e-ISSN: 2349-9745
p-ISSN: 2393-8161
Scientific Journal Impact Factor (SJIF): 1.711
International Journal of Modern Trends in Engineering and Research
www.ijmter.com

A Study: The Analysis of Test Driven Development And Design Driven Test

Shalini Priyadarshini[1], Madhu Nashipudimath[2]
[1,2]*Computer Department, PIIT*

Abstract—There are various types of testing methods. In this paper, we have discussed TDD and DDT. Test driven development method is a type of agile methodology. In this method, test drives the design. Basically, TDD is a unit testing. It is a test-first approach method[15]. This method can be a never ending process, because we are writing the test first, then writing the code to make the test pass. If test fails then again we will change the code and then we will test again. Using this method, defect rate can be reduced to 50%. But DDT cannot be a never ending process, because we design first, then we implement that design any way. TDD is more complex to implement, while DDT is a bit easier

Keywords-Test Driven Development; Design Driven Test; agile software; extreme programming; acceptance testing; unit testing; test-first approach.

I. INTRODUCTION

Software Engineering is applying Technical skill at each level of the of the software development process. SDLC Model is a framework that describes the activities performed at each stage of a software development project. Many phases are included within Software Engineering, such as Requirement Analysis phase, Designing phase, Coding or Development phase, Testing phase, and maintainence phase.

Software engineering projects are requiring many software engineers so that they can coordinate their efforts to produce a large software system. Software development method is known as a subset of the system development life cycle. Some common methodologies are there. Some model comes under **Traditional Model** some comes under **Agile methodology**[1][20].

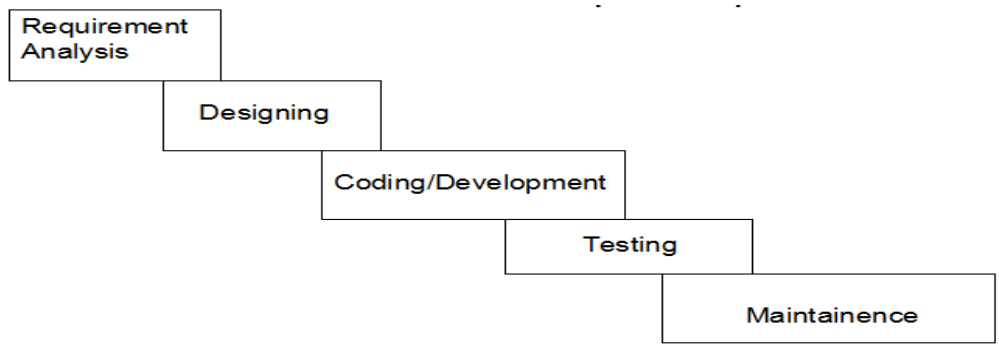

Fig.1.1 : Software Engineering

Table 1: *Software Project Development Model*

Software Project Development Model	
Traditional Methodology	**Agile Methodology**[1][20]
Waterfall Model	XP (Extreme Programming)[21][22][23][13][7]
Spiral Model	Crystal Clear[9]
V-Model	Scrum[10][5]
Structured Evolutionary Prototyping Model	RAD (Rapid Application Development)
Incremental Model	FDD (Feature-Driven Development)
	ASD (Adaptive Software Development)
	DSDM (Dynamic Software Development Model)
	RUP (Rational Unify Process)
	TDD (Test Driven Design)

II. TEST DRIVEN DEVELOPMENT

TDD is among the corner-stone practices of the XP. TDD is said to be the mixture of test-first development (in which unit tests are written before the implementation code needed to pass those tests) and refactoring (which includes restructuring a piece of code that passes the tests in order to reduce its complexity and improve its clarity, understandability, extendibility and maintainability). The use of TDD can bring improvements in code quality and productivity[17]. There are three possible outcomes ---positive, negative and neutral. These possible outcomes have been reported for both quality and productivity improvements obtained with TDD.

