume more than 15% of the total project time. Two-thirds of subjects played around with the new ideas, trying what-if scenarios etc. without being sure what will result – but no more than 10% of the

time was used up for this activity. Only one person identified an "other" activity, which was navigating between tests, and it occupied 5% of that person's time.

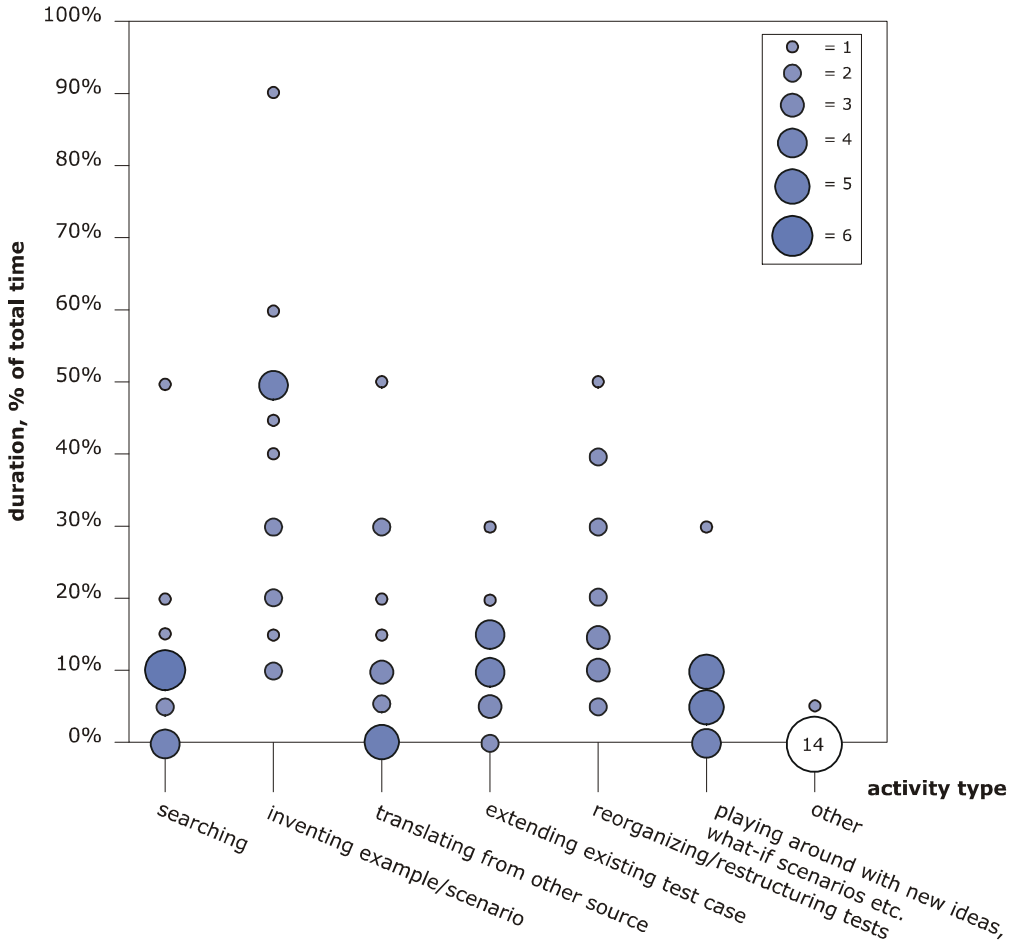


Figure 17. Activity Categories Duration Data.

IV.3.5.2 Effort

Table 16 contains information on the effort of the customer team members spent on writing the executable acceptance-test specification and communicating with the development team. We expected each person to contribute about 4 hours a week on this task (16 hours in total per person; or 32 hours per team).

Table 16. Effort Spent (N=17).

Team	Effort own, hrs	Effort pairing, hrs	Effort, partnership with development team, hrs	Total indivi-dual effort, hrs	Total effort per team, hrs
1	8	3	3	14	**35**
	18	1	2	21	
2	15	3	1	19	**19**
	N/A	N/A	N/A	N/A	
3	5	2	1	8	**13**
	3	1	1	5	
4	10	2	5	17	**26**
	5	2	2	9	
5	0	0	3	3	**16**
	10	2	1	13	
6	3	3	1	7	**23**
	10	3	3	16	
7	8	1.5	1	10.5	**26.5**
	12	4	0	16	
8	15	4	1	20	**29**
	8	1	0	9	
9	4	2	3	9	**15**
	1	5	0	6	

The total time spent per team (between 13 and 35 hours, with the median of 23 hours) suggests that customer teams were able to achieve their goals in reasonably-expected time frame or even sooner, with little overtime occurring.

IV.3.5.3 Usefulness perceptions.

At the end of the project, subjects were asked about their perceptions of the technique. Figure 18 shows the distribution of the answers to how likely they would recommend using executable acceptance tests (in FIT, for example) for specifying business requirements, to a colleague.

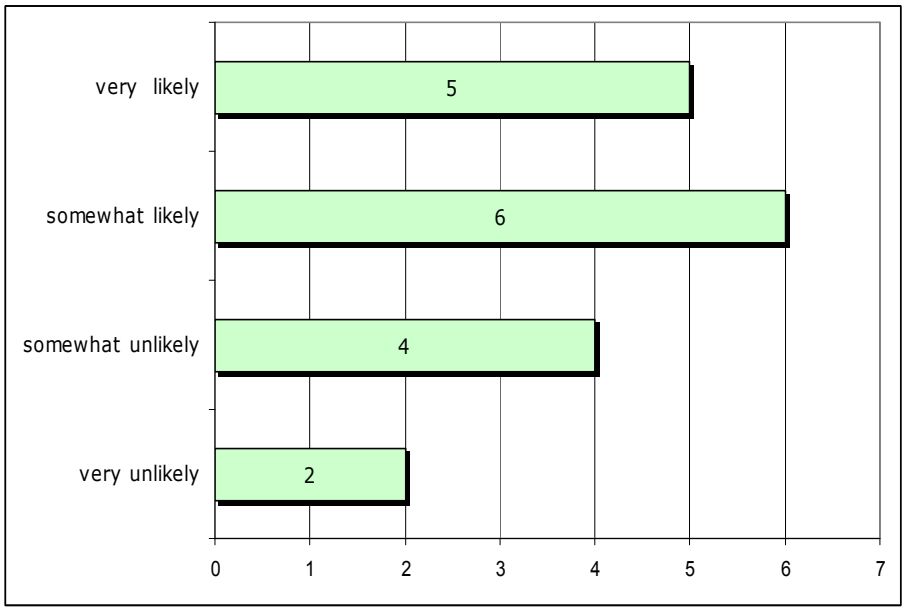


Figure 18. Likelihood of Recommendation to a Colleague.

Noticeably, no correlation between these perceptions and the quality of the produced specification during the course of the experiment was found. This is remarkable as it seems that the subjects are almost unaware of the quality the approach produced for them.

When asked whether they would use FIT/FitNesse on a regular basis at work (assuming it would be available), the opinions were split half and half. Some people expressed their strong desire to adopt this practice, as can be seen from the following testimonial:

> *"Yes, I would like to use FIT/FitNesse for specifying requirements provided my organization supports it. It is easy to use but the whole team (i.e. analysts and developers) must be committed to using it and communicating though acceptance tests".*

Several people indicated their preference to use FIT on a regular basis but with more practice. One respondent pointed out:

> *"Yes,[I would use it] but as a clarification and elaboration of narratives/event flows. The leap from story to executable acceptance test is too large to easily overcome".*

A member of the customer team that did an extremely good job with their specification (see footnote to Table 13), describes his experience:

> *"I had a difficult time with describing simple functionality. In some cases, I would find it easier to tell (verbal) a developer what I want. Also, I found the "setup" and pre-conditions to be tedious and painful. However, in my line of work, I have noticed a gap between business requirements and technical spec. I can see FIT providing the missing functional layer".*

This goes along with the notion that acceptance tests are meant to support communication, not to replace it.

One subject indicated that he "*liked to use FIT/FitNesse for simple scenarios*". However, for more complex ones, he indicated the preference for use cases. Another person considered executable acceptance testing to be complementary to "more formal requirements elicitation techniques". Several people, who did not express a clear 'yes' or 'no' preference, said that their decision would "*depend on the project and the maturity of the customer*".

Among those who rejected the practice was one person from a team that, nevertheless, created a high-quality specification:

> *"Writing FIT/FitNesse almost felt like writing code. I felt like I was literally specifying [to the] development team the details down to function names. On top of that I had to explain what I wanted in paragraphs before the test cases. I felt like I was doing double the amount of work".*

This speaks of an expectation of the role of the customer and the work involved.

To sum up, half of the sample found this technique promising and would consider using it at work and also recommend it to their colleagues; while the other half were more skeptical and did not plan on adopting the practice.

IV.3.6 Validity

Several threats to the internal and external validity have been identified. The biggest concern is the small size of the sample (18 individuals, 9 teams). We addressed this risk by using statistical procedures that are recommended for the small samples.

Another concern is the type of the sample – it was a convenience sample based on self-organized/self-selected teams. Because we did not have a random sample, we caution the reader to be conservative in the interpretation of our study results. In fact, we cannot claim direct cause-effect relationships that will hold in general, in many settings. We only report the trends that manifested themselves in the course of this experiment.

The fact that the teams were composed of the graduate students suggests two potential biases. The first one is that volunteers (students enrolled in the course based on their interests of the topic) may bias the results because they are often more motivated and skilled than average industry customer. Graduate students are usually pre-selected and represent a highly motivated group of the student population. Even though, many of them will end up in the industry, this sample may not be indicative of the population of the customer representatives from the industry. Despite this, we used customer teams external to the development teams, and this constitutes, in our opinion, an improvement to the sampling approaches in a previous study (see discussion in §IV.3.2 and §IV.1).

Business-grad students may also have had a strong expertise in computer science as they were taking an elective course in software engineering, which may point to some previous interest in computer science and technical topics.

We are also aware that cross-team discussion may have taken place and those may have skewed our results. In addition, there might be a possibility of an unequal workload when one pair partner may have contributed more than the other one but did not report it in the post-experiment questionnaire – this would have obscured the results and would not have created the main effect.

The project provided to the system also poses a potential bias. Though it was not trivial, it may not have been complex enough to approximate the real world. This would have produced weaker results-effects. In addition, the short-term involvement of the customer team (only 4 weeks on the part-time basis) may have lessened their level of engagement with the project implementation towards the end of the iteration.

Ultimately, we recognize the limitations of the study and therefore intend to conduct further case studies in the industry to reinforce and extend the findings of this research.

IV.3.7 Academic Study Three Summary

The study addresses a need to understand if and how executable acceptance testing technique can be used directly by customers, to support the elicitation and development of system requirements. To test this, we conducted a quasi-experiment with teams of customers, composed of both business and computer science graduates, who used FIT and Fitnesse to develop requirements for a human resources information system in the form of acceptance tests, which were communicated and used by development teams composed of undergraduate students. Our results show that customers can specify functional requirements clearly, despite some initial difficulty in doing so. Two-thirds of the subjects will recommend and use the executable acceptance testing technique in specific future practice scenarios, while the other one-third will not.

Chapter V Qualitative Analysis

V.1 Impetus.

An initial set of theoretical ideas about the nature of the EATDD process emerged from the early pre-experimental observational studies (§§IV.1-IV.2) and the quasi-experiment (§IV.3). As a result, we were able to refine our research questions and determine what further data should be collected in order to explore and elaborate on these ideas. This required working on another level - "grounded theory analysis" – in order to gain insights into how EATDD is really used in the field. Thus, the ultimate objective is to enrich the findings, produce a theoretical framework which is better grounded in the industry, and empirically support it.

V.2 Research questions and propositions

The objective of this investigation is to characterize and evaluate the main hypothesis in the contexts of several case studies, that executable acceptance testing is an effective tool for discovering, articulating and validating business requirements on a software project. Even though we did not impose our definition of "effectiveness" on the respondents, we recognized that an effective practice and/or tool must address several aspects of communication. Those aspects include the following: clarity of requirements, the ability to deal with complex end-to-end scenarios, ease-of-learning and ease-of-use. Therefore, we structured the interview-guides accordingly to address these points. Specifically, the following propositions were evaluated:

A) *The learning curve for executable acceptance testing is short (2-4 weeks).*

B) *The business experts and the technology experts experience no difficulty in following the executable acceptance test-driven development method and in using the FIT framework.*

C) *Writing executable acceptance tests is a collaborative endeavor. Even though certain tests can be written or extended by an individual team member, the predominant way of writing the tests and communicating requirements they depict is via a group discussion.*

D) *The team writes executable acceptance tests early (first) and iteratively.*

E) *The business experts find executable acceptance tests and the FIT framework useful because they can specify, understand and extend the test scenarios.*

F) *The business experts and the technology experts would recommend using executable acceptance tests to a colleague.*

G) *Overall, the clarity of the requirements that are supported by examples in the form of executable acceptance tests increases.*

Additionally, we sought answers to the following questions:

- Why did the team decide to adopt the method of executable acceptance test-driven development?
- How and why do business experts and technology experts use acceptance tests?
- How do they collaborate?
- How difficult is it to specify requirements in the form of executable acceptance tests?
- Who typically writes the tests?
- How do they decide what needs to go into the test?
- What kinds of challenges arise and how do team members resolve them?
- What kinds of additional tools were used?

- What types of tests were mainly written (positive – "happy paths", or negative – "sad paths", representing exceptions, misuses and/or abuses)?
- What types of test styles were the most popular?
- How often and why did the business experts check the progress of the development team by executing the acceptance tests?
- Were there any scenarios that seemed especially complex or difficult to work out and to specify as an acceptance tests?

V.3 Case Study as Grounded Research Method

We chose the case study method for our grounded qualitative research because of the following factors:

- the type of research questions being posed (mainly "how" and "why"),
- the lack of control investigators had over actual behavioral events,
- the focus on contemporary phenomenon within a real-life context,
- and the desirability of using multiple sources of evidence.

Moreover, "the case study method allows investigators to retain the holistic and meaningful characteristics of real-life events" [140] and "to generate knowledge of the particular" [126], which were additional reasons for this choice of research method.

V.3.1 Units of analysis

For the purpose of analysis of the case studies that follow, the following units of analysis were selected:

- iteration planning meeting when a story was written, acceptance criteria discussed, and acceptance tests (potentially) specified;
- individual members of the team;
- engineering team as a whole.

V.4 Scoping and sampling

Beanlands defines scoping as an anthropocentric value judgment of what is important or what is not [13]. Throughout all studies that were used to produce this research, our focus was to determine the range of issues to be addressed. In our qualitative studies we continuously followed the process of "theoretical sampling" as defined by Glaser and Strauss [53]. During this process, we collected and analyzed the data and determined what data to collect next and where to find it, in order to develop our theory as it emerged. Essentially the process of scoping and data collection was controlled by the emerging theory: initially, data was selected to explore maximum possibilities of variations in the area of EATDD and the sampling gradually became more focused to expand emerging concepts and relationships and to fill in gaps in categories and emerging theory.

More important, the sampling of specific subjects is not based on the usual criteria and techniques of statistical sampling. The representativeness of a sample is guaranteed neither by random sampling nor by stratification. Rather, individuals, teams, etc. are selected according to their expected level of new insights for the developing theory, in relation to the state of theory elaboration so far. Sampling decisions aim at the material which promises the greatest insights, viewed in the light of the material already used and the knowledge drawn from it [42]. If, in our quantitative studies (Chapter IV), we followed a conventional methodology when sampling preceded analysis, in grounded theory, sampling decisions were made based on the preceding analysis, thus adapting to the evolving emergent theory.

Another critical question is the selection of companies (which served as sites for case studies). Which of these companies would provide appropriate comparable data to extend and deepen the merging conceptualizations? In grounded theory, "the criteria for selection therefore revolved around comparison in terms of the concepts being investigated – rather than selection in terms of other factors

which might delimit the populations or control the variables being studied."[31]. Therefore, our comparison proceeded in terms of the theoretical value of sites for generating categories rather than their representational value (as cases from which to generalize).

We followed a generic theoretical sampling framework (as per [131]): the sampling method matched the type of coding used in the data analysis (see Table 17).

Table 17. Coding and Sampling Methods.

Type of coding	Type of theoretical sampling method	Purpose
Open coding	Open sampling	Maximize the variation in the data in order to define categories according to their dimensions and properties
Axial coding	Relational & variational sampling	Obtain data that demonstrates relationships between categories
Selective coding	Discriminate sampling	Select data that will maximize opportunity for comparative analysis and help define and refine a central concept

Open coding was used to create categories on the data properties and dimensions. This is an inductive method. At the beginning, we did not precede our data with provisional start-list of generic codes. As a result, we were not constrained by the generic codes, but allowed our coding to be more open-minded and context-sensitive. A line-by-line analysis of the interview transcripts was performed, and the concepts were organized into tables of categories and sub-categories (Appendix E, Table 19 – Table 30).

The open coding analysis phase of the current research resulted in the creation of conceptual categories and subcategories that cover various aspects of participants' activities, experiences, perceptions and challenges. We have performed this coding twice (same person) to account for all new categories added during the first pass. The codes were also reviewed (by the same person) every time a new piece of data was analyzed.

The next major path through the data was an axial coding analysis. This time we focused on the initial themes (formed from the categories) more than on the raw data. Axial coding allowed us to deepen our understanding of the categories, and discover important relationships between major categories.

V.5 Data collection

A variety of sources was used for data collection. Interviews were chosen as the primary source of qualitative data. The interview format was selected since it allowed a semi-structured approach to gathering data. Generally, our interviews were based on a set of pre-determined open-ended questions which were designed to meet the research goals, but could be and were easily modified and/or added to, during the interviews themselves, based on the subject's responses. The nature of our qualitative research incited numerous exploratory activities, for which the interviews were a well-suited instrument. Our intent was to allow the interviewees to express their own personal experiences.

We interviewed 6 members of software development teams from 3 companies. To gain a multi-dimensional perspective, it was highly important to involve both business and technology experts with various degrees of experience and expertise. The data from the first 2 companies were used for conceptualizing and theory building, while the last case was used for its validation.

Table 18 summarizes sites, projects information together with individual participants' backgrounds, roles on the project discussed, and levels of prior experience with EATDD and the FIT framework.

Interviews and observations were conducted from March 2006 through April 2007. The mean duration of the interviews was 73 minutes, which, in transcribed form, constitutes 10,332 words or about 21 pages (standard Letter page, single spacing, 12pt font).

The answers obtained to the open-ended questions of the in-action and exit surveys were used as the secondary source. These surveys were conducted during the academic studies (Chapter IV).

In Case Alpha, we intermixed interviewing and document analysis. For document analysis, we were provided with a sample of test artifacts and "info sheets" (described in V.9.2.2) from the project. These became our third data source.

V.6 Data collection logistics

The interviews were given verbally, either over the phone, over Skype, or face-to-face. All interviews were digitally recorded. All interviewees were informed about the research through a phone script or a research consent form designed in compliance with the University of Calgary Conjoint Faculties Research Ethics Board (The Ethics Board). The phone script was read to interviewees who participated in interviews via the phone/Skype. Alternatively, if an interview was conducted in-person, interviewees were given a choice of their interview being recorded or not, and they were handed a research consent form (written in compliance with the Ethics Board, see Appendix A) for their signature prior to conducting the interview. All interviewees were informed that the research was anonymous and that their participation was voluntary. Interviewing only began after their consent was obtained. Interviewees were informed that they were in full control of the process and could terminate it at will, or decline to answer any question or remove an answer. They were also offered to review interview scripts upon request. There were no reported cases of interviewees feeling uncomfortable during the data collection process and thus none of them used any of the opt-out options outlined above.

Participant confidentiality was strictly maintained. The reports and presentations used in our analysis only referenced the participant alias, not his or her actual name. No personal information was released.

Table 18. Sampling: Sites, Participants, Roles, and Experiences.

Company/ site (alias)	Case pur-pose	Country	Project description	Process	Iteration size	Team size	System size	Subject (alias)	Primary role[20]	Years of experi-ence	Involved with the project since…	Prior experience with FIT?
Alpha	ETB	USA	B2B communicati on system	XP	2 weeks	14-17	large	**Cadmus**	dev lead	14	start	no
								Jacinda	tester	5	1/3	no
								Neo	customer/ product manager	15	start	no
Gamma	ETB	Canada	Metabolism analysis system	XP	1 week	13-15	medium	**Chrysander**	dev lead	11	start	yes
								Talos	developer	8	start	no
Epsilon	V	USA	Forensic code analysis system	XP	2 weeks	7-11½	medium	**Teodor**	dev lead	12	start	yes

Notes: a) **ETB** = evidence/theory building, **V** = validation; b) ½ person in the case Epsilon is an indication of a part-time person

c) System size grouping (# test assertions on the date of the interview): small =< 5,000, medium = [5,000, 20,000], large = > 20,000.

