Test Driven Development: The State of the Practice

Susan Hammond
Auburn University
3101 Shelby Center
Auburn University, AL 36849-5347
334-386-7575
sah0017@auburn.edu

David Umphress
Auburn University
3127E Shelby Center
Auburn University, AL 36849-5347
334-844-6335
davidumphress@auburn.edu

ABSTRACT
Test-Driven Development has been a practice used primarily in agile software development circles for a little more than a decade now. In software development circles, this is a relatively young and immature practice. How much acceptance has it gained it its short life span? What do we know about its effectiveness? This paper will explore these topics.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques – *Evolutionary prototyping, object-oriented design methodologies.*

General Terms
Design

Keywords
Test-driven development, agile software development

1. INTRODUCTION
Test Driven Development (TDD), also referred to as test-first coding or Test Driven Design, is a practice where a programmer is instructed to write production code only after writing a failing automated test case [1]. This approach offers a completely opposite view of the traditional test-last approach commonly used in software development, where production code is written according to design specifications, and, typically, only after much of the production code is written does one write test code to exercise it.

Kent Beck coined the moniker Test Driven Development (TDD) in his book *Extreme Programming Explained: Embrace Change* in 1999. Beck began using the practice in the early 90's while developing in Smalltalk. But, as Beck himself points out, the idea of writing test code before production code is not original to him. While test-first coding is easily done with modern, sophisticated IDE's where one can write test and production code and get immediate feedback, the practice was in use with earlier modes of programming. Beck describes being a child and reading a programming book that described test-first coding using paper tapes, where the programmer would produce the expected paper output tape first and then write code until the actual paper tape output matched the expected paper tape output.[2]. In fact, the earliest references to teams using a TDD-style approach are in NASA's Project Mercury in the early 1960's, which employed test-first practices in the development of the spaceship's software [3].

ACMSE'12, March 29–31, 2012, Tuscaloosa, AL, USA.

Test Driven Development has been described using general guidelines [4] [5], but does not provide rigorous guidance for its implementation. Some of Beck's initial instructions were as ambiguous as "never write a line of functional code without a broken test case." [2]

William Wake adapted a traffic light metaphor to give developers an idea of how the practice should work. It has been modified and adapted numerous times, but his original example followed a green, yellow, red pattern. Starting to write the test was the initial green light. When the test failed to compile because there was no production code available, the programmer was experiencing the yellow light. Once a stub was written for the production code, the test code would fail because the stub did not do anything, resulting in a red light. Once the production code was written and the test case passed, the developer would return to a green light status. [6]

The most common implementation of the traffic light example is shortened to Red, Green, Refactor [4]. The red light represents the failing, or possibly non-compiling, test code. Green light is the end-result of writing the minimum amount of code to make that test code pass. Refactoring is used to eliminate any code duplication or bad programming techniques that were introduced by getting to green as expeditiously as possible.

After the 2001 book introducing XP, Beck wrote a second book to provide more detail about the practice of TDD, summing it up with a five step process [4]:

1. Write a new test case.
2. Run all the test cases and see the new one fail.
3. Write just enough code to make the test pass.
4. Re-run the test cases and see them all pass.
5. Refactor code to remove duplication.

TDD is a fundamental practice in the Extreme Programming approach to developing software [1], but it can be used separately from the XP methodology. In a 2010 survey of self-described Agile practitioners, 53% of the 293 respondents used what the poll referred to as 'Developer TDD' for validating software. [7] Kent Beck polled attendees of the 2010 Leaders of Agile webinar regarding their use of TDD, and 50% of approximately 200 respondents indicated they used TDD [8]. In both of these polls, the respondents are pre-disposed to TDD usage because they develop software in an Agile manner.