TDD is all about testing at the detailed design level. It is separated from Extreme Programming, so TDD lost a valuable companion- Acceptance Testing. It is also called test-first approach. But end up with an awful lot of unit tests of questionable value, and it's tempting to skip valuable parts of the analysis ans design thought process because "Code is Design". Unit testing will become too hard, if the project has been written in such a ways to make the code difficult or nearly impossible to test. TDD is a technique for improving the software's internal quality. TDD well-written code has good design, smooth evolution.

2.1 TDD Process

The term refactoring is used because the last step is about transforming the current design toward a better design. TDD can be described as "red-green-refactor cycle". The steps of Red-green-refactor cycle are as follows:-

1. Design and add a test
2. Run all tests and see the new one fail (red)
3. Add enough implementation code to satisfy the new test
4. Run all tests, repeat 3 if necessary until all tests pass (green)
5. Occasionally to improve code structure, we refactor
6. To ensure that all the tests pass, run all tests after refactoring .

First we write a test. Then we write code to make the test pass. Then we will find the best possible design among whatever we have gone through - refactoring (Relying on the existing tests to keep us

from breaking things while we are at it) .TDD is a development and design technique that helps us build up the system incrementally, knowing that we're never far from a working baseline.

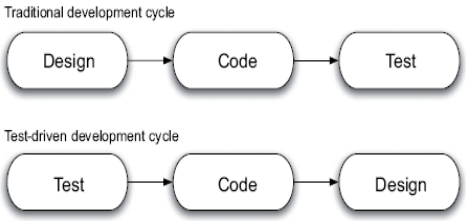

Figure 2.1.1 : Test-Code Refactor

Figure 2.1.2 : Test-Code Refactor

2.2. Characteristics of TDD

Top ten characteristics of TDD are as follows:-

- Tests drive the design.[20][5]:The first item, that of tests driving the design during coding, rather than an up-front process of thinking and modeling driving the design, is the defining characteristic of TDD.
- There is a Total Dearth of Documentation: Documentation is not an inherent part of the TDD process. The mantra that "the code is the design" means that very little design documentation may be written.
- Everything is a Unit Test: If it's not in JUnit, it doesn't exist. The mindset of a test-first methodology is to write a test first. Then add something new to make the test pass.
- TDD Tests are not quite Unit Tests[5]: TDD tests (also called programmer tests) have their own purpose; therefore, on a true test-first project[18] the tests will look subtly different from a "classical" fine-grained unit test. A TDD unit test might test more than a single unit of code at a time.
- Acceptance Tests provide feedback against the requirement: Acceptance tests form part of the specification. Unless we have layered another process on top of TDD (such as XP or Acceptance TDD), we will be driving unit tests directly from the requirements.
- TDD lends confidence to make changes[19]: Confidence to make the changes that is "design by refactoring". A green bar means "all the tests what we have written are not failing."
- Design emerges incrementally: First you write a test, and then we write some code to make the test pass. Then to improve the design, we refactor the code without breaking any tests that have been written so far. Thus, the design emerges as we grow the code base incrementally through the test/code/refactor cycle.
- Some Up-front design is Fine: It is definitely within the TDD "doctrine" to spend time up-front thinking through the design.
- TDD produces a lot of tests: The test-first philosophy underlying TDD is that we first write a test that fails, before we write any code. Then we can write the code to make the test pass. The net result is that aggressive refactoring, which will be needed by the bucket-load. TDD also doesn't really distinguish between "design level" (or solution space) tests and "analysis level" (or problem space) tests[19].
- TDD is too damn difficult: From the Department of Redundancy, TDD is Too Damn Difficult. The net effect of following TDD is that everything and its hamster gets a unit test of its own.

TDD has an image of a "lightweight" or agile practice.

2.3 Login Example

Understand the Requirement.

1. As a website owner, we want to provide a secure login capability so we can gain revenue by charging for premium content.

