UNIVERSITY OF CALGARY

Supporting Agile Teams of Teams via Test Driven Design

by

Kris Read

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

FEBRUARY, 2005

© Kris Read 2005

ABSTRACT

Agile methods are increasingly popular for small, co-located teams. However, large or distributed teams do not hit the sweet spot for the application of agile methods. A divide and conquer approach can be used to employ multiple teams on such a project, however there are several obstacles to employing agile methods with teams of teams. Issues of up-front design, communication of dependencies and integration need to be overcome in order to employ agile methods in such a scenario. The use of test driven design practices, in particular acceptance testing, can directly support agile teams of teams and overcome these challenges.

ACKNOWLEDGEMENTS

The author would like to thank Frank Maurer for providing invaluable feedback and suggestions during this research work, and for his comments during the editing of this thesis. Thanks also to Thomas Chau for keeping me on-track and focused. Lastly, thanks to Grigori Melnik for co-authoring several papers on this subject, assisting me with my research, and brewing tea.

TABLE OF CONTENTS

1. Introduction ... 1
   1.1. A History of Software Development Methodologies ... 1
   1.2. The Divide and Conquer Approach ... 8
   1.3. Research Motivation ... 11
   1.4. Thesis Goals ... 12
   1.5. Structure of this Thesis ... 14
2. Related Research in Agile Methods ... 15
   2.1. Agile Practices ... 15
      2.1.1. Iterative Development ... 16
      2.1.2. A Flexible Feature Set ... 17
      2.1.3. An Involved Customer ... 18
      2.1.4. Developer Empowerment ... 19
      2.1.5. Lean Documentation ... 20
      2.1.6. Refactoring ... 21
      2.1.7. Test Driven Development ... 22
      2.1.8. Continuous Integration ... 25
      2.1.9. Acceptance Testing with FIT ... 25
   2.2. Empirical Evidence for Agile Methods ... 28
   2.3. Summary ... 28
3. Conceptual Approach ... 29
   3.1. Supporting Divide and Conquer using TDD ... 29
   3.2. Restrictions of the Approach ... 31
   3.3. Example Scenario ... 32
   3.4. Summary ... 36
4. Tool Support ... 37
   4.1. COACH-IT ... 37
      4.1.1. Purpose ... 37
      4.1.2. Design ... 38
      4.1.3. Implementation ... 41
      4.1.4. Anecdotal Evidence ... 42

4.1.5. Summary 44
4.2. MASE 44
4.2.1. Purpose 45
4.2.2. Design 45
4.2.3. Implementation 50
4.2.4. Anecdotal Evidence 51
4.2.5. Summary 51
4.3. Rally Testing Tool 52
4.3.1. Purpose 52
4.3.2. Design 54
4.3.3. Implementation 55
4.3.4. Anecdotal Evidence 56
4.3.5. Summary 57
4.4. Summary 57
5. Empirical Evaluation 58
5.1. Suitability of FIT Acceptance Tests for Specifying Functional Requirements 59
5.1.1. Purpose 59
5.1.2. Subjects and Sampling 60
5.1.3. Instrument 60
5.1.4. Results 63
5.1.5. Interpretation 65
5.2. Examining Usage Patterns of the FIT Acceptance Testing Framework 66
5.2.1. Purpose 66
5.2.2. Subjects and Sampling 67
5.2.3. Instrument 67
5.2.4. Results 71
5.2.4.1. Strategies for Test Fixture Design 71
5.2.4.2. Strategies for Using Test Suites vs. Single Tests 73
5.2.4.3. Development Approaches 75
5.2.4.4. Robustness of the Test Specification 75
5.2.5. Interpretation 76

5.2.5.1. Strategies for Test Fixture Design 76
5.2.5.2. Strategies for Using Test Suites vs. Single Tests 77
5.2.5.3. Development Approaches 78
5.2.5.4. Robustness of the Test Specification 79
5.3. Student Experiences with Executable Acceptance Testing 80
5.3.1. Purpose 80
5.3.2. Subjects and Sampling 80
5.3.3. Instrument 80
5.3.4. Results 81
5.3.5. Interpretation 85
6. Conclusion 86
7. Future Work 89
References 90
Appendices 96
Appendix A. Participant Consent Form 96
Appendix B. Questionnaire 98
Appendix C. Ethics & Co-Author Approval 99
Appendix D. Technical Detail 103
Appendix E. Test Suite 109
Appendix F. Raw Data 110

LIST OF TABLES

Table 1. Customer test statistics by teams 63
Table 2. Percentage of attempted requirements... 64
Table 3. Additional features and tests statistics 64
Table 4. Common FIT fixtures used by subjects 72
Table 5. Statistics on fixture fatness and size 73
Table 6. Frequency of test suites versus single test case executions during ramp up....... 74
Table 7. Frequency of test suites versus single test case executions during regression ... 74
Table 8. Ratio of valleys found versus total assertions executed 75
Table 9. Possible ramp-up strategies 77

LIST OF FIGURES

Figure 1. Cost of change over time in a software development project. 4

Figure 2. Theoretical cost of change over time in an agile project. 8

Figure 3. Four critical factors in project success and quality 18

Figure 4. Sample FIT table and ColumnFixture in Java. 26

Figure 5. Simple FIT table and ActionFixture in Java.. 27

Figure 6. A timeline of two teams working in parallel 35

Figure 7. A conceptual drawing of how COACH-IT works.. 39

Figure 8. COACH-IT allows the definition of components and links between components 40

Figure 9. The MASE web-based wiki tool for supporting agile projects 46

Figure 10. Simplifying how component dependencies are represented 47

Figure 11. Tests can be run on-demand in the MASE tool by clicking “Run” on the toolbar 48

Figure 12. Overview of how MASE functions conceptually... 50

Figure 13. Adaptations for Rally require that tests be run on a remote server 53

Figure 14. Overview of components used by both server and client programs................ 55

Figure 15. Empirical research completed versus outstanding 59

Figure 16. Assignment specification snapshot.. 61

Figure 17. A partial FIT test suite from the student experiment...................................... 62

Figure 18. Typical iteration life-cycle 68

Figure 19. Samples of how a given fixture could be implemented 69

Figure 20. A pattern of what incremental development might look like (left) versus what mocking and refactoring might look like (right)... 79

Figure 21. Student responses judging FIT for defining functional requirements............. 82

Figure 22. Student responses comparing FIT with a written specification....................... 83

Figure 23. Student responses comparing difficulty of FIT with JUnit 83

PUBLICATIONS

Materials, ideas, and figures from this thesis have in part appeared previously in the following publications:

Read, K., Melnik, G., Maurer, F. (2005) Examining Usage Patters of the FIT Acceptance Testing Framework, *Proceedings of XP2005*, Sheffield, UK, Springer

Melnik, G., Read, K., Maurer, F. (2004) Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective, *Proceedings of XP/Agile Universe 2004*, Calgary, Canada, Springer

Read, K., Maurer, F. (2003) Issues in Scaling Agile Using an Architecture-Centric Approach: A Tool-Based Solution, *Proceedings of XP/Agile Universe 2003*, New Orleans, US, Springer

1. Introduction

This thesis addresses how agile methodologies can be applied to teams of teams using a divide and conquer strategy on a software development project. The introductory chapter starts out with some history and background about agile methods. I then identify the value of teams of teams and of a divide and conquer strategy, and explain why this is relevant to the contemporary software development industry. Following this is a discussion of my research goals, which center on identifying and overcoming the challenges presented by the merging of these two approaches. These challenges are broken into specific problems addressed by the thesis. Finally, the introduction ends with a summary of the structure of the remainder of this thesis.

1.1. A History of Software Development Methodologies

Before discussing how agile methods can applied to teams of teams, it is necessary to have some understanding of what agile methods are, how they came to be developed and why they are now becoming more widely accepted.

The strategies by which teams of software developers create software can be separated in to three very general categories. The first strategy is, oddly enough, a lack of any real strategy. This thesis will refer to it as “chaos”. Many teams today still follow the chaos development strategy. Chaotic development has many different characteristics, and teams may meet one, many, or all of them. Often teams are working without any plan or means of measuring their own progress towards their goal. They also may not be aware if they are behind or ahead of any kind of schedule, and moreover have little to no knowledge of how realistic their goals may be. There may be no clear idea of who is responsible for what, and little accountability. Essentially, these kinds of teams are making things up as they go. The truth is that this can work for extremely small or simple projects. For example, students often cobble together a small project without any sense of methodology. However, as a project grows, small problems and issues begin to snowball into project failures. Bugs will start occurring more frequently and be harder and harder

to fix. Testing for those bugs may be insufficient and many will not be uncovered before the software is released. It will become increasingly difficult to add new features or make changes in the developing product. It also is harder go gauge project progress (no effort is being made to do so) and as a result not only might the team be behind schedule but they may not know it. A good analogy for this strategy is cooking without a recipe. For simple things, like toast and jam, we don't need a recipe or much experience to get an adequate result. For more complicated dishes, without a recipe it will just taste terrible.

A very common type of methodology used in all software development is document-centric development. In this type of methodology, the main idea is to formalize the entire project into a documented plan up front, and then to follow the plan as precisely as possible. The origin behind these approaches, also called Big Design Up Front (BDUF), Waterfall, or Tayloristic approaches, relates to the early 1900's work of Frederick Winslow Taylor [Taylor, 81]. Taylor performed now-famous "time in motion" studies which lead him to promote workplace specialization in order to optimize cost effectiveness in manufacturing. The most famous example of these theories in practice is the assembly-line manufacturing pioneered by Henry Ford [Austin, 8]. Ford's approach was to design the product (car) as well as the assembly-line and all tools needed to produce the project, then to break down the manufacturing into simpler and simpler sub-steps. Eventually these steps could be performed over and over by individuals and required little expertise or knowledge by those individuals. The steps were so simple and repetitive that anyone could do them. This assembly-line yielded highly efficient and predictable results. The analogous approach to software development is to create a comprehensive design up-front, and break that design into smaller and smaller tasks to be accomplished by specialized but not necessarily skilled workers. The initial design needs to include all requirements that the customer has and all details that must be included in the product. The assembly line for the software is then a "waterfall" process that starts with gathering the requirements and cascades through documenting the design, doing production and finally post-production or maintenance. This design methodology can and has worked in the past, and on the surface Tayloristic approaches are suited to teams of teams. By designing the software in advance, it does not matter who the "assembly

line worker" is, or where they are located. The workers merely need to follow the documentation provided. For this reason, Tayloristic approaches are often chosen for larger or distributed teams working on software projects.

There are nonetheless a number of problems and criticisms of Tayloristic methodologies. These criticisms are not all supported with research, but have been driven by anecdotal evidence and common experience. One criticism is that software development is too different from manufacturing to apply this production theory [Beck, 11]. In manufacturing, design is done once and production thousands of times, and therefore it makes sense to optimize production. In software development, the design essentially is the production, and it is done only once. Ford's Tayloristic approach to producing the car did not optimize how the car was actually designed, only produced. By designing a software system once as a document and then converting that design to source code, we may actually be doing our production twice (in the very least there is some overlap). In addition, it is easy to reverse development effort in a software project, unlike in manufacturing where once something is made it cannot easily be unmade. Another major criticism of this approach is that it is next to impossible to come up with a satisfactory design for software in advance—that doing so is basically an attempt to predict the future [Favaro, 23]. A complete design would require complete requirements. Customers simply do not know what they want, and it is not possible to say that we have elicited 100% of the customer's requirements such that they will be satisfied with what we produce [Thomas, 83]. It is estimated that 85 percent of the defects in developed software originate in the requirements [Young, 90; Ben-Menachem, 12]. "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" [Van Vliet, 85]. Moreover, the cost of change always increases over time. The cost of making changes to software in such a project increases exponentially as the project proceeds (see Figure 1) [Austin, 8]. BDUF projects are most likely to discover problems in the later, more expensive stages of development. Not being able to deal with change in a cost-effective way increases the risk of the project failing. Even by guessing that some change may need to be made down the road, it is impossible to totally predict what all changes will be

and how they will affect the project. Although we might guess correctly some of the time, other times we will waste time and resources on what we will never end up using [Beck, 10].

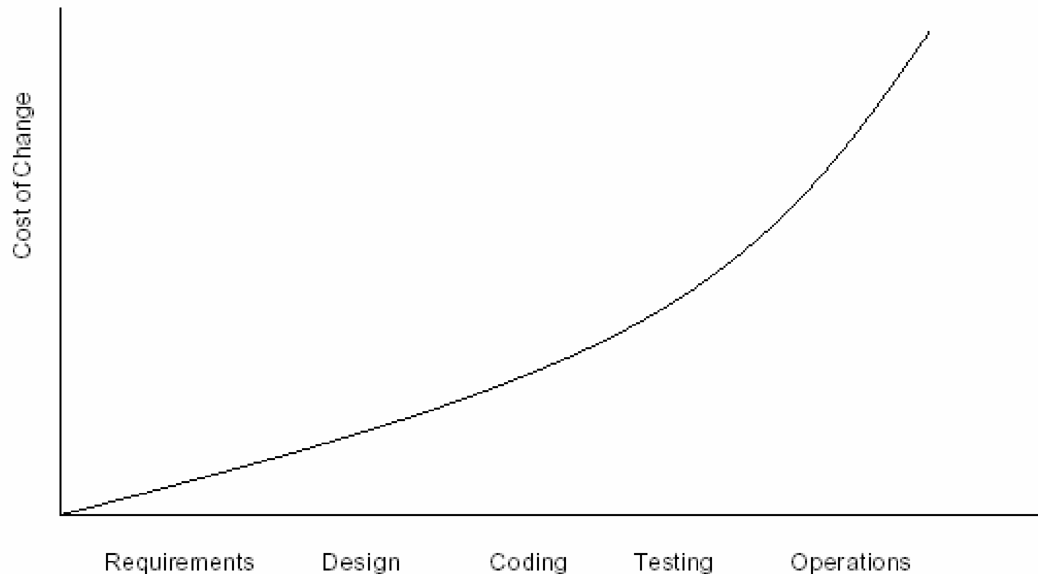


Figure 1. Cost of change over time in a software development project. Source: [Austin, 8]

A possible problem in projects using a "waterfall approach" to development – proceeding from one state through to the next - is that they may not deliver any kind of valuable product until the entire process is complete. Only after the very last step in a "waterfall" type document-centric process does the customer receive ready-to-use software. Therefore, if the project runs over time or over budget, it is likely that final stage of software production (often testing and quality control) will be left incomplete due to a lack of time or resources. At the end of a project it is silly to have a perfectly documented design but only 70% of the features complete – customers would doubtless prefer to have 70% of the design documented but 100% of the features.

An additional criticism of this type of methodology is that even with a perfect design, the translation of the design into the product is open to errors in human interpretation. Some such problems of interpretation and representation are listed by Meyer as "deadly sins"

[Meyer, 61]. The first such sin is noise, which manifests 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 make up 13 percent of requirements problems [Hooks, 37]. A second hazard is silence, in which important aspects of the problem are simply not mentioned. Omitted requirements account for 29 percent of all requirements problems [Hooks, 37]. Over-specification can happen when aspects of the solution are mentioned as part of the problem description; requirements should describe what is to be done but not how they are implemented. Wishful thinking is when prose describes a problem to which a realistic solution would be difficult or impossible to find. Ambiguity is common when natural languages allow for more than one meaning for a given word or phrase. Often this is problematic when jargon includes terms otherwise familiar to the other party [Meyer, 61]. Prose is also prone to reader subjectivity since each person has a unique perspective (based on their cultural background, language, personal experience, etc.). Forward references mention aspects of a problem not yet mentioned, and cause confusion in larger documents. Oversized documents are difficult to understand, use and maintain. Customer uncertainty appears when an inability to express specific needs results in an inclusion of vague descriptions. This, in turn, leads to developers making assumptions about "fuzzy" requirements: it has been estimated that incorrect assumptions account for 49 percent of requirements problems [Hooks, 37]. Efforts to make requirements understandable and complete might lead to the creation of multiple representations of the same requirements, and preserving more than one document can then lead to maintenance, translation and synchronization problems. Requirements are sometimes lost, especially non-functional requirements, when tools for requirements capture only support a strictly defined format or template. Lastly, requirements documents are often poor when written with little to no user involvement, instead of being compiled by requirements solicitors, business analysts, domain experts or even developers [CHAOS, 82]. The risk of misreading the requirements increases as the project team becomes larger and distributed over geographical, cultural or organizational boundaries [Konito, 47].

Finally, Tayloristic methodologies can be criticized for their improper use of labour. Essentially, software developers are best at writing code and not at maintaining design documents [Martin, 55]. Even assuming that developers are trained to be better at documentation, it is something they must do in addition to producing code. Developers also prefer to write code over documentation, as shown from studying students [Melnik, 58]. Essentially, the majority of the labour force likely to be working on a given software project will be unsatisfied, alienated or unskilled at the processes necessary in a document-centric approach to software development. Although one could argue that education is the problem, creating more skilled document authors is only a partial solution.

Another type of methodology that addresses many of the criticisms of big design up front or Tayloristic methods is agile. Agile software development also has been influenced by manufacturing production theory. In the 1970's, Michael Porter came up with the concept of value-chains in production [Austin, 8]. The idea of these value-chains is that at each stage of production, labour is adding value to raw materials, and the total value of the product is the result of the chain. In 1984, Eli Goldratt introduced his Theory of Constraints [Goldratt, 32], which states that the overall production capacity of a system is defined or limited by the bottlenecks in that system. According to this theory, the role of management is to merely identify and remove bottlenecks in production. In his book "The Fifth Discipline" [Senge, 76], Peter Senge brought together these ideas in 1994 and advocated considering the production system as a complex whole rather than the sum of its parts. This is in many ways an opposing viewpoint from Tayloristic theories of production. All of these ideas became slowly more and more popular as western manufacturing began to fall behind the management and production techniques used in Japan for products such as electronics and automobiles. At Toyota, if at any stage a team member finds a problem, they can stop the entire line and correct the problem before proceeding; their process incorporates the unpredictability of future changes in order to minimize risk and maximize quality [Womack, 87]. These new theories of production have largely replaced Tayloristic views in modern manufacturing. Agile software development incorporates these same philosophies into software development

[Poppendieck, 69]. The agile manifesto states four maxims that express the priorities of agile software development: Individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan [Manifesto, 54]. In agile development, software is produced in short iterations over which it is easy to manage predictability and efficiency. At the beginning of each iteration, the immediate future is planned, and customers prioritize and choose the features they would like most. Developers provide feedback about how much they can practically accomplish in the upcoming iteration. During each iteration workers add value to a deliverable product. At the end of each iteration, software is delivered that provides value to the customer, and changes are considered for the next iteration. Management removes obstacles and bottlenecks but trusts the team to deliver quality. By developing and planning the software in short iterations, customers can work with developers to come up with a product that is acceptable with a low risk of project failure. Constant re-analysis of time and budget goals help reduce this risk, along with the constant delivery of real useful features. The cost of change over time becomes much more manageable through this repeated cycle of development and feedback [Larman, 50]. Essentially the design and overall product evolves from a number of small changes rather than being built as a monolithic unit.

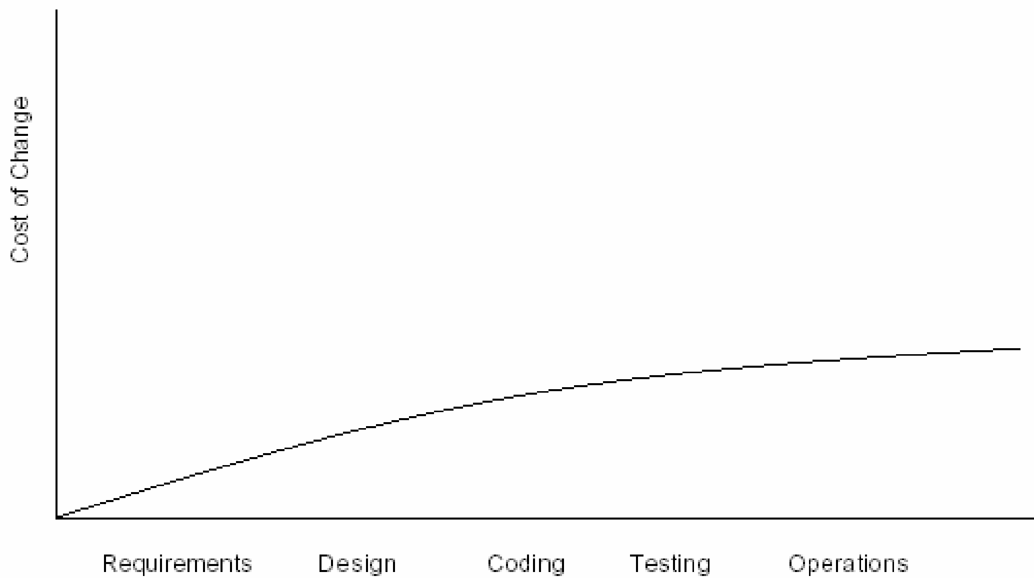


Figure 2. Theoretical cost of change over time in an agile project. Source: [Austin, 8]

In summary, software methodologies have been born slowly out of a need to create order from chaos. Traditional development methodologies were largely document-driven. These document driven design methodologies are still popular today, however over time a large body of criticisms have been assembled in both academia and among practitioners. These criticisms motivated a different perspective on software development, namely agile methods. Agile methodologies incorporate the philosophies of the agile manifesto and/or rely on an iterative, lightweight model for software development.

1.2. The Divide and Conquer Approach

In simpler times, a single person could produce a computer program. But as software and hardware have become more complicated, and as our expectations of what can be done have increased, teams of individuals have become necessary to produce a software product. Teams are not just necessary due to increasing product complexity, but also due to business demands [Karolak, 43]. A project that could be developed by one or two people over several years may not be valuable with such a long time-to-market. If a team

needs to deliver a lot of functionality but also has a lot of time, the team size can be quite small. Likewise the team can be small if it has not much time but can reduce the scope of the project. Large projects are out there, projects for which a small team is not ideally suited. However, to deliver a lot of functionality in a short amount of time, the business solution is to add more people so that work can be conducted concurrently. Therefore, greater resources (meaning larger teams) are employed in order to shorten the development time and deliver large systems in short timeframes. These teams are made up of people with different skills and different experiences, different personalities and histories. Not all of these people are always available in the same place. In a global market, resources are sometimes scarce or distributed in different geographic locations. Languages and cultures can separate members of a team [Carmel, 15]. Alternatively, corporate organization and structure can form boundaries between individuals just as effectively as geography. It may be more cost-effective to employ resources in different locations rather than re-locating them to a new common team (i.e., outsourcing). According to the 2005 Global Outsourcing Report [Global Outsourcing, 30], "Three-quarters of U.S. companies outsourced some or all of their information technology activities in 2004, and that percentage is likely to increase". META Group reported in late 2004 [METAspectrum, 60] that "The [US] offshore outsourcing market will continue to grow 20% annually through 2008". People telecommute, or there can be overstaffing or understaffing issues at corporate locations. Teams often collaborate with each other on projects, forming larger teams or by sharing responsibilities on a project. Large projects can now span multiple teams at multiple sites involving multiple companies and organizations. As people have come together to develop software in this way, they have adapted their software methodologies to help with the management of these complex interactions. These methodologies impose a disciplined process upon software development with the aim of making software development more predictable and more efficient. Efficiency, in the course of this thesis, can be taken to mean maximum return of value to the customer for their investment of time and/or resources.