In a more mainstream poll, Forrester reported that 3.4% of 1,298 IT professionals polled used a TDD approach to development. [9] The difficulty with these numbers is the question: "Please select the methodology that most closely reflects the development

process you are currently using." TDD was listed as a methodology along with Scrum, XP, Waterfall and other development methodologies. TDD is not really a methodology, but is a practice that could be used in conjunction with Scrum or XP, so this could result in an under-reported number of TDD practitioners. Alternately, since TDD is not well defined, it is possible that some respondents may be incorrectly claiming to use TDD, resulting in over-reporting. Regardless, the numbers indicate enough interest in TDD to treat it as a serious technique.

As TDD has entered into the software development vocabulary, academics have been assessing how to incorporate it into the university curriculum. In a 2008 paper, Desai and Janzen surveyed the results of 18 previous studies trying to assess the best time to introduce TDD into the curriculum. [10] Five studies were conducted with graduate students, five studies used upper classmen, seven used freshmen and one used a mix of all age groups. Overall, the results were more promising at the higher levels, but Desai and Janzen feel strongly the concepts can and should be introduced sooner. After assessing the methods used with the freshmen, Desai and Janzen concluded that the key to teaching TDD at the lower levels was reinforced instruction where the concept was introduced early on and then continually modeled throughout the course. They followed up the survey paper with an experiment to validate their hypothesis and a proposal for integrating TDD into the curriculum as early as the CS1/CS2 classes [11]. Kollanus & Isomottonen conducted an experiment with upperclassmen and also concluded that reinforced teaching with examples is a necessary approach, and that instruction should begin early in the curriculum. Their recommendation was to introduce it after the first programming course [12].

Numerous books have been written with extensive examples in an attempt to educate developers in the practice [4], [5], [13], [14]. Despite all these elaborations, TDD remains deceptively simple to describe, but deeply challenging to put into practice effectively. One reason for this may be found in an experiment conducted by Janzen and Saieden [15]. They found that while some programmers saw the benefits of the test-first approach, several of the programmers had the perception that it was too difficult or too different from what they normally do. Another study found that 56% of the programmers engaged in a TDD experiment had difficulty adopting a TDD mindset [16]. For 23% of these developers, the lack of upfront design was identified as a hindrance. A third study compiled answers from 218 volunteer programmers who participated in an on-line survey. In the study, programmers self-reported on how often they implemented (or in some cases deviated from) specific TDD practices. The study found that 25% of the time programmers admitted to frequently or always making mistakes in following the traditional steps in TDD. [17] Aniche and Gerosa observe that while the technique is simple and requires only a few steps, the programmer must be disciplined in his approach in order to garner the benefits. When the programmer is not disciplined, he does not experience the full benefit of TDD, and as a result may be less inclined to use the practice.

The TDD practice at a unit test level also leaves many questions unanswered. John McGregor states, "Design coordinates interacting entities. Choosing test cases that will adequately fill this role is difficult though not impossible. It is true that a test case is an unambiguous requirement, but is it the correct requirement? More robust design techniques have validation methods that ensure the correctness, completeness, and consistency of the design. How does this happen in TDD?" [18]

After producing a book with in-depth empirical research on the subject of TDD, Madeyski wrote, "a more precise definition and description of the TF (Test First) and TL (Test Last) practices, as well as a reliable and automatic detection of possible discrepancies from the aforementioned techniques, would be a valuable direction of future research." [19].

2. PREVIOUS STUDIES

Kollanus performed a systematic literature review of TDD research where empirical methods were employed [20]. She found 40 papers that report some kind of empirical evidence on TDD as a software development method. These empirical exercises resulted in inconsistent and often contradictory results from one study to the next.

The studies focused on different aspects of the practice and the resulting code, including defect density, programmer productivity, and object cohesion and coupling. In many of the articles, evidence was reported on TDD related to 1) external quality, 2) internal code quality, and/or 3) productivity. Kollanus summarized her findings with regard to these three factors by stating there is [20]:

1. weak evidence of better external quality with TDD.
2. very little evidence of better internal quality with TDD.
3. moderate evidence of decreased productivity with TDD.