[20] Members of agile teams are generalists, playing multiple roles. Here we list their primary roles as identified by the subjects.

V.7 Data analysis

The key steps of the analytic process that guided this qualitative research include the following: data management, data review, data reduction, drawing conclusions, and verifying conclusions [101]. Specific tools that informed the analytic process included: textual analysis and coding to break down and describe units of data or concepts derived from that data; researcher's memos to document thoughts and describe findings based on the textual analysis; mind maps to organize those findings; data matrices that summarized data and facilitated comparisons across different roles; and network diagrams to display connections between categories and concepts arising from the data.

Atlas.ti, specialized software for qualitative data analysis was used to aid the tasks of document and data organization, systematic coding and memoing, powerful search based on co-occurrence analysis, and reporting (see Appendix C).

It should be noted that in the data analysis discussion, all citations are given verbatim, with no correction for grammar or style. Any particularly strong points made by the respondents were denoted with asterisks.

V.8 Evolution of the instrument

For the interviews, we used the general interview guide approach that involves outlining a set of issues that are to be explored with each respondent before interviewing begins. This guide was prepared to ensure that the same basic lines of inquiry were pursued with each person interviewed. Still, the interviewer had a lot of flexibility in probing and exploring.

Our first industrial case was used to characterize and validate the main propositions previously examined in the academic observational studies and the quasi-experiment, i.e. business experts in collaboration with technology experts can effectively articulate functional requirements in the form of executable

acceptance tests. Therefore, it primarily focused on the process of requirements specification. It had 19 questions. As the interviewing process progressed and new concepts were identified, we updated the guide with questions about the common vocabularies, regression testing, duration of test runs, tabular representations, improvement ideas and whether the interviewee would recommend EATDD to his/her colleagues. Its final version had 25 questions and can be seen in Appendix D.

V.9 Industry Multi-Case Alpha: B2B Communication System

V.9.1 Case study context

This case study investigates the dynamics of a software engineering team using EATDD on a real-world project. The distinguishing characteristics of this case are listed below:

1) high complexity B2B system;
2) the size of the acceptance tests;
3) the use of instruments complementary to acceptance tests; and
4) the use of acceptance tests beyond functional requirements.

The team consisted of an on-site full-time product manager who was a domain expert (the "Customer"), a project manager, 10–12 developers, 1–4 quality assurance engineers, and one consultant who introduced the methodology, the practices, and who also performed specialized work. All software development was done in-house. The team was collocated in an open area. Informal communication occurred freely.

The team worked on a single "greenfield" project (new development, no project switching). The purpose of the project was to implement an EDI transaction platform to allow users to define business rules around the delivery of certain

critical documents (for example, purchase orders) through a Web interface, to execute those rules, and to provide notifications. It is important to stress that this was not a simple rule engine. This platform not only utilized but also extended many features of standard EDI. Upstream from the Customer, there was a product strategist (a person in marketing). Marketing provided the Customer with a product requirements document, which, according to the Customer, *"alternated between vagueness and specificity, sometimes requesting large swaths of functionality with a single sentence and at other times making detailed recommendations of deployment platform."* In addition, the Customer was given access to retail industry veterans, but no one on the team had any experience in retail supply chain management. As the Customer stated, there was *"a development team that had never had a customer."* It *"had to build an application that no one had thought too much about."*

The team adopted the extreme programming (XP) methodology and was coached by a consultant (not the researcher). The team diligently carried out all 12 practices of XP. The iterations were two weeks long.

The project lasted 10 months, and despite some difficulties and growing pains, it was successful – the team was able to release a high-quality, feature-complete application on time (as unanimously recognized by all interviewees). In addition, marketing was satisfied and accepted the system. The existing clients of this vendor were happy with the product and the vendor even managed to sign up new clients.

Three members of the team took part in this case study. In order to get multiple perspectives, the investigators targeted one representative from each role:

(1) Neo, *the Customer*, whose job was to identify a high-value set of user stories for each iteration and do what was necessary to help the developers understand and implement these stories;

(2) Jacinda, a lead quality assurance engineer (hereby referred to as "*the Tester*"), whose job was to review acceptance tests specified by the

customer, suggest new tests cases, find problems, and help the team to understand what was going on; and

(3) Cadmus, a lead developer ("*the Developer*"), whose job was to implement the system that met the business requirements of the Customer.

It is important to note that the Customer had an information systems background. While he was not a software developer, he was performing a job similar to what a business analyst would do. Therefore, this study does not make any speculative generalizations on whether a non-technical customer would be as capable as the one we interviewed. In reality, the chances are likely that it would not be the case.

To provide a rough caliber of the project, the total number of the acceptance tests produced was about 7,000 (with test pages typically aggregating between 5 and 100 tests[21]).

V.9.2 Findings

V.9.2.1 Learning the practice

Upon recommendation of the consultant, the team adopted the EATDD practice and the FIT framework to make the process of writing and executing the tests easier. No one on the team had any prior experience with FIT. All team members received a four-hour long introduction to both test-driven development and executable acceptance testing with FIT. During the first iteration, the consultant assisted the team with writing user stories and acceptance tests. Gradually, the team felt comfortable specifying their business rules in the form of acceptance tests. According to all three respondents, learning the technique and the framework was easy.

[21] The issue of test aggregation or test chaining is considered by some practitioners as an undesirable practice.

Some of the initial questions that technology experts had were about the difference between the unit tests and acceptance tests. The developers did their best to make acceptance tests more readable and to make the Customer and the Tester write those tests ahead of time.

According to the Tester, after a couple of days of "playing" with FIT, the team could operate the framework and write basic test scenarios. Everyone on the QA team learned the basis within a week and were able to work individually on writing and running acceptance tests. The learning curve was quite short. Jacinda, the Tester enthusiastically noted that "*FIT is simple!*"

V.9.2.2 Using the practice

Because of the inherent complexity of the domain, for each iteration meeting, the Customer would prepare an "info-sheet" (one- to three-page informally-written document with plenty of diagrams, callouts, and, most importantly, mock screen shots). It was used to describe characteristics, behavior, and logic around a coherent set of features. It was not meant to be an authoritative specification and no official signoffs were required.

The iteration planning meetings involved the following actions:

a. Discuss the info sheet and talk about functionality.

b. Resolve any general questions about functionality.

c. Define a user story.

d. Define acceptance tests (criteria) for that story.

e. Repeat c, d (until sufficient number of stories and acceptance tests are specified).

In this case study, defining acceptance tests did not mean coding them in FIT. Initial *"sketching"* of the test was done on the back of an index card. As the list of possible test cases grew, the testers suggested recording them in a spreadsheet – *"something that we could later go back to"* (Jacinda). Later, either the Customer or the Tester would create an actual FIT table. Neo, the Customer, explains: *"We*

defined all requirements in general groups. I went into the planning meetings with well-described 'featurelets' and came out with stories and ideally acceptance tests." An example of a story could be "Rule: deadline dates are all treated as Eastern time." The team then identified all places in the system where the time was relevant (the UI, database, email, etc.) After these steps they stopped because they had enough information to write an acceptance test.

The technology experts generally would start their work based on the user story and acceptance test summary. They would have the detailed tests before their implementation was completed.

This story-by-story procedure, including the invention of the info-sheets, matches the pattern of specifying business rules with executable acceptance tests the investigators expected. Importantly, the test-first paradigm of development was truly adopted and followed throughout the project.

The researchers pursued the line of inquiry to understand why the info-sheets were necessary. The Customer produced and focused the material necessary for the info-sheets *"at where they needed to be – to communicate the context to developers."* The stories were isolated. So the team talked about the stories, but *"stories and talking are not great for communicating the details that should persist"*. The info-sheets would help to answer the questions why the developer was doing this or that and what this piece connected to. This is illustrative of a common concept that the business experts need to come to the iteration planning meetings prepared and have a very concrete understanding of what they want from the upcoming iteration. This understanding can be documented upfront (if a business expert thinks that it is beneficial – which was the case with this project and this particular customer).

In a separate study on the impact of Scrum on overtime and customer satisfaction [77], Mann and Maurer reported an early issue with some teams only adopting agile methods – the customers did not come to the iteration planning meetings prepared. When they realized the problems this caused, business analysts were used to make sure the situation was rectified in the future.

Testers and the Customer paired up often with developers when specifying test scenario details (what data to use, what actions to execute, etc.) *"Sitting down with the developers and giving feedback to them – they didn't need much more than that."* Everybody agreed that *"it was very interactive between the developers, the QA, the Customer – everyone!"* Jacinda, the Tester, pointed out that the *"open space led a lot to XP thinking and very open communication. Everybody knew what everybody else was doing."* It is worth reiterating that the team size was ideal for this type of the process (13 – 18 people) and in a different setting (larger team or non-collocated team), the results may have varied.

While working on a story, the team may have realized that they had missed several cases. In this event, additional acceptance tests would be written. The interviewees estimated that this occurred 30-50% of the time. The phenomenon can be explained by the nature of continuous learning about the domain and the system through testing (this aspect of continuous learning is emphasized by the thought leaders and practitioners of the context-based school of testing, and exploratory testing, in particular [10]). During iteration planning, one cannot often think of all acceptance test scenarios, but as one dives in the story implementation, other things become more apparent and new scenarios are added to the test suite.

V.9.2.3 Acceptance test authoring

All acceptance criteria were specified by the Customer and the QA. When it came to actual authoring of tests in the form of FIT tables, about 40% of all FIT test pages were written by the Customer, 30% by developers, and the remaining 30% by testers (based on the estimates provided by the Customer and the Tester). Neo, the Customer found that *"in practice, it was best if the Customer wrote acceptance tests. This is related to the fact that going from a general description to a test has some fluidity in interpretation."* Because of the domain complexity, the customer either had to communicate in greater detail what the test should be and then review it, or simply do it himself or herself. The Tester reported

specifying acceptance tests in pairs with the actual developers of a story or with the Customer. If it was with the Developer, the acceptance tests would be reviewed by the Customer in an informal review session (that usually took no more than 10 minutes and was done on the fly). This was possible due to team collocation (no offices, no cubes) and an informal communication flow.

The Developer indicated that for negative tests (that cover deviant behaviors, misuses or abuses), they wrote sophisticated error messages (and comprehensive checks) to convey the meaning of what may have caused that error. Moreover, the developer went beyond functional tests in FIT. They extended the FIT framework to capture runtimes and do basic performance and load testing.

V.9.2.4 Acceptance test types and patterns

Though it is commonly believed that business customers predominantly think about the positive scenarios (or "happy paths"), they also think of negative scenarios. We verified this claim and found sufficient support for it: "*For each individual story, there is a single positive flow, and an infinite number of negative ones. And, so, you [a team member] want to verify the positive, but especially for lifecycle tracking, the variation that can get you an error is huge – so we [the customer and the testers] had to think a lot about negative testing*". A typical negative test would deal with a received document that did not match a certain rule set, and as a result no tracking instance would be initiated (which would be verified by the test). The Tester confirmed writing both positive and negative tests in the proportion of 20/80.

With regard to the FIT test types, the analysis of the testing artifacts revealed that the `ActionFixture` was the most popular type. This was no surprise since transactions and workflows represent the major functionality of a system developed (an EDI transaction system).

The following common test pattern emerged:

- *Build* – several tables are used to reset the database, populate it with the data etc.;

- *Operate* – a table to operate the data;
- *Check* – one or more tables to check the results of the operation.

Figure 19 contains a sample acceptance test – with the *build*, *operate* and *check* sections clearly denoted. Notice how setting a system up to a certain state (the *build* phase) takes most of the test page, even though several common tasks (for example, resetting the database) are delegated to specialized fixtures (in the Figure 19 example, `itm.fixture.ITMDatabaseUtil`).

Each test page would include multiple test cases. We noticed that some of them might not be independent and, thus, potentially be affected by the preceding tests.

V.9.2.5 Challenges in specifying requirements in the form of executable acceptance tests

Several experts in the industry question the expectation of the agile teams for the customer to write acceptance tests (see, for example, [123]). Therefore, the customer's opinion of the difficulty of specifying executable acceptance tests was especially important to this investigation. Neo, the Customer, testifies: *"[It was] not particularly hard… Because we were all there (developers, testers, and I [the Customer]) talking about the story. So, the acceptance test was a natural segway*." Apparently, the difficulty was not the practice itself, but the discipline of doing it. "*Once functionality was discussed and the stories were defined, the team wanted to be done. Forcing ourselves to think in detail about what tests needed to be performed and what the logic of those test scenarios should be, was hard*". The team had to work on devoting proper attention to the tests at the beginning. This question of discipline was intriguing, so the researchers pursed this line of questioning further. Neo, the Customer, recognized that putting off writing an acceptance test was *"a dangerous thing"* (even if it did not happen frequently). He paraphrased from the book "Zen and the Art of System Analysis" by Patrick McDermott [40]: *"We delay things because they are either difficult or unpleasant. Difficult things become easier over time, and unpleasant thing become more so."* The question was whether the team was postponing writing the

LifeCycle Tracking – Tracking Instance Acceptance Tests

When a PO matches the sender Org's configured rule, user sees a new row in the tracking view. The state value is "waiting". When a PO matches the receiver Org's configured rule, user sees a new row in the tracking view. The state value is "waiting".

Each row in the tracking view displays the originating document's track date.

Reset Database

itm.fixture.ITMDatabaseUtil
ensureBase

Create Organizations & TPShips:

itm.OrganizationStepFixture					
create An Organization	Retailer1	11	RECEIVER1	retailer1@retailer.com	Immediate
create An Organization	Vendor1	11	SENDER	vendor1@vendor.com	Immediate
create An Organization	Retailer2	11	RECEIVER2	retailer2@retailer.com	Immediate

itm.TPShipValidator						
create a TPShip	11	SENDER	11	RECEIVER1	850	004010VICS
create a TPShip	11	RECEIVER1	11	SENDER	856	004010VICS

Build

Create a rule in the system for Vendor1 with the specified parameters. (Organization S/R Qualifier, Organization S/R ID, Rule Name, PO Type, Track Date Type, Warning Interval Hours, Warning Interval Hours, Default (true) or Selected (false), Active (true) or Inactive (false), Selected TP's).

itm.LCTStepFixture					
createRule	11	SENDER	PO expects ASN	SA	001

Test No. 1

Negative - Send in a document that DOES NOT match the configured rule for the DTM01, verify that no tracking instance is initiated in the LCT tracking view.

itm.fixture.DocLibFixture		
createDoc	PO1	PO_EDI
setDocSenderID	PO1	SENDER
setDocReceiverID	PO1	RECEIVER1
setDocPONum	PO1	987654
setDocDTM1Qualifier	PO1	038

Operate

Verify LCT Tracking View based on view of sender and receiver identified by S/R ID (Rule Name, Sent Ref #, Return Ref #).

itm.LCTValidator		
getLCTView	11	SENDER
check	noExtraLCTInstances	Success
getLCTView	11	RECEIVER1
check	noExtraLCTInstances	Success

Check

Figure 19. Snippet of a Sample Acceptance Test on the Alpha Project.

acceptance tests because they were “difficult” or because they were “unpleasant.” The result was that it was usually because of the “unpleasant” aspect. The Customer explained: *“It was complicated stuff to test, and the thought of diving into that complexity, just when we thought we were done, was unpleasant.”* The team finally realized that they had to put discipline into their acceptance test writing.

All in all, both the Customer and the Tester were quite enthusiastic about EATDD and, specifically, FIT. The following testimony of the Customer illustrates one of the reasons for this enthusiasm: *“FIT is definitely more accessible and I could write FIT tests. That was huge!”* Acceptance tests helped the Customer and the team to discover many missing pieces or inconsistencies in a story. The FIT tests were concrete.

V.9.2.6 Test execution

The Customer executed acceptance tests frequently. As the Customer created the tests, he would run them right away to ensure that they were internally valid (get to the “yellow” unknown state – a test without an implementation could not possibly pass or fail). Then the Customer would notify the developers and tell them that the tests are ready and the developer would implement the necessary functionality and the “glue” (in the form of FIT fixtures) to hook the tests up to the system. From time to time, developers may need to make changes to a test. When a change is needed, the developers would inform the Customer and the rest of the team about it. The Customer would perform spot-checking (though quite often that was not necessary). The team implemented continuous integration with an automated build and notification system (they started with `CruiseControl` and then implemented a home-grown solution).

The Tester executed the acceptance tests with an `ant` script. The developers ran tests daily and also ran tests on every check-in to the source code repository.

V.9.2.7 Test navigation and management

Considering that most test pages were quite long (5-40 pages if printed from the browser, normal font size) and contained multiple test cases and tables (in some cases up to a 100 tables in one test page), the navigation, management, and maintenance of such acceptance tests were, as a result, expected to be an issue. The investigators' line of inquiry confirmed this supposition with the members of the team recognizing that their tests "*exploded in size and number,*" resulting in a test suite of unmanageable size, that they were "*either too scared or too busy to refactor*." Neo, the Customer, expressed a desire for a meta-layer FIT management tool defined as some kind of an interface that allows correlating stories with acceptance tests and individual FIT tables.

Jacinda, the Tester, recalled that they did their *"own little [test] management"* by separating each test by function. This way "it was easy for us to locate the tests we needed." Also the naming convention of the files containing tests was very straightforward (using the function of the system).

V.9.2.8 Acceptance tests vs. unit tests

As the team was transitioning from a waterfall-like process to an agile process, testing became of paramount importance. Unit testing (in JUnit) was always quite diligently completed by the developers. Sometimes unit tests became indistinguishable from the acceptance tests. The developers started to lean towards the use of unit tests as opposed to acceptance tests. Unit tests provided a more natural way for them to code test cases and assertions. Besides, as the project progressed, the developers were learning more and more about the domain. So, when new issues where found, it would necessitate the acceptance tests to be re-written or simply thrown out, *"causing a lot of churn"* (according to the Developer and the Customer). As a result, developers thought that they had *"to invest a lot of effort into the development of FIT pieces"* (fixture implementations) while adding more methods to those fixtures so that they could become more human-readable. To no surprise, JUnit was what the developers were more comfortable with. Figure 20 shows an example of a de facto

```
public class BusinessRulesRoleAccessTest extends TxITMDatabaseTestCase {

   private UserWorkflow _userWorkflow;
   private LifecycleTrackingWorkflow _lctWorkflow;

   private static OrganizationID XYZ_ORGANIZATION;

   private static int __counterToEnsureUniqueness = 0;

   private static final String USER_EMAIL = "email@foo.com";
   private static final String USER_LOGIN = "login";
   private static final String USER_PASSWORD = "password";


   //**************** TEST CASES **********************//

   /**
    * This method asserts that only the proper security roles can launch the user
    * picker through process tracking of the business rules.
    */
   public void test_process_tracking_launch_user_picker_privileges() throws Exception {

      UserID XYZAdmin = createUser(XYZ_ORGANIZATION, UserRoleEnum.XYZ_ADMIN);
      checkCanLaunchUserPicker(XYZAdmin);

      UserID customerAdmin = createUser(XYZ_ORGANIZATION, UserRoleEnum.CUSTOMER_ADMIN);
      checkCanLaunchUserPicker(customerAdmin);

      UserID businessUser = createUser(XYZ_ORGANIZATION, UserRoleEnum.BUSINESS_USER);
      checkCannotLaunchUserPicker(businessUser);

      UserID endUser = createUser(XYZ_ORGANIZATION, UserRoleEnum.END_USER);
      checkCannotLaunchUserPicker(endUser);
   }

   /**
    * This method asserts that only the proper security roles can add
    *   a lifecycle tracking rule.
    */
   public void test_lct_add_rule_privileges() throws Exception {

      UserID XYZAdmin = createUser(XYZ_ORGANIZATION, UserRoleEnum.XYZ_ADMIN);
      checkCanAddLCTRule(XYZAdmin, TrackDateType.NONE);

      UserID customerAdmin = createUser(XYZ_ORGANIZATION, UserRoleEnum.CUSTOMER_ADMIN);
      checkCanAddLCTRule(customerAdmin, TrackDateType.PROMOTION_START);

      UserID businessUser = createUser(XYZ_ORGANIZATION, UserRoleEnum.BUSINESS_USER);
      checkCanAddLCTRule(businessUser, TrackDateType.DELIVERY_REQUEST);

      UserID endUser = createUser(XYZ_ORGANIZATION, UserRoleEnum.END_USER);
      checkCannotAddLCTRule(endUser, TrackDateType.REQUESTED_SHIP);
   }
//...
}
```

Figure 20. Example of an Acceptance Tests written in the syntax of a Unit Testing Framework.

acceptance test written in the language of the unit testing framework (JUnit). Though this snippet can be easily read and interpreted by any technology expert (even one unfamiliar with Java), it is more challenging and less friendly for

business experts. Even in this case study, in which the Customer did not have problems reading JUnit tests due to his prior IT background, he did not write them. Therefore, in the Customer's view, *"it was much better with FIT since I [the Customer] could write FIT tests"*.