Predictability and efficiency are even more important for large or distributed teams than in small, co-located teams. As the team size grows, there is a communications and

management overhead cost that was not present in the small co-located team [Herbsled, 33]. This communications overhead really means increased risk of project failure. A project is much more likely to fail or at the very least run over-budget or behind-schedule when there is no way to predict and gauge progress. Insufficient communication leads to unpredictability in both the present state of development as well as to the future of the project. There is a need then to mitigate this risk, and this is done through the adoption of methodologies. However, no matter what methodology is adopted, there will be a loss of communication bandwidth as teams grow or become distributed. No matter what methodology is used, if people are present in a co-located environment then some communication will be conversational. If you are in the same room as a co-worker you are very likely to know the state of their work and how it relates to your work. You are also more likely to solve problems or have work related discussions with that person. However, as the number of individuals increase or as the team is distributed or divided, it is not practical for all individuals to have conversations with one another. Certainly the frequency of casual conversation will decrease. Telephones and networks will be used more and more, but communicating this way does not transfer as much information as a face-to-face conversation [Kraut, 48]. Moreover, when given a choice of how to communicate, people prefer face-to-face over email, phone, or other means [Finholt, 24]. Methodologies universally must address this issue by mandating increased communication. Some common methodological ways to increase communication are scheduled conversations, meetings, and formalizing information and communication into documentation. All methodologies therefore mitigate the risk of larger or distributed teams to some extent.

On software projects developed by large or distributed teams, one strategy is to scale up from a single team to a team of teams. This can also be called the divide and conquer approach, because the problem is being divided among multiple groups. Divide and conquer is nothing new, it simply means breaking the problem down into several smaller problems, then solving these smaller problems in order to find a solution to the whole. This strategy is particularly resource effective because more than one team can work in parallel, maximizing the output over a given period of time. Dividing the problem does

help with the communication issues previously mentioned, since the smaller pieces can be developed by smaller, more cohesive units. Many people believe that the most effective teams are smaller teams [Katzenbach, 44; McConnel, 57], because these teams can stay focused, develop interpersonal relationships, can easily co-locate in a single space, and can frequently communicate with each other and update one another about project progress and status. By splitting a project into smaller projects and assigning them to different teams, we partly overcome the communications problems associated with too many people in one place. Any number of methodologies can be applied while using a divide and conquer strategy. Divide and conquer does not tell us how to develop the software, it is just a term that indicates the presence of many smaller problems that need to be solved in order to solve the overall problem. This implies the presence of more than one team working in parallel, each doing their own part. How software is developed in agile teams following this approach, as well as how these parallel teams communicate and interact is the topic of this thesis.

1.3. Research Motivation

As previously mentioned, agile methods address many of the common criticisms of document centric or Tayloristic development. For these reasons agile methods are gaining popularity among industry practitioners. However, agile methods have only recently begun to be investigated by the software engineering research community. The majority of this research has been directed at validating the claims made by grass-roots practitioners [Highsmith, 35], through observing agile projects and collecting empirical data about them [Abrahamsson, 1]. There is also increasing interest [METAspectrum, 60] in the software engineering community surrounding the topic of distributed software development, and one commonly proposed strategy is using teams of teams [Kruchten, 49; Schwaber, 75]. However, existing research is based predominantly on document driven methodologies. Recent introductory work has been done on applying agile methods outside of their most common context [Lippert, 53], such as to large [Fowler, 27; Elssamadisy, 21; Reifer, 73; Erdogmus, 22] and distributed [Holz, 36; Simons, 78; Rees, 72] teams. This suggests that there exists a demand to expand the application

domain of agile methods. However, the following problems are yet unanswered by the research community:

1. It is not known if agile methods can be applied in a divide and conquer setting and yet remain agile, or whether these approaches are complementary;
2. It is not known what adaptations can be or need to be made to agile methods in order to enable such an approach;
3. There exists no specialized tool support to enable or assist in using agile methods with a divide and conquer strategy.

By examining the needs of a divide and conquer project and applying agile philosophies and practices to meet those needs, it may be possible to suggest a scheme for incorporating both approaches. Such a scheme would allow agile methods to be applied across a broader scope of projects. Moreover, organizations solving problems using teams of teams may be provided with additional options for choosing a software methodology.

1.4. Thesis Goals

Agile methods address many of the criticisms of big up front design. However, these methodologies were not developed to be a good fit for large or distributed software development teams [Turk, 84]. There will be three major incompatibilities between agile processes and the divide and conquer approach. These incompatibilities are the need for up-front design, team inter-communication and integration. The first problem is how we can define the responsibilities of each team so that everyone knows what they are responsible for, with well-defined success criteria. The second problem, namely communication, is how to deal with the inevitable interactions and dependencies that will exist between the sub-projects under parallel development. Thirdly and lastly, there needs to be a mechanism of ensuring that the final products developed by parallel agile teams do work together properly to form a complete solution. All three of these issues are solved in big design up front projects through creating detailed and heavyweight prose design documents, which are not acceptable on an agile team. In order to make agile

methods a viable choice on such projects, they need to be adapted to overcome these challenges without introducing the problems of big up front design, and without creating new problems. The unanswered question is if agile methods can be adapted in such a manner and still remain agile.

There might potentially be many ways in which agile methods can be applied to a divide and conquer project. It is beyond the scope of a Master's thesis, and indeed beyond the scope of several theses, to attempt to identify and validate every possible scheme or situation in which agile methods can be applied. Instead, it is the goal of this research to come up with a single solution for how agile methods might overcome the three aforementioned challenges. For this solution, I intend to:

1. Explain how the solution overcomes the problems of a divide and conquer approach
2. Make clear that this solution does not violate the definition of an agile methodology
3. Provide working tool support to enable adoption of this solution
4. Validate (in part) the practicality of the solution

This is meant to be an exploratory thesis, which examines the problem and proposes ideas leading to a potential solution. The development of tool support is also exploratory, in that multiple tools will be developed to demonstrate how different approaches to the problem could be concretely supported. These tools will not be evaluated in any scientific study. The empirical studies and validation of ideas included in this thesis are a bonus and are meant to be treated as merely an indicator of what more complete studies might reveal.

The solution investigated in this thesis will be whether some form of test driven development can replace document-centric design as a way to enable multiple teams to work together using agile methods on a project. I hypothesize that a lightweight representation for the system can be found in which dependencies between projects and

teams can be identified, communicated and verified through some form of automated test. If this is true, then these tests can be created in a test driven development environment that is consistent with agile philosophy, and incorporated into the iterative development practice of agile methods. The end goal is to enable the use of agile methods for a team of teams when using a divide and conquer project strategy.

1.5. Structure of this Thesis

Following this introduction, the thesis will take the following form: Chapter Two imparts some background knowledge about the specifics of agile methods, including an overview of relevant agile practices and tools. Also included is an overview of the small but growing body of empirical research supporting the claims of agile methods. Some related work is provided which relates to expanding the context in which agile methods can be applied. In Chapter Three, I outline the proposed solution for applying agile methods along with a divide and conquer strategy. This chapter includes a description of the approach as well as a hypothetical situation and how the approach might be applied in a step-by-step way. Chapter Four presents three tools developed to support and validate the proposed approach. This is followed by a series of empirical studies in Chapter Five which attempt to validate the claims of this thesis. Chapter Six concludes with an analysis of the results of the empirical findings and judges the practicality and viability of the entire solution as presented in this thesis. Finally, Chapter Seven discusses potential future work in this area.

2. Related Research in Agile Methods

The first purpose of this section is to familiarize the reader with the common practices of agile methods, including the motivation for using each practice and how it relates to the philosophies of the agile manifesto. Following this overview will be a short discussion of the ongoing research surrounding agile methods, specifically focusing on recent publications more closely related to the ideas of this thesis.

2.1. Agile Practices

There are many different "Agile methods" defined in books or papers or purveyed by professional consultants. These include popular approaches such as eXtreme Programming (XP) [Beck, 10], Feature Driven Development (FDD) [Palmer, 6694], Scrum [Schwaber, 75], Dynamic Systems Development Method (DSDM) [Stapleton, 79], and Adaptive Software Development (ASD) [Highsmith, 34]. The majority of these methods have a lot in common. It would not be very useful to create a complete atlas of the practices and processes for all agile methods; however it will be useful to define agile methods within the scope of this thesis, and to describe the workings of "typical" agile team that can henceforth be used as a reference. More detail will be used to describe test driven development, acceptance testing practices, and several important testing tools as they are integral to the approach proposed in this dissertation.

In this thesis, an agile method will be defined as one which:

1. Uses an iterative development process
2. Empowers the developer
3. Involves the customer
4. Is flexible to change
5. Does not require comprehensive up-front design documentation

2.1.1. Iterative Development

Agile methods are typically built around the idea of an iterative development schedule. Iterative development means delivering small, working parts of a project on a regular basis. In general, agile methods use very short iterations of development – as short as one or two weeks [Beck, 10]. By keeping the iterations short, agile methods keep the customer, developers and management informed about project progress, and allow opportunities to make changes as necessary [Basili, 9]. At the end of each iteration, the goal is to be able to deliver value to the customer; this means that it should be possible to roll out the software as-is and for the customer to benefit from it (although this is not always done every iteration). Each iteration is planned just before it starts, incorporating any changes discovered in the last iteration. At the beginning of each iteration the team collectively defines "user stories". User stories serve the same purpose as a document of collected requirements. However the difference is that each user story only is discussed in enough detail to identify the task and create a reasonable work estimate. User stories are written by the customer or with customer input, and should also be prioritized by the customer so that the most valuable stories come first. In general, a user story will consist of about three sentences of text written on an index card, by the customer, using the customer's terminology. Stories are devoid of technical details such as algorithms or graphical user interface layouts, but rather simply identify user needs. After user stories are written and prioritized for the next iteration, developers estimate how long the stories might take to implement. If a story takes more than an iteration to complete, it needs to be broken into smaller user stories first. After the developers give their estimates, those user stories that are of sufficiently high priority and fit within the available timeframe of the next iteration are chosen for development. Those user stories that do not fit within the scope of the next iteration remain in the product backlog for future iteration planning meetings. During an iteration, each day developers can themselves choose which user stories to work on from those making up the current iteration. Management exists to remove bottlenecks and problems that may be keeping the team from completing their work. An agile manager might provide tools for the team or solve problems with the physical work environment. Every day agile teams hold what is called a stand-up

meeting. At a stand up meeting everyone can communicate problems, solutions, and promote team focus. Participants remain standing up to avoid long discussions. The idea is that it is more efficient to have one short meeting daily rather than several long meetings infrequently.

Iterative development is important to understand in the context of this thesis because is it necessary to preserve the workings of the iterative development cycle in order for the process to remain agile. The nature of iterative development necessarily spreads the effort of planning and design over the whole project timeline. After each iteration, new decisions are made and disseminated among team members. These decisions include what to work on next, who will work on what, and how to resolve any issues from the previous iteration. This process relies on the assumption that planning and communication can be done often and quickly.

2.1.2. A Flexible Feature Set

In many software projects, the attempt is made to make constant the budget, the schedule, software quality, and project scope in terms of the number of features (see Figure 3). With all of these being fixed, there is a high probability of one of them being exceeded [Ambler, 4]. For instance, the project will run overtime, over budget or not be completed when change is required. Even if the features are completed on-time and budget, the quality may suffer due to rushed development, cutting corners, lack of quality assurance practices, and so forth. In an agile project, no attempt is made to prescribe the features being developed [Patton, 67]. By not fixing the scope, agile projects aim to always be on-time, on-budget and delivering high quality software. Any setbacks will result in features being postponed – and with prioritized, iterative development schedules it should be the least important features that are cut. Should a user story prove to be more time-consuming than estimated, a new estimate should be communicated to the customer. At that time the customer can choose whether to drop lower-priority items from the iteration to make more time, or to simply move the problematic user story into the product backlog for a future iteration. This minimizes the negative impact of change and chance of project failure. In an XP environment "velocity" is a measure of person-hours spent on

completed user stories per iteration, divided by the number of total person-hours invested in that iteration [Alleman, 3]. Project velocity for the previous iteration can be used to predict the velocity and estimate a target size for the next iteration. As the project progresses, this number will become more stable as developers get better at making estimations.

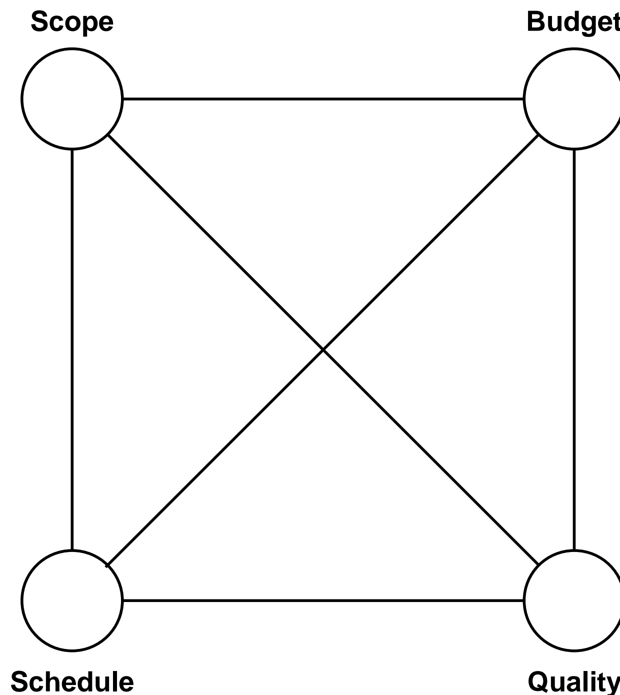


Figure 3. Four critical factors in project success and quality: Budget, Scope, Schedule and Quality

The concept of a flexible feature set is dangerous in a divide and conquer project because without a fixed scope, it is difficult to determine and resolve dependencies. When software is being developed in parallel, dependencies say which pieces might need to be developed first, so that other pieces may be built "on top" of those pieces.

2.1.3. An Involved Customer

"Lack of user involvement traditionally has been the No. 1 reason for project failure. Conversely, it has been the leading contributor to project success. Even when delivered on time and on budget, a project can fail if it doesn't meet user needs or expectations" [Johnson, 41]. It is important in agile methods to have a customer or customer

representative available as much as possible. Because user stories only provide enough detail to identify the feature, developers need to go to a customer when they start working on the story to receive a detailed description of the requirements in a face to face conversation. Often the on-site customer will need to be consulted by developers continually while they develop the software. This continuing feedback ensures that the customer representative always knows what is being developed, and therefore both developers and the customer have the opportunity to bring up issues at any time. This may seem like a lot of the customer's time. However, when employing an agile team the customer needs to be willing to contribute their resources in order to make the project work. "If the client won't give you full-time, top-flight members, beg off the project. The client isn't serious" [Peters, 68]. If reluctant to participate, the customer can be reminded that their time is spared initially by not requiring detailed requirements elicitation or the negotiation of a requirements specification. Time is also saved later when the agile team delivers a helpful system rather than one requiring re-work.

This concept is also challenging to a divide and conquer type of project because it becomes increasingly difficult to have a customer on-hand if there are multiple project teams or sites. Moreover, as a project scales to multiple teams, a single customer representative may be stretched too thin. Beck [Beck, 10] argues that if the project is valuable to the customer, they will be willing to pay with increased involvement.

2.1.4. Developer Empowerment

Agile methods universally empower the developer [Cao, 14]. Although there are many different approaches to how the developer is empowered, all agile methods agree that the people on the team are more important than the process. The overarching idea is that happy developers are productive, successful developers [Murru, 63]. One commonly used idea is sustainable development, which sets a limit on the number of working hours each week and mandates that no overtime be asked for. Kent Beck put it thus:

> “I want to be fresh and eager every morning, and tired and satisfied every night. On Friday, I want to be tired and satisfied enough that I feel good

about two days to think about something other than work. Then on Monday I want to come in full of fire and ideas." [Beck, 10]

Problems should be resolved through the iteration planning meetings and through customer collaboration, not by torturing your developers and making them regret that they are working on the project. Other forms of empowerment include collective code ownership, in which all developers are trusted to make changes to the entire code-base and to commit their changes to the final product, and developer rotation, where developers are moved around frequently to get a chance to work on and learn about different aspects of the project rather than encouraging that each person become a specialist on one tiny detail. Developer empowerment, the ability of each developer to make decisions involving the project and what they themselves do, is an aspect of how agile a project is. Any adaptation to agile methods should keep this as a central concept.

2.1.5. Lean Documentation

In general agile methods are known for their minimalist approach to written documentation. In fact, it is a common misperception that agile methods mandate that no documentation is written at all. To the contrary, agile methods simply place higher value on delivering actual working code to the customer in order to maximize the value delivered. In order to minimize waste, documentation is minimized when the customer is unlikely to need it, or when the investment effort in documentation exceeds the value that the customer places on that document. Because software is produced through the effort of design, designing software once as a written document and then writing code can be redundant. Nonetheless, agile developers will do documentation if warranted. For example, documentation can be used when developers need to communicate asynchronously, such as through email. Documentation is also useful when the customer specifically asks for it and places value on it as a product – for instance a user manual. Ron Jeffries writes "Outside your extreme programming project, you will probably need documentation: by all means, write it. Inside your project, there is so much verbal communication that you may need very little else" [Jeffries, 40]. However, detailed documentation of what the code does or how the code works is generally looked down

upon. The philosophy of agile methods is that most often the code is the documentation. Developers are assumed to be highly trained experts who can read code and understand its meaning. Therefore when a developer seeks to understand part of the system, they are likely to examine the source code, and if the developer cannot understand the source code, they are likely to get help from their team to refactor it into a more understandable form. The source code is also more reliable then the documentation, since it cannot be out-of-date or obsolete. In a team of teams, interpersonal communication may not be present to the same degree as in a single small, co-located agile team. Assuming that we wish to remain agile, the principles of lean documentation should be retained despite this challenge.

2.1.6. Refactoring

Refactoring is a process which involves re-working existing and new code as the system is developed, in order to maintain "good design" such that the product remains maintainable and expandable while minimizing waste. What constitutes "good design" is left as a subjective term for the developer, however it commonly means code that is easy to read, where everything is expressed once and only once. This practice is important because unlike in BDUF projects there is no "master design" that everyone is following, and there is a risk that code will be duplicated or that code will be forgotten or become obsolete. Refactoring takes time, but agile philosophy claims that this time pays off [Lindvall, 52]. By refactoring code, we come to understand that some parts of our software are difficult to understand and refactor. These areas are the very areas that often have a higher density of design defects or bugs. When these areas are not refactored, developers may spend more time fixing bugs and trying to understand how the code works than they would doing the refactoring in the long run [Du Bois, 20]. Ralph Johnson and William Opdyke were the first to use the term refactoring in print [Opdyke, 65], but acknowledge that it came out of the Smalltalk programming community. A related concept to Refactoring is the idea to "do the simplest thing that could possibly work". According to this maxim, developers are encouraged to always take the simplest approach when implementing a solution, and not try to predict the future or build-in additional features that may or may not be needed later. Refactoring is used when code

needs to expand, and therefore expending effort to build in purposeful upgradeability may be a waste of effort. Refactoring has become a popular practice among all software developers, and many tools now exist to assist with refactoring, including support in most modern programming environments such as IntelliJ Idea, Eclipse, IBM Websphere, and Microsoft Visual Studio. Refactoring is an important consideration for agile teams of teams because code changes can break dependency relationships. However, in this thesis an approach is discussed in which tests themselves communicate between teams; refactoring may therefore be much less risky.

2.1.7. Test Driven Development

Automated testing is a cornerstone of agile methodologies. The underlying concept of test driven development is the automated test written before the production code is developed. An automated test is a bit of software that can be run at any time to test some specific aspect of the system. Developers create automated tests so that they do not need to continuously manually check that their code is working according to their expectations. An automated test is made up of a set of input parameters as well as some representation of what the expected output will be. The test applies the input parameters to the target code and checks that the result matches the expected output. Many types of automated tests exist. One type of test commonly written by developers is the unit test. A unit test is a type of automated test at the granularity of code-level detail. Because these tests are written entirely in the language of the program being tested, they do require developer expertise to create, and are platform-specific. Developers write unit tests to check the quality of fine-grained bits of code, and can use suites of unit tests or larger unit-tests to check coarse-grained larger areas of code. A standard framework used for creating unit tests is xUnit, for example JUnit [JUnit, 42] for Java. Another type of automated test is the acceptance test. Unlike unit tests, acceptance tests are not designed to test that code is working according to developer expectations. Rather, acceptance tests check to make sure that features are good enough for the customer. For this reason, acceptance tests are written with customer input, and include input parameters in the customer’s language as well as expected output criteria from the customer’s perspective. Acceptance tests are so named because they are a means of verifying that a feature is

complete from the customer's perspective: if it passes it is acceptable to the customer. Acceptance tests are very "coarse-grained" and test software at a higher level than unit tests. One acceptance test might use code tested by dozens of unit tests. Because these tests are high level, they are not necessarily tied to the same environment or platform as the target software.

Simply writing automated tests provides many benefits [Kaufmann, 45]. Automated unit tests provide instant feedback, and can act as a "safety net" when changes are made. If tests are executed after refactoring or new features, we can be sure that these changes have not caused problems (this process is known as regression testing). Automated tests also verify that any assumptions we have made are correct; code that is technically correct may act incorrectly if it depends on external assumptions.

Automating tests are not exclusive to agile methods. However, a more agile practice is writing the automated tests before writing the actual code. A motivating factor to write the tests first is that doing so helps the developer to think about and understand his or her design before starting to code [George, 28]. A criticism of methodologies that are light on documentation is that the developers simply write whatever first comes to mind, without thinking or designing in advance. Doing test-first development (also called test driven design) ensures that ideas are thought through, and that success, failure, and exception conditions are well considered. Writing the tests first also has the side effect of making sure that the tests get written at all – it is very dangerous to assume that you will have time to write the tests later, since you may never get around to it. Maintaining a very high level of test coverage is important when doing constant code refactoring and following the agile tenet of responding to change [Kaufmann, 45]. Typically, most agile methodologies advocate unit test driven development practices, and this has also been the focus of the majority of research [Geras, 29]. Of late, it has been suggested that creating acceptance tests along with user stories can help drive the design and implementation of a project (story-test driven development). Supporting this idea are Johan Andersson in his paper "XP with Acceptance-Test Driven Development" [Andersson, 6] and Richard Watt and David Leigh-Fellows in their paper "Acceptance Test Driven Planning" [Watt, 86].