2. As a website user, we want to be able to log in securely so we can access the premium content.

3. As a website owner, we want the system to lock a user account after 3 failed login attempts.

4. As a website user, we want our password entry to be masked to prevent casual shoulder-surfers from seeing it.

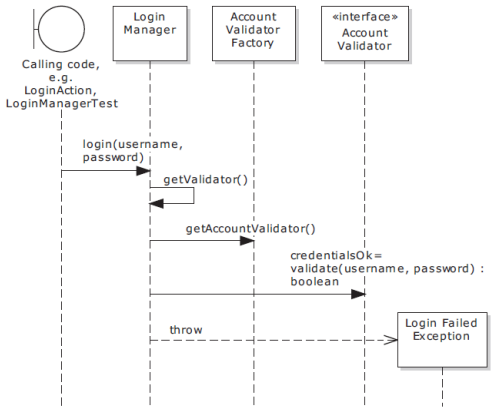

Fig. 2.3.1: Sequence diagram for website login

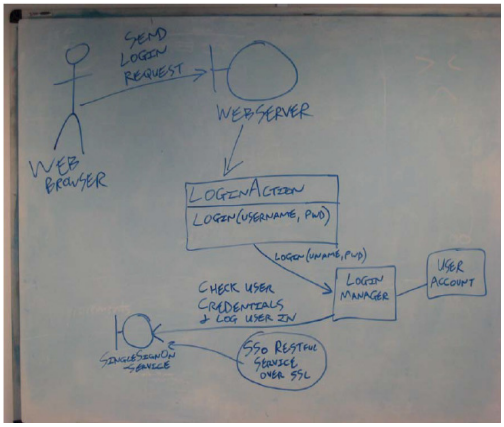

Fig.2.3.2: Initial design sketch for login

In fig.2.3.1, user is sending login request. Login Manager will check whether the pattern of username and password is correct. If pattern correct, it will check in the account database lists for the existence of the account. Then it will check for the username and password, if the username or password is incorrect, it will throw Login Failed Exception. If it throws Login Failed Exception 3 times, then the account will be locked. The same thing we can see in the fig.2.3.2.

In the fig.2.3.3, if its not failing, we can see the green bar that occurs. In the fig.2.3.4, we have created a mock SSO service for test environment and SSO service is for live environment. User send login request. Login Manager will accept the username and password if it is correct. It will validate the Mock SSo service for a test environment and it will validate the SSO service for a live environment. We have created such code which will run before and after each test case to set up the mock validator an the Login Manager. This set up code will be repeated each time, i.e. Refactoring.

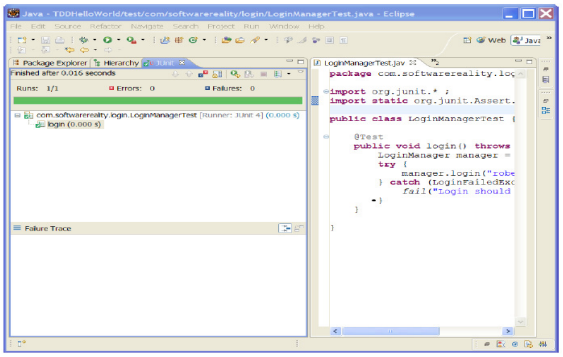

Fig.2.3.3: Green bar

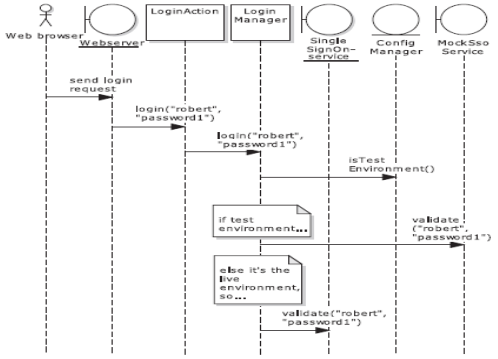

Fig.2.3.4: A common anti-pattern

III. DESIGN DRIVEN TEST

With Design Driven Testing we generate unit test code from UML/SysML models before code is written[21]. DDT includes the Developer Testing as well as the Quality Assurance Testing. DDT is a solution for answer to the problems that become apparent with other testing methodologies. Also, it is very much answer to the much bigger problems that occur when: no testing is done or no automated tests are written; some testing is done or some tests are written but it's all a bit aimless and adhoc; too much testing is done or too many low-leverage tests are written. With DDT, tests are driven directly from the conceptual design. Design Driven Test inherently promotes good design and easily testable code. Controller tests is developer tests that look like unit tests, which operate at the conceptual design level. They provide a highly beneficial glue between analysis (the problem space) and design (the solution space). The purpose of Design Driven Test is to prove systematically that the design fulfills the requirements and the code matches up with the design.