Consider Figure 21 with the same acceptance test refactored by the author in the style of the FIT framework – a) using the workflow style of the test; b) using the calculation rule table style of the test. When the refactored versions were shown back to the Customer, he agreed that those were much easier to understand and to interpret – the characterization applicable not only to the assertions but also to the results of execution – the implementation of the last rule, that is "End users are not allowed to launch user pickers", is implemented incorrectly as shown by red cells of the test tables.

user is logged in	
user role is	**host_admin**
ensure	user can launch user picker

user role is	**customer_admin**
ensure	user can launch user picker

user role is	**business_user**
reject	user can launch user picker

user role is	**end_user**
reject	user can launch user picker

a) Refactored Test in the Workflow style

role	*can launch user picker?*
host_admin	yes
customer_admin	yes
business_user	no
end_user	no *expected* / true *actual*

b) Refactored Test in the Calculation style

Figure 21. `test_process_tracking_launch_user_picker_privileges()` **from the Example depicted by Figure 20 refactored in the syntax of FIT.**

The Developer's view was such that, if they *"had not found FIT, we [the developers] would have tried to use JUnit for writing acceptance tests as well."* The important thing is not which type of the framework was used (FIT or JUnit), but the fact that executable acceptance tests were actually written. This, in our opinion, illustrates a maturity of the development team.

It is important to keep in mind that this seeming preference to unit testing was not overwhelming. Cadmus, the Developer, did recognize the value of FIT: *"for*

the most part, it was nice to run those tests and see the system-level tests that would run exactly how they would run in the real world (but with lots of things mocked out) pass or fail. And even more – to see where they fail." Sometimes the Developers had unit tests that came in line with the level of system tests, moved to FIT and vice versa.

According to the developers, there were apparent situations when the use of FIT was advantageous. For example, Cadmus explains, *"when we needed to provide multiple values for something (more specifically: our system processes various types of files – binary, XML, etc.) Those would become various inputs for the system and via the acceptance test you could see how the system would react to those values. This is where the FIT framework really excels. To write this in JUnit is pretty painful and the JUnit tests are hard to follow."*

Thus, on the one hand, the technology experts demonstrated some minor skepticism of the FIT framework due to the fact that *"FIT required a little bit more effort than unit tests"* and also lack of tool support and integration with the IDE (like JUnit has, for example). However, on the other hand, the technology experts recognized the value of the executable acceptance tests specified in FIT because of their readability and intuitiveness, and their ability to provide an easy way for exercising various what-if scenarios. In fact, Cadmus, the Developer, emphasized the latter as *"the best part of FIT – when you throw in different types of inputs to see how the same piece of code falls out*."

This is typical of any framework. It can generally allow you to test anything. Therefore, it is a matter of pragmatics and the purpose of the test that helps select a framework. If a customer can read and write tests in JUnit, then acceptance tests can also be specified in JUnit. But if a customer cannot (which is a usual case), then it makes sense to provide an extra level of abstraction. The researchers have seen this phenomenon on other projects, where JUnit tests have been even called from the FIT fixtures.

The communication power of the executable acceptance tests, their clarity and the ease of reading and following the logic (that all three interviewees alluded to)

were also confirmed by the random examination of several test pages provided by the company. With the exception of a few acronyms, the researcher (who had no prior experience with intricacies of the domain) was able to comprehend and walk through the test scenarios.

V.9.2.9 Executable acceptance tests vs. other requirement specification techniques

The Customer's phrase "*I pray to God I will never have to write [in prose] another functional requirements spec again*" is the strongest indication of his preference.

V.9.2.10 Executable acceptance tests vs. manual acceptance tests

The Tester was familiar with other types of testing prior to this project, but none of them were automated. "*All manual, all through UI. It took two days to run four regression tests! And that was a fast cycle, without finding too many defects.*" If the team had not made the active decision to incorporate EATDD, they would have had many more manual regression tests. The result, according to the Tester, would have been "*a way worse quality of the product.*"

It should be noted that certain acceptance tests on the project were, in fact, manual. The system had a sophisticated presentation layer, and those manual tests were for testing just that[22].

V.9.2.11 Process Effectiveness

The Customer and the Tester decisively recognized the effectiveness of the executable acceptance test-driven development for specifying and communicating functional business requirements. In his own characterization,

---

[22] This is consistent with informal observations we made in several other projects that also did not automate user-interface level acceptance tests. In addition, Robert C. Martin in [83] makes a case for a good, testable system that can access the API independent of the UI. He advocates the acceptance tests as an alternative form of a UI.

the Customer *"was happy."* The Tester also enthusiastically declared *"It [EATDD] made the whole testing process more focused. It made it more unified – everybody agreed on the tests – it was the same tests running over and over again. It made our code a lot cleaner. When we found bugs in our system, we would go and update our FIT tables related to that particular function, so that we could catch it the next time it [the bug] transpires… It was just a good, fresh, new way to run the testing process. The other thing that I loved about it is, when you found a defect and you wrote a test around it, if it was a quality test, it didn't happen again – it was caught right away. Obviously, it made my job [as a QA] much easier and made the code a lot better."*

Furthermore, the Customer did an internal survey of the team and found that the developers felt that the info-sheets together with iteration planning meetings were quite effective. As mentioned earlier, the developers may have been less enthusiastic about FIT from time to time as they deemed writing acceptance tests in FIT required more effort than implementing them in JUnit. However, there was no argument about the value of FIT tests from the perspective of making the tests *"as English as possible"* (i.e. readable and intuitive). This is remarkable, as it clearly demonstrates the consensus among all three interviewees on the value and effectiveness of executable acceptance testing.

V.10 Industry Multi-Case Gamma: Metabolism Analysis System

V.10.1 Case study context

This is the second case study investigating how EATDD is used on a real-world project and what kind of benefits and limitations the practice holds. The following characteristics make this case particularly interesting:

1) the highly regulated environment the company operates in (health care/pharmaceuticals),
2) the presence of the dedicated full-time user experience specialist, and

3) the high planned internal turnover of technology experts during the project.

On this team, business experts were represented by a senior scientist with a Ph.D. in Chemistry (the "Customer"), one domain expert, and one user experience designer. There were also technology experts: a project manager/coach, six developers, one technical writer, lastly a number of testers varying from two to four.

The project involved implementation of a Metabolism Analysis System for the pharmaceutical market. This software system was to be used in conjunction with one of the medical devices that the company produces (mass spectrometer). Importantly, the team discovered that more than a 100 people who used their software were not necessarily experts in drug development, but lab technicians who more than likely graduated from community colleges and not university medical schools. Therefore, one of the objectives for the software development was to make it simple and intuitive enough to be used for somebody who has not been educated or is inexperienced in the field of pharmaceutical research and development.

Business experts provided necessary domain knowledge and were heavily involved in the development process.

The team followed extreme programming methodology (XP) and two professional XP coaches provided the necessary training and initial guidance during the first several iterations. The team members had no prior experience with XP or EATDD.

There was no turnover among business experts. On the technology side, however, a high degree of employee turnover took place. Only two from the original group of 12 programmers stayed until the project was completed. This was partially due to the way resourcing of other projects was done in this particular company and also because the company allowed to use this project as a testing ground to train programmers in the domain and in the new methodology. However, during the term of the project, the technology team was fully engaged only on this project.

Near the end of the project (the last 3 months), when another large project was getting spun off, some project-switching took place.

Reportedly, a big culture mismatch occurred between testers and the rest of the team. Testers who were originally assigned to the project were accustomed to the old-fashioned way of working: *"the programmers would create software over the months, they would then throw it over the fence to get it tested for 2 months."*

When the team started to demand testers to produce a more rapid feedback – *"programmers are going to work for a couple of hours and they are going to build a feature, and we want you [the testers] to start testing right away and provide feedback right back into the team"* – many of the testers did not adapt well to that mode of work. A recruitment drive for new testers took place to bring more easily adaptable testers to the team.

Business experts and technology experts were collocated in a big open space with plenty of surrounding walls that were used as whiteboards. In addition, there were 6 movable whiteboards that could be used as partitions if necessary.

The project lasted two years and the team shipped a working, good-quality and feature-complete system to the customer's satisfaction (as per respondents' testimonials). This particular system (software plus device) is still being offered to their customers on the market today.

Two members of the team were interviewed for this case: (1) Chrysander, the project manager (who was also the coach) and (2) Talos, the user experience specialist (referred by the team as the "usability architect").

To get a sense of the project size, the total number of the acceptance test pages produced was about 500, with each page containing between 2 and 30 test assertions.

V.10.2 Findings

V.10.2.1 Learning the practice

Expert consultants introduced the practice of EATDD to the entire team along with other XP practices. A three day training was offered to technology experts and was sufficient to get them started: *"It was easy – it was just a technical problem that [we] had to solve"* (Talos). Business experts, apparently, required a bit more coaching. Chrysander elaborates: *"When doing a storytest, you have to really step back and think: What is that that I really want to test and what is that that I really don't want to test. The customers had a hard grappling with a notion of "I don't have to set my entire system up through test just to test one little thing or to specify one thing". So, for instance, if they wanted to test that an algorithm was working, they had a hard time thinking that, well, "I have to get the software to open up a file, then I guess I have to get a mouse push the "Find Metabolite" button, and I guess I have to get a table to go through each metabolite, and then I can finally look at metabolites that I want to look at". And we had ehhh….,you know, it was a rough road trying to get them to understand that we can set up everything programmatically – you just have to tell us what you want to look at. So that was a bit of a struggle. But they soon got over that by working with the programmers a lot. And sort of seeing how software is working… the customers who never really programmed before, started to learn more about how the software is put together and what things you can actually do with it."* Evidently, it is the potential of the software that the business experts were realizing. This increased understanding of what they can do with the software incited the discovery of additional features.

Importantly, this difficulty was more of a cognizant nature (thinking about the possibilities, thinking about the user needs, and deriving requirements from those). The operational and syntactic difficulties associated with using the FIT framework were quickly overcome in less than a month.

V.10.2.2 The process of requirements discovery and articulation

We now direct the line of inquiry toward the process of requirements discovery and articulation while providing a rich account of the ways this team specifically went about conducting these activities.

When a business expert comes up with a new idea, they typically meet every Wednesday for what is called "the customer team meeting". These meetings usually took about an hour and a half. During these meetings, the business experts get an opportunity to hash out these new feature ideas among themselves. The reason for a separate customer team meeting was explained by the project manager, Chrysander: "*One other thing we've noticed: when you have programmers, they tend to be like-minded – they, sort of, think alike and they come to an agreement very quickly; customers, because they have various backgrounds, they all have different points of view… so we give them a special meeting off, on their own, where they hash out the details of the feature that they want.*"

At the end of the week, the team holds an iteration retrospective and planning meeting during which business and technology experts discuss how well they did in the past iteration, calculate project velocity[23], and then discuss and plan features for the following iteration. The prioritized stories were placed on the board. At that point, business experts did not know which individual from the technology team would be working on which story. The technology experts did not know either. All that was known is that "*a set of programmers will *work* on it*". During the iteration, a pair of technology experts would *"walk up to the board, and put their name on the story, and find out, ok, which customer is going to help us [programmers] write the storytest [acceptance test]."* It was commonly known which business expert was going to write which acceptance tests, because "*if it's, let's say, a usability story, then we typically know it's going*

[23] The project velocity is a measure of how much work is getting done on the project, calculated by adding up the estimates of the user stories that were finished during the iteration.

to be Talos, our UI guy, our user experience architect. If it's a horrible algorithmic story, we know it's going to be (Carmelita), she is our domain expert… It's that type of thing" (Chrysander). Programmers can work on any type of story, because "*we [the team member] don't have our specialized areas, we are all *generalists**" (Chrysander). Once the technology experts (in fact, a pair of technology experts) identify which story they are going to work on, Chrysander explains, "*the customer comes over and that's when the conversation starts, that's when they start to write the storytest [acceptance test] together*." Afterwards, "*once the storytest is finished… well, I shouldn't say "finished" but… once the storytest is in *good enough shape* to start fixturing*[24] *it, the programmer will write up a fixture … they won't get it passing… they'll just bake any… anything that goes on the form*". Then, the programmers use acceptance tests plus the Test-Driven Development approach (described in §II.6) to implement the chunk of the system required to make the tests pass. The process continues with a demo to the business experts that the acceptance test was running and passing all the requirements. "*During all of that, the customer may come back, …and they may make changes, they may change their mind and we adjust to that …"* by replacing some of the originally planned but yet unimplemented functionality with the new one.

V.10.2.3 The meaning of "completed"

Once the coding is finished and the acceptance tests are passing, those get marked off for the task. As can be seen from the following passage, the mere fact of passing acceptance tests does not constitute the completion of the story:

"…In order for a task, for a feature to be complete, there is more requirements than just a [passing] storytest…., customer also needs to make sure that any UI is ok, that any technical writing, help, messages are done, that any performance criteria are met, and that any manual, or sorry, system testing is done by testers. So, there is a number of extra things on top of the actual acceptance

[24] Chrysander refers to the process of writing code that connects the acceptance test to the actual system under test.

tests…that mean "the story is done". And once every one of those criteria are finished, we then put a big green checkmark on the story to indicate that it has been accepted by the customers." (Chrysander)

V.10.2.4 Acceptance test authoring

Business experts usually drove the authoring of user stories and acceptance tests. "T*hey'll use all the domain terminology and they'll write the tests in their domain way; and by having conversations with the programmer at the same time, they'll think of things or new ways that they wonna test feature" (Chrysander).*

Business experts worked on acceptance tests in two modes. In the first mode, business experts start writing the tests on their own and if they find a similarity with other tests in the suite, they would have no problem in adding a new one by analogy. This typically involves modifying the dataset.

The prevalent modus operandi, however, is for a business expert to pair up with a technology expert. "*…If it's really something brand new, they [business experts] like to get a programmer's insight*" *(Talos).*

In order to make acceptance tests easier to follow, they were accompanied by embedded commentaries and occasional diagrams. Microsoft FrontPage was used for authoring and modifying FIT pages in the format of HTML.

V.10.2.5 Evolution of ubiquitous language

Chrysnader describes how the ubiquitous language evolved: *"We always encourage the customer team to use their domain language. We don't want them speaking in programmer's speak. We want them speaking in good old-fashioned chemistry talk or metabolism talk. We want them using words like "spectrometer", "chromatogram", "metabolite"…. we don't want them using words like "object" or "class" or "event" or anything like that…. we don't want to hear that."* Talos confirms that *"So, we use the storytest at that point with all the domain concepts in it that they[customer]'ve written, we use those to drive the development of our domain objects and all the objects that go into our source*

code, and ….” This demonstrates a solid commitment of the team to enhance their communication on the project. Even further, through this experience the team became convinced that *“the emergence of a domain language is unavoidable.”*

When we asked about transferability of the domain language to other projects, Chrysander’s opinion was as follows: *“I think it would be hard to re-use that domain language with a different customer (even if the domain is the same) because the dictionary in use comes from the people involved in the project … The proof might simply be that, when the tests are expressed in concrete terms (specific to a given project), the chance of re-using them as they are goes down because there are less chance of another product using the same terms.”*

Talos’s judgment was even more straight-forward: *“I'm quite convinced that the more concrete the language, the less reusable the fixtures, especially out of their context.”* This presents a challenge for knowledge transfer and reuse.

V.10.2.6 User interface acceptance tests

In the course of the present research, this was the first case study where a user experience specialist was involved full-time, and where some of the stories were clearly focused around the usages and usability. Let’s consider the following example of the usability acceptance story as described by Talos: “*[In] our software you have to open up a study. And if you can’t open up a study, you can’t process anything. Ok? Well, so, what’s the usability expert would say? I wonna write a storytest that says “If you don’t have a study open, then you are not allowed to process”. That’s really what type of story you are looking at there. Ehmm…. “if you have a sample and a control file opened, the software should have these buttons disabled, otherwise, if you have the vault open, it would have these ones all enabled”. That’s the type of things we are looking at. It’s really a workflow story - it’s like that. There could also be other things like “This dialog box will appear at certain times; and if it appears these conditions will be on, these port settings will be set on”. It’s those types of stories.*” Based on this and other descriptions, it appears that the “usability storytest” is really a type

of a workflow acceptance test, with heavy validation of control states and with all actions modeled and executed through the user interface. There is no higher abstraction present there. Without doubt, some of these stories are very difficult to implement. Despite this difficulty, the project manager and the team did see a clear benefit from automating acceptance testing of the UI layer: "*Once we get them programmed, it's very good, because over the years we all learnt that, you'll program something that if this checkbox is clicked, then those 4 buttons are enabled or disabled, but it's very easy to make changes in your code that will break all that stuff, because you rarely look into UI when you are programming. Ehh… it's just very nice to test this stuff in a storytest - to always make sure that your UI is working*."

Notice, these UI tests do not dominate. They are specifically designed to test the UI. The team indicated that they designed the system in such a way that most of implementation work is done "*just below the UI*", at the business logic layer[25].

V.10.2.7 Economic factors

On agile teams, it is typical to consider cost estimates during the feature prioritization process. On this team, the cost was also a prominent topic in conversations around the acceptance criteria. While business experts together with technology experts were involved in collaborative writing of these acceptance tests, they were also making decisions about the feature feasibility. The team called this activity "*bartering*":

*"… because they [business experts] are really worried about… well, not worried, but they are really *thinking* about the *costs* … a lot of the times…. And through the conversation with the programmer, they'll say "I think I want it to do this…", and the programmer says "If you do it that way, it will cost this much" (they'll think of this many hours or this many days) …. "but have you thought of, maybe, doing this for your …customer… your software? Because it*

[25] The team followed the Model-View-Presenter pattern [45], keeping the UI layer separate and extremely thin.

*actually gives you the same thing but a little cheaper…" and they [business experts] say "Oh, yeah"… and so, there is a constant *dialog*, back and forth, between the programmer and the customer… and then other customers may pop in and get involved, or other testers as well…" (Chrysander)*

Talos contributes his opinion on this issue: "*Our customers are very good at reducing costs now. Once… once they see that they get things cheap and fulfill the end-user requirements, they started really barter well. So they keep cost in mind, and they also keep end user need in mind. And by bartering back and forth and disagreeing, they usually come to a cheaper solution, that's relatively easy to implement and always fulfills the end user's needs. I've seen that work time and time again.*"

Both the program manager and the developer agreed that *"FIT is a good tool to use for getting your requirements down, but it also is a good tool for invoking a conversation that needs to happen to have the features implemented in a way that is economically feasible, but also in a way that's right."*

V.10.2.8 Resolving disagreements

Evidently, both the project manager and the user experience specialist indicated that during the process of communicating the requirements and hashing out acceptance criteria, the business experts frequently disagreed with technology experts. *"In those cases, sometimes a story gets dropped or it changes so drastically that it is difficult to recognize the original idea.*" (Talos). Despite this "constant disagreeing", the team morale was high. In fact, the respondents indicated that throughout the project they have learned to embrace the disagreement: "*We've learnt over time that *disagreement is good* because it's really,… it brings out *better solutions*… so, what will happen is … they[business experts]'ll go in with, sort of, "here's my vision of the feature should be"… and then they can get, sort of, a technical point of view [from technology experts] like, if you ask them "if you want it to be done that way, it's gonna cost a bit more than if you want it done the other way", that type of thing. Now when the customers are going back and forth, they are really*

trying… they never lose the spirit of what the feature is supposed to do, they never lose sight,… that's one thing they are good at doing, they never lose sight of what it is that they are trying to get accomplished for the end user. And just that they often go down different paths of exploration in their disagreement, to say: "You know, I really don't think we should be doing it this way; I think we should be doing it this way" and then "Why do you think that?", "Well, because of this", or "I didn't consider that… yeah, let's look down this path", or "No, we should be doing it this way"… They go down many paths with some technical guidance on what the costs of things will be." This process may sound disorganized as described, but in practice is quite rhythmic and productive, as the team is more comfortable with dealing with the levels of disagreement by constantly seeking resolutions through effective evaluation of alternatives and negotiation. This evaluation of alternatives leads to further inquiries and facilitates the discovery of the business requirements that may have been missed otherwise.