One question that may come to mind is how to write tests before code. There is no assumed "API" or interface from which to construct the tests. The test developer is essentially inventing and planning how their code will work based on how they create their tests. With some testing frameworks, the syntax used to specify the tests is not relevant to how the actual code ends up; an extra step is needed to "map" the code to the test. In other frameworks, the syntax used in the test specifically mandates that the same syntax be used when the code is created.

An idea implicit in agile methodologies but not necessarily exploited by agile teams is that just as the code can act as its own "documentation" for a system, the tests can also act as "documentation" [Kohl, 46]. Watt and Leigh-Fellows have observed "When first introduced to acceptance testing, many think of them merely as a form of black box testing and miss their greater significance as a multi-use form of requirements specification" [Watt, 86]. Developers who are able to read unit tests can determine from the expected input and expected output what that bit of code is supposed to do. Similarly, developers who can read acceptance tests can determine from the customer's input samples and expected outcomes approximately what the feature is supposed to do. When we think about what a test is, we realize that it is an expression of a developer or customer expectation. Tests represent what the business process is and where the business value comes from, and the code to make the test pass implements that business process. Tests have input conditions and output is produced. All of these are similar to the characteristics of what in software engineering we call a functional requirement. The ANSII/IEEE standard 729-1983 defines a functional requirement as "a requirement that specifies a function that a system or system component must perform" [Palmer, 66]. Functional requirements therefore also are an expression of an expectation. They represent what the business process is and where the business value comes from, and have input conditions and output that needs to be produced. This similarity is the keystone behind what holds up the argument in this thesis – that agile methods can be employed on a team of teams by communicating using tests (specifically by representing functional requirements as acceptance tests).

2.1.8. Continuous Integration

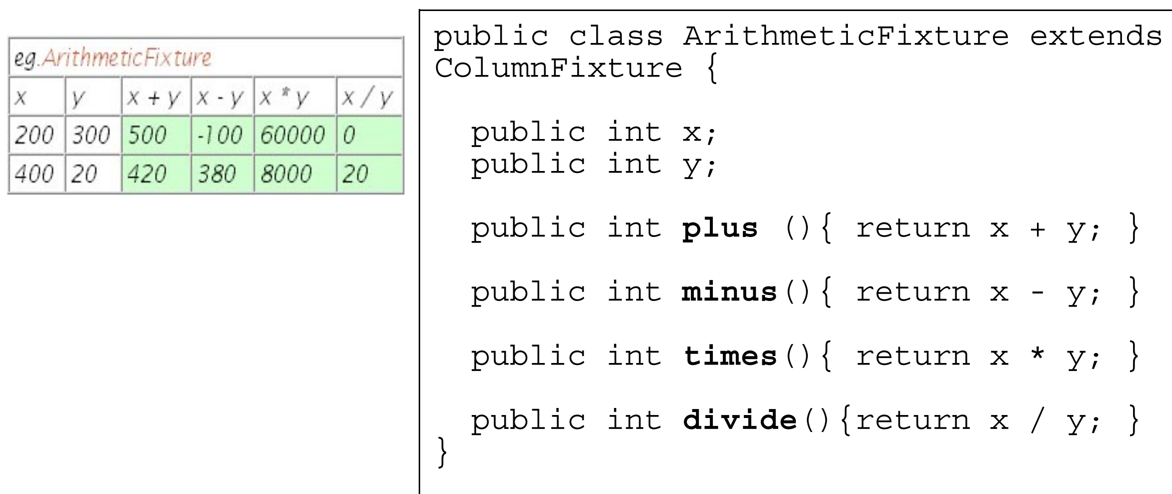
The aforementioned idea of regression testing has been extended in test-driven development to include a practice known as continuous integration. Continuous integration is a process that automatically rebuilds the system and runs all of the automated unit or acceptance tests whenever any developer commits a change to the code repository. If the developer "breaks the build" by introducing problems or making tests fail, the entire team is notified and the problem needs to be fixed before work can continue. This practice ensures that there is always a completely working system in the code repository that can be deployed at any time. It also ensures that developers are careful about running and writing tests, since they do not want their peers to be upset that they are constantly introducing errors. Continuous integration tools such as Cruise Control [CruiseControl, 19] send out emails or instant messages about build status, but agile teams also often use fun devices such as sirens, lights or alarms to indicate that a problem has been detected. Continuous integration helps support developer empowerment, making it easy to trust anyone when it comes to committing changes to the code repository. Some suggestions for scaling continuous integration are available in "Staging: Scaling Continuous Integration to Multiple Component Teams" [Appleton, 7] and one of the tools developed as part of this thesis (see Chapter 4) is based heavily on the Continuous Integration idea.

2.1.9. Acceptance Testing with FIT

By definition, acceptance tests assess whether a feature is working from the customer's perspective [Abran, 2]. Acceptance tests are different from unit tests in that the later are modeled and written by the developer, while the former is at least modeled and possibly even written by the customer. There is an ongoing debate about who should write acceptance tests [Sepulveda, 77], and the differences between acceptance testing and unit testing has been examined by Rogers [Rogers, 74]. 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. Acceptance tests can be specified in many ways, from prose-based user stories to formal languages.

Because the manual execution of acceptance tests is time consuming and costly, it is highly desirable to automate this process. Automating acceptance tests gives a yes-or-no answer when functional requirements are fulfilled [Crispin, 18]. At the same time, making the requirements too formal alienates the user, as in the case of definition using formal languages.

FIT [FIT, 25] was named from the thesaurus entry for “acceptable”. The goal of FIT is an acceptance test that a non-developer can read and write. To this end, FIT tests come in two parts: tests are defined using ordinary tables (usually, written by customer representatives, see Figure 4 and Figure 5, left side), and later fit fixtures are written to execute code using the data from table cells (implemented by the developers, see Figure 4 and Figure 5, right side). By abstracting the definition of the test from the logic that runs it, FIT requires only knowledge of the business domain and its own syntax to author tests.



eg.ArithmeticFixture					
x	y	x + y	x - y	x * y	x / y
200	300	500	-100	60000	0
400	20	420	380	8000	20

```
public class ArithmeticFixture extends
ColumnFixture {

  public int x;
  public int y;

  public int plus (){ return x + y; }

  public int minus(){ return x - y; }

  public int times(){ return x * y; }

  public int divide(){return x / y; }
}
```

Figure 4. Sample FIT table and ColumnFixture in Java. Excerpt from fit.c2.com

FIT tables can be created using common business tools, and can be included in any type of document (HTML, MS Word, MS Excel, etc). This idea is taken one step further by FitNesse [FitNesse, 26], 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. The FitNesse wiki allows anyone to contribute content to the website without knowledge of HTML or programming technologies.

fit.ActionFixture		
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 5. Simple FIT table and ActionFixture in Java. Excerpt from fit.c2.com

Although acceptance tests are often written based on user requirements, I believe that with FIT it is not necessary to create a written requirements document before creating an acceptance test. FIT tests are a representation of customer expectations that aim to be understandable by just reading tables. Mugridge and Tempero discuss the evolution of acceptance tests to improve clarity for the customer. Their approach using tables for acceptance test specification (i.e. FIT) was found to be easier to use than previously developed formats [Mudridge, 62]. All that is needed to write a FIT table is a customer expectation and the ability to precisely and unambiguously write it as an example. In this way they are very similar to written functional requirements. If the expectations themselves adequately explain the requirements for a feature, can be defined by the customer, and can be read by the developer, there may be some redundancy between the expression of those expectations and written system requirements. Consequently, it may be possible to eliminate or reduce the size of prose requirements definitions. On a related note, Steinberg has already suggested how acceptance tests can be used by instructors to clarify programming assignments and by students to check their progress in introductory courses [Steinberg, 80]. He found that FIT acceptance tests can be useful in this way without sacrificing other teaching materials.

2.2. Empirical Evidence for Agile Methods

Agile methodologies and their practitioners make many claims. Among these claims are that developers are more content, that productivity is as high or higher than other approaches, that quality is as high as or higher than other approaches, and that customer satisfaction and project success rates are improved. The vast majority of support for agile methods is in the form of published books and guides, along with word-of-mouth and anecdotal evidence. Consultants are also pushing agile methods on the basis of their own experience. Research into agile methodologies, especially empirical findings, is very recent. Publications in IEEE Computer by known authors such as Barry Boehm [Boehm, 13] and Jim Highsmith [Highsmith, 35] are making agile methods more mainstream and helping researchers and practitioners understand the facts and myths of agile software engineering. However, these articles and most others are generally focused on introducing agile methods and presenting initial evidence. There are no long-term or significant empirical studies on the use of agile methods or agile practices. Early studies such as those done by Abrahamsson [Abrahamsson, 1], Melnik et al. [Melnik, 59] and Lindvall et al. [Lindvall, 52] offer encouraging evidence that agile methods are viable in the short term.

2.3. Summary

By now the reader should have a good grasp on the core agile concepts and practices, including the motivation for using agile methods. In particular, the reader should be familiar with the concepts of continuous integration, and test driven development (including different types of tests and how FIT acceptance testing works). Moreover, the references in this section should have left an impression about the current state of agile methods research. All of this background information builds up the foundation of a solution to how agile methods can be used to help teams of teams.

3. Conceptual Approach

In this section of the thesis, I will describe my proposed approach enabling the use of agile methods on teams of teams using a divide and conquer approach. As previously mentioned, there are three principal challenges that need to be overcome in applying agile methods in such a scenario. Firstly, there is the problem of communicating the initial design and assignment of responsibilities. Secondly, team inter-communication during development, specifically to resolve dependencies, is difficult. Finally, there is the problem of integrating parts as they are completed into a unified whole. The first section of this chapter will describe the proposed approach in general, and discuss how it addresses these three aforementioned challenges. The second section of this chapter discusses restrictions to using the approach, and the third and final section applies the approach in a step-by-step way to a hypothetical project.

3.1. Supporting Divide and Conquer using TDD

The central concept of this thesis is that a lightweight representation for a divide and conquer project exists, in which dependencies between project parts and teams can be identified, communicated and verified through some form of automated test. In general the idea is to adapt the Test-Driven Design practice from agile methods to meet the needs of using a divide and conquer strategy resulting in teams of teams. Test-driven development is already used in agile teams to replace up-front design at the code level, and so it should be intuitive for agile teams to spontaneously plan their architecture and dependencies using tests. Moreover, because agile developers are used to the philosophy of "the code being the design", tests themselves should be a readable and understandable medium of communication. Finally, automated tests can be executed to ensure smooth integration, just as is currently done with continuous integration. This approach integrates with existing agile practices unobtrusively, essentially adapting current practices to meet new needs.

The first major challenge is minimizing up front design and overhead. There are several reasons why we need to minimize the initial design work that needs to be done. Agile methods rely on flexibility, and in fact exist in part as an avenue to address project risk through increased flexibility. If too much design is done before beginning iterative development, it will highly restrict the evolution of the product and how the iterations operate. Ideally the team should have complete control over what is done in any given iteration. On the other hand, we do need some representation of the dependencies between teams. The solution to this challenge is to rely on the concept of implicit architecture. It is too heavyweight to expect agile teams to build and maintain any sort of explicit or written architecture of all system dependencies. However, if we simply ask developers to create acceptance tests for dependencies they discover, as they are discovered, the collection of tests that is slowly built up will represent the evolving system architecture. Each test is in essence a link between two modules or components, just as if it were a link on a graph of connected nodes. There is therefore no need to represent the system up-front, because it will make itself apparent over the course of the project. This evolution of the architecture should also involve actual customer representatives, who can make decisions about the entire deliverable system if conflicts or questions of priority arise.

Because we are doing development on many parts of the project in parallel, and because we do not know in advance what dependencies exist between these parts of the project, there needs to be a way to identify and communicate dependencies as they are revealed. Communicating conversationally or sharing knowledge in a face-to-face format is impractical in teams of teams who may involve many people or not be co-located. Acceptance tests fit the criteria for agile teams and are capable of communicating project dependencies. Jack knows he is finished when all of Jill's tests for his components pass. Likewise, Jill knows Jack's code will integrate with her own when the tests she provided are successfully run by Jack. In essence, the author of a test becomes a customer for the module developer. A hierarchy of customers is formed, with one or more actual on-site customers at the "top". The real customers speak with some of the teams, who then define user stories and tests related to the child components. At each level the developers

have their own product backlog of user stories defined by the customers with whom they interact. These user stories are complimented by the automated tests.

The final challenge is that of integration. More specifically, it must be certain that the products each team develops in parallel will work together when it is time to deliver the project. This problem of integration is ideally solved by using automated tests and test driven development. As long as the tests are passing, and if tests exist for every dependency, then logically the divided project will come together in the end. The iterative nature of agile development will ensure that any neglected dependencies or misunderstandings will be resolved early rather than late and new tests will be put into place to ensure that such problems are not recurring. Changing the system drastically is potentially a source of difficulty, but to address this we can recall how refactoring handles changes to code. Changing a small amount of code can sometimes have sweeping effects, but now and again we need to evaluate the cost-benefit tradeoff and make a decision. If our general strategy is to make changes little by little, and keep the architecture healthy, then flexibility is not necessarily lost.

3.2. Restrictions of the Approach

This thesis purposely does not include any advice or methodology to determine how to divide the responsibilities of a project. This omission is largely because the way in which a project is divided is highly specific to the type of project and type of teams involved. For example, two companies working together may wish to divide the project according to the specializations of each company. Alternatively, a project may be divided according to an obvious division of the design, such as graphical interface and back-end. Suffice to say that for a large number of projects the broadest division of project responsibility will likely be obvious based on the project, goals and teams involved. If this is not the case, a divide and conquer strategy may not be the best approach to consider. Moreover this thesis does not address the issue of load-balancing between teams. Maintaining that a fair or equal amount of work is in the backlog of each team at any one time may require some additional planning or communication.

We also need to assume that these teams are willing to employ agile practices and meet the typical criteria for agile methods. Teams are free to use any agile methodology they choose within each team, but should be willing to commit to whatever methodology they choose. For example, an on-site customer or customer representative is an important aspect of XP, and so XP may not be appropriate without such a representative. If teams are not willing or not able to follow common agile practices, again this methodology may not be the right choice. This thesis does not attempt to argue that agile methods are universally applicable, but rather that they can be a viable option for teams of teams specifically.

3.3. Example Scenario

Given the prerequisites discussed above in Section 3.2, teams can begin the development process using a typical agile system of iterations. In an agile project, typically detailed planning is done one iteration at a time at the beginning of that iteration. At the beginning, each team needs to elicit user stories from the customer and decide what user stories they are going to be implement. At this stage, these teams will basically be working independently, since they will have little idea of what lies in store until they have finished their iteration planning. The only restriction is that the teams (alongside the customer) elicit and choose user stories that fit into their broadly defined project responsibilities. This leaves open the possibility for redundancy between teams. However, this risk of redundancy may vary from being small on some projects with well understood divisions, to being larger on projects with hazy divisions. This risk can be mitigated through communication between the teams, and if judged to be too high, it may be worth re-evaluating whether this approach is useful.

As user stories are drawn up, it is inevitable that dependencies will be identified between a given user story and some aspect of another team's project. Alternatively, developers may discover these dependencies while implementing a user story which they believed had no dependencies. When such dependencies are identified, they will be written up as new user stories for the features needed. At this time, the discoverer of the dependency is acting as a customer creating a user story. This person is committing themselves to act as

customer for this new story for the remainder of the project. These user stories also require that acceptance tests be created, either immediately or during the next iteration (teams will need to make this decision based on their own estimations). It is very important that the acceptance test be well constructed, since it will be used both as the primary means of communicating the story, and also the means to verify that the story is complete. Acceptance tests can be represented in many ways, one of which is the FIT framework evaluated as part of the research of this thesis. Regardless of the format, these acceptance tests are to be delivered to the external team deemed responsible for that feature upon their completion.

An important idea that needs to be clearly understood is what sort of dependency relationships developers could identify. There are two possibilities for dependencies between teams. One scenario is when Jack realizes he will need to provide some interface to the other team, so that they may eventually use his API as a consumer. The other scenario is when Jack believes that he needs himself to be an API consumer of some functionality yet to be provided by another team. When Jack is developing some code, it does the project little good if Jack writes acceptance tests for the interface he is providing, as in the former case (in fact, if Jack has already written an interface, and then writes acceptance tests, he is also not doing test driven design). Jack may be very well aware of what functionality he is providing, but likely has no knowledge or incorrect knowledge of the functionality that other modules are expecting him to provide. It is thus very probable that Jill's module, which uses the module written by Jack, will have some specific need Jack knows nothing about. We can therefore dismiss the first possibility of trying to predict what other teams might need. The more effective arrangement would be for Jill to act as a customer for Jack. Jill will write tests for the functionality that she expects from Jack, and for her to do this *before* Jack writes his actual code. Jill doesn't need to test Jack's entire interface, just the features that she herself will be using. The idea of API consumers writing tests is similar to that already discussed by Newkirk [Newkirk, 64] for doing test first design of third party software. Newkirk asserts that in addition to writing tests before writing code, you should write tests before *using* code written by others. However in this case, the third party software itself may not have been

written yet. You are tailoring the tests as much to your own requirements as to the functionality that will finally be provided. Developers should therefore only create user stories when they identify that they need to use some functionality provided by another team, whether or not that functionality has been provided yet.

When the next iteration is reached by the team, there will be a number of user stories and acceptance tests written for functionality that is expected to be developed externally. Teams will not know if this functionality has been developed yet, since one or more other teams are working in parallel and completing user stories in the order of their choosing. It is possible that these new dependencies have already been fulfilled, but that is irrelevant to the discoverers who do not know it. All such stories and acceptance tests will be sent to the remote team to be included in their next iteration, just as though they were defined by a customer. When the remote team is planning their next iteration, all such developer-created user stories need to be weighed and prioritized just as those user stories describing customer requirements are weighed and prioritized. The team who receives these acceptance tests will essentially treat them as user stories provided by a customer representative. During their next iteration planning meeting, they can then follow the agile procedure of prioritizing and choosing user stories from those provided both from the original customer and from their new customer (from the other team). During this planning meeting it is necessary to communicate with the customer, be it a real customer or a member of another team, in order to determine the relative value of the user story. The acceptance tests are also already available as a means of asynchronous communication with the customer. If the second team chooses to include an acceptance test for dependant functionality, they then need to make that test pass before they can finish the iteration.

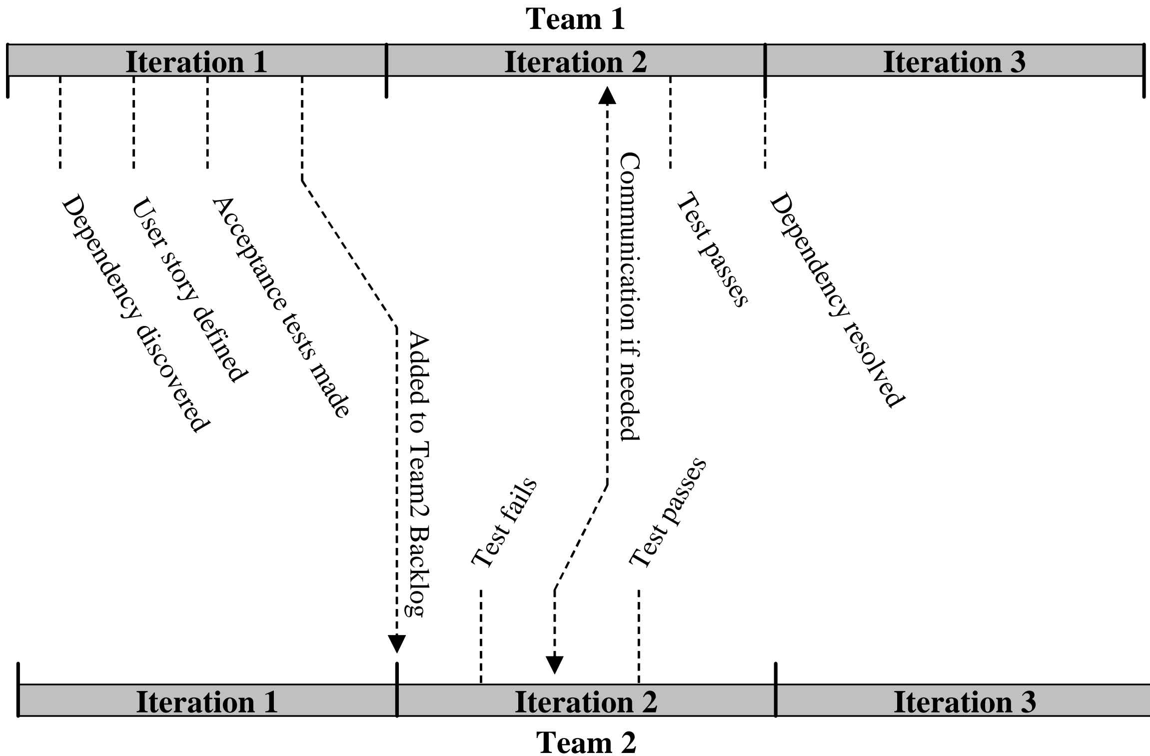


Figure 6. A timeline of two teams working in parallel

Above, Figure 6 shows a timeline of events as they might happen in order to resolve a dependency between two teams. Initially, the first team discovers a dependency in which they need functionality from the second team. To solve this dependency, the first team writes up a user story and creates acceptance tests. The acceptance tests do not test the complete "feature" as envisioned by team 1, but instead test the functionality related to the dependency relationship. Any expectations team 1 has should be tested. This story should be then added to the product backlog of team 2. It should be noted that this necessarily creates a delay in the delivery of team 1's code. However, with complete acceptance tests, team 1 should be able to continue development on that feature assuming that team 2 will provide the expected functionality according to the test. When the second team begins work on their user story, their goal will be to make the acceptance tests pass. Depending on the tests and the user story, additional communication between the teams may be required – this communication is outside the scope of this thesis. Once the acceptance tests pass, both teams should be satisfied. However, if team one finds that

they are not satisfied, they can always modify the acceptance tests or add new acceptance tests in yet another story for team 2.

3.4. Summary

What has been described so far is a variation on test driven development to address issues arising from a team of teams approach. Initially some need drives the creation of a test. This new test will fail, because that need has not yet been satisfied. The test drives the design of the code such that it satisfies the original need, and the test passes. Passing tests serve as a check that the final products will integrate properly and work together as intended. It should be noted that tests can be modified; they are never carved in stone. If a customer decides that their needs have changed, then the acceptance test is changed and delivered as a new or refined user story for the developers providing that functionality. The person who discovers or originates the new “need” is responsible for changing the test, and is also responsible for assuming a customer-like role in interacting with the provider of that test in the future. This cycle of user stories and acceptance tests driving the design of the system and the dependency relationships between teams is not perfect. It is certain that at some point other types of communication may be necessary to clarify requirements and communicate intricacies of how one program interacts with another. However, this system of acceptance test driven design across multiple teams, in which one team can act as a customer for another does overcome the challenges of minimizing up-front design, communicating dependencies during development and ensuring successful project integration.