A confounding complication Kollanus experienced in performing the systematic review was that, frequently, the TDD process was very briefly or not described within the source papers, so it is very difficult to compare the results. As mentioned previously, TDD has not been rigorously defined, and if the studies do not express how it was practiced, there is no guarantee the results between studies can be accurately compared. Another possible reason for the wide variation of findings may be an inconsistent understanding or application of TDD by the people participating in each individual study [21]. These studies have a "process compliance problem" in that there is no validation that the participants actually used the TDD practices as set forth in the individual experiments.

Approximately half of these TDD experiments were conducted with college students who had training in software development and perhaps software design. Frequently, these students were then provided a brief introduction to TDD and given a toy solution to develop under a specific time constraint. Muller and Hofer found that studies such as these that use novices to perform TDD are not easily generalized because of inconsistent process conformance [22].

Even experienced programmers do not conform to the process consistently. Examples abound in Aniche's survey of TDD programmers. For instance, the survey revealed that, on a regular or frequent basis, 33% of programmers do not begin with the simplest test, a violation of one of the most fundamental concepts of TDD [17].

Another complication related to the results of previous empirical studies lies in the skill and experience of the subjects. Ward Cunningham's statement "test-first coding is not a testing technique" [2] illustrates a major misunderstanding of TDD. To use the technique properly, the programmer must be developer and designer at the same time. This type of task is not for every

developer at every skill level. Designing and developing code being led by specific tests requires a certain degree of maturity in understanding how software interacts.

Boehm provides a classification system for personnel with regard to their ability to perform within a particular framework [23]. Within his five levels, he asserts that developers in the top two have appropriate skills to function on an Agile team. He feels that developers in the bottom two require a much more structured environment, and those in the middle level can perform well on an Agile team if there are enough people from the top two tiers to provide them guidance. The implication is that to be successful using an Agile approach, including using TDD to write code, requires people who are relatively experienced and very talented and flexible.

3. TDD Extensions

3.1. Agile Specification-Driven Development

This is a technique that combines both TDD and Design-by-Contract (DbC) [24]. DbC is a concept introduced by Bertrand Meyer which allows a language to explicitly declare pre- and post-conditions for a method and invariants for a class [25]. Meyer created the programming language Eiffel to support DbC [26], but more mainstream languages are being extended to include DbC as well. Leavens & Cheon [27] introduce the Java Modeling Language as a way to support DbC in Java, and Microsoft has introduced their version of DbC called Code Contracts into the .NET framework beginning with Visual Studio 2008 [28].

Meyer also introduced an approach to software development called the Quality First Model [29]. His approach pre-dates XP by 2-3 years, yet shares many similarities with it. The Quality First Model values working code above all else, but it also takes advantage of formal methods for development and tools that allow models to generate code and vice versa.

Ostroff, et al, summarize the quality-first approach as [24]:

1. Write code as soon as possible, because then supporting tools immediately do syntax, type, and consistency checking.
2. Get the current unit of functionality working before starting the next. Deal with abnormal cases, e.g., violated preconditions, right away.
3. Intertwine analysis, design, and implementation.
4. Always have a working system.
5. Get cosmetics and style right.

The Quality First Model has at its core the DbC approach, and it shares some commonalities with conventional TDD practice. For instance, the emphasis in step 2 is to finish one unit of functionality before moving on to the next. This is also a tenet of TDD. However, a big difference is also immediately obvious in the second step because TDD asks the developer to focus on the most common case, whereas DbC expects the developer to focus on abnormal cases first.

In concept, both TDD and DbC are specification tools. Ostroff, et al., point out that the two approaches can actually be complementary. They state that using TDD early in the development cycle allows the developer to describe the functionality and formalize the collaborative specifications, while adding contracts later provides an easier way to document and enforce pre- and post-conditions [24].

This approach is being explored further in blogs on the web by Matthias Jauernig [30] and David Allen [31]. The introduction of Code Contracts in .NET stimulated the interest of combining the use of TDD and DbC for these bloggers. Specifically, they are encouraging developers who currently use TDD to incorporate the use of Code Contracts into their development practices.