V.10.2.9 Improved communication

Both participants clearly articulated that EATDD improved communication among business and technology experts. Evidently, the practice provokes the team members to engage into conversations about the requirements and acceptance criteria. Chrysander enthusiastically states *"We value conversations more than the actual tool itself [FIT]"*.

Business experts also praise the FIT framework because they feel very comfortable with using tables: *"You know, they [business experts] can understand tables and they could fit their requirements in there. I am just trying to say, that we really value FIT because it helps to have those conversations [about requirements]… and the, of course, we value it, because it proves that we've met the requirements of the customer anyway" (*Chrysander).

V.10.2.10 Regulatory compliance - traceability

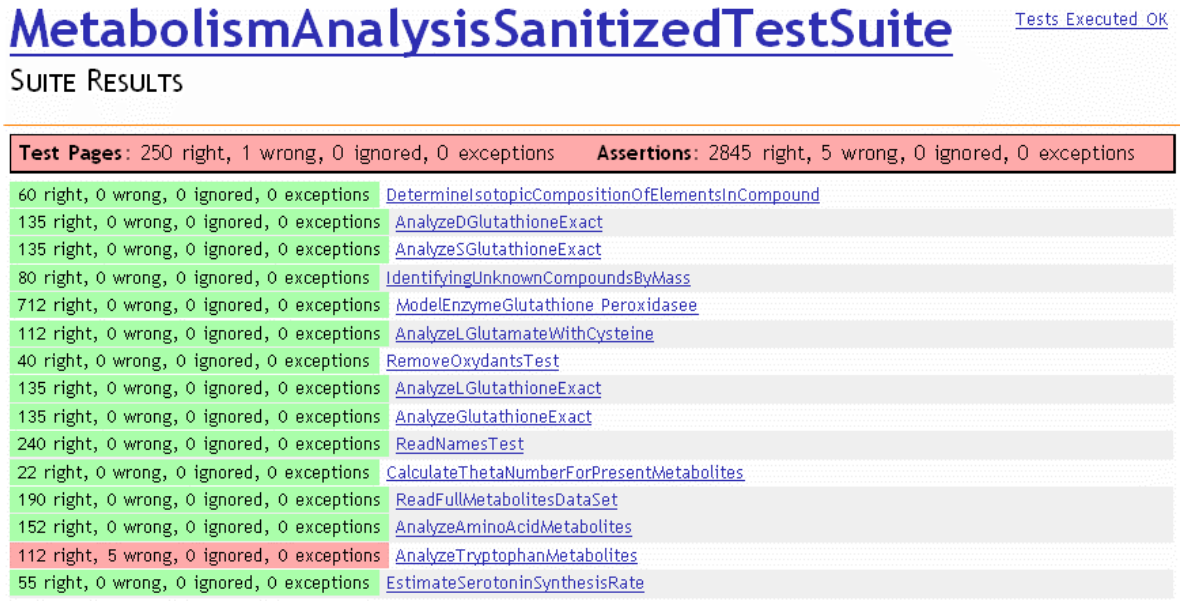
Clearly, as a company that builds software-intensive systems for pharmaceuticals, it has to adhere to Canadian, U.S. and European regulations and provisions (including those mandated by Health Canada (Health Products and Food Branch), U.S. Food and Drug Administration (FDA) and European Agency for the Evaluation of Medicinal Products (EAEMP)). Consequently, requirement traceability, as well software verification and validation activities are of utmost importance. According to the principle of the independent review, the company is regularly audited by the officials of the corresponding regulating bodies.

Prior to adopting EATDD, the development team was focused on producing detailed functional specifications, and implementing systems strictly to those specifications (as per ISO 8402:1994 standard). The auditors would *"come and read our functional spec, and make sure that if we said we were going to do this, we are doing it in the software"* (Chrysander). Now, when the team follows the XP process and articulates their functional requirements, mainly in the form of user stories and executable acceptance tests, they actually use those artifacts as their baseline specification. *"Now they [auditors] can look at our storytests [executable acceptance tests] - and we've run them for them! - and they look at them and say "oh, ok! I can see you are following all this because your documentation is completely in sync with your executable code"*. Essentially, auditors are looking for how the traceability of requirements is achieved, where "traceability" is typically viewed as "the ability to describe and follow the life of a requirement, in both a forward and backward direction, i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases." [54]

The team addressed the requirement traceability issues in the following way. In their centralized repository, web pages with the high level story descriptions are stored. Those pages include embedded hyperlinks to the corresponding executable acceptance tests. *"So, we have some folders,... HTML pages that we auto-generate. And we then have another program that we custom-wrote that*

has links in it; and at that point it's HTML files that have stories that are linked to the higher-level requirements. That seems to work fine. Our customer actually manages it fairly well now." Considering the fact that the executable acceptance tests are run against the actual system, the required level of traceability can be achieved. According to the participants' testimonies, the team was able to track relationships between the high-level requirements, individual user stories, acceptance tests, and code, and analyze the impact when changes occurred. In fact, any change that breaks the system will be visible on the test run status page (see Figure 22).

Traceability at the level of changes authoring ("who made which changes and when") is achieved through employment of revision controls systems (such as `cvs`, `Subversion`, `SourceSafe` etc.)[26]. This is possible because the acceptance tests-requirements are specified in the plain text as opposed to some proprietary format.



MetabolismAnalysisSanitizedTestSuite

Tests Executed OK

SUITE RESULTS

Test Pages: 250 right, 1 wrong, 0 ignored, 0 exceptions	**Assertions**: 2845 right, 5 wrong, 0 ignored, 0 exceptions
60 right, 0 wrong, 0 ignored, 0 exceptions	DetermineIsotopicCompositionOfElementsInCompound
135 right, 0 wrong, 0 ignored, 0 exceptions	AnalyzeDGlutathioneExact
135 right, 0 wrong, 0 ignored, 0 exceptions	AnalyzeSGlutathioneExact
80 right, 0 wrong, 0 ignored, 0 exceptions	IdentifyingUnknownCompoundsByMass
712 right, 0 wrong, 0 ignored, 0 exceptions	ModelEnzymeGlutathione Peroxidasee
112 right, 0 wrong, 0 ignored, 0 exceptions	AnalyzeLGlutamateWithCysteine
40 right, 0 wrong, 0 ignored, 0 exceptions	RemoveOxydantsTest
135 right, 0 wrong, 0 ignored, 0 exceptions	AnalyzeLGlutathioneExact
135 right, 0 wrong, 0 ignored, 0 exceptions	AnalyzeGlutathioneExact
240 right, 0 wrong, 0 ignored, 0 exceptions	ReadNamesTest
22 right, 0 wrong, 0 ignored, 0 exceptions	CalculateThetaNumberForPresentMetabolites
190 right, 0 wrong, 0 ignored, 0 exceptions	ReadFullMetabolitesDataSet
152 right, 0 wrong, 0 ignored, 0 exceptions	AnalyzeAminoAcidMetabolites
112 right, 5 wrong, 0 ignored, 0 exceptions	AnalyzeTryptophanMetabolites
55 right, 0 wrong, 0 ignored, 0 exceptions	EstimateSerotoninSynthesisRate

Figure 22. Fragment of a Sample Test Suite Execution Results Page with One Test Failing.

[26] Note that `FitNesse` [40] supports this kind of traceability by automatically versioning all changes made to the acceptance tests.

There are additional activities performed by the team (such as, risk management, problem resolution procedures, threat modeling etc.), which are out of scope of this dissertation.

V.10.2.11 Test execution

Similarly to Case Alpha (described in V.9), the team diligently followed the practice of continuous integration, where technology experts integrate their work frequently – multiple times per day. Each integration is verified by an automated build that includes running all tests to detect integration errors as quickly as possible. Talos, the user experience specialist, explains: *"Whenever we do our integration cycle, the programmers run them… and any time, an installer, sorry, anytime a tester wants to pick up the latest installer, they build the installer and the build automatically runs it [FIT]. So, our build [script] is written in such a way that you can't build software without running all unit tests and all storytests*." And the build script does not submit any new or modified code to the repository, unless all tests pass.

During the development cycle, pairs submit their updates about once every two hours. There were 3 pairs of developers working on the project. Hence, at a minimum, the acceptance tests were executed 12 times a day, each run taking about 3 minutes. There were several acceptance tests (referred to as "infrequent tests") that were excluded from the automatic build process. The reason is because they ran over very large datasets which took 15 minutes or more to finish. Chrysander reveals that in spite of the technology team trying to "*convince our customer and say "you don't have to run really big dataset through – a small dataset will do", …they [business experts] really want a confidence to say "no, here's a really big dataset and we want to see the real results from it*". These infrequent tests ran during nightly builds.

V.10.2.12 Test retirement and test maintenance

The majority of the acceptance tests produced over two years of development, remained in the regression suite. Few test cases were removed from the suite

("retired") but only for the reason of a certain feature being taken out. The project manager gives a concrete example of this in the following statement:

"If our usability expert goes out into the field and finds out that a certain feature is not necessary. Let's say… I am going to use some words from my domain, but let's suppose "smoothing a peak" is not longer needed. So, what we'll do, we'll take the "smoothing the peak" feature out and we'll take the tests out as well, because the tests can no longer run - there is no feature running… and that's the only scenario under which the tests will be retired."

Sometimes the tests changed – those changes were motivated by the real feedback from the field (the team referred to these changes as "macro level changes"). The project manager explains: *"Because we may find, through usability studies or through interviewing of real paying customers, that they do things in a different ways, or things that we did, don't quite work - so, we will modify the existing tests to reflect any changes to features."* Further investigation into the nature of the change reveals the following details. The team found that while there was practically no change to the core algorithms (those were implemented from the very beginning and remained fairly stable), there was substantial change around the workflow and interactions – in other words, how users interact with the software. This finding was not surprising. UI interactions are considered to be one of the most fragile operations on software projects. That is why earlier approaches to automated acceptance testing which were based on the "record & playback paradigm" fail. Meszaros in [99] describes the "fragile" test problem and analyzes in detail their common pitfalls.

A different type of change ("micro-level change") is necessitated internally, when the business experts introduce new ideas to the stories after the initial story is written up:

"When you start developing of a feature, the customer will sit down with a programmer and together they'll write a storytest. Now, the customer usually drives there…. So, once they are finished writing their storytest in FIT, in HTML, they give it to the programmer, the programmer fixtures it up,

*implement the feature, but then the customer might come back a few hours later or an hour later and say "You know what? I was just thinking about that… and we should really change this". So, the whole pack in the storytest changes a little bit… Make it a little different, make it more suitable. And as a programmer, we'll go back and start coding *that* storytest."*

V.10.2.13 Executable acceptance tests vs. other requirement specification techniques

Respondents were asked to compare and contrast their current process of specifying, communicating and verifying requirements using executable acceptance tests with the other techniques they have used in the past.

Chrysander explained that in the previous 12 years of his career, the predominant way of dealing with requirements was through functional specification documents written in the style of "The system shall…" His opinion of the EATDD was that *"it's a lot less ambiguous!"* Chrysander supported his assertion with the following statement: *"… because you have that conversation, because the tests are being written as you are programming and as you are speaking, there is a lot less ambiguity in there. Your customer can see right away … that the requirement been met because it's executable. And because it is surrounded and supported by the conversations and the community, the rest of the team, you know that you are producing the right thing. So, what we end up getting is a lot less requirements-based defects.*" In fact, it turned out that the company performed a study using the data from their defect tracking system (with over 1,000 data points analyzed) and they discovered a large percentage of those defects to be related back to incorrect requirements specifications. In contrast, "*now, in general, our projects get a very low amount of defects anyway, and that's due to the fact that we are getting our requirements right through TDD. But any defects we have, a smaller portion of those are due to requirements specifications, you know. And so, this type of specifying with Fit really forces everybody think about exactly what they really want and get it down to write in an unambiguous way that can be executed.*" Chrysander concludes "*I think the*

*execution is the thing that makes it unambiguous. It's either *green* or *red**! <laughs> *... and I think it's miles ahead of what we used to have!*"

V.10.2.14 Limitations

Despite the evidence that the number of requirements-related defects was dramatically reduced, the respondents indicated that a few of those defects were still present. Subsequent inquiry revealed an insight as to why this may be the case. Chrysander explains: *"At a certain point in our project, we had about 70 defects that piled up. 70 defects - that doesn't sound like a lot for most projects, but for us it was, it was a major concern! And so we wrote them all on the whiteboard and went through each one to find out what the problem was and where they were... and So, most of the requirements ... sorry, most of the defects that were requirements-based, were because they were missing requirements, or assumed requirements. So,... well,...it was like "We assumed that you would check that a number was out of range", or "We assumed that you would follow certain UI guidelines", and "We assumed that..." ... there were always a lot of these "assumed" requirements."*

Another example of an assumed requirement is related to with the consistency-with-other-product-functionality heuristics: "*Well, we see this in all other pieces of software, we assumed you would do it here too"* (Talos).

This is a clear indication that EATDD is not a silver bullet and that a high degree of discipline in communicating and revealing the assumed requirements is needed. This team eventually rectified this problem by agreeing to be even more specific – *"Let's ask ourselves what, you know, what things in here we are going to assume... And so you can have that just like a thought provoker or you can even write executable storytests around those assumed requirements, that just get executed on various features.*" (Chrysander)

V.11 Validity of Qualitative Studies

For qualitative studies to be valid, they must accurately represent the phenomena to which they refer and be backed by evidence. In order to achieve this goal, we have diligently presented evidence from all industrial case studies in great detail.

To establish validity in our studies, we used the following triangulation methods:

1) **Data triangulation.** We used different sources of information: interviews, surveys, coding and testing artifacts.

2) **Methodological triangulation.** We used a mixed approach, combing case studies with the grounded theory analysis, comparative analysis of multiple perspectives. Those were also completed with the results of the surveys and quantitative studies.

3) **Environmental triangulation.** Subjects from three different companies and settings were engaged in this research. This aspect of validity can clearly be improved by expanding the line of inquiry further to other teams, companies and industries.

4) **Investigator triangulation.** Two types of investigator triangulation were employed:

    i. **Internal consistency checks.** We performed internal consistency checking of the first ten pages of each transcript three weeks after the initial coding. The mean value of our internal code-recode reliability (measured as # of agreements / (number of agreements + number of disagreements) was 82%. There were few conflicts, mainly omissions and overlaps.

ii. **External check-coding.** Two individuals (one from industry and one from academia) were asked to recode the same segment of an interview transcript with a given set of codes (produced by the researchers). The findings were compared. The external code-recode reliability of the academic expert and industry experts were 60% and 65% correspondingly. Though not excellent, these are considered to be satisfactory in the qualitative studies

5) **Theory triangulation.** The results of this research were reviewed, published and commented on by multiple professionals outside of the academic research. No obvious conflicts were detected.

Chapter VI Synthesis of Findings from Quantitative and Qualitative Studies

Chapter IV and Chapter V presented two academic observational studies, one academic quasi-experiment and two industrial multi-case studies. The academic studies focused on the abilities of business and technology experts to interpret and to author executable acceptance tests, while the industrial qualitative multi-case studies focused additionally on the essence of how business and technology experts of software engineering teams utilize EATDD and what their experiences look like. Throughout all analyses, we also addressed the questions of learnability and ease of use of EATDD and the supporting frameworks and tools (FIT and FitNesse). In this chapter we synthesize the findings and make analytical generalizations about the ways EATDD facilitates software requirements discovery, articulation, and validation, based on the patterns and trends emerging from the empirical evidence.

VI.1 Emergence of main categories

Through summative analysis of the data and findings of all studies, we have identified the following 4 main categories manifesting the facets of employing EATDD (see Figure 23):

- **Requirements discovery** (includes activities of the problem analysis, fact finding, domain exploration and familiarization, and collaborative discovery of requirements through "warm" communication);
- **Requirements articulation** (includes methods of articulating requirements and representations of the domain (in the form of the

executable acceptance tests) among various stakeholders; types and attributes of the produced executable acceptance tests; plus emerging patterns of their design);

- **Achieving confidence** (includes activities to achieve high quality and confidence in the system's implementation with executable acceptance testing, regression testing, and other types of testing, continuous integration, fast feedback, requirements traceability as well as social implications and project management aspects);
- **Challenges** (summarize difficulties associated with utilizing EATDD based on the experiences and perceptions of business experts and technology experts, as well as limitations on the types of domains and contexts).

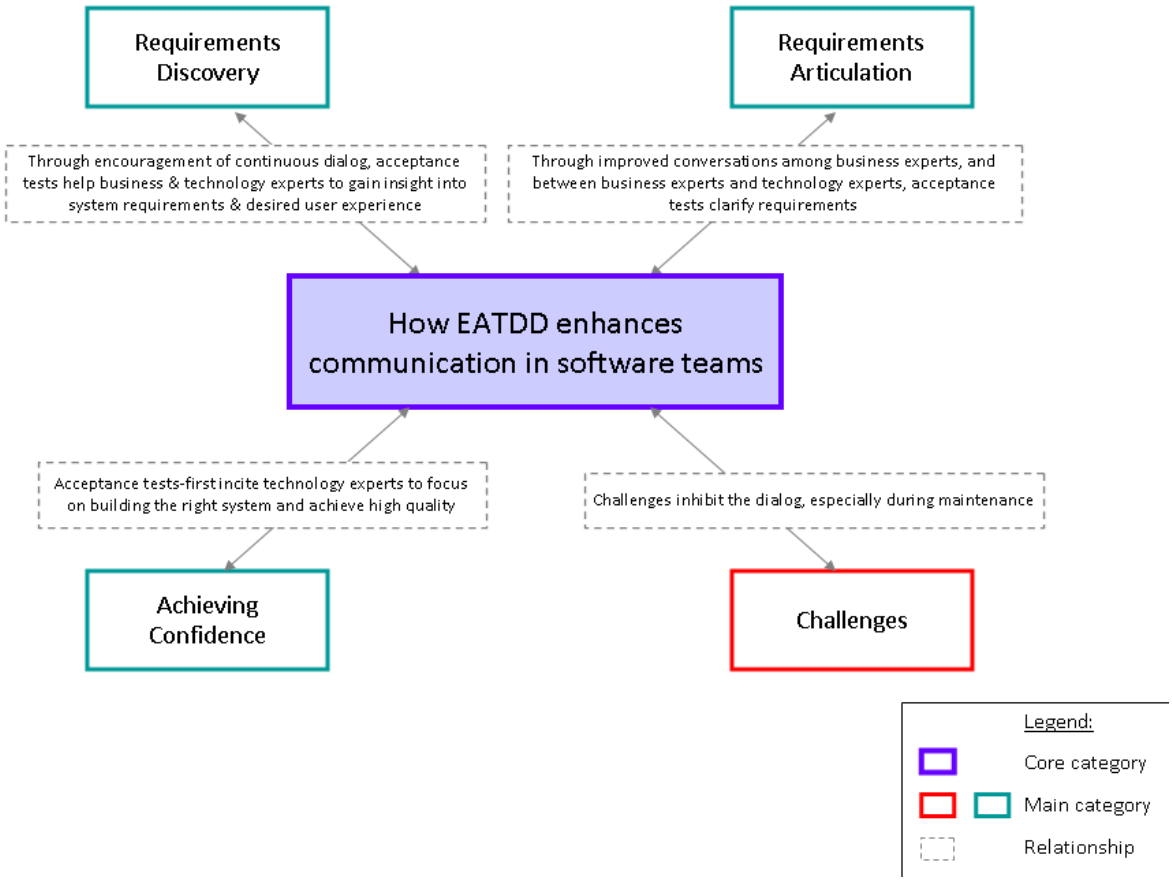


Figure 23. Relationships between main categories

The following subsections summarize each of the main categories.

VI.1.1 Requirements discovery

Requirements discovery activities did not transpire in the first two quantitative studies with undergraduate students. Very little exploration into the problem space and the domain was demonstrated. This can be attributed to the timing of the assignments (the end of the semester), the short time frame given for inventing and authoring of new requirements on the project (one week), and also possibly, to the lack of practical guidance from the TA's involved.

These activities were clearly exhibited by the subjects in the quasi-experiment when business school graduate students together with computer science graduate students performed the role of business experts and undergraduate students were responsible for technical implementations of the system. The central hypothesis of that study (that business experts would be able to effectively describe functional requirements of the system in the form of executable acceptance tests so that a development team could later implement those features) was supported as the majority of the teams were able to produce a high quality executable specification (see §IV.3.4.1). This time, they were given significantly more time – four weeks as opposed to one. In addition, there was a clear separation of responsibilities. While in the first two studies subjects had to play dual roles of both business experts (for other teams) and technology experts (implementing requirements of some other teams), in this quasi-experiment, business experts could focus primarily on requirements articulation.