4. Tool Support

In order to both support and validate the use of test driven design to support agile methods for a team of teams, a series of three tools have been developed. These tools bring together the technologies needed to facilitate the application of the methodology discussed in Chapter Three. Each tool has been created to investigate the potential of using tests to communicate between teams. However, each tool also takes a different approach and has unique characteristics and capabilities. The first tool, COACH-IT investigates using a unit testing framework to test dependency relationships between software components being developed by distributed teams. The second tool, MASE, represents and tests these dependency relationships using FIT acceptance tests. Finally, the third tool, software developed for an industry partner Rally Development, investigates a generalized test framework supporting many testing types and languages.

4.1. COACH-IT

COACH-IT, the Component Oriented Agile Collaborative Handler of Integration and Testing, was an early effort to develop tool support for agile practices in teams of teams using an architecture-centric approach.

4.1.1. Purpose

The initial concept for creating COACH-IT was inspired by Ken Schwaber. His SCRUM methodology [Schwaber, 75] proposes scaling to large teams by coordinating a "Scrum of Scrums". In Scrum, each “Scrum Team” contributes one person, typically the “Scrum Master”, to the Scrum of Scrums meeting. This meeting is used arbitrarily to coordinate the work of multiple Scrum Teams. However, Mr. Schwaber’s methodology does not propose how to distribute the project among the scrum teams, and more specifically how these teams interact when there are dependencies between teams. A weakness of the “Scrum of Scrums” is that the “Scrum Master” cannot be expected to communicate technical specifics of the dependencies between teams. He or she may not have comprehensive technical knowledge, and that there may simply be too much information

to communicate through one person in a reasonable amount of time. This restriction of funneling all communication through one person may mean that teams have their own independent and not necessarily compatible ideas of what is to be developed.

The COACH-IT solution is to explore the concept of lightweight architecture. In this context, lightweight architecture essentially means splitting a single project into several components or "modules" and formalizing their inter-dependencies by unit tests. However, there is in fact an intrinsic contradiction between agile software development and the practice of separating a project into modules in this way. If we need to do up-front design or documentation it may be counter to many of the foundations of agile methods, and may limit team flexibility. The process of creating the modules in COACH-IT therefore needs to be as simple and as minimalist as possible. However, the resulting architecture nonetheless needs to clearly define responsibilities, and the tool needs to communicate those responsibilities to each team. Finally, each module needs to become part of the "whole" and so the tool should assist with that integration process. COACH-IT attempts to address all of these issues.

4.1.2. Design

COACH-IT combines and extends existing continuous integration technologies in order to provide an end-to-end solution for module definition and testing. The sequence executed by COACH-IT is as follows:

1. Users define an architecture using the COACH-IT web application
2. Tests are attached to the architecture to verify dependency relationships; each tests checks if the interface provided by a module fulfills the expectations of a user of that module
3. Multiple source code repositories are monitored for code changes in each module
4. When a change is detected the module and related modules are downloaded
5. The modules are deployed and tests are run to ensure interface compatibility
6. Teams are notified directly of any problems via electronic mail
7. The "health" of the system is available to the teams via a web page

The following diagram shows the interaction of COACH-IT technologies:

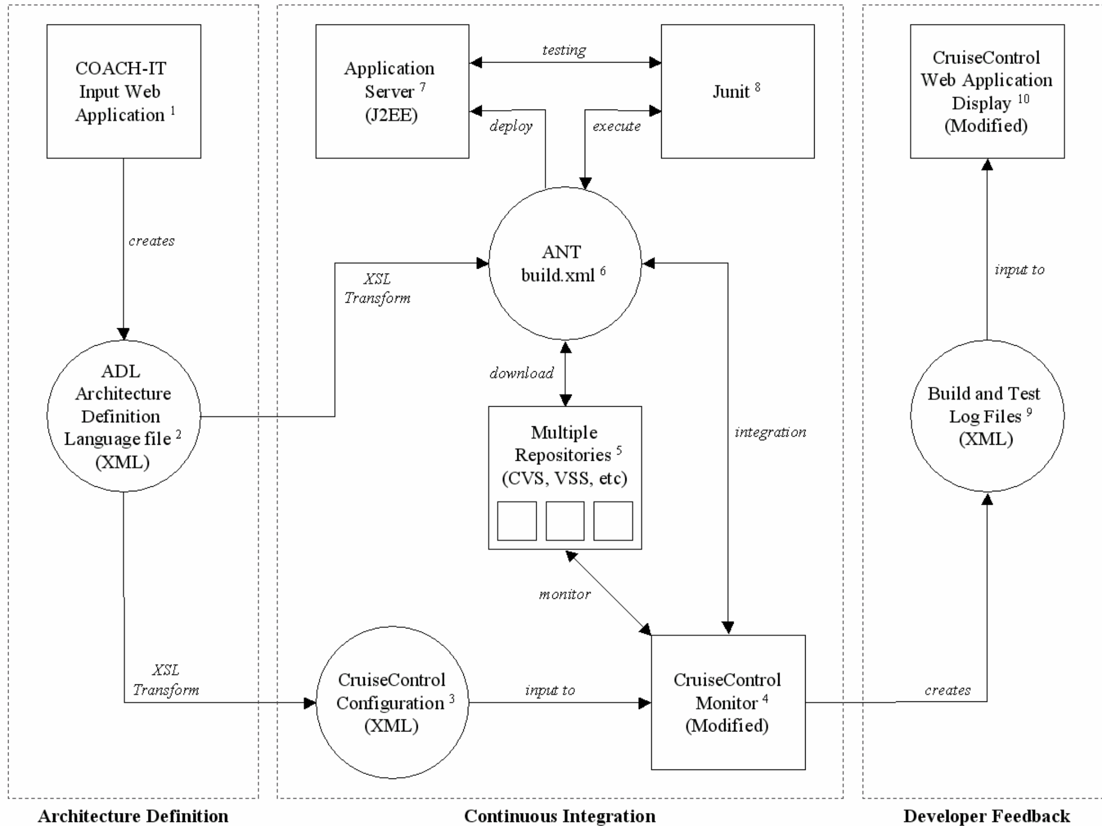


Figure 7. A conceptual drawing of how COACH-IT works

The COACH-IT Input Web Application was designed to assist agile practitioners with managing architecture definitions. Using the COACH-IT tool any developer can define a set of modules and assign JUnit unit tests to the interfaces between those modules (see Figure 8). In this context, a module refers to a piece of the whole software project that has been "divided" according to some divide and conquer strategy. An interface is a link between two modules. Interfaces are unidirectional relationships; "Module X uses Module Y" is a different example from "Module Y uses Module X". The Input Web Application attempts to provide a simple to use, self-documenting user interface. COACH-IT stores these modules and interfaces as an architecture definition. The same application can also load and edit current or previous architecture definitions; architecture definitions in an agile project are likely to change (see Figure 7). Architecture definitions

are recorded using an architecture definition language or ADL (see Section 4.1.3). Although even the minimum necessary ADL can become complex, the web application guides the user through this complexity and lets the developer concentrate on delivering something real.

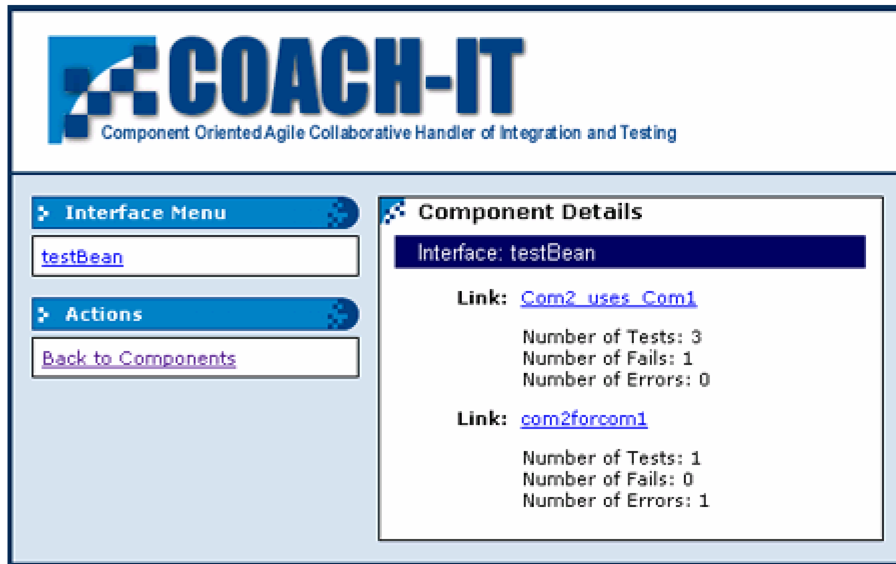


Figure 8. COACH-IT allows the definition of components and links between components

Defined within the ADL file are module names and (optionally) descriptions/annotations, module repository locations, module file locations, module interfaces, module team contact information (e-mail), module relationships (unidirectional), relationship test associations, test repository locations, test file locations and test contact information (e-mail). Only these few items are required as user input to create a simple architecture. An example of the complete ADL file is in Appendix D.

COACH-IT determines when modules are changed using a modified version of the CruiseControl [CruiseControl, 19] continuous integration tool. A modification made to CruiseControl allows it to monitor multiple repositories; each is monitored individually according to custom settings and schedules. Each team is thus able to configure their own repository to suit their unique needs. When COACH-IT detects a changed module it uses ANT [Apache ANT, 6] to perform the integration and testing. ANT is a build tool that

assists with generating deployable software. Each module will have one ANT file generated that will download that module and any other dependant modules, deploy them on the local application server and run the suite(s) of associated unit tests.

COACH-IT is also able to directly notify teams and individual developers via electronic mail. In the event of a test failure, COACH-IT can be configured to notify any and all involved parties, such as the authors responsible for the test, the authors of the involved modules, the developers who last committed changes, the team leaders, or the entire teams of the failed modules.

Each module being managed by COACH-IT has its own logs generated as output. COACH-IT also includes a custom web application based partially on CruiseControl that summarizes the results of tests across the entire architecture. Details and contact information are provided for each test in the event of a failure. There is also a history feature that allows the user to browse through past tests and see when tests passed or failed.

4.1.3. Implementation

COACH-IT is implemented entirely in Java as a J2EE web application, and requires only free, open-source software to run.

The ADL file generated by the COACH-IT web application is stored as XML (eXtensible Markup Language) [XML, 88]. COACH-IT uses XML as its document format so that it can be integrated with existing and future tools that use XML as input and output. The core technologies underlying COACH-IT (ANT and CruiseControl) both rely heavily on XML. However, the XML expected as input by each of these technologies is different. Moreover, the XML produced as output by both CruiseControl and ANT are different. COACH-IT therefore applies XSL [XSL, 89] (eXtensible Stylesheet Language) to map one XML format to another. COACH-IT executes the following steps:

1. Defined architectures are saved in a COACH-IT ADL file using XML

2. Architectures are mapped to CruiseControl configuration files using XSL
3. Architectures are mapped to ANT build files using XSL
4. CruiseControl loads the configuration file and monitors the repositories
5. CruiseControl executes the ANT script(s) when changes are detected
6. CruiseControl generated output in an XML file
7. COACH-IT can displays the generated output

Note that from the steps above, much of the actual labor is performed by either ANT or CruiseControl. A sample COACH-IT ADL and XML Schema are available in Appendix D. Also in Appendix D are a sample CruiseControl configuration file, generated ANT script, and generated output.

Input to the CruiseControl monitor is also via an XML file generated from the ADL file using an XSL script. A sample CruiseControl input file is available in Appendix D. The CruiseControl configuration file follows a standard CruiseControl format but allows multiple project definitions (one for each module). Output from the CruiseControl monitor is also in a standard CruiseControl XML format; output logs generated by COACH-IT are compatible with the standard CruiseControl web application as well.

The ANT file used by COACH-IT is likewise generated via the ADL file using XSL. Because these ANT files are generated using XSL scripts it is simple to add additional ANT tasks if required.

4.1.4. Anecdotal Evidence

A paper on the COACH-IT approach was accepted to XP/Agile Universe 2003 [Read, 71] and was demoed at that conference in order to gather feedback from experts in this field. Sharing COACH-IT with industry and academic agile practitioners, I gathered three major criticisms of the COACH-IT tool and concept.

The first major criticism of COACH-IT is that defining architecture explicitly is too cumbersome. Although COACH-IT makes it fairly intuitive to define a series of modules,

it is nonetheless time consuming defining modules and entering many properties for each module, then defining the participants and properties for each dependency. There is also a danger to defining these modules in a structured form such as the one provided by COACH-IT: once the data is entered, users are reluctant to change or update the data. One reason for this reluctance is that users tend to view the architecture as a plan to follow and adapt their code to fit the architecture rather than vice versa. This negates some of the "agility" of agile teams and in fact also negates the usefulness of the tool. Some users at the conference also pointed out that defining modules is redundant since the dependencies themselves imply the existence of the modules. For instance, defining dependency "A requires B" implicitly tells us that "A" and "B" exist. Modules with no dependencies do not really require communication with other teams, and therefore those teams will have little use for the tool. Moreover, in most cases the "dependency" relationship between modules is only used as a place to attach tests. Although this was the intention, it led me to begin thinking that perhaps the tests alone were the minimalist representation of the relationship between modules, and in fact the only necessary item in the architecture.

A second criticism of COACH-IT is that JUnit tests are not suitable for testing module dependencies. Writing JUnit tests requires intimate knowledge of how the code works. For instance, you need to know method names, method parameters, and class names. Writing these tests ahead of time for dependency relationships of components that do not exist turns out to be harder than it would seem. In order to come up with this specific interface in advance, some design needs to be done. Although this holds true for any kind of TDD, it should be the person who implements the module who defines the technical structure of that module. JUnit is also made to test at a fairly low level of granularity. Rather than testing general features, it would test some specific technical aspect.

It follows that Continuous Integration is not a suitable system on which to base this tool. Continuous Integration is meant to be used when tests that have been checked in are all supposed to pass, and to identify when one fails. It is a tool for regression testing, to make sure that no bugs are introduced into the build. In the case of COACH-IT, the

majority of tests in the repository at any one time are supposed to fail because there is not yet a module satisfying those tests. One at a time they may begin to pass at later stages of the project, but for most of the project life-cycle there will be failing acceptance tests. It is useless to notify stakeholders by email every time someone commits code to a source repository and acceptance tests fail – in early project phases, all tests will fail all the time.

Finally, the agile community suggested that additional communication channels would be useful between the teams. Although having a "minimalist" architecture representation to share is useful, it is also useful to be able to contact other people, or to be able to make additional notes and annotations about the project in a common knowledge sharing environment.

4.1.5. Summary

The COACH-IT tool does provide support for agile teams of teams. It delivers a web based solution for defining a lightweight architecture explicitly. It does allow the definition of modules and dependencies between modules. It also helps overcome integration difficulties by enforcing dependencies using unit tests. However, anecdotal feedback about COACH-IT also suggests that a simpler, non-continuous-integration based solution might be more ideal.

4.2. MASE

In order to address the deficits and criticisms of COACH-IT, a new tool was developed as part of the MASE project [MASE, 56]. MASE is a collaborative web-based environment and tool for agile teams to which I have contributed but of which I am not the sole author. MASE includes a wiki for knowledge management and collaboration as well as some agile specific tools for story card management, estimation, and prioritization, iteration planning, progress tracking, notification, and pair programming. However, I will use MASE as shorthand notation to refer to my own contributions to the tool henceforth.

4.2.1. Purpose

This new tool is not simply a new version of COACH-IT but a new application to support teams of teams using agile methods, based on refined ideas and using different technologies. The purpose of MASE is to address the criticisms of COACH-IT. Specifically, these criticisms were:

1. Working with an explicitly represented architecture is cumbersome
2. Unit tests are unsuitable for representing and validating dependencies
3. Continuous integration is not a suitable basis for executing these tests
4. Insufficient support for other communication and knowledge sharing

MASE supports and demonstrates the practicality of working with an implicitly represented architecture rather than an explicit one. MASE also supports using FIT acceptance tests rather than unit tests. A hypothesis that MASE can help test is if acceptance tests are rich enough to describe actual functional requirements. MASE runs tests on-demand and supports additional communication and knowledge sharing.

4.2.2. Design

Basing this tool on MASE (see Figure 9) means there are more features available to the end user, and allows the features of MASE to solve some of the criticisms of COACH-IT. MASE is a wiki: a website which allows readers to interactively add and update its own text. The wiki should aid in knowledge sharing between teams. One extension made to the wiki is user notification. If you have enabled notification on a given wiki page, you will receive email any time that page is modified. This helps teams keep track of tests they may have authored or that they may be executing against their code base. Teams can self-organize sets of wiki pages to share knowledge about their work, and can also use the project planning tools to create online user stories and iteration plans. However, studying the effect of a wiki is outside the scope of this thesis (see [Chau, 16]). MASE also provides hooks to Microsoft NetMeeting to allow users to chat, audio/video conference, or use application sharing.

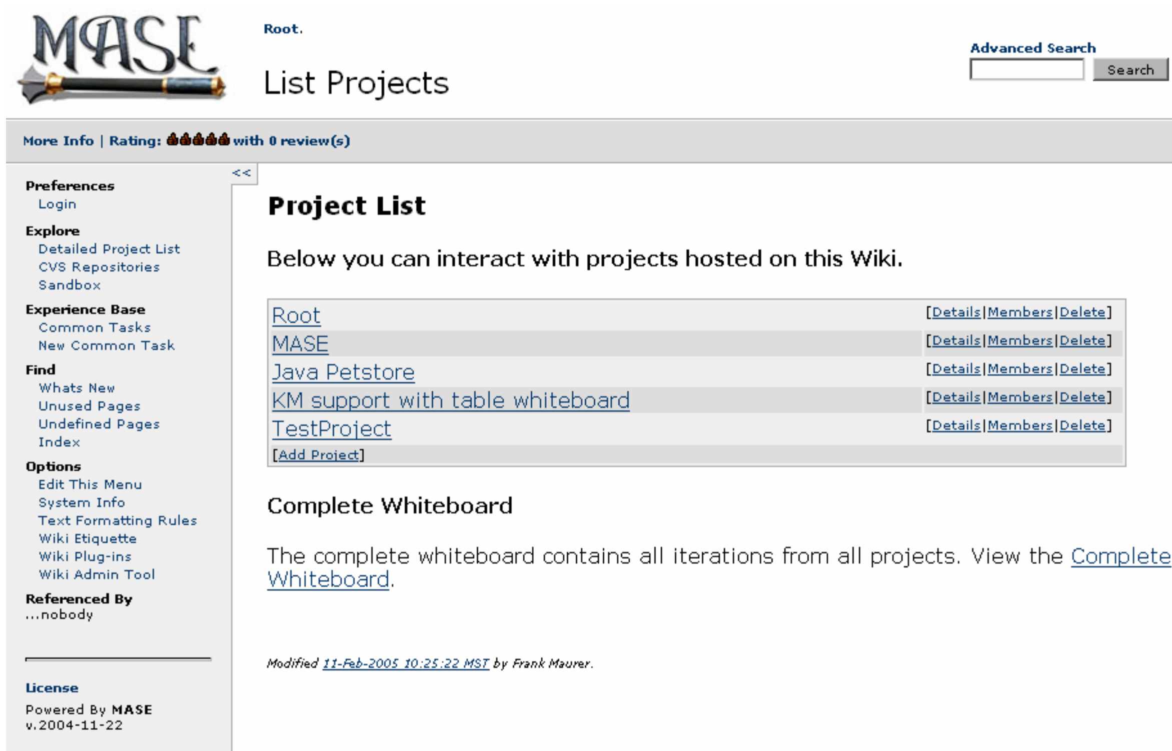


Figure 9. The MASE web-based wiki tool for supporting agile projects

A major difference between COACH-IT and MASE is a move from a continuous-integration based approach to an on-demand approach (see Figure 11). As discussed previously, monitoring a repository for changes and then executing tests is not practical, since we may be writing the tests well in advance of the actual code. Tests are not necessarily going to be satisfied immediately, and so it does no good to run them repeatedly when we know they are going to fail. Moreover, it is useless and annoying to have a system such as CruiseControl sending failure notifications for large numbers of failing tests when we have simply not chosen to work on their user stories yet. It makes more sense to simply run the tests on-demand. By putting acceptance tests into test suites, they can be executed in groups or all at a time.

Another difference is that MASE does not attempt to use a lightweight architecture of any kind. COACH-IT was based around the concept of defining a minimalist representation

of system modules and dependencies between those modules. Users could then attach tests to the dependency link in order to communicate and verify that relationship. However, it is not necessary to define the components, merely the links. Each link implicitly names two components, one at each end. The complexity can further be reduced by eliminating the idea of link; each unique test implies the existence of a link between components (see Figure 10). This holds true so long as multiple tests of interactions between the same two components can be distinguished. However, this can be accomplished by naming the tests descriptively.

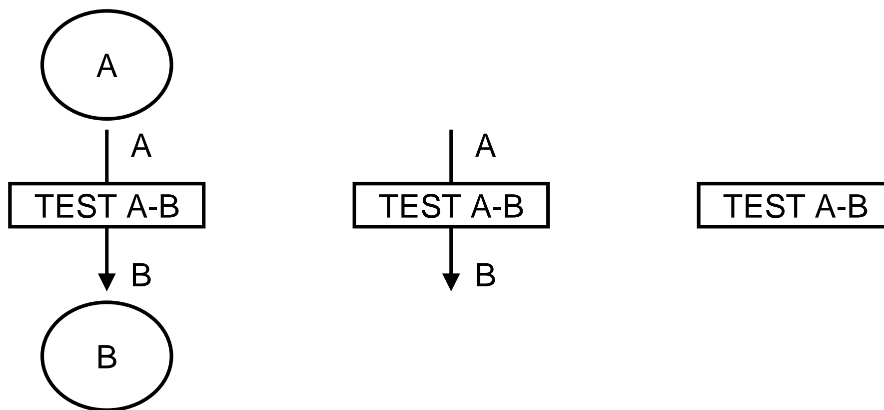


Figure 10. Simplifying how component dependencies are represented

Using MASE we can create sets of tests such that each set of tests represents a link between components. Because MASE uses a wiki, it is easy to annotate or comment tests and test suites to describe additional information about what the tests do or what modules or projects they involve. The advantage of this approach is that it takes less time for a team to define dependencies, and the architecture of the system evolves as the tests evolve, rather than being defined explicitly in a traditional document-centric way.