3.2. Behavior-Driven Design (BDD)

BDD was first introduced by Dan North [32]. The initial reason for the name shift from *Test*-Driven Development to *Behavior*-Driven Development was to alleviate the confusion of TDD as a testing method versus TDD as a design method. Astels notes, "Behavior-Driven Development is what you were doing already if you're doing Test-Driven Development very well." [33] By changing the developer's focus to the behavior of the code, it was posited that the developer would shift his mind-set away from validation to the design aspects of the technique.

To illustrate the need for BDD, the BddWiki provides a life-cycle to the learning and adoption of TDD [34]:

1. The developer starts writing unit tests around their code using a test framework like JUnit or NUnit.
2. As the body of tests increases the developer begins to enjoy a strongly increased sense of confidence in their work.
3. At some point the developer has the insight (or is shown) that writing the tests before writing the code, helps them to focus on writing only the code that they need.
4. The developer also notices that when they return to some code that they haven't seen for a while, the tests serve to document how the code works.
5. A point of revelation occurs when the developer realizes that writing tests in this way helps them to "discover" the API to their code. TDD has now become a design process.
6. Expertise in TDD begins to dawn at the point where the developer realizes that TDD is about defining behaviour rather than testing.
7. Behaviour is about the interactions between components of the system and so the use of mocking is fundamental to advanced TDD.

Rimmer asserts that most developers, with some assistance, reach step 4, but very few progress beyond it. He does not provide any empirical research to support this learning curve; it is merely based on observation and experience.

Proponents of BDD have developed new testing frameworks that change the nomenclature in order to support the approach and to get developers to think beyond the viewpoint that TDD is about testing. The rationale is based on the Sapir-Whorf theory that the language used influences your thought [33]. Dan North developed a Java-based framework called JBehave that removed any reference to testing and modified the language to focus on behavior. For example, instead of using the testing terminology 'assertion', the framework uses 'ensureThat'; for example

'ensureThat(actual, equalTo(expected); [35]. For Ruby, the framework is called rSpec, and it replaces the 'assertion' terminology with a sentence-type structure where the actual object is the subject and the assertion statement is the verb; for example, 'actual.should.equal expected' [33].

Some key aspects of BDD include [32]:

1. The word 'test' should not be included in the test name.
2. Test method names should be sentences and should begin with the word 'should', for instance, 'shouldTransferBalanceFromAccountWithSufficientFunds'.
3. Tests should not correspond one-to-one with classes, but should focus on behaviors. This makes it easier to change the structure as classes grow and need to be broken out into separate classes.

3.3. Acceptance Test-Driven Development (ATDD)

Traditional TDD focuses on the smallest test that could possibly be written [4]. The developer begins with the failing test case and writes code until the test case passes. Test cases are written to assist in the development of various low-level requirements, but there is no mechanism by which a developer can put the requirements in context. Approaching development from this low-level viewpoint can produce results like Madeyski experienced, where programmers' code was significantly less coupled than comparable Test-Last code, but the percentage of acceptance tests passed was not significantly impacted [19]. One could say the code quality was better, but the code didn't necessarily meet the high-level requirements of the project.

In order to ensure that the code is actually providing the functionality that the users desire, some software engineers have been focusing on a higher level test to set the context for the overall development. Beck mentions the concept of application test-driven development [4], whereby the users would write the tests themselves. There have since been off-shoots of TDD that focus on allowing the customer to write acceptance tests that the developers can use to see if their software is providing the correct functionality. As described above, BDD is one of those approaches. ATDD is another.