In addition, the industrial case studies provided sound evidence of the motivating power of executable acceptance tests as business experts found new ideas about the desirable features of the system while discussing business rules and flashing out acceptance criteria for those (in the EATDD way) (§V.10.2.2). After all, acceptance tests are examples of how the software system will be used. There is a high probability of discovery through elaboration and experimentation with examples, which is increased when business experts are paired with technology experts.

VI.1.2 Requirements articulation

VI.1.2.1 Interpreting executable acceptance test specifications

The communicating power of EATDD was demonstrated throughout all five studies. In the first two academic studies, the requirements were articulated by the instructor (playing the role of a business expert) completely in the form of executable acceptance tests and the technology experts (students) found little or no problem in comprehending those tests and deriving the requirements from them with very limited clarification from the business expert. It is important to recognize that the validity of those findings may be threatened by the fact that the instructor was an expert in the areas of requirements engineering, software testing, and the EATDD process itself.

VI.1.2.2 Authoring executable acceptance test specifications

In the third academic investigation, we decided to mitigate this risk by assigning the role of business experts to a separate group of graduate students. Teams of business experts (graduate students) and technology experts (undergraduate students) were randomly formed. They had to work together to implement a chunk of business functionality within one iteration. Business experts worked in pairs. Their communication with the technology experts was predominantly asynchronous (mainly due to scheduling constraints). As evident from the implementations produced by the technology experts, all team members accomplished their tasks very well. This is worth noticing because it exemplifies the potential of the EATDD approach for adoption by distributed teams.

Surprisingly, however, the quality of the produced executable specifications did not correlate with the quality of the resulting implementations. One possible explanation to this is the fact that the quasi-experiment was conducted over a single iteration only. Another explanation could be based on the nature of the academic settings and the type of student motivation that is different from the practitioners'.

Overall, subjects of academic and industrial studies (with the ones from industry being more enthusiastic) praised the fact that the executable acceptance tests were very concrete in nature and, as such, they significantly diminished (but not eliminated) ambiguity in specifying software requirements.

VI.1.2.3 Suitability executable acceptance tests for specifying functional requirements

"Suitability" was evaluated as a degree to which the functional requirements in the form of executable acceptance tests (in FIT) are found to be unambiguous, verifiable, consistent, and usable by all project stakeholders – business and technology experts – for understanding the software system. The studies evaluated how EATDD performs with regard to mitigating risks discussed in §II.1. *Noise* is greatly reduced when using executable acceptance tests to represent requirements. Irrelevant information is more difficult to include in well structured tables than in narratives. Also, tests which shade or contradict previous tests are easily uncovered at the time of execution – and vice versa. As a result the conflict is highlighted to the developer who can then discuss it with business stakeholders to resolve it. Acceptance tests can be used as regression tests after they have passed in order to prevent problems associated with possible noise.

We discovered that *silence* is not well addressed by the EATDD approach – the problem of implied but not externalized requirements still exists. Technology experts might assume that the tests are complete – although they are not. This was well demonstrated by the failure of our teams in the academic study one to test at least 50% of the requirements for which no tests were given. Our example of case-sensitive document types also clearly demonstrates how a lack of explicit tests can mislead developers and create a false sense of completeness. Prose documents may be obviously vague, and by this obviousness incite additional communication. Executable acceptance tests are very concrete and require proactive thinking about missing requirements.

Overspecification is not a big problem since executable acceptance tests do not allow any room for embedded solutions in the tests themselves. Executable acceptance tests represent customer expectations, and the underlying plumbing (fixtures) becomes the agent of the solutions. Although it can be argued that specifying workflows (like it is done in ActionFixture or DoFixture style tables in FIT) describes a sequence of actions (and therefore a solution), when writing acceptance tests these actions should be based on business operations and not code-level events.

Wishful thinking is largely eliminated by EATDD, since defining acceptance tests requires that the business experts think about the problem and make very specific decisions about acceptance criteria for the solution being built. In turn, technology experts get concrete requirements that allow them to highlight technical issues resulting from the requirements. If the team follows agile estimation practices (where technology experts estimate the effort for each requirement), these issues will typically lead to high estimates (i.e. high development costs). In turn, business experts can decide if they are willing to pay the price for fulfilling their wish.

Ambiguity may still be a problem when defining requirements using executable acceptance tests if keywords or fields are defined in multiple places or if these identifiers are open to multiple interpretations. However, acceptance tests diminish ambiguity simply because they use fewer words to define each requirement.

Forward references and *oversized documents* may still be an issue if large numbers of tests are present and not organized into meaningful test suites. Manageability and maintenance of large suites is also an issue as evident from the industrial cases.

Reader subjectivity is greatly reduced by executable acceptance tests. As long as tests return their expected results when executed, the developers and business stakeholders know that the corresponding requirement was correctly interpreted regardless of the terminology used. A test either succeeds or fails – there is

nothing left to interpretation. Acceptance tests can be executed by the business expert or in front of the business expert, and business experts can quickly evaluate project progress based on a pass or fail condition. Some tools are even able to chart the number of pass/fails over time and show if more and more tests are passing over the course of an iteration.

Customer uncertainty may manifest as the previously mentioned problem of silence, but it is impossible for a defined test not to have a certain outcome. Tests are executable, verifiable and easily readable by the business experts and technology, and therefore there is no need for *multiple representations* of requirements. All necessary representations have effectively merged into a suite of tests.

In addressing the characteristics of suitability, our findings demonstrate that executable acceptance tests can be used as functional requirements specifications and are in fact unambiguous, consistent, verifiable, and usable (from both the business experts' and technology experts' perspectives).

VI.1.2.4 Tabular representations

The results of post-mortem surveys of the academic subjects and the industrial case studies show a clear preference of the business experts to use tabular format for specifying their acceptance tests. A simple refactoring of an acceptance test written in xUnit-style into a tabular form was perceived by business experts to be a significant improvement in readability (V.9.2.8). Another factor why this may be the case is because most business experts are very familiar with spreadsheets and accustomed to using tabular representations.

VI.1.2.5 Normal and deviant scenarios

With regard to the types of test specified, our hypothesis that positive tests are prevalent in the executable acceptance test specifications was supported by all studies. It is important to note that industry participants were much more aware of negative scenarios (deviant cases, misuses, abuses) – this was expected as their experiences dictate more serious attention to the negative test cases.

VI.1.2.6 Formation of ubiquitous language, motivation for reuse

Through analysis of the industrial cases, we found that EATDD played an important role in normalizing domain languages that business experts and technology experts used for their respective projects; yet this was not obvious from the student data (perhaps due to a short-temporal nature of the academic projects and lesser focus on standards and reuse). In the industrial cases, however, it became apparent that EATDD motivated a great deal of reuse of action verbs, terms, and test cases. It also provoked discussions about the meanings of the specific terms. In case Gamma, participants spoke eagerly about the formation of the ubiquitous language and the benefits of all members of the team communicating in the ubiquitous language. It seems that in the context of EATDD, the complexity of a domain is not an inhibitor but, in contrary, it creates an additional positive motivational influence towards formation of the ubiquitous language.

VI.1.2.7 Patterns

The reuse theme proclaimed itself also in the test design patterns, specifically in the "Build-Operate-Check" pattern, which was common across most academic and industrial workflow tests. Other patterns detected are "Fixture setup", "Common includes", and "Transaction rollback".

In the academic study two, we also detected two other patterns – incremental addition of passing assertions (which is consistent with an incremental approach to software development) and a common use of preferred FIT fixture types (no confirmation in the industry as the participants seem to use a diverse set of styles).

VI.1.3 Achieving confidence and improving quality

VI.1.3.1 Credibility and business focus

By their nature acceptance tests do not possess a high detective power (i.e. the ability to detect the defects in the system), for which other testing techniques such as stress, risk-based, and domain testing are so good for [70]. Rather, the main objective is to convey and clarify the intentions and desires of the business experts to technology experts, and then later verify that those were in fact implemented. When authored by business experts (even if paired with technology experts – which we found to be a more common work pattern), these tests carry the credibility and help steer the project towards what is important to the business. This way, the business perspective is always at the forefront.

VI.1.3.2 Early test design leads to better requirements

Test-first is a powerful paradigm that requires a great deal of discipline. The test-first aspect of EATDD (early specification of executable acceptance tests before coding starts) evidently helps to avoid unnecessary work and re-work caused by requirements bugs (which are the costliest). It creates a clear context for business experts and technology experts to communicate and to weed out misunderstandings. Participants in our industrial studies reported fewer numbers of defects in the production code related to incorrect requirements. This finding is consistent with the experience report from Neilsen Media Research [116], which concludes that this type of conversation results in reducing the risk of building the wrong system.

VI.1.3.3 Frequent feedback

Another important aspect of EATDD that helps in building confidence in the systems being built became apparent through empirical studies. This aspect deals with the frequent demos of the working functionality to the business experts and reflections on the latter. It is common during the demos to run the suites of acceptance tests. Through this execution, the system provides an immediate and unambiguous feedback on what works. After all, there can be no ambiguity about

a requirement expressed as an acceptance test, if that acceptance test can turn a light red or green. These test runs provide visibility to the project. At any moment, any member of the team (that includes business experts) can execute the entire suite and get a status of the project.

VI.1.3.4 Related activities

It is also common for teams to incorporate executable acceptance test runs in their build process. This provides an additional rigour and discipline as usually no changes are allowed to be committed until all tests pass.

To balance the low detective power of the acceptance tests, other types of testing are necessary. Interviewees stated that exploratory testing and specialized testing (e.g. fuzz testing, interruption testing, compatibility testing, system stress testing etc.) were utilized both by the in-team testing professionals and the external testing teams.

VI.1.3.5 Traceability

The issue of traceability was not on the research agenda until the second industrial multi-case study. Therefore, our findings are primarily based on the data from that case. It was significant that the auditor found the prerequisite of traceability satisfied even though no formal requirements specification (in the traditional sense) was produced. The live demos of executing acceptance tests directly against the software system serve as a powerful evidence of the traces between requirements and code.

Since executable acceptance tests are in textual format, traceability at the level of who made which changes and when can be achieved through employment of revision controls systems (such as CVS, Subversion, SourceSafe, etc.). Note that FitNesse supports this kind of traceability by automatically versioning all changes made to the acceptance tests.

VI.1.3.6 Embracing change

One of the common problems with functional specifications is that after a substantial amount of time has been invested in their production, a simple requirements change may render them out of date. In addition, when different teams and tools are used to produce acceptance tests, the problem of keeping those functional specs and suites of acceptance tests in sync arises.

In the case of EATDD, if the requirements changes, the acceptance test that defines or accompanies it will change as well. This is a part of the process and team members are disciplined about it – no functional requirement change can take place unless the acceptance criteria for it, change.

EATDD executable specs are active and alive at all times. Change through experimentation is encouraged through "what-if" scenarios, which anyone on the team can easily and safely produce by modifying the existing acceptance tests and testing how the system would react.

VI.1.3.7 Social implications

Our empirical evidence from the industry suggests that since EATDD encourages conversations, diverse talent collaboration is strongly promoted. As a result, technology experts learn more about the domain, and the business experts learn more about the underlying technology. This is a two-way relationship. It also facilitates and promotes serious training-on-the-job. We have not seen this effect in student teams though – largely due to the limitations of the academic environments.

The test-first aspect of EATDD also seems to play a role in confidence boosting by providing a safety net comprised of acceptance tests that will notify the technology expert as soon as a failure occurs. The fear of touching other peoples' code or legacy code is reduced.

VI.1.4 Challenges and limitations

Besides seemingly optimistic results, our findings discovered a set of challenges in adopting and using EATDD. Figure 24 summarizes the challenges identified from the data.

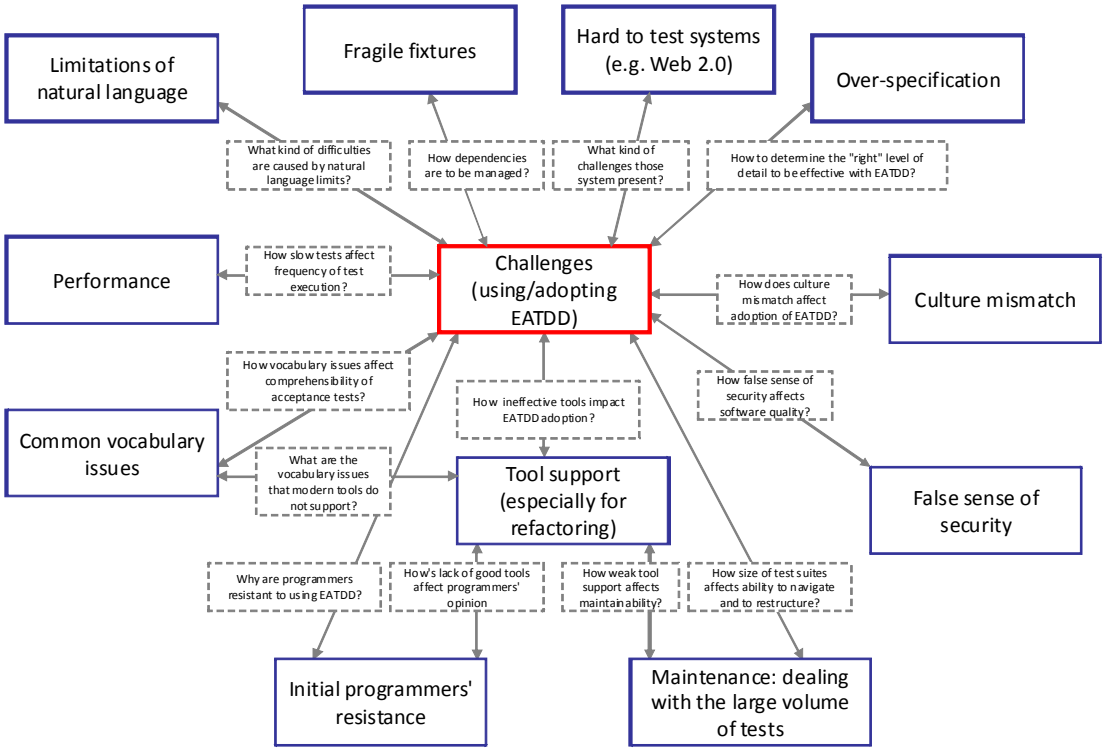


Figure 24. EATDD challenges.

As expected, the top challenge of academic subjects was different from the one perceived by industrial subjects. Specifically, undergraduate students found specifying acceptance tests difficult. This is not surprising because thinking of a well defined scope and example requires practice and experience, which these subjects were lacking. As evident from section IV.3, a number of graduate students experienced difficulties with expressing their requirements as acceptance tests. They suggested that using alternative notations (like diagrams)

would have been easier. From the standpoint of writing fixtures, the frequent error was related to the "fatness" of fixtures, which made them very fragile.

Students also struggled with the question of how one knows when one has collected an appropriate set of acceptance tests to describe the current problem. This is related the issue of silence but is not the same. Students got confused whether the set of examples they've provided was sufficient to explain what they wanted the system to do. This is a big question of representativeness and it applies equally to industrial teams. It is also typical of other types of scenarios (see e.g. [132]).

When teams of business experts were assembled (for academic study two), one of the issues had to do with using common vocabulary. Analysis artifacts (students' acceptance test suites) revealed numerous cases of synonymic use of terms (which resulted in growing the fixtures) without any attempt to reconcile them.

As discussed in sections V.10.2.14 and IV.1.6, assumed requirements are still a problem. Even though due to increased communication, there should be fewer of those.

Apparently, the last two problems described above also occur in the industrial teams. The biggest concern for industrial team members, however, turns out to be the issue of maintaining the suites of acceptance tests. Weak tool support is one of the factors impacting maintainability of the acceptance test suites. Navigation and refactoring support was among the key pain points. Artifact analysis (Industrial Case Alpha) proved to be fairly difficult even for the researchers who are familiar with EATDD and FIT. The test pages were so long (dozens of screen scrolls) and wide (dozens of columns), that navigating and deriving meanings from the tests was extremely difficult.

If the teams want the tests to survive and to be used in the post-release operations (during either corrective or perfective maintenance conducted by a different group of technology experts), it is imperative that the technology experts are able to locate the tests quickly, read and understand them, make necessary tweaks, and then execute them. Andrea advocates operations teams to

take on the test-first approach when doing maintenance [7] Simple refactorings such as renaming fixtures/tests, moving columns, reshuffling the rows, context replacement, parametrizing test cases, along with IntelliSense support are desired. The tool smith community is taking notice (new tools are introduced but are still far behind the levels of support and integration, unit testing has got).

Another concern that manifested itself in both industrial and academic studies is the initial resistance and even pushback by technology experts. It is explained by a perceived extra work that is assigned to the technology experts ("fixturizing" acceptance tests and linking them to the actual software system). This challenge is similar in nature to the challenge of unit testing done by programmers (when JUnit was first introduced), i.e. some perceived it to be extra work. As the current state of the practice shows, this challenge was overcome and programmers' unit testing is now a standard practice of many software development companies. Besides, as the technology experts begin to see a long term benefit of improved quality (due to clearer requirements and regular system regression testing performed with acceptance tests), the initial resistance disappears: *"…even though developers complained that acceptance tests created more work for them, … at the end they were saving a lot of time – because of regression testing"*.

The cultural mismatch of old-fashioned testers and agile testers may also impact the levels of adoption and penetration of EATDD. On one hand, old-fashioned testers, who are not used to highly iterative approaches, find the volatility of the acceptance test suite disturbing. On the other hand, it seems to be hard for them to think outside of the box: **"…***so, when they [old-fashioned testers] think 'automated test', they think 'I want to automate the way I test manually'…Anything that doesn't look like automation of a manual test feels funny to them"* (Chrysander).

Requirements gathering tools can be problematic when they limit the types of requirements that can be captured. Executable acceptance testing frameworks and tools are no exception. It can be difficult to write some requirements as acceptance tests, and it is often necessary to extend the existing plumbing, or to

utilize prose for defining para-functional requirements and making clarifications. However, prose can be embedded in acceptance tests or defined through a collaborative wiki such as FitNesse, and this may help overcome the limitations of bare tests.

VI.2 Core category

In addition to the main categories, the qualitative data suggests that ***EATDD is correlated to the enhanced communication in software teams*** (based on the perceptions of the subjects). Without strong experimental evidence, however, no causality can be inferred. This relation of EATDD and enhanced communication is the central category and it is associated with all other main categories identified above: requirements discovery, requirements articulation, achieving confidence, and challenges.

VI.3 EATDD from a socio-technical perspective

A socio-technical system is hybrid in nature. It is made up of individuals, technologies, processes, and information. It requires successful integration of all these elements for its proper functioning. Based on the analysis of the activities performed by both business experts and technology experts in student teams and industrial teams, we propose the following socio-technical view of EATDD (Figure 25). The key elements of the system are:

- business experts
- technology experts
- EATDD process
- executable (acceptance test) specification
- ubiquitous language
- software system

- quality characteristics of the system perceived by business and technology experts.

In the case of the regulated environment, additional elements may include auditors and their perceptions of the process quality and system quality (denoted in blue in the diagram).

The key relationships and interactions among elements of this system include:

- using the EATDD process, business experts and technology experts discuss and refine functional requirements in a precise way by providing examples of the intended system usages;
- these examples serve as acceptance criteria;
- these acceptance criteria are specified in the form of executable acceptance tests ;
- these executable acceptance tests are aggregated into test suites and suites of suites;
- at the highest aggregate level, these suites represent an executable specification;
- the process of EATDD facilitates formation of the ubiquitous language;
- the ubiquitous language improves communication between the technology experts and the business experts;
- as more and more requirements are discussed and acceptance tests are specified, the emerged ubiquitous language affects the EATDD process and the executable (acceptance test) specification;
- the executable specification provides details about the business functionality of the software system;
- the code traces back to the executable specification (via underlying fixtures);
- this executable specification is accompanied by additional commentaries and diagrams if necessary;

- the executable specification also verifies and validates the system built, and affects its quality as perceived by business experts, technology experts, and auditors;
- the executable specification provides immediate feedback to business experts, technology experts, and auditors about the system in the form of test results;
- the software system itself provides feedback to business experts, technology experts, and auditors;
- the software system possesses qualities that business experts, technology experts and auditors perceive in their own ways; an agreement of those perceptions must exist for the system to be successful.

Though the process of acceptance tests execution is automated, it is important to note that the process of requirements discovery and articulation is not mechanistic. It is an intellectual process.