In MASE the dependency tests are defined using FIT acceptance tests rather than as JUnit tests. A unit test makes the programmer satisfied that the software does what the programmer thinks it does. Although unit tests do provide input parameters and expect certain output, they also require knowledge of the inner workings of the code being tested such as method names, method parameters, and class names. Moreover, they do not necessarily describe acceptance criteria, but rather exercise certain pathways or execution

scenarios. When doing test driven development with unit tests, how you develop your unit test may influence how you develop the solution; it is not the intention in this tool to influence how the solution is developed, only to communicate what the requirements of that solution are. Acceptance tests on the other hand do exactly what is intended. An acceptance test describes the acceptance criteria for a feature, and automated acceptance tests can be executed to ensure that the given criteria are met. Although simple, FIT is powerful because it separates the input and output expectations from any source code or executable entity. The tool can therefore support communicating and representing of functional requirements using FIT tables, and later users can develop FIT fixtures to verify those requirements against the product.

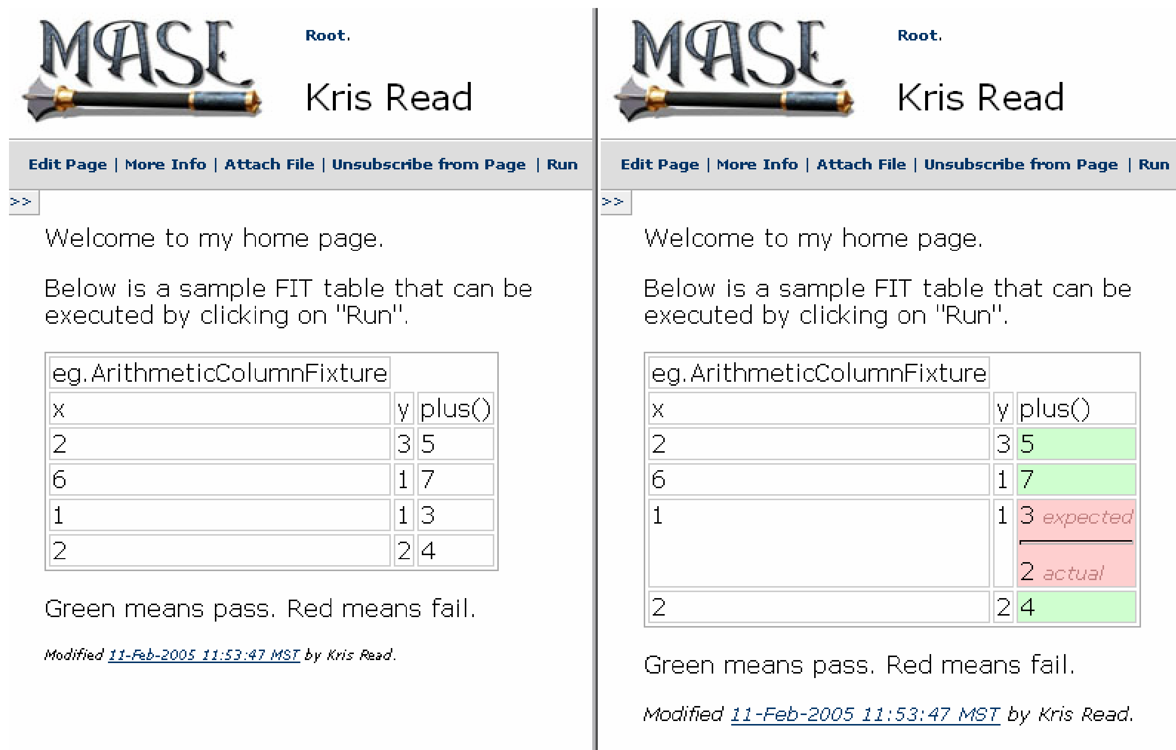


Figure 11. Tests can be run on-demand in the MASE tool by clicking “Run” on the toolbar

The sequence executed using MASE is as follows:

1. Users define FIT tests using the MASE wiki
2. Test suites (FIT tests referencing other FIT tests) can be made to implicitly define the architecture of the system
3. CVS Repositories can be browsed and viewed through MASE in order to identify and mark relevant executable code modules
4. Executable code is explicitly loaded from a CVS location
5. Tests are executed on-demand individually or as suites
6. The modules are deployed on the MASE local server and tests are run to ensure interface compatibility
7. Results are viewed as green or red FIT tables within the wiki page

The MASE wiki editor is the means by which FIT tables are defined. These tables need to be provided using MASE wiki syntax, which is a fairly easy-to-use markup language. The aim of wiki markup is a very user-friendly way to format text that does not require any kind of programming background. Tables are created using the "|" character in a wiki table.

```
|eg.ArithmeticColumnFixture|
| x | y | plus() |
| 2 | 3 | 5  |
| 5 | 5 | 10 |
```

The arrangement of columns and rows in a FIT table are flexible depending on what fixture is used. These FIT tests are stored as regular wiki pages by MASE, and so it is possible to make links to other wiki pages or make annotations on the test page itself.

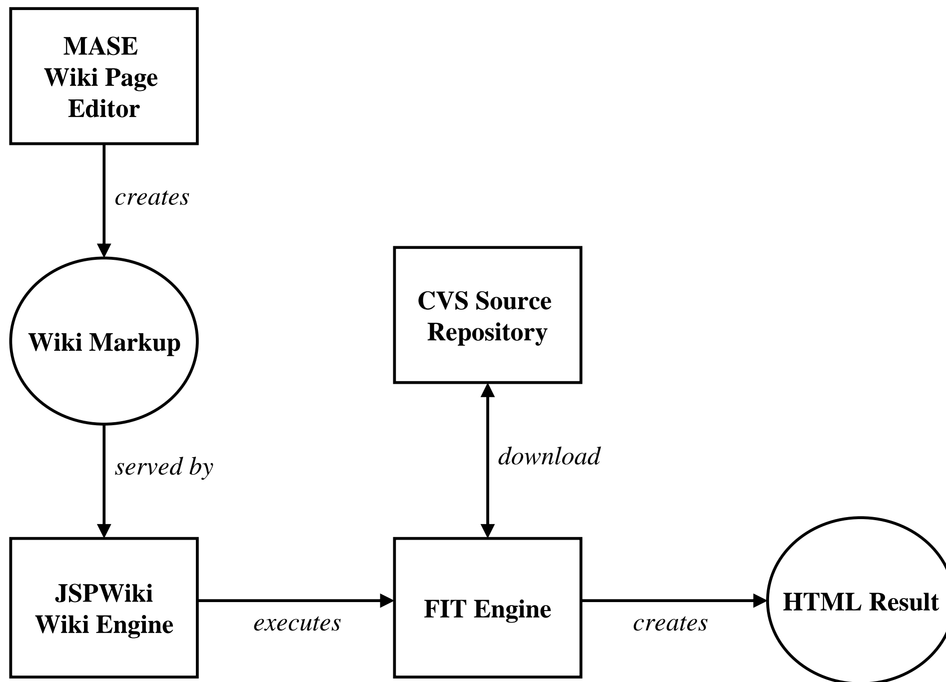


Figure 12. Overview of how MASE functions conceptually

MASE works by translating wiki pages on-demand from wiki markup into an executable FIT test. MASE passes the FIT test or test suite to the FIT framework. FIT parses tabular data and interprets it according to templates called fixtures. Fixtures can be executed using the FIT engine, which then parses the tabular input, executes the project code against that data and outputs green cells when criteria are met or red cells when criteria fail. MASE then displays those test results to the user (see Figure 12).

4.2.3. Implementation

Implementation for MASE is using J2EE, and runs on open-source software. Since MASE was an already existing tool, implementation was less than for COACH-IT.

Wiki pages are processed by the wiki engine. Wiki functionality was not developed for this thesis and therefore will not be discussed here. Once the wiki engine has processed the wiki content, the resulting HTML is passed to the FIT engine. The tool is currently using the Java FIT engine provided by Ward Cunningham [FIT, 25]. FIT processes the HTML tables on the page and feeds the input data through FIT fixtures that run

production code. The FIT fixtures are java classes that handle mapping the tabular data to actual methods, and need to be provided by the implementer of the test. The fixtures themselves as well as the production code come from a CVS repository defined within MASE. CVS repository connections are made in MASE using jCVS, an open-source CVS application [jCVS, 39]. MASE dynamically loads any required files that have been marked as executable and deployable archives in the repository, and deploys those files on its own application server if needed. MASE displays the output from FIT directly, which includes pass or fail information as well as any exceptions or errors that may have occurred while executing the tests.

4.2.4. Anecdotal Evidence

Features of MASE were demonstrated at a product booth at XP Agile Universe in 2004. It was also presented to CAMUG (the Calgary Agile Methods User Group) in 2004. MASE has been available online for public demonstrations since that time, and students at the University of Calgary and SAIT have had internet access to a MASE server to use several of their courses. Feedback about MASE has been very positive. MASE does a lot more than described in this thesis, since it is the work of many graduate students over many years. However, feedback about the FIT integration and specifically about the testing features of MASE has also been positive. The most notable feedback came from an industrial software provider, Rally Development. Rally makes a commercial tool that is essentially a competitor to MASE; the purpose of Rally's tool is to support agile teams. Rally asked for a detailed demonstration and suggested that they would like to investigate integrating the testing features of MASE into their own commercial tool.

4.2.5. Summary

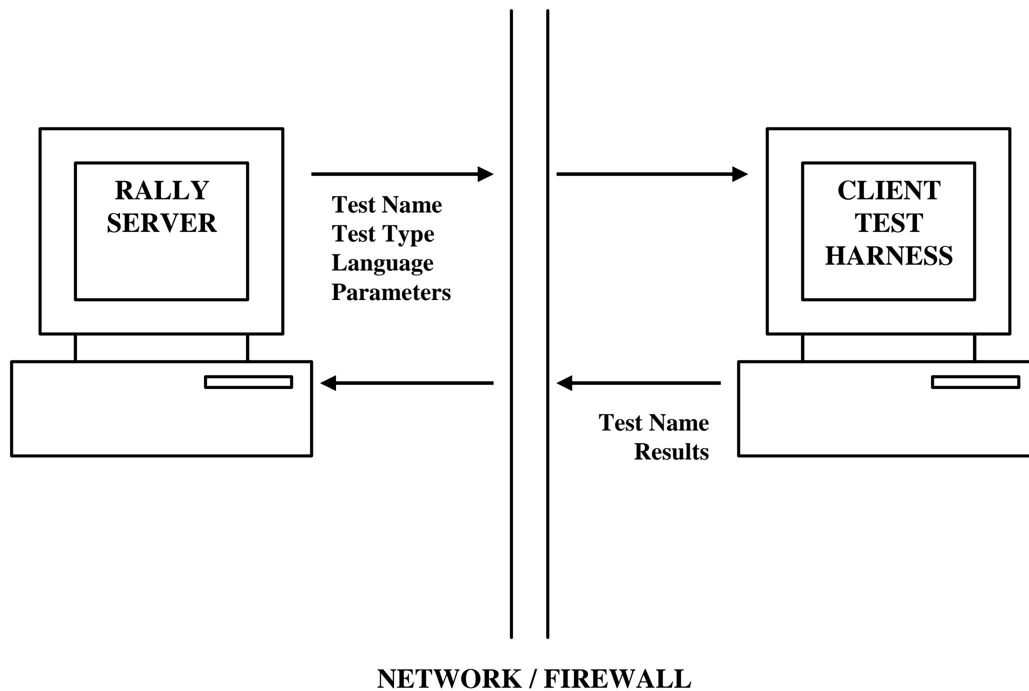
The final result is a simpler and more elegant system than COACH-IT. The MASE wiki handles the input of tests and their prerequisites, then executes FIT, and finally displays the results without any need for external programs like CruiseControl or ANT. This tool can be used to support planning, communication and integration by an agile team of teams on a divide and conquer project.

4.3. Rally Testing Tool

At the XP/Agile Universe 2004 conference, Rally Development [Rally, 70] expressed interest in the approach and implementation used in the MASE tool. However, Rally also raised some questions about how this approach could be integrated with their commercial tool. Therefore a third tool was developed to show that indeed the approach is compatible with the business model chosen by Rally.

4.3.1. Purpose

There are several conceptual differences between this tool and MASE or COACH-IT which deserve examination. One major difference is that this tool assumes a hosted solution for project planning. With such a solution, the organization of the tests can be done on a remote server, yet tool users need to be able to execute their own tests at their local site behind their own firewall. This implies that unlike MASE, where the server has information about the tests and executes them as well, the Rally system has information about the tests but gives the clients the freedom to execute their own tests in an uncontrolled environment. This difference creates several specific challenges. First, there exists demand for a client-side environment that can execute tests. Secondly, tests need to be abstracted so that the server can represent a general test while the client runs a specific instance of a test. Thirdly, the server must communicate with the client (see Figure 13). The move from a purely controlled server environment to a split client/server environment also highlights a major security flaw in the MASE design. Executing any test on the server side is a security problem, since the automated tests could contain arbitrary and potentially harmful code that might disrupt the server machine. If the development team owns the server, then this is only a problem if they do not trust their developers. For hosted solutions (like Rally's), or for virtual companies in which multiple organizations collaborate, a single server is not appropriate. Running the production code and acceptance tests on the server means that the customer needs to relinquish possession and control over their intellectual property. For most companies, code under development is kept in-house and under-wraps rather than on a remote server (especially a server owned and operated by a third party).



NETWORK / FIREWALL

Figure 13. Adaptations for Rally require that tests be run on a remote server

Another important issue identified with the help of Rally Development is that the tool should be programming language independent. Because COACH-IT and MASE were prototypes, the language supported for the tests was limited to just Java. However, this is not acceptable in a business environment where clients may already be using another language or possibly several languages. Therefore, a test harness needs to be able to execute arbitrary types of acceptance tests on code written in any language. This again creates several specific challenges that need to be overcome. First, the server needs to represent tests and test results in an abstract way, but at the same time the server needs to contain data for each test identifying what language the test uses and how that test can be executed. Second, the client needs to be able to give the test client tests of their own choosing, and the test client needs to be able to execute them. Third, the client needs to report the results of different tests back in a consistent way.

Yet another issue is that tool users may not be comfortable using only FIT acceptance tests. Although this thesis contains evidence suggesting that FIT is a good match for

specifying functional requirements (see Chapter 5), users of the Rally tool may have a broader use for testing and therefore a goal is to be able to run all kinds of tests, from unit tests to acceptance tests, or any other kind of test deemed useful. The direct impact of this goal on the tool is that the client side harness needs to support some system of dynamically loading different modules to run different types of tests.

4.3.2. Design

The MASE system is the basis of the tool for Rally. The first adaptation made is the addition of an object on the server that represented a generic "test". Users of MASE can create "test" objects and even suites of tests, and for each test they can provide a name, test type, and parameters or resources needed by that test. At execution time, MASE can provide this test data to the remote client so that the client knows what test to run. MASE has also been modified to have a "test result" object, so that the client can return generic information such as whether the test passed, failed, or what if any errors might have occurred. The server is therefore responsible for managing tests, test suites, and the results of tests, but does not know how to run a specific test. The addition of tests, test suites and test results require the addition of pages to MASE to add, view, update and delete these entities.

Experience on COACH-IT helped to create the client test-harness that can run any type of test. The tool chosen to overcome the problem of arbitrary test types and different programming languages is ANT. ANT supports many kinds of tests already, and also can be extended with new tasks for new test types. Moreover, ANT also allows the tool to run tests from any programming language, against code in any programming language.

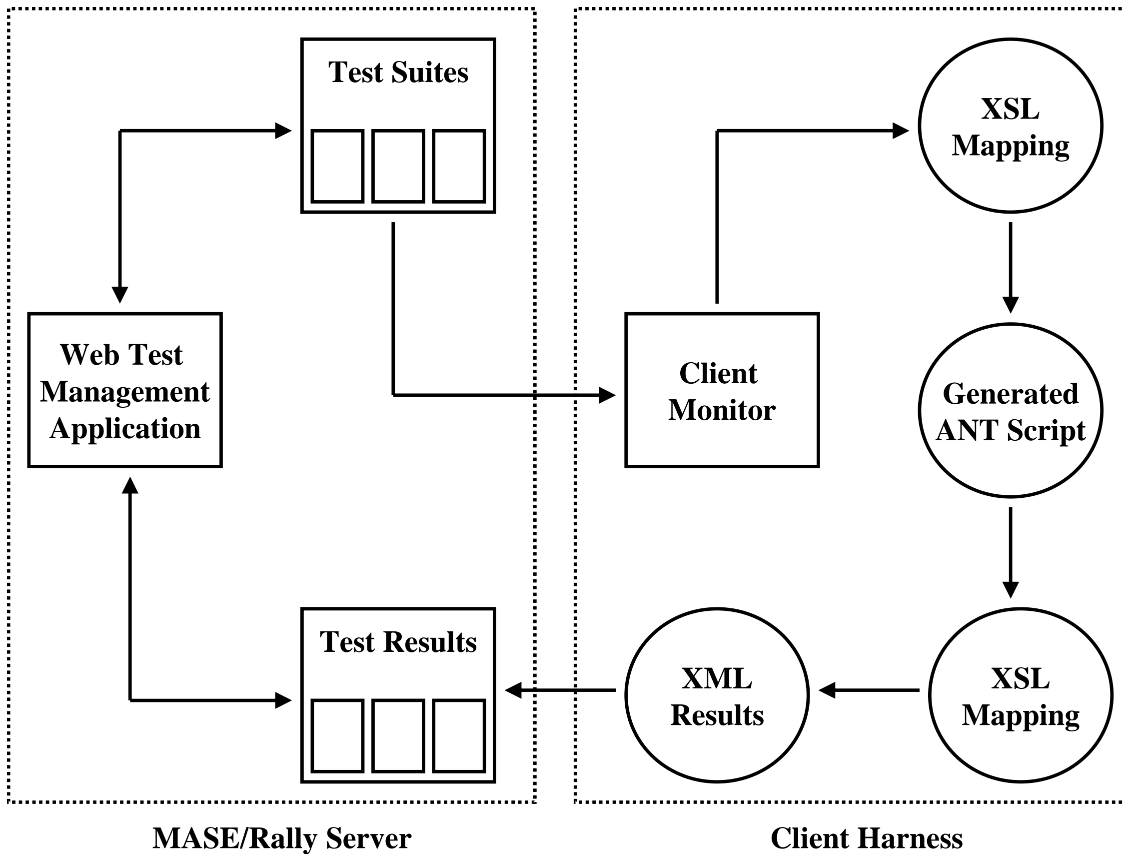


Figure 14. Overview of components used by both server and client programs

Like COACH-IT, the client now constantly watches for new input (see Figure 14 for an overview). However, instead of watching a source code repository, it is waiting for on-demand requests from the MASE server. When a request arrives, the client generates an ANT script to execute the tests and output the results. Those results are then returned to the MASE server.

4.3.3. Implementation

Extending the MASE server required three major changes. The first change required the addition of persistent entities to store test cases, test suites and test results. These are implemented as Enterprise Java Beans following the example of other entities in the MASE project. The second change required the addition of add, view, update and delete user interfaces for each of the entities, as already mentioned. Thirdly, the MASE server needed to be able to send messages to the client. These messages are sent as HTTP

POST requests so that internet or company firewalls will not provide a hindrance to communication.

On the client an ANT script is constantly monitoring for requests from the server. When a request arrives, the “monitor” loads an XSL mapping that converts the generic test data into a second executable ANT script, one that is capable of running the tests and outputting the results. The XSL script is also responsible for formatting the test results into a generic XML form (see Appendix D for a schema). Each time a test is run, a unique ANT test script is created and executed on-the-fly. After the tests are executed, the parent “monitor” script sends the data back to MASE/Rally using HTTP.

This elegance of this system comes from the interaction of XSL technology and the ANT tool. Using XSL the end user can supply many pre-defined maps to create the tiny scripts that will run each kind of test on code from each programming language. As an example, the system includes XSL scripts to run JUnit tests. Using this tool, users would not need to disclose what type of tests they are running or how their testing system worked – they can keep the details of how tests are run to themselves, but organize their tests and store the results of those tests on a secure, remote server.

A limitation to this system is that the users who are executing the tests need to be able to author XSL scripts to map the generic test structure sent by MASE into an executable ANT script. Over time, more and more examples of these scripts could be collected and distributed with the tool, so that new users do not have to expend so much effort on startup.

4.3.4. Anecdotal Evidence

Because the Rally tool was developed with and for Rally Development, it has not seen as much public exposure as either MASE or COACH-IT. However, feedback from Rally indicated that they were pleased overall with the adaptations made in this third tool and with their investment in this research. It remains to be seen if this technology is released

as part of a future version of Rally's product, and whether there is positive feedback from Rally's customer base.

4.3.5. Summary

The tool for Rally builds upon the support offered by MASE through enhanced client control of their code-base, improved security, and greater flexibility in terms of what types of tests and programming languages are supported. These changes come at a price: not having the tests on the server side is an inconvenience for the end user. In the MASE tool users can quickly add and update their FIT tests within a wiki editor. However, in the Rally tool clients are not given any support for creating or managing the tests.

4.4. Summary

All three tools address important issues about how agile teams of teams can communicate within a divide and conquer scenario. COACH-IT focuses on resolving integration issues through automated tests, and enables communication of dependencies between teams using unit tests to represent functional requirements. MASE uses acceptance tests rather than unit tests for communication, and represents dependencies in an implicit architecture of tests rather than using an explicit XML architecture. In the Rally tool, the system was distributed between client and server environments, and supports the use of any type of test on any programming platform. All three tools assist teams with storing and communicating functional requirements in an agile way using automated testing, and are an important part of showing the practicality of this approach.

5. Empirical Evaluation

Ideally, it would be desirable to directly test the tools and ideas developed in this thesis in a long term, industrial divide and conquer project. However, it is impractical to perform such a study due to limitations of time (for a M.Sc. thesis) and subject availability. Therefore, several smaller yet pre-requisite questions have been examined over a number of experimental and observational studies: Suitability of FIT Acceptance Tests for Specifying Functional Requirements, Examining Usage Patterns of the FIT Acceptance Testing Framework, and Student Experiences with Executable Acceptance Testing. These studies offer strong indicators about the viability of the concepts and tools proposed in this thesis. They are not intended to completely validate the proposed solution for supporting agile teams of teams via test driven design. They are also not intended to validate the tools developed as part of this thesis. Data gathered will support or refute the use of FIT acceptance tests to represent functional requirements, and thereby more broadly support or refute the use of the proposed methodology and supporting tools for agile software development across a team of teams.

For a more complete empirical evaluation, many more questions would need to be tackled (see Figure 15). A study below examines the understandability of FIT for specifying functional requirements. This is only a pre-requisite to the real question of whether FIT tests are understandable as representations of dependencies between teams. Likewise, in this thesis the question of whether teams can author FIT tests to represent dependencies between teams is not answered. Moreover, in this thesis FIT is used as an example testing framework, however it would be preferable to determine if other frameworks would be suitable or more suitable. Lastly, this thesis is missing any evaluation of the tools themselves.