3.1 Structure of DDT

There are four principle tests. These are: Unit test, Controller tests, Scenario tests, Business requirement tests. Unit tests are fundamentally rooted in the design/solution/implementation space[11][21]. Controller tests are sandwiched between the analysis and design spaces. Controller tests also help to provide a bridge between the analysis and design spaces. Scenario tests are manual test specs containing step-by-step instructions for the testers to follow. Business requirement tests are manual test specs. They facilitate the "human sanity check" before a new version of the product is signed off for release into the wild.

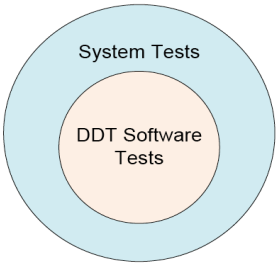

Fig.3.1.1: DDT System[11][21]

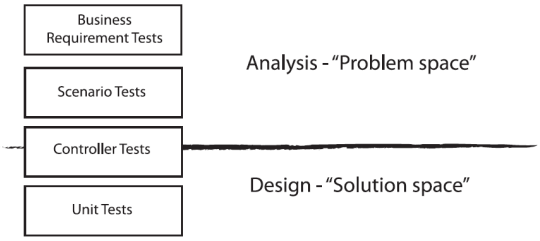

Fig.3.1.2 : Structure of DDT

Test Granularity is the tests vary in granularity and in the number of automated tests that are actually written. Fig.3.1.3 shows that each use case is covered by exactly one test class. It is actually one test per use case scenario. Use cases are divided into controllers, and each controller test covers exactly one controller. The controllers are then divided into actual code functions and the most important method gets unit test method.

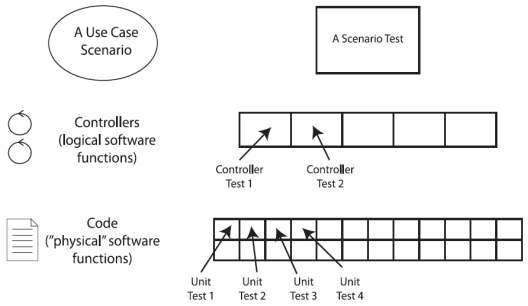

Fig. 3.1.3: Scenario Test[20]