Also, writing requirements in the form of acceptance tests should not be confused with some earlier approaches that auto-generated test scripts from requirement specifications, finite state machines, activity diagrams, etc. These approaches were not very successful in practice. *"The main problem was that the scenarios developed during requirements engineering and system design were out of date at the time the system was going to be tested"* [137]. Neither should these requirements-tests be confused with "operational specifications" that support formal reasoning (such as Gist [25], Statemate [56] or PAISley [144] with derivatives), which are powerful but quite cryptic for an ordinary business person to comprehend (not to mention to write). On the other hand, the coded nature of the operational specifications does not require any additional manual mapping; while FIT acceptance tests require such mapping (of fixtures to code).

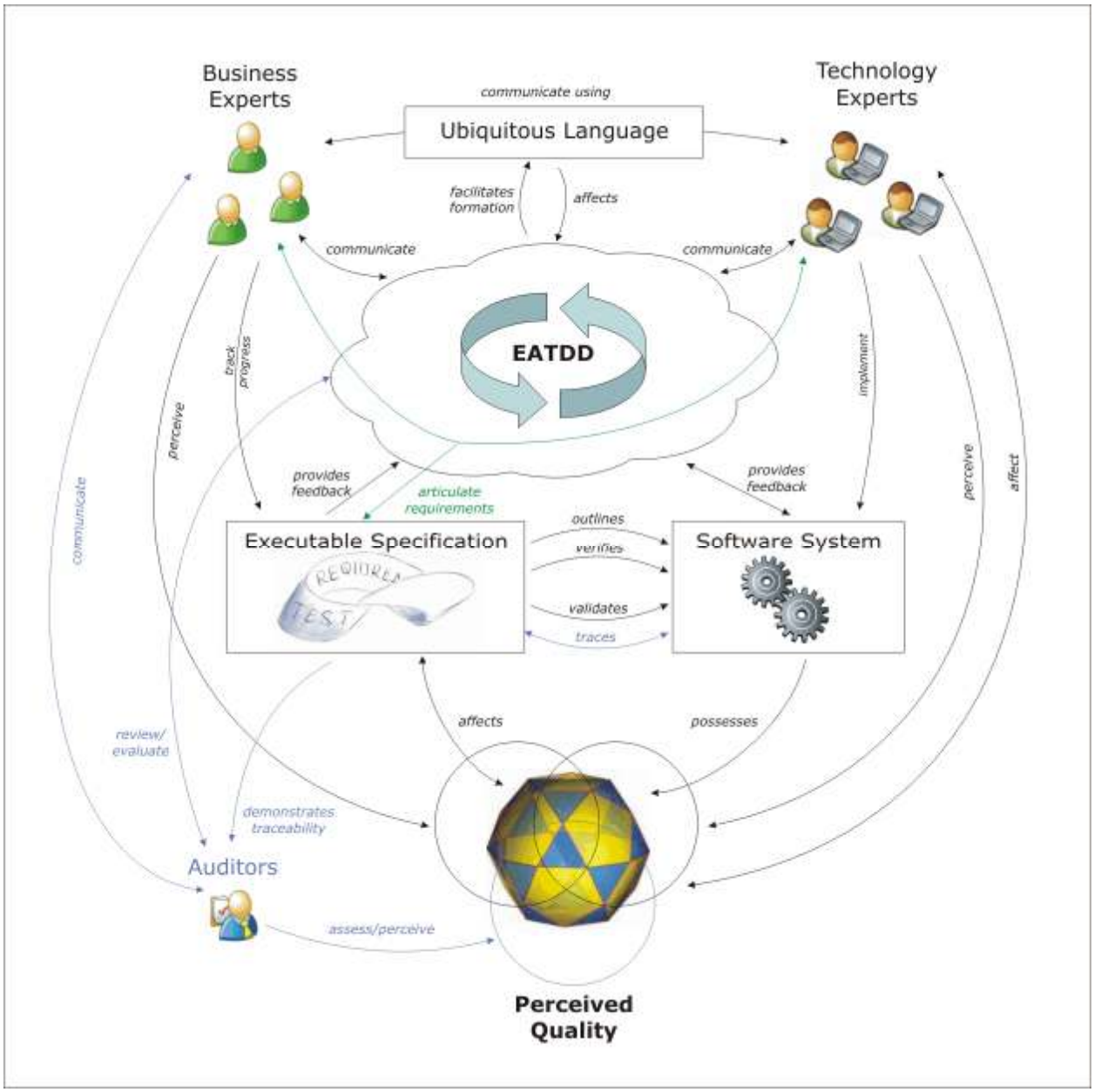


Figure 25. EATDD in the Realm of a Socio-Technical System.

When using EATDD, the test suite itself is a form of the executable specification. The requirements/tests evolve with the system. Indeed, in an environment where continuous integration and rigorous testing are practiced, an executable acceptance test specification could never get out of sync with the application itself, because any disagreement between the requirements and the code would cause the tests as well as the build, to fail.

VI.4 Artifact Model.

The Artifact Model (Figure 26) presents another generalization of how EATDD is used in practice.

In this model you can see a separation of artifacts produced by business experts or technology experts, or a combination of thereof (placed on the separation line in the diagram). Artifacts in red brackets are optional because they were not present on all projects. An arrow denotes an impact of one artifact onto the other.

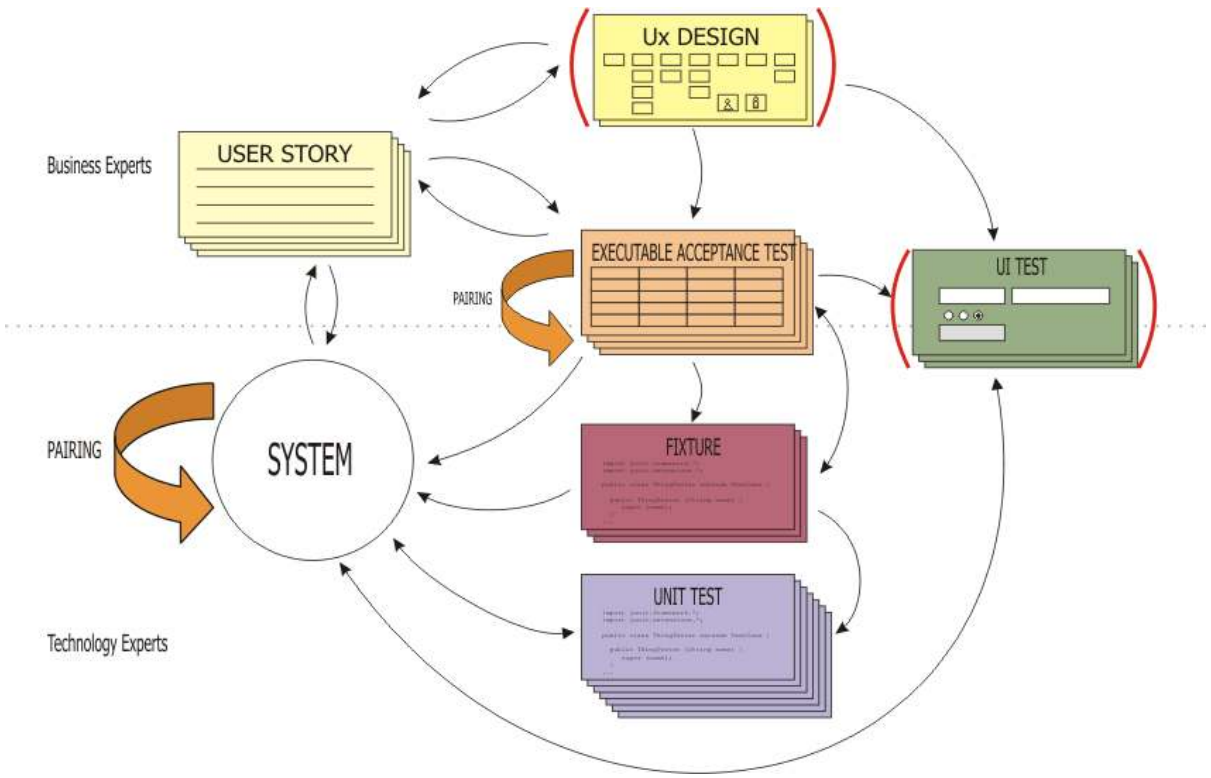


Figure 26. EATDD- Artifact relationship map.

Typically, for each user story (feature) written by the business expert(s), there will be a set of executable acceptance tests specified by business experts alone in collaboration with technology experts. If a user experience (Ux) designer is involved, there may be additional activities (such as storyboarding, affinity mapping etc.) with the corresponding artefacts produced, which will influence

the acceptance tests written. A separate suite of UI tests may also be produced for exercising the logics of the system through the actual user interface. These are typically derived from the acceptance tests and from the Ux artifacts (if any).

There are three artifacts produced by the technology experts. No direct collaboration with business experts takes place when working on 1) executable acceptance tests fixtures (fixtures), 2) unit tests, and 3) implementation code itself. Through fixtures, executable acceptance tests exercise the logics of the business system. A reciprocal link shown from the fixture to the executable acceptance test manifests the impact an existing fixture may have on the choice of the test style and vocabulary when a new acceptance test is being specified. The fixture will call methods of the business logic, which, in turn, must be tested at the unit test level.

Ultimately, all artefacts interact with and form the software system.

Importantly, our industrial evidence showed that tests (of all kinds) are now becoming to be considered and treated as assets and not liabilities like before. This is an important trend that we are also beginning to see on some open source projects that are shipped with the sets of tests (e.g., Eclipse, Lucerne, Spring). It is hoped that more customers would start demanding testing artifacts to be submitted by the software engineering teams as deliverables of the system, along with the source and assembled code.

VI.5 Validation of the synthesized models

In order to validate our synthesized models, we have performed an additional industrial case study (Case Epsilon). Considering the precedent set by the industrial case Gamma, we employed the purposive sampling strategy, which guided our search for another case from a regulated environment. Unlike the first two industrial multi-case studies, our goals were not explorative, but largely confirmative.

VI.5.1 Context

Case Epsilon involved an ongoing project that, on the day of the interview, was under iterative development (with multiple frequent releases) for 4.5 years. The software system was developed in .NET as a "fat" Windows client. In a nutshell, the system was used for performing comprehensive analyses of the DNA codes. The software is meant for the use by forensic departments of the government authorities. The system's size can be envisioned by the number of acceptance tests (in thousands) and ≈200,000 lines of C# code.

The distinguishing characteristics of this case are as follows:

1) a restricted and regulated environment the company operates in (government forensic labs);
2) a long duration of the project (4.5 years);
3) the use of FitNesse (a wiki with integrated acceptance test runner)

Most of the team was experienced in object-oriented programming but had no prior experience in .NET environment. Teodor, the development lead on the project, was interviewed. Specifically, we have inquired about the main categories and findings of the EATDD process. The team started doing executable acceptance testing (not EATDD, the test-first aspect of it was missing) with NUnit: "*We would actually go through and instantiate forms and subcontrols, and we've implemented some of our own extensions NUnit, so that we could do that more efficiently.*" Shortcoming of writing acceptance tests in the unit testing framework were discussed in §V.9.2.8. Even though the team was able "to draw some value out of the tests, but as the application grew they [acceptance tests] became more and more cumbersome to generate." Eventually, the team adopted EATDD and started using FitNesse upon a recommendation of one of the external consultants. They had been using FitNesse for over 2 years at the time of the interview.

VI.5.2 Requirements discovery

Teodor recognized the inciting power of the EATDD with the quote: *"When discussing acceptance criteria, our customer proxy and programmers participate… and learn significant new things about their requirements… and about design… so did the testers… Brand-new aspects of the familiar concept emerge"* On several occasions, when the customer proxy went back to the customers for clarifications on acceptance criteria, *"he discovered he was not 100% on the same page with them [actual customers]."* Reportedly, these new requirements ended up having an impact on system's design. This is consistent with our synthesized model.

VI.5.3 Requirements articulation

Executable acceptance tests along with commentary and scientific illustrations were primarily used for communicating requirements to the programmers. The commentary was not extensive – Teodor commented that *"we don't write narratives, but we write like purpose statements of the table. So, kind of, one sentence description"* In this case study, the customer proxy did not actually write the acceptance tests even though he was the main source of the system's requirements for the team. Instead, he *"paired with the QAs [testers] and … they talked about the acceptance criteria, which QAs would later detail in the [acceptance test] tables."*

Since no customer was present on site at all times and even the customer proxy traveled frequently back and forth between his team city and customers' locations, the team kept "a week of a buffer" of acceptance tests to always have enough articulated and clarified requirements in the form of acceptance tests, for at least one week of work. Prior to this one-week buffer strategy, the development process was rather chaotic: *"You are done and you are waiting for new requirements or something … and they would change as soon as he [the customer proxy] comes back…. So, it was actually good – it was not only giving us the buffer, it was also a chance for them [clients] to think, to reflect and to make the change before we actually started working on this."*

VI.5.4 Achieving confidence and requirements traceability

The team achieves confidence through executable acceptance testing, automated unit testing, and additional manual UI and exploratory testing. In addition, they *"have a nightly procedure, which runs … the autobuild, the script. We actually have it running on two different machines right now – we have …. we obfuscate the code"* (the obfuscation is required by the customer and FitNesse tests sometimes fail due to the way the obfuscator modifies the code).

The team also believed that they were getting better feedback from the tests and the code, following EATDD: "Yeah, I think the amount of [test] data that you get with FitNesse is *much* better than any other thing that I've used before."

Another aspect of accomplishing confidence was the fact that the team achieved a "*much better coverage with FitNesse than [they] did with the acceptance tests that [they] were writing in NUnit style.*" This, reportedly, was "one of the most positive things".

These testimonials emphasize different priorities in the ways of achieving quality, but, in general, support our theory that EATDD indeed helps the team achieve confidence in the system.

Furthermore, executable acceptance tests specified in FitNesse served as the primary functional specification for the project. Acceptance tests were annotated with explanations. In addition, the team *"handle[d] FitNesse pages just like we handle code (version and the whole thing)."* This provided traceability of changes and modifications. According to Teodor, the traceability criteria of the regulators were met. This confirms the experience of the team in Case Gamma and validates our finding regarding requirements traceability.

VI.5.5 Improved communication and collaboration

Teodor recognized that the process of EATDD gradually improved communication. It also reduced tension because *"everybody could see the tests. They provided good contexts for conversations…. Both devs and QAs are looking at the same documentation now … "*

Evidently, a ubiquitous language evolved. It had an effect on the semantics of the acceptance tests: *"QA staff take that requirements on story and basically build a dataset that would be used to test that scenario. We now have our own little language where we can build datasets internally into application. It makes it really easy to them: I want one of these, 3 of these, here's the profile, here's what the result should be."*

EATDD also helped the team to pair more. *"The QA guys [who were used to solitary work] come out now and pair with developer when they've gotten done developing the first run-through of their FitNesse page"*.

This is aligned with our findings about the improved team communication.

VI.5.6 Challenges

Maintainability was recognized to be one of the main challenges. While in the previous two multi-case studies, the maintainability issue was primarily related to the size of the test and the test suite, in this case, it was primarily about the maintainability of the ubiquitous language used and the underlying fixtures: *"That's our biggest problem right now – we don't have a good way of keeping track of the dictionary so that they do not duplicate things that were done before."* This was a different type of the maintainability challenge than we've identified in our previous studies.

Another challenge manifested itself in the way QA was authoring the acceptance tests. Teodor explains: *"The hardest part of that was giving the QA staff to *not* write things from the user interface point of view, to write things from a "here's how I want the application to work" point of view. But in their minds that was the same. Educating that to them was the hardest hump to get over, to getting them to write the tests without the user interface involvement."*

VI.5.7 Validation summary

Through the additional environmental triangulation achieved through the industrial case Epsilon, validity of our generalized results and conceptualizations is enhanced. As discussed in §V.11, other types of methodological and data triangulation (such as a larger number of industrial cases employing other methods of data collection) could be used to evaluate and expand upon these results.

Future Work

This dissertation established a correlation between utilization of EATDD and enhanced communication. Even though the industrial data suggests a perceived impact of EATDD on team communication, other confounding factors may have played out (e.g. the fact that teams followed some agile method). This deserves further investigation, which will require additional experiments to proof or refute such causality. Other rival hypotheses will also need to be evaluated, including the reverse relationship of the enhanced communication facilitating the adoption of EATDD.

In addition, future work will need to address an intriguing finding from the academic experiment three that some teams who were dissatisfied with EATDD produced some of the best executable specifications.

Additional evidence of the use of EATDD in regulated environments is needed.

The theory of how teams utilize EATDD needs to be expanded beyond the use of the FIT framework. Particularly, frameworks that do not use the tabular syntax are of interest.

A large area for future research involves the application of EATDD to specifying and communicating para-functional requirements. Some initial evidence suggests that performance requirements can be effectively described by executable acceptance tests.

Manageability and maintainability of the acceptance test suites was identified as one of the major challenges of EATDD. This issue deserves a thorough investigation.

Conclusions

Three academic quantitative studies and three industrial multi-case studies were at the core of presented research. The findings were guided by three key research questions, which investigated the ways the practice of Executable Acceptance Test-Driven Development (EATDD) was used, what kind of benefits and limitations EATDD manifested, and what kind of quality improvements EATDD contributed to.

All findings are in the context of implementing EATDD for specifying functional requirements using the FIT framework, when developing line-of-business applications.

Our main unequivocal finding is the use of **EATDD is correlated with the enhanced communication in software teams**. It helps to foster creativity, facilitates thinking about the domain, and helps focus attention on the business perspective and the goals that the software system under development is meant to accomplish.

In addressing the characteristics of **suitability** (our second contribution), the findings demonstrate that executable acceptance tests can be used as functional requirements specifications and are in fact unambiguous, consistent, verifiable, and usable (from both the business experts' and technology experts' perspectives). EATDD adequately mitigates the following risks: noise, overspecification, wishful thinking, forward references, ambiguity, customer uncertainty, lack of customer involvement, and multiple representations. The risk of assumed or missing requirements, however, is not effectively mitigated by EATDD.

Our third major contribution is **the socio-technical model of the EATDD process** that was derived from the syndicated data from all studies. It provides a generalized view on the main players and elements and their inter-relations.

Additional findings include the fact that executable specifications produced in the course of EATDD can serve as **sufficient evidence of requirements traceability.** This sets a precedent of such approval by the regulatory authorities.

Finally, our investigation recognizes the **weak tool support** of the executable acceptance tests (especially for refactoring). This results in serious issues with acceptance test **maintainability** and **scalability**, especially once the system is handed over from development to operations.

Bibliography

1. "Acceptance Test". Online: http://c2.com/cgi/wiki?AcceptanceTest
   Last accessed: July 20, 2007
2. Abrahamsson, P. et al. "Improving Business Agility Through Technical Solutions: A Case Study on Test-Driven Development in Mobile Software Development, Business Agility and Information Technology Diffusion," *IFIP TC8 WG 8.6 Intl. Working Conf., IFIP International Federation for Information Processing*, Vol.180: 1-17, 2005.
3. Alexander, I., and Maiden, N. *Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle*. New York, NY: Wiley, 2004.
4. Alexander, I. "Initial industrial experience of misuse cases in trade-off analysis", *Proc. IEEE Int. Conf. on Requirements Engineering* (RE'02): 61–68, 2002.
5. Alexander, I. "Positive Results from Negative Scenarios". *Pres. for IDEX Project Challenge*, May 2002. Online:
   http://easyweb.easynet.co.uk/~iany/consultancy/negative_scenarios.ppt
   Last accessed: July 20, 2007
6. Ambler, S. "Test-Driven Development of Relational Databases", *IEEE Software,* 24(3): 37-43, 2007.
7. Andrea, J. "Envisioning the next generation of functional testing tools", *IEEE Software*, 24(3): 58-66, 2007.
8. Andrea, J. "Generative Acceptance Testing for Difficult-to-Test Software," *Proc. XP2004,* Lecture Notes in Computer Science, Springer-Verlag, Vol. 3092: 29 – 37, 2004.

9. Astels, D. “A New look at Test-Driven Development”, 2006; http://blog.daveastels.com/files/BDD_Intro.pdf Last accessed: July 20, 2007

10. Bach, J. “Exploratory Testing Explained”. Online: http://www.satisfice.com/articles/et-article.pdf Last accessed: July 20, 2007

11. Bach, J. *Private correspondence* with Grigori Melnik, 2007.

12. Basili, V.R. “The Role of Experimentation in Software Engineering: Past, Current, and Future”. *Proc. 18th Int. Conf. S/W Engineering (ICSE’96),* IEEE Computer Press: 442–449, 1996.

13. Beanlands, G. “Scoping methods and baseline studies in EIA”, in P. Wathern (Ed) Environmental Impact Assessment: Theory and Practice, London, Routledge, 1988.