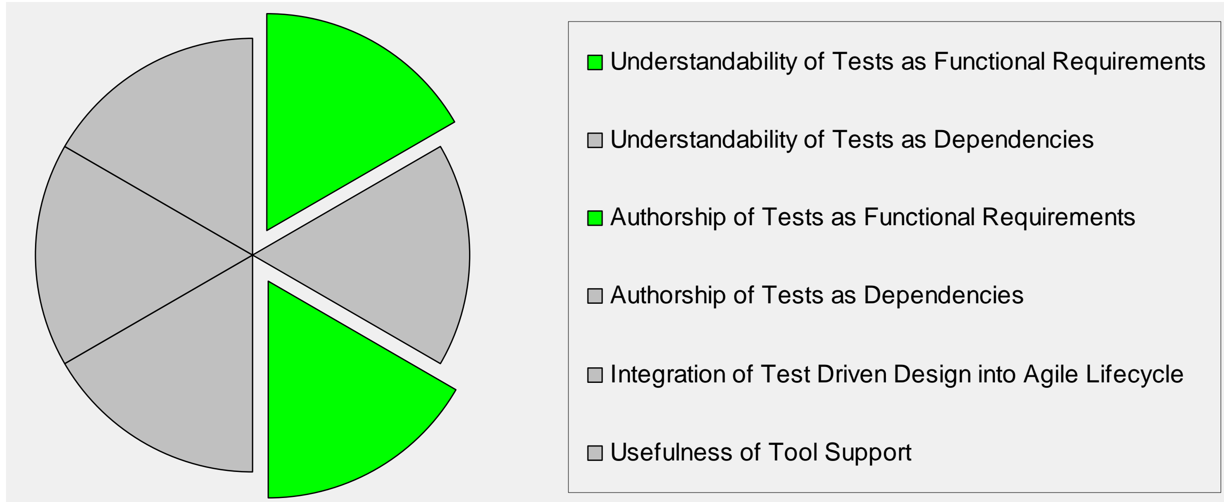


Figure 15. Empirical research completed versus outstanding

5.1. Suitability of FIT Acceptance Tests for Specifying Functional Requirements

5.1.1. Purpose

The first study determines the suitability of FIT user acceptance tests for specifying functional requirements. In order to suggest that dependencies between one project and another can be represented and communicated as acceptance tests, we need some knowledge of whether acceptance tests can be used to communicate those dependencies. Many dependencies between teams should fall into the category of functional requirements, since they will have concrete input and output parameters. Some non-functional requirements could also be expressed through FIT acceptance tests by expressing them in terms of functional requirements. For example, performance can be expressed in terms of expected or maximum execution times. A goal is to find out if FIT acceptance tests can communicate functional requirements to developers who do not already have an understanding of those requirements or a definite idea of how the final component will be consumed.

5.1.2. Subjects and Sampling

Students of computer science programs from the University of Calgary and the Southern Alberta Institute of Technology (SAIT) participated in the experiment. All individuals had previous knowledge about programming and testing (based on course pre-requisites), 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, 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, at SAIT. The material from both courses was presented consistently by the same instructor (Grigori Melnik) in approximately the same time frame. This experiment spans 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 teams had 3 members each; however, they did not have access to additional help outside of classroom lectures. SAIT teams had fewer members so that we would have an equal number of teams at each location. In total, there were 12 teams and a total of 42 students.

5.1.3. Instrument

A project was conceived for students 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 experiment, subjects were required to work on only a partial implementation concentrating on the submission and review tasks. 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 (Fig. 16).
3. A suite of FIT tests (Fig. 17)

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 16. Assignment specification snapshot

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 17. A partial FIT test suite from the student experiment

Requirements in the FIT Test Suite of this experiment can be described generally as sorting and filtering tasks for a sample XML repository. The provided suite (see Appendix E) 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. 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 was set up and all students could utilize and contribute to this knowledge repository. An iteration planning tool and source code management system were available to all teams if desired. In this experiment I hypothesized that:

1. FIT acceptance tests describe a customer requirement such that a developer can implement the feature(s) for that requirement.
2. Developers with no previous FIT experience will quickly be able to learn how to use FIT given the time provided.
3. 100% of developers will create code that passes 100% of customer provided tests.

4. More than 50% of the requirements for which no tests were given (that we judged to be needed to complete the DRS) will be implemented and tested.
5. 100% of implemented requirements will have corresponding FIT tests.

5.1.4. Results

Our first hypothesis was that FIT acceptance tests describe a customer requirement such that a developer can implement the feature(s) for that requirement. This 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 consider the 10 teams who actually made attempts to implement the required FIT tests (SD=24%) (Table 1). Informal student feedback about the practicality of FIT acceptance tests to define functional requirements also supports the first and second hypotheses. Students generally commented that the FIT tests were an acceptable form of assignment specification. 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%

Table 1. Customer test statistics by teams

Seventy-three percent (73%) of all groups managed to satisfy 100% of customer requirements. Although this refutes the second hypothesis, 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 (Table 2).

	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%

Table 2. Percentage of attempted requirements

No teams were able to implement and test at least 50% of the additional features expected (Table 3). Those requirements defined loosely in prose but given no initial FIT tests were largely neglected both in terms of implementation and test coverage. This disproves my 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 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*
University	1	19	100%	208	100%	49%	32%	32%	67%
University	2	157	100%	5225	100%	403%	795%	26%	100%
University	3	0	0%	0	0%	0%	0%	0%	0%
University	4	116	100%	2218	100%	297%	338%	32%	75%
University	5	9	100%	99	100%	23%	15%	16%	100%
University	6	41	93%	616	95%	105%	94%	37%	100%
SAIT	1	0	0%	0	0%	0%	0%	0%	80%
SAIT	2	0	0%	0	0%	0%	0%	0%	100%
SAIT	4	56	100%	1085	100%	144%	165%	11%	66%
SAIT	5	0	0%	0	0%	0%	0%	0%	100%
SAIT	6	5	100%	64	100%	13%	10%	5%	100%

Table 3. Additional features and tests statistics

Customers do not always consider exceptional cases when designing acceptance tests, and therefore acceptance tests must be evaluated for completeness. Even in this experiment's 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 the 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.

The raw results for this study are available in Appendix F.

5.1.5. Interpretation

Our hypotheses (1 and 2) that FIT tests describing customer requirements can be easily understood and implemented by a developer with little background on this framework were substantiated by the evidence gathered in this experiment. Considering the short period of time allotted, it can be concluded 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, my 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 my 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. Given more time and advice I believe that a higher rate of fulfilled tests can be achieved. The lack of tests for new features added by teams may, in my opinion, be accredited 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. At the very least, the 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 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. This leads to a belief 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). Thus, it may be that non-developers can NOT provide comprehensive tests (one of the goals of FIT). 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.

5.2. Examining Usage Patterns of the FIT Acceptance Testing Framework

5.2.1. Purpose

It is not only desirable to know if FIT acceptance tests have the capability of communicating functional requirements, but also to understand how FIT acceptance tests are interpreted and used by teams. The second study looks for patterns in the use of FIT and FitNesse, in order to identify if there are commonalities between how teams use and understand both acceptance testing and commonly used tools for acceptance testing. A better understanding of how FIT acceptance testing is undertaken and how FIT acceptance tests can drive design will help determine if using these tests as a communication medium is a good fit for agile teams. The goal was to identify usage patterns and gather information that may lead to better understand the strengths and weaknesses of the tool. Furthermore, examining these patterns can lead to

recommendations on how FIT can best be used in practice, as well as for future development of FIT and related technologies.

5.2.2. Subjects and Sampling

Data was gathered from two different projects in two different educational institutions over four months. As in the first study, students were enrolled in computer science programs from the University of Calgary and the Southern Alberta Institute of Technology (SAIT). All individuals were again knowledgeable about programming and testing (based on course pre-requisites), however, no individuals had any advance knowledge of FIT or FitNesse (based on a verbal poll). The material from both courses was presented by the same instructor in an approximately consistent way over a similar time period. Data was not gathered on an individual basis but rather collected from observation of teams. Four teams at the University of Calgary and five teams at SAIT were involved in the sample group.

5.2.3. Instrument

In this experiment, the projects attempted by the teams differed between institutions. Students at SAIT developed an interactive game, and students at the University of Calgary worked on 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.

At the University of Calgary FIT was introduced immediately, and at SAIT FIT was introduced in the third iteration (half way through the semester). After FIT was introduced, developers 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.

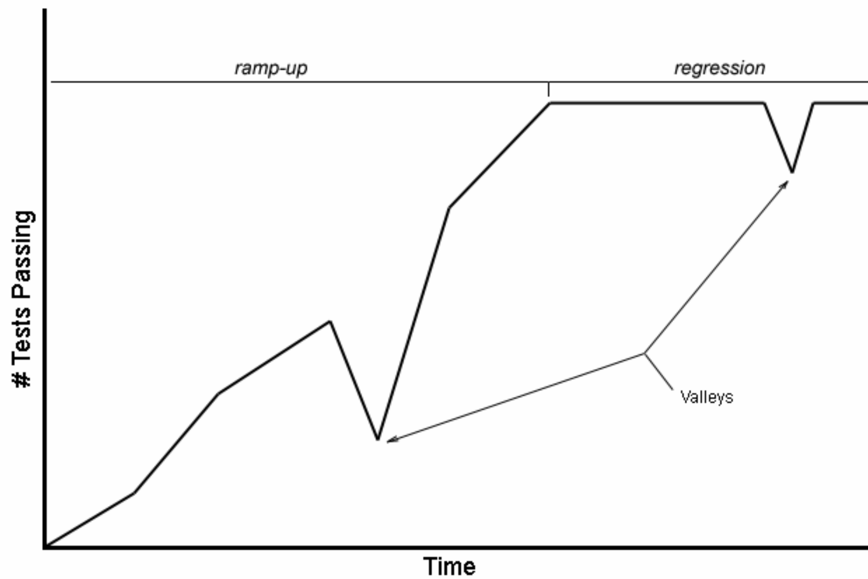
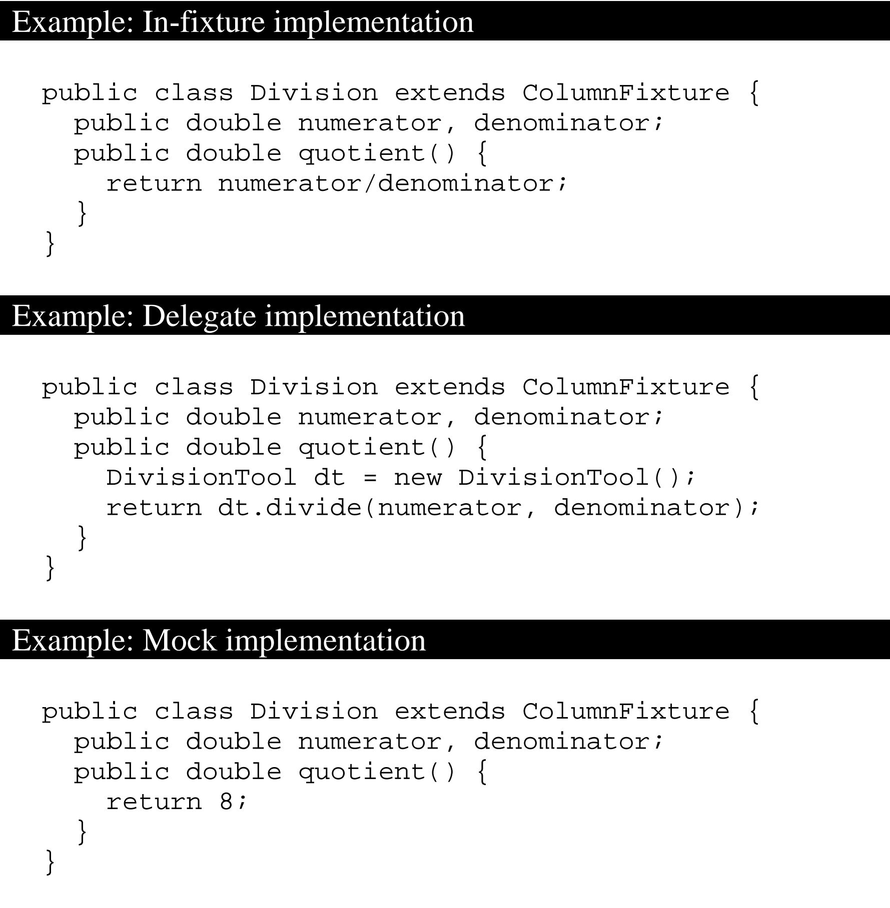


Figure 18. Typical iteration life-cycle

The timeline of both projects can be split into two sections (see Figure 18). The first time period begins when students receive 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 (see Figure 19).