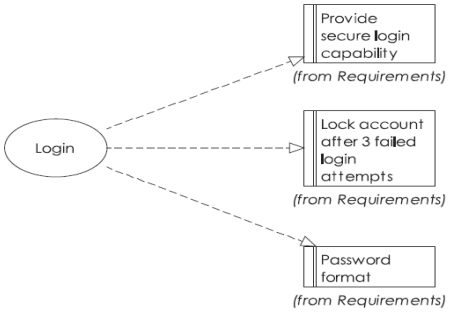

Fig.3.3.1:Login use case satisfies requirement

3.2 Characteristics of DDT

They are as follows:-

- DDT Includes Business Requirement Tests: We will typically create "requirement" elements in your UML model, and allocate those requirements to use cases.
- DDT Includes Scenario Tests: A key feature of DDT is that you create scenario tests by expanding use cases into "threads." DDT includes a **thread expansion** algorithm that helps us to create these acceptance tests automatically from the use cases.
- Tests Are Driven from Design: The tests are driven from the design, and, therefore, the tests are there primarily to validate the design. That said, there's more common ground between this and TDD. With DDT the tests are not the design, and the design is not driven from the tests.
- DDT Includes Controller Tests: DDT because the ICONIX Process supports a "conceptual design" stage, which uses robustness diagrams as an intermediate representation of a use case at the conceptual level.
- DDT Tests Smarter, Not Harder: DDT takes a test smarter, not harder philosophy, meaning tests are more focused on code "hot spots." With true DDT, we don't take an incremental approach to design; therefore, we don't need to refactor nearly as much, and we don't need eleventy billion tests poking into every nook and armpit of the deepest recesses of your code. Instead, the tests are targeted precisely at the critical junctures of the code that we have highlighted in the course of the design and requirements modeling as being the most cost-effective areas to test.
- DDT Unit Tests Are "Classical" Unit Tests: DDT unit tests are driven directly from the detailed design. You'll write one or more unit tests for each method you've drawn on a sequence diagram
- DDT Test Cases Can Be Transformed into Test Code: There are a few steps involved in DDT. But at the high level, we first identify test cases from the design, and then we transform the tests into UML classes, from which actual test code can be generated.
- DDT Test Cases Lead to Test Plans: Once we identify a test we can populate it with things like inputs, outputs, descriptions, and expected results. We can essentially generate test plans from our cases. These will get forward-generated into xUnit test code, and we can also generate test plan reports.
- DDT Tests Are Useful to Developers and QA Teams: DDT test cases are useful to both developers and QA teams—and the customer and business analysts, too. Since the scope of DDT includes identifying requirements tests and acceptance tests, it's quite naturally relevant to quality assurance folks. So, while still serving the "green bar" needs of comprehensive unit

testing by the development team, it's also going to be useful for independent QA teams.

- DDT Can Eliminate Redundant Effort: One of the big benefits of doing a model-based design that includes identifying requirements and allocating those requirements to use cases, as opposed to refactoring a system into existence, is that you can eliminate a lot of redundant work. A little later in this chapter you'll see an example of a requirement (Validate Password syntax) that is germane to both a Create Password use case and to Login. It's easy to see how someone could assign one of these "stories" to one team and the other to a different team, each of whom could come up with a separate (and, we hope, similar) definition for the requirement and two redundant sets of tests. This sort of duplicate effort is guarded against with DDT, as the requirement (and associated tests) would be defined once, and the tests would be re-used.

3.3 Login Example Implemented using DDT[20]

A simple Login Use case, which satisfies requirements as show in the Fig.3.3.1[11][21]

Step 1: Create a robustness diagram (conceptual design).

Step 2: Create the controller test cases from your robustness diagram.

Step 3: Add scenarios to each test case.

Step 4: Transform your controller test cases into test classes.

Step 5: Generate the controller test code.

Step 6: Detailed design: create a sequence diagram for the use case.

Step7: Create unit test cases from your sequence diagram, and cherry-pick the non-redundant ones.

Step 8: Fill in the test code using the generated comments as guidance, and write the code to make the tests pass.