Ward Cunningham introduced the concept of FIT tables in 2002 [36]. The general idea is that the customers enter test data into a table using a standard word processor or spreadsheet, and the developers write a test fixture that uses the data to test the software under development [37]. This approach has been extended with multiple variations and tools, including Fitnesse [38], which allows the customer to enter test cases into a wiki and then translates the data through a FIT client and server process into standard FIT fixtures, to get a FIT environment up and running very quickly. Custom fixtures written by the development team complete the configuration [38]. Other solutions include Easy Accept, which is a script interpreter and runner offering another approach to automating the creation of the acceptance tests [39] and AutAT, which focuses on acceptance testing of web applications [40].

Beck objects to ATDD because tests would not be under the control of the developers. He predicts that users and/or organizations will not be willing or able to take on this responsibility in a timely-enough manner for the software development. He objects to the lag in time between test and feedback, preferring the more immediate feedback of TDD [4].

3.4. Growing Objects, Using Tests

Freeman & Pryce advocate a combination of the developer control and immediate feedback of TDD with the functional focus of ATDD [14]. The developer begins with a functional test case that is derived from a user story that represents a system requirement. This differs from the Acceptance Test-Driven Development approaches in that the tests are written by the developers, not by the end users. It puts the onus on the developer to make sure that he understands the user stories sufficiently to create the correct test. But it addresses Beck's concern that ATDD would create delays or cause the developers to lose needed control.

The fundamental TDD cycle typically looks like this [14]:

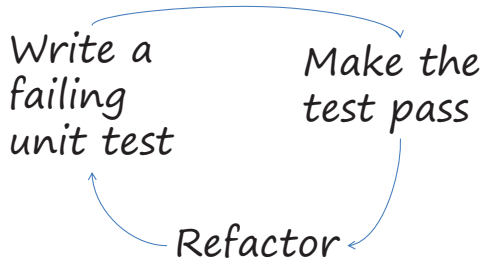

Figure 1: Fundamental TDD Cycle

By adding the functional test, the cycle now looks like this [14]:

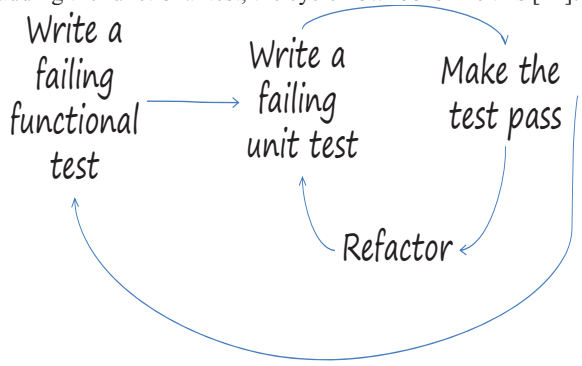

Figure 2: Functional Test TDD Cycle

Freeman & Pryce's book *Growing Objects, Using Tests* is a long example of how to approach development in this fashion.

4. FUTURE WORK

TDD has many proponents and detractors. It is perceived as a counter-intuitive approach to developing software. Various

research studies have yielded conflicting results. More research is needed, particularly as it relates to two significant questions: 1) . "How do we know we are developing the correct functionality?"[18], and 2) "How can TDD be given a more precise definition and description?" [19]. In other words, TDD focuses on small, simple tests, but how does the developer know that he is developing the correct functionality, and when does the developer know that he is finished?

We propose a new approach to address these issues. Using means-ends analysis, a problem-solving approach that originated in the AI arena, TDD can be performed using a more structured approach with the end goal in mind. In the means-ends analysis technique, a current state and a goal state are defined, and the actor chooses an action that should reduce the difference between the two. [Wiki 2011b]

The Means-Ends Analysis Approach (MEAA) uses a functional test case to represent the goal state. When this functional test passes, then the programmer has completed the development of a functional system requirement. In support of that functional test, the developer is expected to follow the traditional TDD approach to develop each new component as the need for it arises. As new components are identified, their unit-level tests become intermediate goals. Each TDD cycle represents an action to get the developer closer to the desired state of fulfilling the functional requirement.