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 DivisionTool();
    return dt.divide(numerator, denominator);
  }
}
```

Example: Mock implementation

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

Figure 19. Samples of how a given fixture could be implemented

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, developers 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 the subjects might make, but possibilities range from refactoring to adding new functionality.

Subjects used the FitNesse tool for defining and executing their tests. For the purposes of this study, I provided a binary of FitNesse that was modified to track and record a history of FIT test executions, both successful and unsuccessful. Specifically, I recorded:

- Timestamp
- Fully-qualified test name (with test suite name if present)
- Team (made anonymous)
- Result: number right, number wrong, number ignored, number of 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 of 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. This study observed 25,119 different data points about FIT usage, over four months.

Additional information was gathered by inspecting the source code of the test fixtures. Code analysis was looking at:

- 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
- A subjective rating from 0 to 10 of the "fatness" of the fixture methods (judged by myself and Grigori Melnik)

Fatness of the fixture methods was judged with a 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). 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.

The following hypotheses were formulated prior to beginning observations:

1 Common patterns of ramp-up or regression will be found between teams working on different projects in different contexts.
2 Teams will be able to identify and correct 100% of "bugs" in the test data and create new tests to overcome those bugs (with or without client involvement).
3 Even when no external motivation is offered, teams will refactor fixtures to properly delegate operations to business logic classes.
4 Even when no additional motivation is given, students will continue to the practice of executing their tests in regression mode (after the assignment deadline).
5 Students will use both suites and individual tests equally to organize and execute their tests.

5.2.4. Results

Results for the second study have been divided into four sections, each discussing a pattern of use or other observable behavior from the teams studied. These sections are Strategies for Test Fixture Design, Strategies for Using Test Suites vs. Single Tests, Development Approaches and Robustness of the Test Specification. Raw results are available in Appendix F.

5.2.4.1. Strategies for Test Fixture Design

It is obvious that there are many ways to develop a FIT fixture such that it satisfies the conditions specified in the FIT table. 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 (see Table 4).

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

Table 4. Common FIT fixtures used by subjects

Some tests involved a combination of more than one fixture type, and subjects ended up developing means to communicate between these fixtures.

Another design decision made by teams was whether to develop "fat", "thin" or "mock" methods within their fixtures (see Table 5). "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. These methods 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 to 10, in which 0 is entirely thin and 10 is entirely fat.

Team	Fatness (subjective)		NCSS[1]	
	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

Table 5. Statistics on fixture fatness and size

The most significant observation that can be made from Table 5 is that the UofC teams by and large had a much higher fatness when compared to the SAIT teams. No teams were observed delivering "mock" fixtures.

5.2.4.2. 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 developers 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.

In this experiment data on the frequency of test suite executions versus single test case executions was gathered (Table 6). Teams used their own discretion to decide which approach to follow (suites or single tests or both).

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

Team	Single Case Executions	Suite Executions	Single/Suite Ratio
UofC T1 (***)	454	650	0.70
UofC T2 (***)	253	314	0.80
UofC T3 (**)	459	169	2.72
UofC T4 (*)	597	0	Exclusively Single Cases
SAIT T1 (**)	501	258	1.94
SAIT T2 (**)	735	314	2.40
SAIT T3 (**)	160	49	3.27
SAIT T4 (*)	472	8	59.0
SAIT T5 (**)	286	47	6.09
SAIT T6 (not included due to too few data points).	25	8	3.13

Table 6. Frequency of test suites versus single test case executions during ramp up

During the regression testing stage I also measured how often suites versus single test cases were executed (Table 7). There is a clear difference between the ratio of single tests run to suites run between the UofC and SAIT teams.

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

Table 7. Frequency of test suites versus single test case executions during regression (post ramp-up)

5.2.4.3. Development Approaches

The analysis of ramp up data demonstrates that all teams likely followed a similar development approach (Table 8).

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

Table 8. Ratio of valleys found versus total assertions executed (valleys occur whenever a previously passing test fails).

Noticeably, there were very few peaks and valleys during development. A valley is measured when the number of passing assertions actually goes down from a number previously recorded. Peaks occur conversely when the number of passing tests goes up. The number of peaks always equals the number of valleys, since any occurrence of a test failure will be followed by a fix in that failure (and a corresponding peak). Henceforth I refer only to valleys.

5.2.4.4. 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 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 the professor and TA were easily able to clarify the requirement in question.

5.2.5. Interpretation

Like the results, interpretation has been dividend into four sections based on the patterns observed in this study.

5.2.5.1. Strategies for Test Fixture Design

The choice of fixtures likely relates to the education received by the students. All participants were introduced to RowFixtures and ActionFixtures in advance, and therefore these fixtures were heavily used. Other types of fixtures were available, but only the TableFixture was heavily chosen over the aforementioned two. The TableFixture allows the test implementer to specify how the table is parsed, and therefore is highly customizable. Students may have found it easy to understand and use in any situation.

The degree to which teams chose "fat", "thin" or "mock" fixtures could possibly be explained by commonalities between teams resulting from the environments they were in at each of the two locations. At the UofC, teams implemented the test fixtures in advance of any other business logic code (more or less following Test-Driven Development philosophy). 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" (this would result in tests not checking against real code). No refactoring was done on the fixtures in these cases. This can in my 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 5, UofC T2).

5.2.5.2. Strategies for Using Test Suites vs. Single Tests

Several strategies were identified from the team projects and are listed in Table 9.

Strategy	Pros	Cons
(*) Exclusively using single tests	- fast execution - encourages development in small steps	- 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

Table 9. Possible ramp-up strategies

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. 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

my 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 evenly in the range of 1-8% for all teams. Moreover, the number of total test runs is not deterministic of strategy chosen.

For UofC teams, there is 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 there is a radical difference in regression testing strategy. All SAIT teams use single test case executions much more than test suites. In fact, the ratios of single cases to suites are so high for SAIT that the UofC teams in retrospect appear to be using these two types of test execution more equally. Obviously, even after getting tests to pass initially, the 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 (the instructor of the course indicated that computer equipment of SAIT was slower).

5.2.5.3. Development Approaches

From the observations, it can be seen that there is a similar pattern for all groups. 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. It can be inferred from this pattern that features were being added incrementally to the system (Figure 20, left). Another approach could have included many assertions initially passing followed by many valleys during refactoring. That might illustrate a mock-up method in

which values were faked to get an assertion to pass and then replaced at a later time (Figure 20, right). However, such a mock-up method was not discovered in the student data. If students did use a mock-up method, they must have replaced it before handing in their assignment.

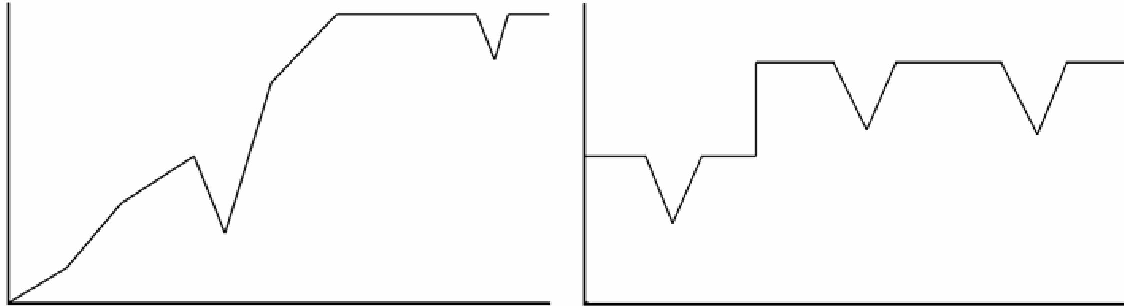


Figure 20. A pattern of what incremental development might look like (left) versus what mocking and refactoring might look like (right)

There were relatively few valleys discovered in the gathered data. A valley would indicate code has broken or an error has occurred. Analysis of the actual results would seem to indicate that in most cases as features and tests were added, they either worked right away or did not break previously passing tests. In my opinion, this is an indication that because the tests were specified upfront, they were driving the design of the project. Because 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 of tests being the main criteria of success.

5.2.5.4. Robustness of the Test Specification

Surprisingly, typos or omissions did not seem to affect subjects' ability to deliver working code. This might demonstrate that even with errors in the test specification, FIT adequately describes the requirements and/or makes said errors immediately obvious to the reader.

5.3. Student Experiences with Executable Acceptance Testing

5.3.1. Purpose

It would be useful to know how using FIT acceptance testing to drive agile development compares to traditional, document-centric communication of functional requirements. Ideally this study can analyze both the strengths and weaknesses of FIT compared to prose as well as how users perceive this new medium. User perceptions are very important, in particular in agile environments when developers are both empowered and often opinionated. This study examines the experiences of students while learning, reading and writing acceptance tests during the course of their projects. Objectives during the course of this study were:

1. To educate students about acceptance testing techniques
2. To investigate student perceptions of acceptance testing
3. To determine student opinions about FIT and FitNesse tools
4. To determine if executable acceptance tests can be used to communicate requirements.

5.3.2. Subjects and Sampling

Subjects for this third study are the same as for the first study mentioned in section 5.1 of this thesis. Namely, 28 senior undergraduate UofC Computer Science students from the Web-Based Systems course and 14 SAIT Bachelor of Applied Information Systems students from the Internet Software Techniques course took part in the study. In total, 9 teams with 4-6 members were formed.

5.3.3. Instrument

As previously mentioned (see Section 5.1), the practical component of both courses included a series of assignments to design and build a document review system. Each assignment required participants to deliver an increment of working functionality.

Assignment One required participants to design a data model using XML Schemas and to specify several transformations using XSLT. In this assignment, students' exposure to FIT was limited to understanding an assignment specification (given in the form of a suite of FIT tests) with the assistance of the instructor and teacher's assistants. Assignment Two involved building business logic using Enterprise Java Beans (session beans); Assignment Three a Web-based user interface; and in Assignment Four they developed an implementation of a persistence layer (entity beans). Assignments two through four did not involve any new acceptance tests (although other types of testing were introduced). In Assignment Five, participants built a Web-service for their document review system. In this fifth assignment teams were required to write acceptance tests for the Web-service functionality and then exchange tests cases with another randomly chosen team. Some teams received acceptance test suites from teams at another institution. In this final assignment, I observed students' ability to both write and understand acceptance tests independently.

Following the completion of their semesters, feedback was gathered using an "opt-in" survey distributed to subjects. The opt-in survey was distributed to all 42 subjects. The results of the survey were entirely anonymous. Each question described in the results notes the response rate relative to the total number of returned surveys. Questions asked in this survey are listed in Appendix B. Some additional qualitative information was informally gathered from in-class polls and during weekly stand-up meetings.

5.3.4. Results

Students could choose which questions to answer, and therefore response rates are noted where appropriate. The following student perceptions were gathered:

When asked to rate the appropriateness of FIT acceptance tests for defining functional requirements, students on average answered "adequate". Fully one-third answered

"good" or "excellent" to this question. Zero students answered "inappropriate" (see Figure 21).

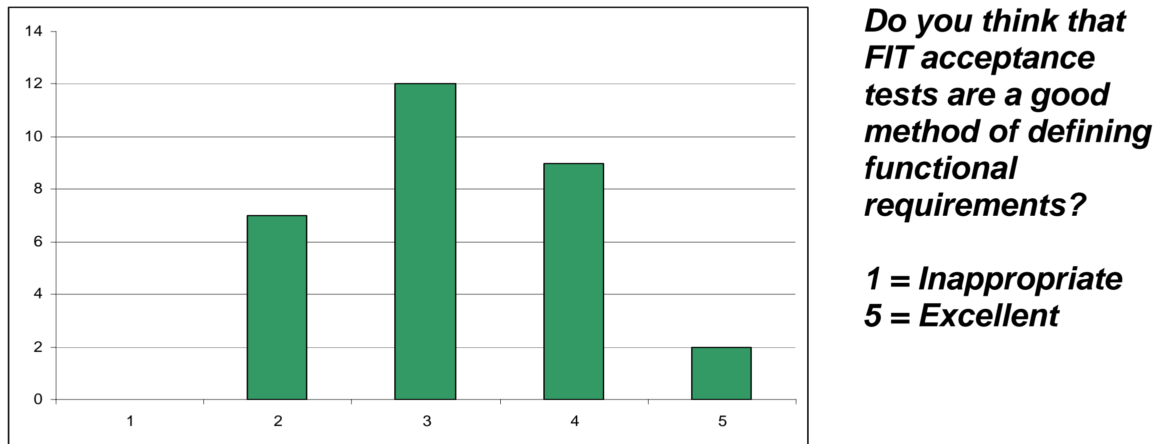


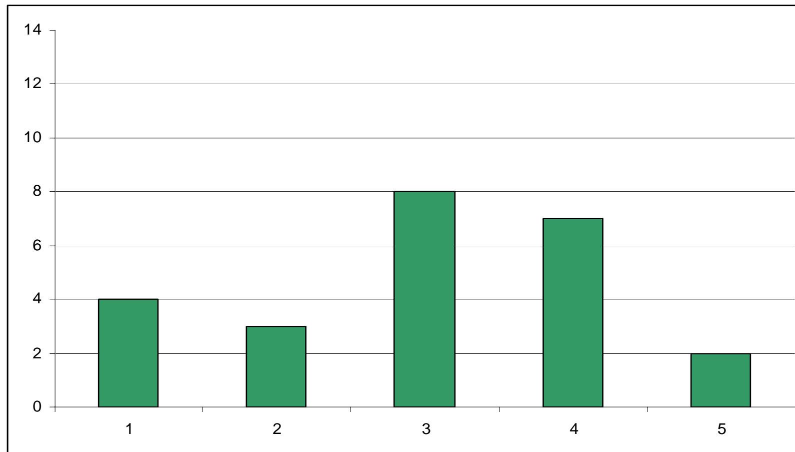
Figure 21. Student responses judging FIT for defining functional requirements

A second question was "Did you require additional written resources to complete the assignment, and if so what resources?" Students responded that no additional written resources were needed in 36% of replies (with a response rate of 73%). Some of the resources mentioned included: more FIT examples, FitNesse tool documentation, with only one person desiring "a better project description".

Subjects were surveyed to see if they required additional verbal instruction to complete the assignment. Thirty percent of students who replied answered that they did not need any additional verbal help to understand the assignment when specified using FIT. The remainder indicated they had sought TA or instructor assistance at some point. This question had a 70% response rate.

Asked "Would you have preferred to have this assignment specified entirely as prose (text) instead of as FIT acceptance tests?" subjects answered "no" in 16 out of 20 responses. This indicates a clear preference for using user acceptance tests over pure prose requirements specifications. The majority of those who answered "yes" explained that for them FIT was difficult to learn.

Subjects compared defining FIT acceptance tests with writing a specification in prose. On average, students found these tasks to be equal in the level of difficulty (see Figure 22). One third found FIT to be easier or much easier to use. This question had a response rate of 83 percent.

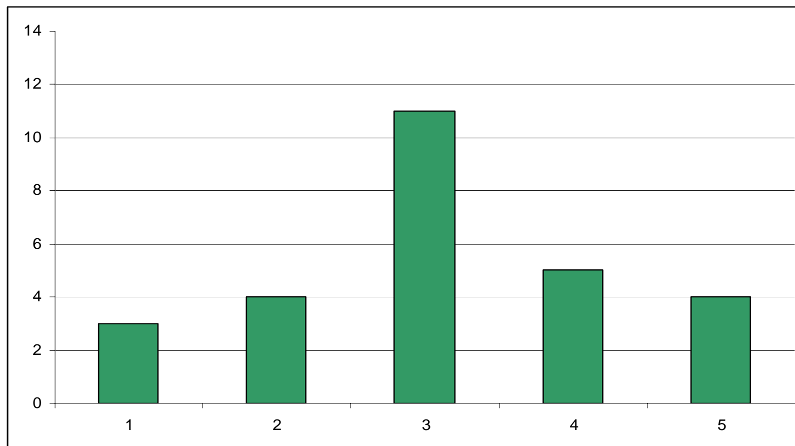


Do you think that defining FIT acceptance tests are easier or more difficult than defining a written specification?

1 = Much Harder
5 = Much Easier

Figure 22. Student responses comparing FIT with a written specification

Similarly, subjects compared defining FIT acceptance tests with writing JUnit tests (see Figure 23). Again on average results indicated that these tasks were equally difficult. Twenty-seven out of 30 subjects responded to this question.



Do you think that defining FIT acceptance tests are easier or more difficult than writing JUnit tests?

1 = Much Harder
5 = Much Easier

Figure 23. Student responses comparing difficulty of FIT with JUnit

Furthermore, I was interested in knowing if subjects believed that someone else could use FIT acceptance tests that they defined to create working software. A majority of sixty-five percent of responses were affirmative. Only three responses making up fifteen percent indicated that they did not think FIT acceptance tests could be used to communicate the specifications to create working software, and two out of three negative responses indicated an incomplete understanding of FIT or the question asked. The response rate for this question was 66%.

Following up on gathering the level of student's knowledge about FIT, I asked whether better knowledge of FIT acceptance tests would allow subjects to create a better specification. Seventy-five percent of respondents believed that it was the case (with an 80% response rate). Some opinions included "[FIT] is much clearer than writing requirements in prose", and "If one knows FIT acceptance tests first then he/she understands the system well".

Since the subjects randomly exchanged test suites between teams, I inquired if the acceptance test given to them by the other team was sufficient to create the Web service. Obviously the replies to this question would vary depending on the quality of specification received. Examples of poor-quality specifications included misuse of FIT which tightly coupled the tests to a specific implementation. In forty-three percent of cases students answered with a definite yes. Twenty-six percent answered that they managed to create the service but had problems. Thirty-one percent indicated that the FIT tests they were given were insufficient.

With regard to whether subjects communicated with the other team (who provided the test suite for the required web service) by means other than the FIT acceptance tests, 41% of respondents indicated that they did not. Of those who did communicate with the other team, common means included email, face-to-face, sharing files, phone, and using an online collaborative space. The response rate to this question was 73%.

Lastly I queried if subjects would choose to use FIT acceptance tests as a means to define functional requirements in the future. More than half (52%) said a definitive "yes" and three more indicated maybe. This cumulatively means that over two-thirds of respondents would consider using FIT in the future.

The raw results for this survey are available in Appendix F.

5.3.5. Interpretation

The observed results speak for themselves, and give a good idea of how the students perceive acceptance tests and acceptance testing tools.

Results from the perceptions survey were on the whole positive. For most questions the majority of responses praised FIT or indicated that FIT was as useful for describing functional requirements as a prose assignment definition. Negative or criticizing comments about FIT and related technologies were in the minority. There are some questions about the validity of this perceptions study. It is impossible to say if these perceptions are altogether honest, or if student experience goes deep enough to form an unbiased opinion. The only measures taken to try to ensure honesty were to assure the students that the surveys were anonymous and not related to their evaluation. It is also possible that there is a recency bias, in which students favoured FIT related approaches simply because they have used them more recently. At the very least this sort of response indicates that more study is needed and that FIT is a technology worth pursuing both in the educational arena and as a developer tool.

6. Conclusion

At the beginning of this thesis, I identified three motivating factors for this research.

1. It is not known if agile methods can be applied in a divide and conquer setting and yet remain agile, or whether these approaches are complimentary;
2. It is not known what adaptations can be or need to be made to agile methods in order to enable such an approach;
3. There exists no tool support to enable or assist in using agile methods with a divide and conquer strategy.

I believe it can be said that in this thesis, all three factors have been adequately addressed.

An analysis of the problem revealed three obstacles to supporting agile teams of teams. These three obstacles are the need for up-front design, team inter-communication and module integration. In order to confidently say that agile methods can be applied on a divided project, all three obstacles will need to be overcome.

A conceptual approach has been proposed that uses adaptations to agile practices to address all three of these obstacles. This approach relies on an adaptation of test driven design, in which a developer can act as a customer for another team in order to resolve and communicate dependency issues. This process of "acting as a customer" involves supplying user stories for another team as well as creating tests for those stories. This method is driven through identification of one's own expectations for other modules, rather than through any predictive or comprehensive design process. Tests can be used by the implementing team as a source of information about feature requirements, yet also can be executed to determine the status of the dependency, as well as to verify compatibility and ensure integration between the products of different teams.

A further adaptation to agile methods proposed in this thesis advocates that tests be used to represent an implicit architecture for the system. Recognizing that tests alone can

represent dependency relationships lead to the idea that an overall picture of architecture can be generated as tests evolve over time. This approach is meant to maintain the flexibility of agile methods and to encourage an iterative evolution of how different components interact. The idea that automated acceptance tests can be used to represent and communicate functional requirements between teams was moreover suggested by this thesis. In particular, the FIT acceptance test framework was proposed and given as an example.

The question of how suitable such automated acceptance tests are for representing functional requirements is an important pre-requisite to the validation of ideas in this thesis, since dependencies between teams exist as functional requirements. Empirical data was gathered in three studies which support the claim that FIT can be used to express functional requirements. Specifically, student participants were able to understand and implement code to satisfy requirements provided as FIT tests (see Section 5.1). Looking in further detail at how teams use FIT acceptance tests, it was determined that common patterns exist in how teams used the framework, and also that teams were able to identify and make corrections for errors in requirements when specified as FIT tests (see Section 5.2). This thesis also examined whether teams do actually execute acceptance tests as regression tests (see Section 5.2). This is an indicator of whether acceptance tests can be used as a means of measuring progress and determining project state over time. In addition, how teams organize their acceptance tests was examined as a pre-requisite question to determining if the idea of an implicit architecture based on the evolution of tests is rational. Finally, user perceptions of FIT for acceptance testing and representing functional requirements were assessed (see Section 5.3).

Three tools were developed to support agile teams of teams via test driven development. These tools can be seen as proof of concept implementations for the aforementioned ideas and adaptations to agile methods. COACH-IT supports defining an explicit lightweight architecture for using test driven development to communicate and enforce dependencies using unit testing. COACH-IT also addresses the problem of up-front design by simplifying the architecture of dependencies between project components as

much as possible, and by providing a web-based user interface for making changes and updates. MASE implicitly defines an architecture from a loose collection of acceptance tests, allowing users of the tool to define tests and test suites as well as to load executable tests and code from remote repositories. MASE does not restrict how or to what degree the relationships between project components are tested or represented, and relies on an evolution of tests over the course of the project. The MASE tool is also a proof of concept for the communication of functional requirements using FIT. The Rally tool distributes the management of testing and the execution of tests across client/server boundaries. Such increased flexibility means that more control over the implementation and authorship of the tests is available to the end-user. This tool also generalizes the proof of concept for FIT tests and Java projects to work with any type of project using any type of test. All three tools support agile teams of teams using test driven development, and might enable agile methods to be applied on large scale or distributed teams using a divide and conquer strategy.

By now it should be clear that the four goals outlined at the start of this thesis have been achieved. I have **proposed a solution that I believe overcomes the problems of a divide and conquer approach**, through using test driven design to address up-front design, communication and integration. I have done so **without violating the terms of what makes a software development methodology agile**. Furthermore I have indicated **the viability of this solution**, by showing that FIT tests can represent functional requirements, by finding common patterns in the use of acceptance tests and acceptance testing tools, and by demonstrating that perceptions of both acceptance testing and using acceptance tests as a communication medium are not negative. Finally, I have **provided working tool support for this solution** with three different proofs of concept: COACH-IT, MASE and the Rally tool.

7. Future Work

There is much future work to be done in this area. The empirical data gathered in this thesis only supports pre-requisite questions regarding whether or not the proposed approach really works. This empirical data suggests that FIT tests can be read and understood as a representation of functional requirements; however this does not prove that automated acceptance tests can be used to communicate dependencies within agile teams of teams. Likewise, the empirical data suggests that there are common patterns in how teams use FIT acceptance tests from the time the requirements are gathered to the completion of the project. However this is not the same as asserting that acceptance tests are proven to be effective at overcoming the problems of validation and integration of dependencies. Empirical data in this thesis provides some groundwork and gives an initial indication about the reasonableness of the claims herein. However, in order to empirically back up these claims, they would need to be studied directly. Doing so would ideally require monitoring a real-world project in which agile teams of teams use the described approach. Moreover, the initial evidence in this thesis comes solely from student participants; real industry developers would make up a more valid group. It would be of great value to also perform direct testing of the tools developed during the course of this research, since no data was collected on their use. Additional work on the tools supporting this method could improve them, since in most cases emphasis during development was on creating a prototype and not a production system. Rally Development is planning to incorporate the ideas and concepts from this thesis and from the proof-of-concept tools into a commercial product. A study could also be performed using the Rally product or comparing the Rally product with the three different tools already developed.

Supplementary avenues of research branching from this one might include using other types of tests to represent requirements, as well as proposing alternate adaptations to agile methods which might enable the application of those methodologies in distributed or large teams.

References

1. Abrahamsson, P. (2004) Extreme Programming: A Survey of Empirical Data from a Controlled Case Study, ACM-IEEE International Symposium on Empirical Software Engineering, ISESE
2. Abran, A., Moore, J. (2001) Guide to the Software Engineering Body of Knowledge : Trial Version (Version 0.95),IEEE, Los Alamitos, CA
3. Alleman, G., Henderson, M. (2003) Making Agile Development Work in a Government Contracting Environment, Proceedings of the Agile Development Conference 2003, IEEE
4. Ambler, S. (2003) Something's Gotta Give, Software Development Magazine, March Issue
5. Andersson, J., Bache, G., Sutton, P., (2003) XP with Acceptance-Test Driven Development : A Rewrite Project for a Resource Optimization System, Proceedings of XP2003, Springer
6. Apache Ant (2005) Online http://ant.apache.org
7. Appleton, B., et al. (2004) Continuous Staging: Scaling Continuous Integration to Multiple Component Teams, CM Crossroads Journal, Vol. 3, No. 3
8. Austin, R., Devin, L. (2004) The Economics of Agility in Software Development, Harvard Business School Working Paper Series, No. 04-057
9. Basili, V., Turner, A. (1975) Iterative Enhancement: A Practical Technique for Software Development, IEEE Transactions on Software Engineering
10. Beck K. (2000) Extreme Programming Explained: Embrace Change. Addison-Wesley
11. Beck, K., Andres, Cynthia (2005) Extreme Programming Explained: Embrace Change Second Edition, Addison-Wesley
12. Ben-Menachem, M., Marliss, G. (1997) Software Quality: Producing Practical, Consistent Software, International Thomson Publishing, London, UK
13. Boehm, B., (2002) Get Ready for Agile Methods, with Care. IEEE Computer

14. Cao, L. et al. (2004) How Extreme does Extreme Programming Have to be? Adapting XP to Large-Scale Projects. Proceedings of the 37th Hawaii International Conference on System Sciences, IEEE
15. Carmel, E. (1999) Global Software Teams, Prentice Hall, Upper Saddle River, NJ
16. Chau, T. (2005) Inter-Team Learning for Agile Software Processes, M.Sc. Thesis, University of Calgary, Department of Computer Science
17. Cockburn, A., Highsmith, J. (2001) Agile Software Development: The People Factor. IEEE Computer
18. Crispin, L. (2002) Testing in the Fast Lane: Acceptance Test Automation in an Extreme Programming Environment, Extreme Programming Perspectives, Addison-Wesley
19. CruiseControl (2005) Online http://cruisecontrol.sourceforge.net
20. Du Bois, B. (2004) A discussion of refactoring in research and practice, Technical Report, University of Antwerp
21. Elssamadisy, A. (2001) XP On A Large Project – A Developer's View, Proceedings of XP/Agile Universe, Online http://www.xpuniverse.com/2001/pdfs/EP202.pdf
22. Erdogmus, H. (2003) Let's Scale Agile Up, Agile Times, Volume 2, Online http://www.agilealliance.org/membership/vol2.pdf
23. Favaro, J. (2004) Efficient Markets, Efficient Projects, and Predicting the Future, Proceedings of XP2004, Lecture Notes in Computer Science, Volume 3092, Springer
24. Finholt, T., Sproull, L., Kielser, S. (1990) . Communication and Performance in Ad-hoc Task Groups, from: Galegher, J., Kraut, R., Egido, C. (1990) Intellectual Teamwork, Lawrence Erlbaum Press, Hillsdale, N.J
25. FIT (2005) Online http://fit.c2.com
26. FitNesse (2005) Online http://fitnesse.org
27. Fowler, M. () Using an Agile Software Process with Offshore Development, Online http://www.martinfowler.com/articles/agileOffshore.html
28. George, B., Williams, L. (2002) An Initial Investigation of Test-Driven Development in Industry, ACM Symposium on Applied Computing

29. Geras, A., Smith, M., Miller, J. (3004) A Prototype Empirical Evaluation of Test Driven Development, IEEE METRICS
30. Global Outsourcing Report (2005) CIO Insight Magazine, March Issue, Ziff Davis Media Publishing
31. Global Software Development (2001) Special Issue, IEEE Software, IEEE
32. Goldratt, E. (1984) The Goal, The North River Press, Great Barrington, MA
33. Herbsled, D. et al. (2001) An Empirical Study of Global Software Development: Speed and Distance, Proceedings of the 23rd International Conference on Software Engineering, ICSE
34. Highsmith, J. (2000) Adaptive Software Development, Dorset House, NY
35. Highsmith, J. (2002) Does Agility Work, Software Development, 10(6)
36. Holz, H., Melnik, G., Schaaf, M. (2003) Knowledge Management for Distributed Agile Processes: Models, Techniques, and Infrastructure, Proceedings of WETICE 2003
37. Hooks, I., Farry, K. (2001) Customer-Centered Products: Creating Successful Products Through Smart Requirements Management, American Management Association, New York, NY
38. IEEE Standard Glossary of Software Engineering Terminology (1983), ANSI/IEEE Std 729-1983, IEEE
39. JCVS (2005) Online http://www.jcvs.org
40. Jeffires, R. (2001) Essential XP: Documentation, XP Magazine, Online http://www.xprogramming.com
41. Johnson, J. et al. (2001) Collaborating on Project Success. Software Magazine, February Issue
42. JUnit (2005) Online http://www.junit.org/index.htm
43. Karolak, D. (1998) Global Software Development, IEEE, Los Alamitos, CA
44. Katzenbach, J., Smith, D. (1994) The Wisdom of Teams: Creating the High-Performance Organization, Harper Collins, New York, NY
45. Kaufmann, R., Janzen, D. (2003) Implications of Test-Driven Development on Software Quality: A Pilot Study, Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, CA

46. Kohl, J., Marick, B. (2004) Agile Tests as Documentation, Proceedings of XP/Agile Universe, Springer
47. Kontio, J., et al. (2004) Managing Commitments and Risks: Challenges in Distributed Agile Development, Proceedings of the 26th International Conference on Software Engineering, ICSE
48. Kraut, R., Edigo, C., Galegher, J. (1990). Patterns of Communication in Scientific Research Collaboration, from: Galegher, J., Kraut, R., Egido, C. (1990) Intellectual Teamwork, Lawrence Erlbaum Press, Hillsdale, N.J
49. Kruchten, P. (2004) Scaling Down Large Projects to Meet the Agile 'Sweet Spot', The Rational Edge, Online http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/aug04/5558.html
50. Larman, C. (2004) Agile and Iterative Development: A Manager's Guide, Boston, MA, Addison Wesley
51. Larman, C., Basili, V. (2003) A History of Iterative and Incremental Development, IEEE Computer 36(6)
52. Lindvall, M., et al. (2002) Empirical Findings in Agile Methods, Proceedings of XP/Agile Universe 2002, Springer
53. Lippert, M., et al. (2003) Developing Complex Projects Using XP with Extensions, IEEE Computer, June Edition
54. Manifesto for Agile Software Development (2005) Online: http://agilemanifesto.org/
55. Martin, R. (2002) Agile Software Development, Principles, Patterns, and Practices, Prentice Hall
56. MASE (2005) Online http://ebe.cpsc.ucalgary.ca/ebe
57. McConnell, S., (1997) Less is More: Jumpstarting Productivity with Small Teams, Software Development, October Issue
58. Melnik, G., Maurer, F. (2004) Introducing Agile Methods: Three Years of Experience, Proceedings of SPPI04, IEEE
59. Melnik, G., Williams, L., Geras, A. (2002) Empirical Evaluation of Agile Processes, Proceedings of XP/Agile Universe 2002, Chicago, US, Springer

60. METAspectrum Report on the Offshore Outsourcing Market (2004) Proceedings Outsourcing Conference & Technology Showcase, San Francisco CA. Online http://www.metagroup.com/us/metaSpectrum.do?fileName=OffshoreOutsourcing_mktOverview
61. Meyer, B. (1985) On Formalism in Specifications, IEEE Software, 2(1)
62. Mugridge, R., Tempero, E. (2003) Retrofitting an Acceptance Test Framework for Clarity, Proceedings of the Agile Development Conference, IEEE
63. Murru, O., Deias, R., Mugheddu, G. (2004) Assessing XP at a European Internet Company, IEEE Software, 20(3)
64. Newkirk, J. (2002) A Light in a Dark Place: Test-driven Development of 3rd Party Packages, Proceedings XP/Agile Universe 2002, Springer
65. Opdyke, W. (1989) Refactoring Object-Oriented Frameworks, PhD Thesis, University of Illinois at Urbana-Champaign. Technical Report UIUCDCS-R-92-1759, Department of Computer Science, University of Illinois at Urbana-Champaign
66. Palmer, S., Felsing J. (2002) A Practical Guide to Feature Driven Development, Prentice Hall
67. Patton, J. (2003) Unfixing the Fixed Scope Project: Using Agile Methodologies to Create Flexibility in Project Scope. Proceedings of the Agile Development Conference 2003, IEEE
68. Peters, T. (1999) The Professional Service Firm 50, Knopf
69. Poppendieck, M., Poppendieck, T. (2003) Lean Software Development: An Agile Toolkit, Addison-Wesley
70. Rally Development (2005) Online http://www.rallydev.com
71. Read, K., Maurer, F. (2003) Issues in Scaling Agile Using an Architecture-Centric Approach: A Tool-Based Solution. Proceedings of XP/Agile Universe 2003, Springer
72. Rees, D. (2004) Distributed Agile Development", Online http://www.itwales.com/99851.htm
73. Reifer, D., Maurer, F., Erodgmus, H. (2003) Scaling Agile Methods – Ten Top Issues and Lessons Learned, July Edition, IEEE Software, 20(4)

74. Rogers, O. (2004) Acceptance Testing vs. Unit Testing: A Developer's Perspective, Lecture Notes in Computer Science, Volume 3134, Springer
75. Schwaber, K., Beedle, M. (2001) Agile Software Development with Scrum, Prentice Hall
76. Senge, P. (1990) The Fifth Discipline – The Art and Practice of the Learning Organization, Currency Doubleday, New York, NY
77. Sepulveda, C., Marick, B., Mugridge, R., Hussman, D. (2004) Who Should Write Acceptance Tests?, Lecture Notes in Computer Science, Vol. 3134, Springer
78. Simons, M. () Distributed Agile Development and the Death of Distance, Source and Vendor Relationships Executive Report, 5(4), Cutter Consortium
79. Stapleton, J. (1997) DSDM Dynamic Systems Development Method: The Method in Practice, Addison-Wesley
80. Steinberg, D. (2003) Using Instructor Written Acceptance Tests Using the Fit Framework, Lecture Notes in Computer Science, Volume 2675, Springer
81. Taylor, W. (1998) The Principles of Scientific Management, Harper & Brothers (available as a Dover republication in 1998)
82. The CHAOS Chronicles (2004) The Standish Group International, West Yarmouth, MA. Online http://www1.standishgroup.com/chaos/intro2.php
83. Thomas, D., Hunt, A. (2004) Nurturing Requirements, IEEE Software, 21(2)
84. Turk, D., France, R., Rumpe, B. (2002) Limitations of Agile Software Processes, Proceedings of XP2002, IEEE
85. Van Vliet, H. (2000) Software Engineering: Principles and Practice, John Wiley & Sons, Chichester, UK
86. Watt, R., Leigh-Fellows, D. (2004) Acceptance Test Driven Planning, Proceedings of XP/Agile Universe 2004, Springer
87. Womack, J., Jones, D., Ross, D. (1990) The Machine that Changed the World. Rawson Associates, NY
88. XML (2005) Online http://www.w3.org/XML
89. XSL (2005) Online http://www.w3.org/Style/XSL
90. Young, R. (2001) Effective Requirements Practices. Addison-Wesley, Boston MA

Appendices

Appendix A. Participant Consent Form

Research Project Title: Examining Tool Support for Agile Software Development Teams

Investigator(s): Thomas Chau (chauth@cpsc.ucalgary.ca), Kris Read (readk@cpsc.ucalgary.ca), Frank Maurer (maurer@cpsc.ucalgary.ca)

This consent form, a copy of which has been given to you, is only part of the process of informed consent. It should give you the basic idea of what the research is about and what your participation will involve. If you would like more detail about something mentioned here, or information not included here, you should feel free to ask. Please take the time to read this carefully and to understand any accompanying information.

Description of Research:
A purpose of this research is to examine how multiple teams, using agile development practices, share their expertise with each other and utilize tools to support distributed agile practices. This research investigates whether developers exhibit particular preferences in sharing expertise with each other and in using an experience base that is shared among all teams (i.e. use of personal dialogue/on-line chat vs. written documents, amount of retrieval vs. amount of update). This research also aims to determine the feasibility of specifying software requirements as acceptance tests, to establish the relative difficulty for students to create specifications as FIT acceptance tests compared to design documentation or code-based tests and to gather data regarding the effectiveness of FIT acceptance tests as a communication tool between distributed software development groups. MASE tries to support the application of agile methods in non-ideal circumstances including distributed development scenarios. We wish to evaluate the effectiveness and helpfulness of the tool (subjectively) and to determine potential problems or areas for improvement.

Procedure:
You will be asked to complete a questionnaire. Your participation in completing this questionnaire is voluntary. If at any time you feel uncomfortable, you may withdraw from the study and your questionnaire will be discarded.

Likelihood of Discomforts:
There is no harm, discomfort, or risk associated with your participation in this research.

Confidentiality:
Participant anonymity 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 or published without your specific consent to disclosure. All the data

collected will be stored in a password-protected computer and only be accessible to the investigator.

Primary Researcher:
Thomas Chau and Kris Read are M.Sc. students in the Department of Computer Science at the University of Calgary under the supervision of Dr. Frank Maurer. This study is conducted as part of their research and will be included in their theses.

Your signature on this form indicates that you have understood to your satisfaction the information regarding participation in the research project and agree to participate as a subject. In no way does this waive your legal rights nor release the investigators, sponsors, or involved institutions from their legal and professional responsibilities. You are free to withdraw from the study at any time. Your continued participation should be as informed as your initial consent, so you should feel free to ask for clarification or new information throughout your participation. If you have further questions concerning matters related to this research, please contact:

Thomas Chau (chauth@cpsc.ucalgary.ca), Department of Computer Science,
University of Calgary

Kris Read (readk@cpsc.ucalgary.ca), Department of Computer Science,
University of Calgary

Frank Maurer (maurer@cpsc.ucalgary.ca), Department of Computer Science,
University of Calgary

If you have any questions or issues concerning this project that are not related to the specifics of the research, you may also contact the Research Services Office at 220-3782 and ask for Mrs. Patricia Evans.

Participant's Signature   Date

Investigator and/or Delegate's Signature   Date

Witness' Signature   Date

Appendix B. Questionnaire

1. What are you knowledge/experience with web technologies prior to taking this course?

0 months   0 – 12 months   > 12 months

In Assignment 1 you were given FIT acceptance tests. Please answer the following questions about that assignment:
NOTE: Questions 2-10 did not relate to this thesis, but were for an unrelated study.
10. Do you think that FIT acceptance tests are a good method of defining functional requirements? Why or why not?

Poor - 1	2	3	4	Excellent - 5

11. Did you require additional written resources to complete the assignment? If so what resources?

12. Did you require additional verbal instruction to complete the assignment? If so what kind?

13. Would you have preferred to have this assignment specified as prose (text) instead of as acceptance tests? Why or why not?

14. What aspects of the tool (MASE) were help or a hindrance to this assignment?

In Assignment 1 you were asked to create your own FIT acceptance tests. Please answer the following questions about that assignment.

15. Do you think that defining FIT acceptance tests are easier or more difficult than defining a written specification? Why?

Much Harder - 1	2	3	4	Much Easier - 5

16. Do you think that defining FIT acceptance tests are easier or more difficult than writing JUnit tests? Why?

Much Harder - 1	2	3	4	Much Easier - 5

17. Do you think that someone else could use FIT acceptance tests that you defined to create working software? Why or why not?

18. Do you believe that better knowledge of FIT acceptance tests would allow you to create a better specification?

In Assignment 1 you were given another team's acceptance test and asked to implement a web service. Please answer the following questions about that assignment.

19. Was the acceptance test given to you by the other team sufficient to create the web service? Why or why not?

20. Did you communicate with the other team by means other than the FIT acceptance tests? If so, what did you use and why?

21. Would you use FIT acceptance tests as a means to define program criteria between teams in the future? Why or why not?

Appendix C. Ethics & Co-Author Approval

UNIVERSITY OF
CALGARY

MEMO

CONJOINT FACULTIES RESEARCH ETHICS BOARD
c/o Research Services
Room 602 Earth Science
Telephone: (403) 220-3782
Fax: (403) 289 0693
Email: plevans@ucalgary.ca
Thursday, January 22, 2004

To: Kristopher D. Read
Computer Science

From: Dr. Janice P. Dickin, Chair
Conjoint Faculties Research Ethics Board (CFREB)

***Re:* Certification of Institutional Ethics Review:** Examining Tool Support for Agile Software Development Teams

The above named research protocol has been granted ethical approval by the Conjoint Faculties Research Ethics Board for the University of Calgary.

Enclosed are the original, and one copy, of a signed **Certification of Institutional Ethics Review**. Please make note of the conditions stated on the Certification. A copy has been sent to your supervisor as well as to the Chair of your Department/Faculty Research Ethics Committee. In the event the research is funded, you should notify the sponsor of the research and provide them with a copy for their records. The Conjoint Faculties Research Ethics Board will retain a copy of the clearance on your file.

Please note, an annual/progress/final report must be filed with the CFREB twelve months from the date on your ethics clearance. A form for this purpose has been created, and may be found on the "Ethics" website, http://www.ucalgary.ca/UofC/research/html/ethics/reports.html

In closing let me take this opportunity to wish you the best of luck in your research endeavor.

Sincerely,

Patricia Evans

Patricia Evans
Executive Secretary for:
Janice Dickin, Ph.D., LLB., Faculty of Communication and Culture and
Chair, Conjoint Faculties Research Ethics Board

Enclosures(2)
cc: Chair, Department/Faculty Research Ethics Committee
Supervisor: Dr. F. Maurer

CERTIFICATION OF INSTITUTIONAL ETHICS REVIEW

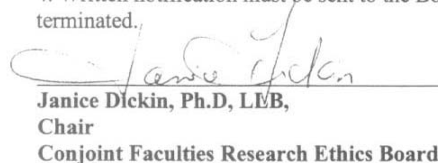
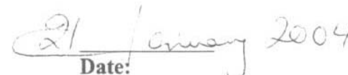
This is to certify that the Conjoint Faculties Research Ethics Board at the University of Calgary has examined the following research proposal and found the proposed research involving human subjects to be in accordance with University of Calgary Guidelines and the Tri-Council Policy Statement on *"Ethical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

File no:	**CE101-3820**
Applicant(s):	**Kristopher D. Read** Thomas Chau
Department:	**Computer Science**
Project Title:	**Examining Tool Support for Agile Software Development Teams**
Sponsor (if applicable):	

Restrictions:

This Certification is subject to the following conditions:

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

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

Date: 21 January 2004

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

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

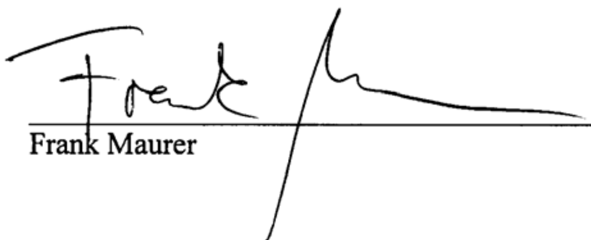
Co-Author Permission

April, 2005

University of Calgary
2500 University Drive N.W.
Calgary, Alberta
T2N 1N4

I, Frank Maurer, give Kris Read permission to use co-authored work from our published papers, "Examining Usage Patters of the FIT Acceptance Testing Framework", "Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective", and "Issues in Scaling Agile Using an Architecture-Centric Approach: A Tool-Based Solution" in this thesis.

Sincerely,



Frank Maurer

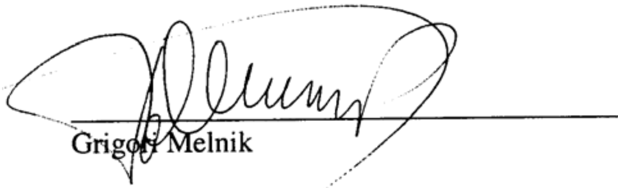
Co-Author Permission

April, 2005

University of Calgary
2500 University Drive N.W.
Calgary, Alberta
T2N 1N4

I, Grigori Melnik, give Kris Read permission to use co-authored work from our published papers, "Examining Usage Patters of the FIT Acceptance Testing Framework", "Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective" in this thesis.

Sincerely,



Grigori Melnik

Appendix D. Technical Detail

COACH-IT Architecture Example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<architecture name="testArch"
        xsi:schemaLocation="http://sern.ucalgary.ca/~readk/COACH-IT/arch.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <description>this is the test arch for the demo</description>
        <component name="comp1-ejb">
                <executable cvspass="cvsebe" file="TestProjectEJB.jar"
                        cvslocation=":pserver:cvs@comokit.cpsc.ucalgary.ca:/E//CVS/TEST"
                        jvm="1.4" type="jar" package="TestComp1"></executable>
                <interface file="TestBean.java"
                        cvslocation=":pserver:cvs@comokit.cpsc.ucalgary.ca:/E//CVS/TEST"
                        cvspass="cvsebe" name="testBean" package="TestComp1"></interface>
                <email address="kdread@ucalgary.ca"></email>
                <description>This is the ejb component for the project</description>
        </component>
        <component name="comp2-web">
                <executable cvspass="pw4c60185" file="TestProjectWeb.war"
                        cvslocation=":pserver:lancet@cvs.cpsc.ucalgary.ca:/scrinium/c60185"
                        jvm="1.4" type="war" package="TestComp2"></executable>
                <interface file="file"
                        cvslocation=":pserver:lancet@cvs.cpsc.ucalgary.ca:/scrinium/c60185"
                        cvspass="pw4c60185" name="interface2"
                        package="Package"></interface>
                <email address="lancet@cpsc.ucalgary.ca"></email>
                <description>This is the web component for the project</description>
        </component>
        <link name="Com2_uses_Com1">
                <uses_interface component="comp2-web"></uses_interface>
                <provides_interface interface="testBean"
                                      component="comp1-ejb"></provides_interface>
                <testsuite className="coachit.test.tests.AllTests"
                        logDir="C:/Documents and Settings/lancet/.coachit/Com2_uses_Com1"
                        name="Testsuite1">
                        <executable cvspass="pw4c60185" file="TestProjectTests.jar"
                        cvslocation=":pserver:lancet@cvs.cpsc.ucalgary.ca:/scrinium/c60185"
                                jvm="1.4" type="jar" package="TestComp2"></executable>
                        <email address="lancet@cpsc.ucalgary.ca"></email>
                </testsuite>
                <description>Link between the EJB and the Web comps</description>
        </link>
        <link name="Com1_uses_Com2">
                <uses_interface component="comp1-ejb"></uses_interface>
                <provides_interface interface="interface2"
                        component="comp2-web"></provides_interface>
                <testsuite className="ca.coachit.Tests.Test1"
                        logDir="C:/Documents and Settings/lancet/.coachit/Com1_uses_Com2"
                        name="testsuite2">
                        <executable cvspass="pw4c60185" file="file"
                        cvslocation=":pserver:lancet@cvs.cpsc.ucalgary.ca:/scrinium/c60185"
                                jvm="1.4" type="war" package="package"></executable>
                        <email address="kdread@ucalgary.ca"></email>
                </testsuite>
                <description></description>
        </link>
        <link name="com2forcom1">
                <uses_interface component="comp2-web"></uses_interface>
                <provides_interface interface="testBean"
                        component="comp1-ejb"></provides_interface>
                <testsuite className="ca.coachit.Tests.AllTests"
                        logDir="C:/Documents and Settings/lancet/.coachit/com2forcom1"
                        name="test3">
                        <executable cvspass="sdgf" file="asfsdf"
                                cvslocation=":pserver:dfg@dfg:qdfg" jvm="1.4" type="ear"
```

```
                                package="sdfg"></executable>
                        <email address="sdf@sdf.com"></email>
                </testsuite>
                <description></description>
        </link>
</architecture>
```