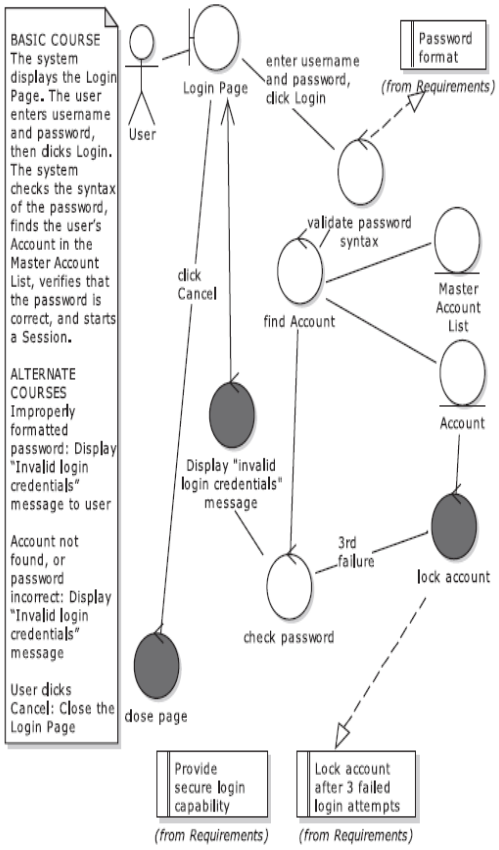

Fig.3.3.2: Complete Robustness Diagram

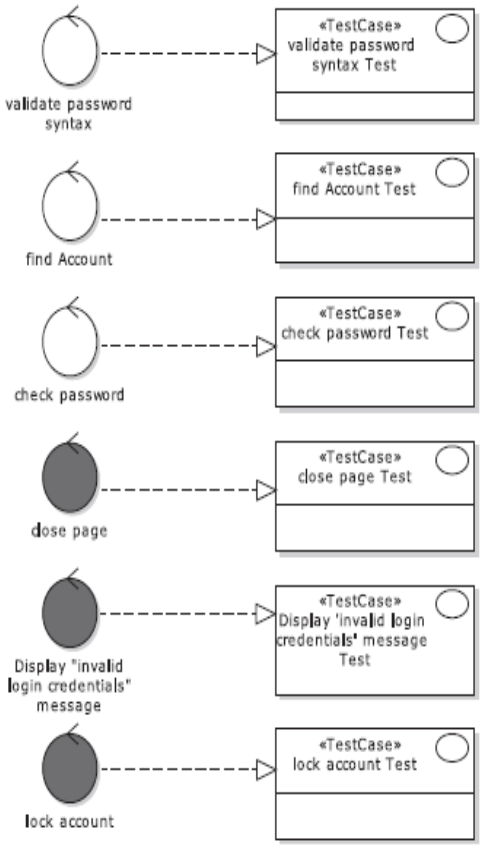

Fig.3.3.3: Generated test case for Controller

In the fig.3.3.2, user send the login request by entering the username and password. Password format will be checked and if the password format is correct, the account will be searched from the master account list. If the account exists, it will check the password whether it is correct or not. If the password fails, it will display the message. The next time again if the password fails, it will display again the same message. But if for the third time password fails, it will lock the account. By clicking on cancel button, the page will be closed.

In the fig.3.3.3, we have created the test cases for all the requirements. The total of 6 test cases are there, out of which 3 are positive test cases and 3 are negative test cases.

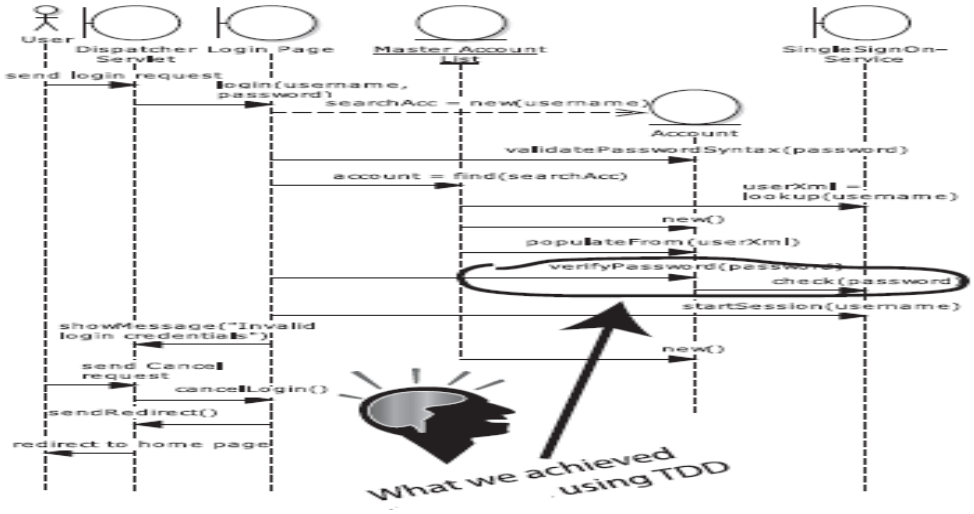

Fig.3.3.4: Designed, Documented and Test-driven vs. Simply test-driven[20]

In the fig.3.3.4, the same thing we got what we achieved using TDD.

IV. COMPARISION

1) TDD drives the design, whereas DDT is driven from the design.[20]

2) In TDD, code is design & tests are documentation; whereas in case of DDT, design is design, code is code & tests are tests[3].

3) In TDD, one can end up with a lot of tests; whereas DDT is not Test-Harder approach, but Test-smarter approach.