Typically in Agile development, system requirements are defined with user stories, a short description of the functionality required. For MEAA , a developer chooses a user story and develops a functional test that represents the requirement. A high-level design is drawn up informally to provide the developer with an initial roadmap for the high-level test. This design grows as the system grows, incorporating existing classes with the anticipated classes for the next iteration and illustrating their anticipated interactions. The remaining user stories remain on the "To-Do list" until the first functional test passes completely.

In addition to functional tests, the developer's To-Do list includes developer tests. Developer tests are derived from system interactions of which the user is typically unaware. In addition, as the programmer is tracking through the TDD process, it may become apparent that the system under development would have a better design by creating factories or following other design patterns that result in a need for an "end-to-end" test similar to a functional test. These developer tests would also go on the To-Do list.

The MEAA borrows somewhat from the Design by Contract approach by focusing on the objects and subsequent interfaces defined in the functional test. This guides the programmer to take an action that will allow him to move from the current state and closer to the goal state.

Future work on the MEAA includes a complete codification of the Means-Ends Analysis Approach. This will address Madeyski's search for "a more precise definition" of TDD. Experiments will be performed to determine if MEAA leads the developer to a better coverage of the user requirements.

5. REFERENCES

[1] Beck, K. 2000. *Extreme Programming Explained.* Addison-Wesley.

[2] Beck, K. 2001. Aim, Fire. *Software,* 18(5):87-89, Sept.-Oct. 2001.

[3] Larman, C., & Basili, V. R. 2003. Iterative and Incremental Development: A Brief History. *Computer* (June 2003), 47-56.

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

[5] Astels, D. 2003. *Test-Driven Development: A Practical Guide* . Pearson Education, Inc., Upper Saddle River, NJ.

[6] Wake, W. 2001. The Test-First Stoplight. http://xp123.com/articles/the-test-first-stoplight/

[7] Ambler, S. 2010. How Agile Are You? 2010 Survey Results. http://www.ambysoft.com/surveys/howAgileAreYou2010.html

[8] Beck, K. 2010. CD Survey: What practices do developers use? http://www.threeriversinstitute.org/blog/?p=541

[9] West, D., & Grant, T. 2010. *Agile Development: Mainstream Adoption Has Changed Agility.* Cambridge: Forrester.

[10] Desai, C., & Janzen, D. S. 2008. A Survey of Evidence for Test-Driven Development in Academia. *inroads - SIGCSE Bulletin , 40* (2), 97-101.

[11] Desai, C., & Janzen, D. S. 2009. Implications of Integrating Test-Driven Development into CS1/CS2 Curricula. *SIGCSE'09* (pp. 148-152). Chattanooga, TN: ACM.

[12] Kollanus, S., & Isomottonen, V. 2008. Test-Driven Development in Education: Experiences with Critical Viewpoints. *ITiCSE* (pp. 124-127). Madrid, Spain: ACM.

[13] Koskela, L. 2008. *Test Driven: Practical TDD and Acceptance TDD for Java Developers.* Greenwich, CT: Manning Publications Co.

[14] Freeman, S., & Pryce, N. 2010. *Growing Object-Oriented Software, Guided By Tests.* Boston, MA: Pearson Education, Inc.

[15] Janzen, D. S., & Saieden, H. 2007. A Leveled Examination of Test-Driven Development Acceptance. 29th International Conference on Software Engineering. IEEE.

[16] George, B., & Williams, L. 2004. A structured experiment of test-driven development. *Information & Software Technology , 46* (5):337-342, 2004.

[17] Aniche, M. F., & Gerosa, M. A. 2010. Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers. *Third International Conference on Software Testing, Verification, and Validation Workshops* , 469-478.

[18] Fraser, S., Astels, D., Beck, K., Boehm, B., McGregor, J., Newkirk, J., et al. 2003. Discipline and Practices of TDD (Test Driven Development). *OOPSLA '03* (pp. 268-269). Anaheim, CA: ACM.

[19] Madeyski, L. 2010. *Test-Driven Development: An Empirical Evaluation of Agile Practice.* Berlin: Springer-Verlag.