COACH-IT Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="architecture">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="description"/>
                <xsd:element maxOccurs="unbounded" minOccurs="0" ref="component"/>
                <xsd:element maxOccurs="unbounded" minOccurs="0" ref="link"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:ID" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="component">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="executable"/>
                <xsd:element maxOccurs="unbounded" minOccurs="0" ref="interface"/>
                <xsd:element maxOccurs="unbounded" minOccurs="1" ref="email"/>
                <xsd:element maxOccurs="1" minOccurs="0" ref="description"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:ID" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="description" type="xsd:string"/>
    <xsd:element name="interface">
        <xsd:complexType>
            <xsd:attribute name="name" type="xsd:ID" use="required"/>
            <xsd:attribute name="file" type="xsd:string" use="required"/>
            <xsd:attribute name="cvslocation" type="xsd:string" use="required"/>
            <xsd:attribute name="cvspass" type="xsd:string" use="required"/>
            <xsd:attribute name="package" type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="executable">
        <xsd:complexType>
            <xsd:attribute name="file" type="xsd:string" use="required"/>
            <xsd:attribute name="cvslocation" type="xsd:string" use="required"/>
            <xsd:attribute name="cvspass" type="xsd:string" use="required"/>
            <xsd:attribute name="package" type="xsd:string" use="required"/>
            <xsd:attribute default="ear" name="type">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="ear"/>
                        <xsd:enumeration value="jar"/>
                        <xsd:enumeration value="war"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute default="1.4" name="jvm">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="1.2"/>
                        <xsd:enumeration value="1.3"/>
                        <xsd:enumeration value="1.4"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="link">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="uses_interface"/>
                <xsd:element ref="provides_interface"/>
                <xsd:element ref="testsuite"/>
                <xsd:element maxOccurs="1" minOccurs="0" ref="description"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:ID" use="required"/>
```

```
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="testsuite">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="executable"/>
                <xsd:element maxOccurs="unbounded" minOccurs="1" ref="email"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:ID" use="required"/>
            <xsd:attribute name="className" type="xsd:string" use="required"/>
            <xsd:attribute name="logDir" type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="uses_interface">
        <xsd:complexType>
            <xsd:attribute name="component" type="xsd:IDREF" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="provides_interface">
        <xsd:complexType>
            <xsd:attribute name="component" type="xsd:IDREF" use="required"/>
            <xsd:attribute name="interface" type="xsd:IDREF" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="email">
        <xsd:complexType>
            <xsd:attribute name="address" type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

CruiseControl Config File

```
<cruisecontrol>
   <project name="helloworld">
      <bootstrappers>
         <currentbuildstatusbootstrapper file="C:\Development\cruisecontrol-2-0-
2\main\example\helloWorldCC\logs\currentbuild.txt" />
      </bootstrappers>
      <modificationset quietperiod="30" dateformat="yyyy-MMM-dd HH:mm:ss">
         <cvs cvsroot=":pserver:readk@cvs.cpsc.ucalgary.ca:/ebe" />
      </modificationset>
      <schedule interval="30" intervaltype="relative">
         <ant buildfile="helloWorldCC/build.xml" target="cleanbuild" multiple="5" />
         <ant buildfile="helloWorldCC/build.xml" target="masterbuild" multiple="1" />
      </schedule>
      <log dir="C:\Development\cruisecontrol-2-0-2\main\example\helloWorldCC\logs" />
      <publishers>
         <currentbuildstatuspublisher file="C:\Development\cruisecontrol-2-0-
2\main\example\helloWorldCC\logs\currentbuild.txt" />
         <email mailhost="mailhost.ucalgary.ca" returnaddress="kdread@ucalgary.ca"
defaultsuffix="" buildresultsurl="http://localhost:8080/buildresults">
            <always address="kdread@ucalgary.ca" />
            <failure address="kdread@ucalgary.ca" />
         </email>
      </publishers>
      <plugin name="cvs" classname="net.sourceforge.cruisecontrol.sourcecontrols.CVS" />
      <plugin name="currentbuildstatusbootstrapper"
classname="net.sourceforge.cruisecontrol.bootstrappers.CurrentBuildStatusBootstrapper" />
      <plugin name="ant" classname="net.sourceforge.cruisecontrol.builders.AntBuilder" />
      <plugin name="email"
classname="net.sourceforge.cruisecontrol.publishers.LinkEmailPublisher" />
      <plugin name="currentbuildstatuspublisher"
classname="net.sourceforge.cruisecontrol.publishers.CurrentBuildStatusPublisher" />
      <plugin name="labelincrementer"
classname="net.sourceforge.cruisecontrol.labelincrementers.DefaultLabelIncrementer" />
   </project>

   <project name="hellocanada">
      <bootstrappers>
         <currentbuildstatusbootstrapper file="C:\Development\cruisecontrol-2-0-
2\main\example\helloCanadaCC\logs\currentbuild.txt" />
      </bootstrappers>
      <modificationset quietperiod="30" dateformat="yyyy-MMM-dd HH:mm:ss">
         <cvs cvsroot=":pserver:readk@cvs.cpsc.ucalgary.ca:/ebe" />
      </modificationset>
      <schedule interval="30" intervaltype="relative">
         <ant buildfile="helloCanadaCC/build.xml" target="cleanbuild" multiple="5" />
         <ant buildfile="helloCanadaCC/build.xml" target="masterbuild" multiple="1" />
      </schedule>
      <log dir="C:\Development\cruisecontrol-2-0-2\main\example\helloCanadaCC\logs" />
      <publishers>
         <currentbuildstatuspublisher file="C:\Development\cruisecontrol-2-0-
2\main\example\helloCanadaCC\logs\currentbuild.txt" />
         <email mailhost="mailhost.ucalgary.ca" returnaddress="kdread@ucalgary.ca"
defaultsuffix="" buildresultsurl="http://localhost:8080/buildresults">
            <always address="kdread@ucalgary.ca" />
            <failure address="kdread@ucalgary.ca" />
         </email>
      </publishers>
      <plugin name="cvs" classname="net.sourceforge.cruisecontrol.sourcecontrols.CVS" />
      <plugin name="currentbuildstatusbootstrapper"
classname="net.sourceforge.cruisecontrol.bootstrappers.CurrentBuildStatusBootstrapper" />
      <plugin name="ant" classname="net.sourceforge.cruisecontrol.builders.AntBuilder" />
      <plugin name="email"
classname="net.sourceforge.cruisecontrol.publishers.LinkEmailPublisher" />
      <plugin name="currentbuildstatuspublisher"
classname="net.sourceforge.cruisecontrol.publishers.CurrentBuildStatusPublisher" />
      <plugin name="labelincrementer"
classname="net.sourceforge.cruisecontrol.labelincrementers.DefaultLabelIncrementer" />
   </project>
```

```
</cruisecontrol>
```

Appendix E. Test Suite

See CDROM

Appendix F. Raw Data

See CD-ROM