4) In TDD, Test-first[3] tests different from fine-grained unit test; whereas in DDT, Unit test is there to validate a single method.

5) In TDD, Acceptance tests can only be done if and only if part of another tests is mixed like ATTD (Acceptance Test Driven Design); whereas in DDT, Acceptance tests are manual test specs.

6) TDD is much finer-grained when it comes to design, whereas DDT is viewed as pretty fine-grained

7) In TDD, a green bar means all the test written are not failing[8]; whereas in DDT, a green bar means all the critical design junctures implemented are passing

8) In TDD, first write the test & then write the code; whereas in DDT, one can write code before the test and vice-versa.

V. CONCLUSION

Basically, TDD is a unit testing. It is a test-first approach method. This method can be a never ending process, because we are writing the test first, then writing the code to make the test pass. If test fails then again we will change the code and then we will test again. But this cannot happen in the case of DDT, because we design first, then we implement that design any way. TDD is more complex to implement, while DDT is a bit easier.

REFERENCES

[1]Aziz Nanthaamornphong and Jeffrey C. Carver, Hope A. Michelsen, Damian W.I. Rouson, Building CLiiME via Test-Driven Development: A Case Study, JUNE 2014
[2]Eduardo Guerra, National Institute for Space Research, Brazil ,Designing a Framework with Test-Driven Development: A Journey, FEB 2014
[3]Roberto Latorre. *Effects of Developer Experience on Learning and Applying Unit Test-Driven Development,* IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 40, NO. 4, APRIL 2014
[4]Yahya Rafique and Vojislav B. Misic,Senior Member, IEEE, The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 39, NO.6, JUNE 2013
[4]Nicholson Collier & Jonathan Ozik. Test-Driven Agent-Based Simulation Development, Proceedings of the 2013 Winter Simulation Conference
[5]Jerod W. Wilkerson, Jay F. Nunamaker Jr., and Rick Mercer. Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 38, NO. 3, MAY/JUNE 2012
[6]Yuefeng Zhang, motorola. Test-Driven Modeling for Model-Driven Development , IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 2004
[7]Jim Whitehead Univ. of California, Santa Cruz, USA (Software Engineering). Collaboration in Software Engineering: A Roadmap, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 2007
[8]Alistair Cockburn Humans and Technology. Crystal Clear A Human-Powered Methodology For Small Teams, including The Seven Properties of Effective Software Projects, JUNE 2004
[9]Jeff Sutherland, Ph.D. Ken Schwaber, Co-Creators of Scrum, Jeff. The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process, OCT 2007
[10]A white paper by Keith Zambelich. Totally Data-Driven automation testing, 2009
[11]Terese Ann Brecklin, Marquette University. Data-Driven Decision-Making: A Case Study of How A School District Uses Data to Inform Reading Instruction , MAY 2010
[12]Lisa Crispin. Driving Software Quality: How Test-Driven Development Impacts Software Quality, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 2006
[13]E. Michael Maximilien and Laurie Willaims, Accessing Test-Driven Development at IBM, 0-7695-1877-X/03 © 2003 IEEE
[]Hakan Erdogmus, Maurisio, Member, IEEE Computer Society, On the Effectiveness of the Test-First Approach to Programming", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 31, NO.3, MARCH 2005
[14]Matthias M. Muller and Oliver Hagner, Experiment about Test-first programming, 2004 IEEE
[15]Nachiappan Nagappan. E. Michael Maximilien, Thirumalesh Bhat, Laurie Williams, Realizing quality improvement through test driven development: results and experiences of four industrial teams, Empir Software Eng (2008)
[16]Matt Stephens. Doug Rosenberg, Design Driven Testing, 2007
[17]Doug Rosenberg, Design Driven Cost Estimation via Design Driven Testing,Copyright © 2011 ICONIX
http://se.cs.depaul.edu/ise/agile.htm
[18]http://www.extremeprogramming.org
[19]http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap
[20]http://www.xprogramming.com
[21]http://www.cs.gmu.edu/~offutt/softwaretest/