14. Beanlands, G. “Scoping methods and baseline studies in EIA”. In Wathern, P. (Ed.) *Environmental Impact Assessment: Theory and Practice*. Unwin Hyman, London: 1988.

15. Beck, K. *Test-Driven Development By Example*. Addison-Wesley, Boston, MA: 2002.

16. Beck, K. *Extreme Programming Explained: Embrace Change*, 1/e. Addison-Wesley, Boston, MA, 1999.

17. Bezier, B. *Software Testing Techniques*. Van Nostrand Reinhold Electrical: New York, NY, 1983.

18. Bhat, T., Nagappan, N. “Evaluating the efficacy of test-driven development: industrial case studies,” *Proc. ISESE2006*, ACM Press: 356–363, 2006.

19. Bloom, B. *Taxonomy of Educational Objectives*. Allyn and Bacon, Boston, MA: 1984.

20. Buwalda, H. “Soap Opera Testing”. *Better Software*, 6(2): 30–37, 2004.

21. Canfora, A. et al. "Evaluating Advantages of Test Driven Development: a Controlled Experiment with Professionals," *Intl. Symp. on Empirical Software Eng.*: ACM Press: 364-371, 2006.

22. Carroll, J.M. (Ed.) Scenario-Based Design: Envisioning Work and Technology in System Development, New York, NY: Wiley, 1995.

23. CenterLine Software, Inc. "A Survey of 240 Fortune 1,000 companies in North America and Europe", Cambridge, MA, 1996. Online: http://www.computerworld.com/news/1997/story/0,11280,17522,00.html Last accessed: July 20, 2007

24. Chau, T., Maurer, F. "Tool Support for Inter-Team Learning in Agile Software Organizations." *Proc. LSO 2004*, Lecture Notes in Computer Science, Springer Verlag, Vol. 3096: 98-109, 2004.

25. Cohen, D. "Symbolic Execution of the Gist Specification Language". *IJCAI*: 17-20, 1983.

26. Cohn, M. "Do-It-Yourself"*, Better Software*, 7(9): 18–22, 2005.

27. Cunnigham, W. "FIT: Framework for Integrated Test." Online http://fit.c2.com. Last accessed on Jan 15, 2007.

28. Damm, L., Lundberg, L. "Results from Introducing Component-level Test Automation and Test-Driven Development", *J. Systems and Software*, 79(7): 1001-1014, 2006.

29. Davis, A. *Software Requirements Revision Objects, Functions, & States*. Prentice Hall PTR, Englewood Cliffs, NJ, 1994.

30. Department of Defense. *Military Standard Defense System Software Development DOD-STD-2167*, section.5.3.3. Online: http://www2.umassd.edu/SWPI/DOD/MIL-STD-2167A/DOD2167A.html
Last accessed: July 5, 2007

31. Dey, I. *Grounding Grounded Theory: Guidelines for Qualitative Inquiry*. Academic Press: San Diego, Ca, 1999.

32. Dohmke, T. and Gollee, H. "Test-Driven Development of a PID Controller", *IEEE Software,* 24(3): 44-50, 2007.

33. Edwards, S. "Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action," *ACM SIGCSE Bulletin*: 26-30, 2004.

34. El Emam, K. "Evaluating ROI from Software Quality," *The Cutter Consortium Report*, 5(1): 20, 2004.

35. Erdogmus, H. et al. "On the Effectiveness of the Test-First Approach to Programming*," IEEE Transactions on Software Eng*., 31(3):226-237: 2005.

36. Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, Boston, MA, 2001.

37. Firestone, W. "Meaning in method: The rhetoric of quantitative and qualitative research". *Educational Researcher*, 16(7), 16-21, 1987.

38. FIT: The Framework for Integrated Testing Documentation. Online: http://fit.c2.com/wiki.cgi?FitDocumentation
Last accessed: August 1, 2007

39. FitLibrary. Online documentation. Online: http://sourceforge.net/projects/fitlibrary
Last accessed: July 10, 2007.

40. FitNesse Documentation. Online: http://www.fitnesse.org/FitNesse.UserGuide
Last accessed: July 5, 2007

41. Fitnesse. Online http://www.fitnesse.org
Last accessed on Jan 15, 2007.

42. Flick, U. *An Introduction to Qualitative Research*. SAGE Publications, Ltd: London, 2002.

43. Flohr, T., Schneider, T. "Lessons Learned from an XP Experiment with Students: Test-First Needs More Teachings," *Proc. PROFES 2006*,

Lecture Notes in Computer Science, Springer Verlag, Vol. 4034: 305–318, 2006.

44. Fowler, M. “Continuous Integration”. Online: http://www.martinfowler.com/articles/continuousIntegration.html Last accessed: July 5, 2007

45. Fowler, M. “GUI Architectures” Online: http://www.martinfowler.com/eaaDev/uiArchs.html Last accessed June 26, 2007

46. Fowler, M. “Specification by Example”. Online: http://www.martinfowler.com/bliki/SpecificationByExample.html Last accessed: July 5, 2007

47. Fowler, M. “Specification by Example”. Online: http://www.martinfowler.com/bliki/SpecificationByExample.html Last accessed: July 5, 2007

48. Gause, D. Weinberg, G. *Exploring Requirements*, Dorset House: 249, 1989.

49. George, B. "Analysis and Quantification of Test Driven Development Approach MS Thesis," North Carolina State University Computer Science, Raleigh, NC, 2002.

50. George, B., Williams, L. “An Initial Investigation of Test Driven Development in Industry,” *Proc. ACM Symp. on Applied Computing*, ACM Press: 1135-1139, 2003.

51. Geras, A. et al. "A Prototype Empirical Evaluation of Test Driven Development," *Proc. METRICS 2004*: 405-416, 2004.

52. Gerrard, P. “Automation Below the GUI”. Online: http://uktmf.com/blog/paulgerrard/2006/07/automation_below_the_gui.html Last accessed: July 5, 2007

53. Glaser, B., and Straus, A. The discovery of grounded theory: Strategies for qualitative research. Aldine, Chicago, IL: 1967.

54. Gotel, O., and Finkelstein, A. "An Analysis of the Requirements Traceability Problem". *Proc. of First Inter. Conf. on Requirements Engineering*: 94-101, 1994.

55. Graham, D. "Requirements and Testing: Seven Missing-Link Myths". *IEEE Software*, 19(5): 15-17, 2002.

56. Harel, D. et al. "Statemate: a working environment for the development of complex reactive systems". *IEEE Trans. Soft. Eng.*, 16(4): 403-414, 1990.

57. Hetzel, B. *The Complete Guide To Software Testing*. QED Information Sciences Inc., Wellesley, Mass., 1983.

58. Hooks, I., Farry, K. *Customer-Centered Products: Creating Successful Products Through Smart Requirements Management*. American Management Association, New York, NY, 2001.

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

60. Janicki, R., Parnas, D., Zucker, J. "Tabular representations in relational documents". In Brink, C., Kahl, W., Schmidt, G. (Eds.) *Relational Methods in Computer Science. Advances in Computing Science*. Springer-Verlag: 1997.

61. Jarke, M., Bui, X.T., and Carroll, J.M. "Scenario Management: An Interdisciplinary Approach." *Requirements Eng. J.*, 3: 155-173, 1998.

62. Jeffries, R. *Extreme Programming Adventures in C#*. Microsoft Press: 2004.

63. Jeffries, R. "What is XP?" Online:
http://www.XProgramming.com/xpmag/whatisXP.htm
Last accessed: July 5, 2007

64. Johnson, M. et al. “Incorporating Performance Testing in Test-Driven Development,” *IEEE Software,* 24(3): 67-73, 2007.

65. Joint Task Force on Computing Curricula, Software Engineering 2004: *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering,* tech. report, IEEE CS and ACM, 2004; http://sites.computer.orgccse.

66. Joint Task Force on Computing Curricula, Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, tech. report, IEEE CS and ACM, 2004; Online: http://sites.computer.org/ccse.
Last accessed: March 15, 2007

67. Jones, C. *Patterns of Software Systems Failure and Success*, International Thompson Computer Press, Boston, USA, 1996.

68. Kaner, C. et al. *Testing Computer Software*, 2/e, New York, NY: Wiley, 1999.

69. Kaner, C., Bach., J., Pettichord, B. *Lessons Learnt in Software Testing : A Context-Driven Approach*. John Wiley & Sons, New York, NY, 2001.

70. Kaner, C. “Cem Kaner on Scenario Testing: The Power of ‘What-If…’ and Nine Ways to Fuel Your Imagination”, *Better Software*, 5(5):16–22, 2003.

71. Kaner, C. “What is a Good Test Case?” *STAR East Conf. 2003*, May 2003. Online: http://www.testingeducation.org/a/testcase.pdf
Last accessed: July 5, 2007

72. Kazman, R. et al “Scenario-Based Analysis of Software Architecture.” *IEEE Software,* 13(6):47–55, 1996.

73. Kerievsky, J. “Storytesting”. Online:
http://industrialxp.org/storytesting.html
Last accessed: July 5, 2007

74. Kruchten, P. "The "4+1" View Model of Architecture." *IEEE Software*, 12(6):42–50, 1995.

75. Madeyski, L. "Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality," *Software Engineering: Evolution and Emerging Technologies, ser. Frontiers in Artificial Intelligence and Applications*, Vol. 130, IOS Press: 113–123, 2005.

76. Mann, C., "An Exploratory Longitudinal Case Study of Agile Methods in a Small Software Company Master's Thesis", University of Calgary, Calgary, AB, 2004.

77. Mann, C., Maurer, F. "A Case Study on the Impact of Scrum on Overtime and Customer Satisfaction". *Proc. Agile 2005 Conference*, IEEE Computer Press: 2005.

78. Marick, B. "Bypassing the GUI". *STQE Magazine,* 5: 41– 47, Sep-Oct, 2002.

79. Marick, B. "Driving Software Projects with Examples"; http://www.exampler.com/
Last accessed: July 5, 2007

80. Marick, B. Exploration through Example.
Online: http://www.testing.com/cgi-bin/blog
Last accessed: July 10, 2007.

81. Marick, B. "Example-Driven Development". Online: http://www.exampler.com, and http://www.testing.com/cgi-bin/blog/2003/09/05#agile-testing-project-4
Last accessed: July 11, 2007

82. Marick, B. Agile Acceptance Testing Workshop Report, *XP/Agile Universe 2002 Conf.* Online: http://www.pettichord.com/XP_Agile_Universe_trip_report.txt
Last accessed: July 10, 2007

83. Martin, R. “The Test Bus Imperative: Architectures that Support Automated Acceptance Testing”, *IEEE Software*, 22(4): 65–67, 2005.

84. Martin, R., Melnik, G. “Tests and Requirements, Requirements and Tests: A Moebius loop”. *IEEE Software*, 24(6), 2007.

85. Maurer, F., Melnik, G. “Driving Software Development with Executable Acceptance Tests”, *The Cutter Consortium Report*, 7(11): 1–30, 2006.

86. McDermott, P. *Zen and the Art of Systems Analysis: Meditations on Computer Systems Development*, 2/e. Writers Club Press, Lincoln, NE: 3, 2003.

87. Melis, M. et al. “Evaluating the Impact of Test-First Programming and Pair Programming through Software Process Simulation,” *J. Software Process Improvement and Practice*, Wiley InterScience, 2006(11): 345–360, 2006.

88. Melnik, G. “Test-Infecting Future Software Engineers”. *Proc. 5th Annual Workshop on Teaching Software Testing (WTST 2006)*, online: www.testingeducation.org/wtst5/WTST5%20GMelnik%20submission%20final.pdf
Last accessed: July 10, 2007

89. Melnik, G., Jeffries, R. ”Test-Driven Development – The Art of Fearless Programming”. *IEEE Software*, 24(3): 24-30, 2007.

90. Melnik, G., Maurer, F. "Multiple Perspectives on Executable Acceptance Test-Driven Development", *Proc. XP2007 Conf.*, Lecture Notes in Computer Science, Springer Verlag, Vol. 4536: 245–249, 2007.

91. Melnik, G., Maurer, F. “A Cross-Program Investigation of Students' Perceptions of Agile Methods”. *Proc. 27th International Conf. on Software Engineering (ICSE 2005)*, ACM Press: 481–489, 2005.

92. Melnik, G., Maurer, F. “Direct Verbal Communication as a Catalyst of Agile Knowledge Sharing”. *Proc. Agile Software Development Conf. 2004*, IEEE Press: 21–31, 2004.

93. Melnik, G., Maurer, F. “Introducing Agile Methods in Learning Environments: Lessons Learnt”. *Proc. eXtreme Programming/Agile Universe 2003 Conf.*, Lecture Notes in Computer Science, Springer Verlag, Vol. 2753: 172–184, 2003.

94. Melnik, G., Maurer, F., Chiasson, M. “Executable Acceptance Tests for Communicating Business Requirements: Customer Perspective”. *Proc. Agile 2006 Conference*, IEEE Computer Press: 35–46, 2006.

95. Melnik, G., Read, K., Maurer, F. “Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective”. *Proc. XP/Agile Universe 2004*, Lecture Notes in Computer Science, Springer Verlag, Vol. 3134,: 60–72, 2004.

96. Melnik, G. “Teaching Acceptance Testing in Contexts of Web Systems Development and Game Programming”. *Proc. 4th Annual Workshop on Teaching Software Testing (WTST 2005)*, online: www.testingeducation.org/conference/wtst4/GMelnik%20Teaching%20Acceptance%20Testing%20final.pdf
Last accessed: July 10, 2007

97. Melnik, G., Maurer, F. “The Practice of Specifying Requirements Using Executable Acceptance Tests in Computer Science Courses”. *Proc. 20th International Conf. on Object-Oriented Programming, Systems, Languages and Applications* (OOPSLA 2005), ACM Press: 365–370, 2005.

98. Meszaros, G. “Agile regression testing using record & playback”. *OOPSLA Companion 2003*: 353-360, 2003.

99. Meszaros, G. *xUnit Test Patterns: Refactoring Test Code*. Addison Wesley, Boston, MA, 2007.

100. Meyer, B. “On Formalism in Specifications”. *IEEE Software*, 2(1):6–26, 1985.

101. Miles, M., and Humberman, A. *Qualitative Data Analysis: An Expanded Sourcebo*ok, 2/e. SAGE Publications, Thousand Oaks, CA: 1994.

102. Miller, R., and Collins, C. "Acceptance Testing". *Proc. XPUniverse 2001 Conf.*, July, 2001.

103. Müller, M., Hagner, O. "Experiment about test-first programming," *IEEE Software*, 149(5): 131-136, 2002.

104. Mugridge, R. and Tempero, E. "Retrofitting an Acceptance Test Framework for Clarity," *Proc. Agile Development Conf. 2003*, IEEE Press: 92-98, 2003.

105. Mugridge, R., and Cunningham, W. *FIT for Developing Software: Framework for Integrated Tests*. Prentice Hall, Upper Saddle River, NJ: 2005.

106. Mugridge, R., MacDonald, B., Roop, P. "A Customer Test Generator for Web-Based Systems". *Proc. XP2003 Conf.*, Lecture Notes in Computer Science, Vol.2675, Springer Verlag: 189-197, 2003.

107. Mugridge, R., Tempero, E. "Retrofitting an Acceptance Test Framework for Clarity", *Proc. Agile Development Conference 2003*, IEEE Press: 92-98, 2003.

108. Nielsen, J., McMunn, D. "The Agile Journey: Adopting XP in a Large Financial Services Organization", *Proc. XP2005*, Lecture Notes in Computer Science, Springer Verlag, Vol. 3556: 28–37, 2005.

109. Osterweil, L. et al. "Strategic directions in software quality". *ACM Computing Surveys*, (4):738-750, 1996.

110. Pančur, M. et al. "Towards Empirical Evaluation of Test-Driven Development in a University Environment," *Proc. EUROCON 2003*, IEEE: 83-86 vol.2, 2003.

111. Patton, M. *Qualitative Evaluation and Research Methods*, 3/e, Sage Publications Thousand Oaks, CA: 342-344, 2002.

112. Perry, W. *Effective Methods for Software Testing*, 2/e, John Wiley & Sons, New York, NY, 2000.

113. Raha, S. Comment in [116], p.19.

114. Read, K., Melnik, G., Maurer, F. "Examining Usage Patterns of the FIT Acceptance Testing Framework." *Proc. 6th International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2005),* Lecture Notes in Computer Science, Vol. 3556, Springer Verlag: 127-136, 2005.

115. Read, K., Melnik, G., Maurer, F. "Student Experiences with Executable Acceptance Testing". Proc. Agile 2005 Conference, IEEE Press: 312-317, 2005 .

116. Reppert, T. "Don't Just Break Software, Make Software: How Story-Test-Driven-Development is Changing the Way QA, Customers, and Developers Work". *Better Software*, 6(6): 18–23, 2004.

117. Rogers, O. "Acceptance Testing vs. Unit Testing: A Developer's Perspective," *Proc. XP/Agile Universe 2004*, Lecture Notes in Computer Science, Springer-Verlag, Vol. 3134: 22 – 31, 2004.

118. Rolland, C. et al "A Proposal for a Scenario Classification Framework," *Requirements Engineering J.*, 3: 23-47, 1998.

119. Rothmann, J. "Managing Product Development". Online: http://www.jrothman.com/weblog/blogger.html Last accessed: July 5, 2007

120. Ruiz, A., and Price, Y. "Test-Driven GUI Development with TestNG and Abbott," *IEEE Software,* 24(3): 51-57, 2007.

121. Sanchez, J. et al. "A Longitudinal Study of the Use of a Test-Driven Development Practice in Industry," *Proc. Agile 2007*, IEEE Press: 2007.

122. Sepulveda, C., Marick, B., Mugridge, R., Hussman, D. "Who Should Write Acceptance Tests?" *LNCS*, Vol. 3134, Springer-Verlag: 184 – 185, 2004.

123. Sepulveda, C. “XP and Customer Tests: Is It Fair?” Online: http://christiansepulveda.com/blog/archives/cat_software_development.html
Last accessed: July 5, 2007

124. Shadish, W.R., Cook, T.D., and Campbell, D.T. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin, 2002.

125. Shore, J. “FIT and User Interface”. Online: http://www.jamesshore.com/Blog/Fit-and-User-Interfaces.html
Last accessed: July 5, 2007

126. Stake, R. *The Art of Case Study Research*. Thousand Oaks, CA: Sage, 1995.

127. Statistical Tables. Critical Values of the Mann-Whitney U (one-tailed testing). Online: http://fsweb.berry.edu/academic/education/vbissonnette/tables/mwu.pdf
Last accessed: July 10, 2007

128. Statistical Tables. Critical Values of the t-Distribution. Online: http://fsweb.berry.edu/academic/education/vbissonnette/tables/t.pdf
Last accessed: July 10, 2007

129. Steinberg, D. “Using Instructor Written Acceptance Tests Using the Fit Framework”. *Proc. XP 2003 Conf.*, LNCS, Vol. 2675, Springer Verlag: 378-385, 2003.

130. Steinberg, D. “Using Instructor Written Acceptance Tests Using the Fit Framework,” LNCS, Vol. 2675, Springer-Verlag: 378 – 385, 2003.

131. Straus, A., and Corbin, J. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. SAGE Publications, Thousand Oaks, CA: 1998.

132. Sutcliffe, A. “Scenario-based requirements engineering”. *Proc. RE 2003 Conf.*, IEEE Press: 320- 329, 2003.

133. Test Driven Development. Online: http://c2.com/cgi/wiki?TestDrivenDevelopment
Last accessed on Jan 15, 2007

134. Van Vliet, H. *Software Engineering: Principles and Practice*, 2/e, John Wiley & Sons, Chichester, UK, 2000.

135. Watt, R. and Leigh-Fellows, D. “Acceptance Test Driven Planning,” *LNCS*, Vol. 3134, Springer-Verlag: 43 – 49, 2004.

136. Watt, R., and Leigh-Fellows, D. “Acceptance Test Driven Planning”. *Proc. XP/Agile Universe 2004 Conf.*, LNCS, Vol. 3134, Springer Verlag: 43-49, 2004.

137. Weidenhaupt, K. et al. “Scenarios in system development: current practice”. *IEEE Software*, 15(2): 34-45, 1998.

138. Weinberg, G. Online: http://www.geraldmweinberg.com/
Last accessed: July 10, 2007.

139. Wiesner, S. “Test-first development with FitNesse: Learn how FitNesse can solve your quality problems,” JavaWorld.com, 2006. Online: http://www.javaworld.com/javaworld/jw-02-2006/jw-0220-fitnesse.html
Last accessed: July 10, 2007.