[20] Kollanus, S. 2010. Test-Driven Development - Still a Promising Approach? *2010 Seventh International Conference on the Quality of Information and Communications Technology* (pp. 403-408). IEEE.

[21] Kou, H., Johnson, P. M., & Erdogmus, H. 2010. Operational definition and automated inference of test-driven development with Zorro. *Automated Software Engineering* , 57-85.

[22] Muller, M. M., & Hofer, A. 2007. The effect of experience on the test-driven development process. *Empirical Software Engineering*, 593-615.

[23] Boehm, B., & Turner, R. 2004. *Balancing Agility and Discipline: A Guide for the Perplexed.* Pearson Education, Inc.

[24] Ostroff, J.S., Makalsky, D., & Paige, R.F. 2004. Agile Specification-Driven Development. In J. Eckstein, & H. Baumeister, *Lecture Notes in Computer Science,* 104-112. Berlin, Germany: Springer-Verlag.

[25] Meyer, B. 1991. Design by Contract. D. Mandrioli, & B. Meyer (Eds.), *Advances in Object-Oriented Software Engineering*, 1-50. Prentice Hall.

[26] Meyer, B. 1991. *Eiffel: The Language.* Prentice Hall.

[27] Leavens, G.T., & Cheon, Y. 2006. *Design by Contract with JML*. The Java Modeling Language: http://www.eecs.ucf.edu/~leavens/JML//jmldbc.pdf.

[28] Ricciardi, S. 2009. *Introduction to Microsoft Code Contracts with Visual Studio 2008*. Steffano Ricciardi: On Software Development and Thereabouts: http://stefanoricciardi.com/2009/06/26/introduction-to-microsoft-code-contracts-with-visual-studio-2008/

[29] Meyer, B. 1997. Practice to Perfect: The Quality First Model. *Computer*, 102-106.

[30] Jauernig, M. 2010. *Specification: By Code, Tests, and Contracts.* Mind-driven Development: http://www.minddriven.de/index.php/technology/dot-net/code-contracts/specification-by-code-tests-and-contracts

[31] Allen, D. 2010. *More on the synergy between Test-Driven Development and Design by Contract*. Software Quality: http://codecontracts.info/2010/02/02/more-on-the-synergy-between-test-driven-design-and-design-by-contract/

[32] North, D. 2006. Introducing BDD. *Better Software* (March 2006).

[33] Astels, D. 2006. Google TechTalk: Beyond Test Driven Develoment: Behavior Driven Development. http://www.youtube.com/watch?v=XOkHh8zF33o.

[34] Rimmer, C. 2010. *Introduction*. Behaviour-Driven Development: *http://behaviour-driven.org/Introduction*

[35] jBehave. (no date). *Candidate Steps.* jBehave: http://jbehave.org/reference/stable/candidate-steps.html

[36] *Framework for Integrated Test*. 2011. Wikipedia: http://en.wikipedia.org/wiki/Framework_for_Integrated_Test

[37] Shore, J. 2005. *Introduction to Fit.* Fit Documentation: http://fit.c2.com/wiki.cgi?IntroductionToFit

[38] Martin, R. C., Martin, M. D., & Wilson-Welsh, P. 2008. *OneMinuteDescription*. Fitnesse.UserGuide: http://www.fitnesse.org/FitNesse.UserGuide.OneMinuteDescription

[39] Sauve, J. P., Abath Neto, O. L., & Cirne, W. 2006. EasyAccept: A Tool to Easily Create, Run and Drive Development with Automated Acceptance Tests. *AST* , 111-117.

[40] Schwarz, C., Skytteren, S. K., & Ovstetun, T. M. 2005. AutAT - An Eclipse Plugin for Automatic Acceptance Testing of Web Applications. *OOPSLA '05* (pp. 182-183). San Diego, CA: ACM.

[41] *Means-ends analysis*. 2011. Wikipedia: http://en.wikipedia.org/wiki/Means-ends_analysis