140. Yin, R. *Case Study Research: Design and Methods*, 2/e, Sage Publications, Thousand Oaks, CA: 2003.

141. Ynchausti, R. A. “Integrating Unit Testing into a Software Development Team’s Process,” *Proc. XP2001*: 79-83, 2001.

142. Young, R. *Effective Requirements Practices*, Addison-Wesley, Boston, MA, 2001.

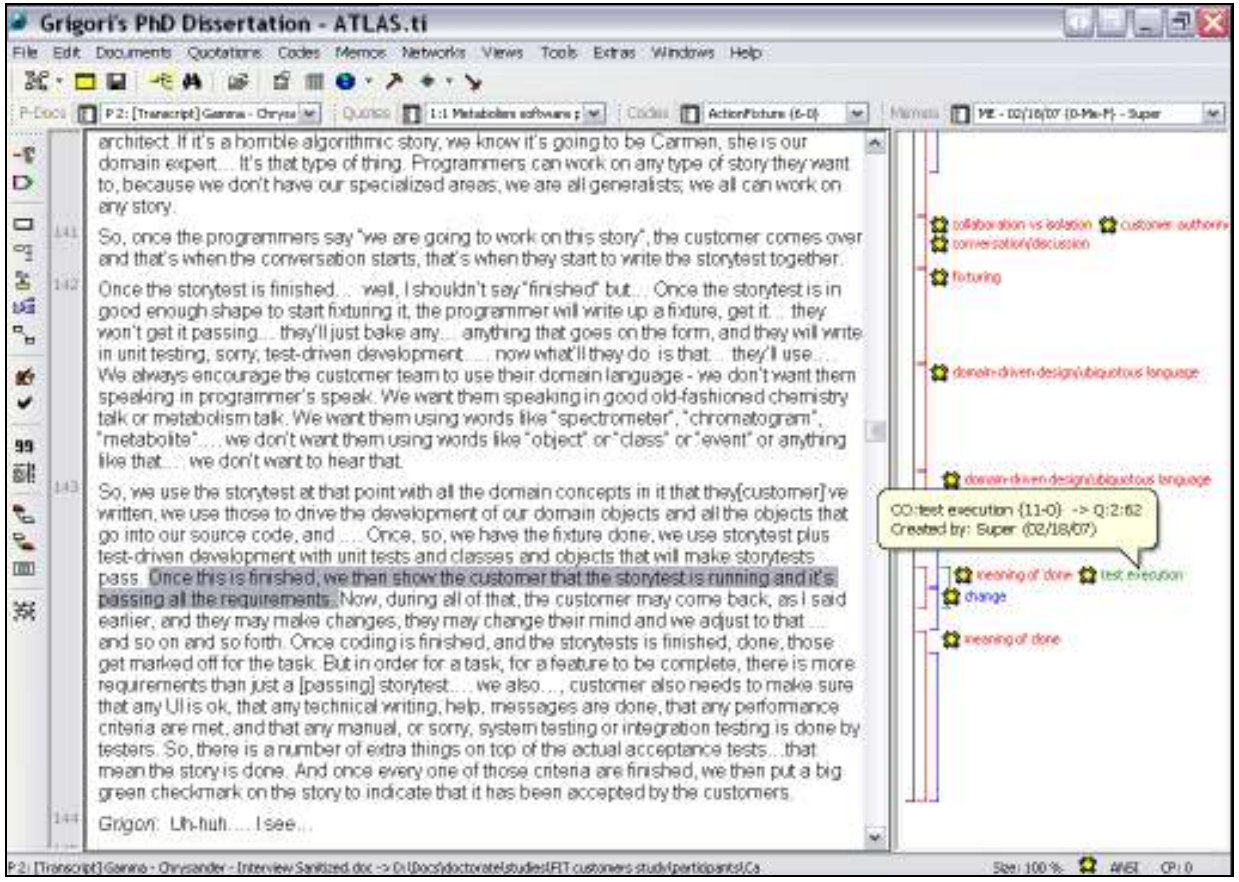
143. Zannier, C., Melnik, G., Maurer, F. “On the Successes of Empirical Studies in the International Conference on Software Engineering”. Proc. 28th International Conference on Software Engineering (ICSE2006), ACM Press: 341–350, 2006.

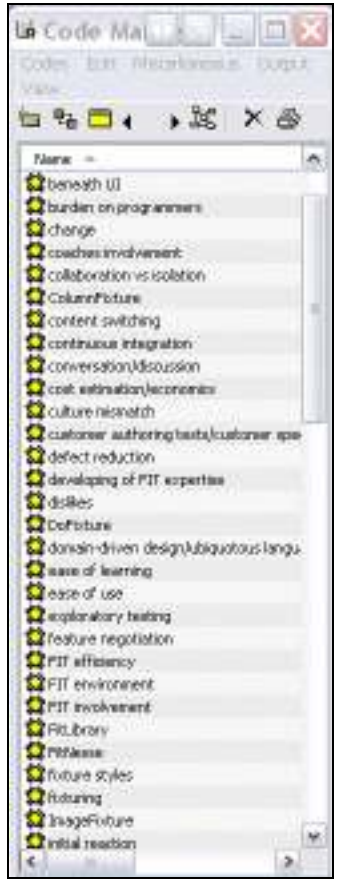
144. Zave, P., Schell, W. “Salient features of an executable specification language”. *IEEE Trans. Soft. Eng.*, 12(2): 312-325, 1986.

Appendix A Ethics Board Certificates

Appendix B. Co-Author Permissions

Appendix C. Open Coding Session with Atlas.ti Screenshot





Appendix D. Interview Guide

Interview Guide

Date/Time:

We are interested in how the requirements and acceptance criteria are communicated to you. This interview is conducted as part of a research project conducted at the University of Calgary, results of which will be published. The interview is subject to your control. Your participation in this research is voluntary. It is your right to decline to answer any question that you are asked or remove an answer. You are free to end the interview at any time. Participant confidentiality will be strictly maintained. Reports and presentations will refer to participants using only an assigned number. No information that discloses your identity will be released.

Do you have any questions before we begin? Do you give me your consent to proceed?

1. In your own words, describe your **development process** for me and **your role** in this process and **how long** you have been involved with the project.
    a. what is your **background**?
    b. Is this your first **agile** project? where you involved with the project since its **inception**?
2. Who is/are the **client(s)** of your system? Who will **use** it?
3. Who do you, as a developer, primarily **interact** with? Do you talk to the product owner? to external customers? directly?
4. How are the requirements **specified** on this project?
5. How do you know you are **"done"**? What does "done" mean ?
6. Are there things that are **especially complex/difficult** to test for completion/acceptance?
7. Tell me about the **domain language**/standard naming conventions?...
8. How do you do **regression testing** of all features, i.e. how do you know that what worked before works now?
    a. How about end-to-end functionality that spans via multiple stories?

9. How do you do **progress tracking**? When do you declare success? How often do **you check the progress** of your whole team by **executing acceptance tests**? (do you actually run them?)

10. How did you become involved with **FIT**? Was it **easy** to learn?

11. One of the things we're interested in understanding better is how customers **use** EATDD. What was this experience of like for you?

    probes: how did you use it? on your own? if not, who else was involved?(in partnership with the development team, in partnership with a tester? someone else?)

    - **how would you usually go about specifying an acceptance test**? Describe for me this process.
        - if I followed you through a typical EATDD specification session, what would I see you doing? what would I hear you saying? what would I see other people doing? Take me to an EATDD session so that I could actually experience it.

12. Types of tests: a) negative vs. positive? b) how large?

13. **How long** does the **entire regression suite** take to run? What about **subsets** that you run locally from your machine – how long can you tolerate?

14. How often you **change** them?

15. How **effective**, do you think, the process of specifying and verifying requirements on <u>your project</u> is?

16. How, in your opinion, the whole **process** (and **specifically acceptance testing** part) can actually be **improved**?

17. How **different** would the process need to be if this was not a **legacy-rewrite** but a green-field development? (or the other way around)

18. **Compare** this process **to other environments** you worked in ?

19. On your next project, would you prefer to **do it the same way**? Would you take on a project that was not acceptance test-driven?

20. Does **tabular** format of FIT tests make it easier to specify?

21. Let me turn now to your personal **likes** and **dislikes** about FIT. What are some of the things that you have **really liked** about FIT?

22. What about **dislikes**?

23. Do you think FIT framework is **more about testing or more about requirement specification, clarification and communication**?

24. Did you feel the going exec. acceptance test-driven way was making you go **slower**?

25. How likely is it that you **would recommend using executable acceptance tests** (in FIT) for specifying business requirements to a colleague?
    - **what advice would you give them?**

Scale [1-10]

Last question: That covers the things I wanted to ask.

Anything at all you care to add?

Thank you!

Appendix E. Results of Open Coding Analysis

Table 19. Open Coding Analysis – Requirements Discovery Activities

#	**Core category**	**Properties and dimensions**
2	Requirements discovery	This category includes methods of domain analysis and collaborative requirements discovery as well as resulting shared external representations of the domain
#	**Sub-category**	**Properties and dimensions**
2.1	Activities	This subcategory contains different activities performed by business experts and technology experts and their idiosyncratic characteristics
	Concepts from data analysis	
a.	Envisioning	
b.	Brainstorming	
c.	Scoping	
d.	Expressing intent	
e.	Customer interaction	
f.	Participatory design	
g.	Collaboration among all stakeholders: – building trust – analysis of somebody else's thinking – dialog with peers – dialog with other stakeholders	
h.	Learning	
i.	Posing useful questions	
j.	Prioritizing important scenarios	
k.	Reuse: – internal – cross-project – patterns emergence	
l.	Exercising the completed functionality of the system: – through UI – through acceptance tests	
m.	Recognizing and managing bias	

Table 20. Open Coding Analysis – Requirements Discovery Facets

#	**Core category**	**Properties and dimensions**
2	Requirements discovery	This category includes methods of domain analysis and collaborative requirements discovery as well as resulting shared external representations of the domain.
#	**Sub-category**	**Properties and dimensions**
2.2	Facets	This subcategory describes idiosyncratic characteristics of the activities contributing to the requirements discovery while specifying, communicating or verifying acceptance criteria for stories/functional requirements.
	Concepts from data analysis	
a.	Focus on business goals	
b.	Systematic approach	
c.	Iterative approach: – business experts specify a small chunk of requirements for a story – business experts use the chunk of the system built – as a result, new ideas are conceived	
d.	Accepting responsibility	
e.	Clearer way	
f.	Evolvability (as understanding of a business rule evolves)	
g.	Productivity: – reduction in the short term – improvement in the long term – relates to the discipline – relates to reduced rework	
h.	Prioritizing important scenarios	
i.	Timing (when to write the tests)	

Table 21. Open Coding Analysis – Shared External Representation of Requirements

#	**Core category**	**Properties and dimensions**
2	Requirements discovery	This category includes methods of domain analysis and collaborative requirements discovery as well as resulting shared external representations of the domain.
#	**Sub-category**	**Properties and dimensions**
2.3	Shared external representations	This subcategory describes elements of tacit knowledge transfer into a shared external representation.
	Concepts from data analysis	
a.	Business value alignments	
b.	Types of acceptance tests: – happy path – variability tour – expecting errors (with calculations/ with actions) – complex transactions – business rule calculations – business forms	
c.	Formation of ubiquitous language	
d.	Independent acceptance tests	
e.	Context-specific acceptance tests	
f.	Motivating (= a stakeholder with influence would push for it to be implemented)	
g.	Inter-scenario relationships: – containment dependency – alternative dependency – temporal dependency – logical dependency	
h.	Increased focus on deviant and alternative behaviors: – failure – misuse – abuse	

Table 22. Open Coding Analysis – Requirements Articulation Attributes

#	**Core category**	**Properties and dimensions**
3	Requirements articulation	This category includes methods of communicating requirements in the form of executable acceptance tests among various stakeholders; types and attributes of the produced acceptance tests; and any emerging patterns.
#	**Sub-category**	**Properties and dimensions**
3.1	Attributes	This subcategory describes attributes of executable requirement specifications stated by the study participants .
	Concepts from data analysis	
a.	Sufficient level of detail: – for business experts – authoring – reading – verifying that the requirements were properly captured – executing	
a.	Sufficient level of detail (continued): – for technology experts – reading – inferring enough specific detail to drive design and coding work – executing – suggesting variations/modifying	
b.	Right-sizing for planning	
c.	Concreteness & preciseness	
d.	Decreased ambiguity	
e.	Improved comprehensibility/clarity: – direct walkthroughs – reverse-order readings	
f.	Non-redundancy	
g.	Ease of authoring	
h.	Comfort with tabular representation	
i.	Relevance/Credibility: – compelling story – real-world usage – comes from business experts – describes problem domain not a solution domain	
j.	Refined ubiquitous language	
k.	Separation of concerns (business modeling beneath UI)	
l.	Domain learning (knowledge acquisition) through collaboration	
m.	Acceptance tests viewed as assets not liability (not by all – see Challenges, core category 6.)	
n.	Adaptability and support for software change	

Table 23. Open Coding Analysis – Requirements Articulation Types

#	**Core category**	**Properties and dimensions**
3	Requirements articulation	This category includes methods of communicating requirements in the form of executable acceptance tests among various stakeholders; types and attributes of the produced acceptance tests; and any emerging patterns.
#	**Sub-category**	**Properties and dimensions**
3.2	Types	This subcategory describes attributes of executable requirement specifications stated by the study participants .
	Concepts from data analysis	
a.	Business constraints	
b.	Workflows	
c.	Temporal (notion of date and time): – sequencing – concurrent transactions	
d.	UI	
e.	Selected para-functional requirements: – performance – security (authentication & authorization) – usability (accessibility)	

Table 24. Open Coding Analysis – Requirements Articulation Patterns

#	**Core category**	**Properties and dimensions**
3	Requirements articulation	This category includes methods of communicating requirements in the form of executable acceptance tests among various stakeholders; types and attributes of the produced acceptance tests; and any emerging patterns.
#	**Sub-category**	**Properties and dimensions**
3.2	Patterns	This subcategory identifies repeatable guides ("patterns") to recurring problems.
	Concepts from data analysis	
a.	Proven good patterns: – Test beneath UI – Build-Operate-Check – Delta assertion – Fixture setup – Transaction rollback – Collections – Grouping into suites	
b.	Smells: – Unnecessary detail – Tangled tables – Long tables – Missing pre-conditions – Laborious action-based tests for calculation – Rambling workflow – Similar setup – Convoluted setup – Many columns – Many rows	
c.	Context-specific classes of acceptance tests	

Table 25. Open Coding Analysis – Achieving confidence

#	Core category	Properties and dimensions
4	Achieving confidence	This category includes methods of achieving confidence in the system's implementation with testing, regression, continuous integration, fast feedback, requirements traceability, as well as social implications and project management aspects.
#	**Sub-category**	**Properties and dimensions**
4.1	Activities	This subcategory identifies various activities performed by business experts and technology experts to achieve confidence in the software system built.
	Concepts from data analysis	
a.	Iteration planning	
b.	Acceptance testing (with FIT, FitNesse, home-grown tools and harnesses)	
c.	Unit testing (with JUnit, NUnit)	
d.	GUI testing (with Selenium, Watir)	
e.	Exploratory system testing – by business experts (what-if analysis; going through the real application UI) – by technology experts (specialized techniques, including complexity tour, interruptions, resource starvation, input constraint attack, blink testing etc.)	
f.	Use of heuristics	
g.	Pairing	
h.	Engagement of external test teams	
i.	Auto-build	
j.	Version control	
k.	Continuous integration	
l.	Reviews: – test case – code	
m.	Iteration/milestone retrospectives	
n.	Perception of testing as part of software engineering hygiene (by all stakeholders!)	

Table 26. Open Coding Analysis – Perceived Quality

#	**Core category**	**Properties and dimensions**
4	Achieving confidence	This category includes methods of achieving confidence in the system's implementation with testing, regression, continuous integration, fast feedback, retrospectives as well as social implications and project management aspects.
#	**Sub-category**	**Properties and dimensions**
4.2	Perceived quality	This subcategory describes various quality aspects of the resulting product.
	Concepts from data analysis	
a.	Defect reduction	
b.	Catching problems earlier	
c.	Building the right system	
d.	Discipline	
e.	Customer satisfaction	
f.	Visibility	
g.	Regulatory compliance: – adequate documentation for audit – traceability	

Table 27. Open Coding Analysis – Social Implications

#	**Core category**	**Properties and dimensions**
4	Achieving confidence	This category includes methods of achieving confidence in the system's implementation with testing, regression, continuous integration, fast feedback, retrospectives as well as social implications and project management aspects.
#	**Sub-category**	**Properties and dimensions**
4.3	Social implications	This subcategory describes team level implications
	Concepts from data analysis	
a.	Diverse talents collaboration: – domain expertise – technical skill – requirements engineering experience – testing experience – project experience – industry experience – product knowledge – educational background – writing skill – cultural background	
b.	Fear elimination/Confidence boosting (due to primarily the safety net in the form of acceptance tests) – "*Green feels really good!*"	
c.	Improved team morale	
d.	Domain knowledge cross-pollination	
e.	Peer training	
f.	Customer involvement	
g.	New perception of testers as "*friends*" as opposed to "*diabolic adversaries*"	
h.	Enhanced communication	

Table 28. Open Coding Analysis – Project Management Implications

#	**Core category**	**Properties and dimensions**
4	Achieving confidence	This category includes methods of achieving confidence in the system's implementation with testing, regression, continuous integration, fast feedback, retrospectives as well as social implications and project management aspects.
#	**Sub-category**	**Properties and dimensions**
4.4	Project management implications	This subcategory describes aspects of EATDD that positively affect project management
	Concepts from data analysis	
a.	Additional support for iteration planning	
b.	Encourages incremental development	
c.	Ease of verification & validation	
d.	Making sense of project status & comparing status against mission	
e.	Progress tracking	
f.	Meaning of "completed"	
g.	Support in making decisions when to ship	
h.	Owning methodology	
i.	Reporting	
j.	Keeping software in good shape & changeability	
k.	Economics: – catching problems early – lower cost of rework – improved customer satisfaction – renewed business relationships – lower risk of tacit knowledge loss – increased awareness of software quality issues	
k.	Economics (continued) – reduced training costs – less unfocused, unproductive work	
l.	Support of other activities in software development lifecycle	

Table 29. Open Coding Analysis – Challenges: Maintainability

#	**Core category**	**Properties and dimensions**
5	Challenges	This category includes business experts' and technology experts' experiences related to challenges in requirements discovery, articulation, validation and maintenance (Categories 2-5).
#	**Sub-category**	**Properties and dimensions**
5.1	Maintainability	This subcategory describes various issues of maintenance and tool support.
	Concepts from data analysis	
a.	Dealing with the large volume of tests: – identification & location/search – grouping/hierarchical structuring – style transformation (e.g. transforming a series of workflow tests in a single calculation test)	
b.	Naming conventions	
c.	Dealing with size of acceptance test cases: – uber-stories (splitting strategies) – width/size of tables (decomposition/fragmentation strategies)	
d.	Fragile fixture – managing dependencies & sensitivities – behavior – interface – data – context – underlying services & infrastructure	
e.	Consolidation strategies	
f.	Tool support: – test case refactoring	

	Concepts from data analysis (continued)
f.	Tool support (continued): – dealing with fragments – selective execution – exploratory branching – freestyle annotating – diagrams – chaining/subroutine support – test tool integration – web testing – GUI testing – B2B testing – runners (through Excel, Selenium, Watir, JWebUnit, JBehave) – IDE integration – Eclipse – IDEA – NetBeans – Visual Studio – Visual Studio Team System

Table 30. Open Coding Analysis – Other Challenges

#	**Core category**	**Properties and dimensions**
5	Challenges	This category includes business experts' and technology experts' experiences related to challenges in requirements discovery, articulation, validation and maintenance. (Categories 2-5).
#	**Sub-category**	**Properties and dimensions**
5.2	Other challenges	This subcategory includes other challenges, complimentary to the main challenge of maintainability and tool support.
	Concepts from data analysis	
a.	Performance of test execution	
b.	Common vocabulary issues (formation of ubiquitous language): – cross-author consistency – scenario recaps – contextual replacement – synonymic equivalence	
c.	Limitations of natural language	
d.	Prepping: – test setup – test teardown	
e.	Assumed/implied requirements	
f.	Culture mismatch (traditional testers vs. agile testers)	
g.	Initial programmers' resistance/pushback – Perceived extra work to fixturize tests and to maintain fixtures	
h.	Inexperienced staff	
i.	Overspecification & the point of diminishing returns (going deeper when actually not needed)	
j.	Acceptance-test-driving Web 2.0 (AJAX) applications	
k.	False sense of security	