2006: PM09

Testing Roadmap for Generalized Agile Framework

CHALLA CHAITANYA

EXAMENSARBETE
Magister Med Inriktning Mot Programvaruteknik Institutionen För Teknik, Matematik Och Datavetenskap

EXAMENSARBETE

Testing Roadmap for Generalized Agile Framework

Sammanfattning

Ingen svensk sammanfattning finns då denna uppsats skrivits av en engelskspråkig student. Se 'ABSTRACT' for mer detaljer (på engelska).

Författare:	Challa Chaitanya		
Examinator:	Dr. Samantha Jenkins		
Handledare:	Dr. Steven R. Kirk		
Program:	Masters Program in Software Engineering, 2006.		
Ämne:	Software Engineering	**Examensnivå**:	MASTER
Datum:	2006-08-18	**Rapportnummer**:	2006: PM09

Nyckelord: XP/Scrum, Integration Testing, Techniques in Integration Testing, Integration Testing in V&V and Quality Assurance, Agile Testing, IEEE Standard for Software Testing, XP Vs. IEEE, TDD, TDD & Traditional Testing ,TDD & Documentation, TDD Development, TDD & AMDD, Test Deliverables, Release Criteria, Suspension Criteria for Failed Smoke Test, Resumption Requirements,

Utgivare: Högskolan Väst, Institutionen för teknik, matematik och datavetenskap,
461 86 Trollhättan
Tel: 0520-22 30 00 Fax: 0520-22 32 99 Web: www.hv.se

MASTERS'S THESIS

Testing Roadmap for Generalized Agile Framework

Summary

Testing in Agile development frameworks has engaged the concentration of software developers throughout the world. The research regarding this testing is however limited.
This paper comprises a literature survey study, which intends to categorize and examine the current existing testing frameworks in agile methodology. The comparative study is done with the Test Driven Development (TDD) and Agile Model Driven Development (AMDD) techniques. The results show the benefits and drawbacks of the current existing testing frameworks in agile methodology in general. However many techniques still make every effort to get the general solutions. Basing on our results general ideas are suggested regarding the testing frameworks in agile methodology.

Author:	Challa Chaitanya		
Examiner:	Dr. Samantha Jenkins		
Advisor:	Dr. Steven R. Kirk		
Programme:	Masters Program in Software Engineering, 2006.		
Subject:	Software Engineering	**Level**:	Master
Date:	August 18th, 2006	**Report Number**:	2006:PM09
Keywords	XP/Scrum, Integration Testing, Techniques in Integration Testing, Integration Testing in V&V and Quality Assurance, Agile Testing, IEEE Standard for Software Testing, XP Vs. IEEE, TDD, TDD & Traditional Testing ,TDD & Documentation, TDD Development, TDD & AMDD, Test Deliverables, Release Criteria, Suspension Criteria for Failed Smoke Test, Resumption Requirements.		
Publisher:	University West, Department of Technology, Mathematics and Computer Science, S-461 86 Trollhättan, SWEDEN Phone: + 46 520 22 30 00 Fax: + 46 520 22 32 99 Web: www.hv.se		

Acknowledgements

I choose this research paper as part of my Masters Program in Software Engineering. The successful completion of this paper involves dedication, strong firm and research work.

I also owe a debt of gratitude to the University staff. This paper stands completed virtually as a result of teamwork and all the attributes already spelled out above. For us it has been a painful, yet satisfying experience, and the output of our dedicated efforts leaves me sufficiently motivated to involve me more in this research paper and complete it with a touch of excellence.

I placed our record & appreciation to my University West, Sweden and for those personals who helped me a lot in which Dr. Steven Kirk (Advisor), Email: steven.kirk@hv.se and Dr. Samantha Jenkins (Examiner & Assoc. Prof., Department of Technology, Mathematics & CS), Email: Samantha.Jenkins@hv.se.

Innehållsförteckning

MASTERS'S THESIS

Testing Roadmap for Generalized Agile Framework

Challa Chaitanya
chaitanyachalla@yahoo.com
Department of Computer Science
University West - Sweden

Abstract

Testing in Agile development frameworks has engaged the concentration of software developers throughout the world. The research regarding this testing is however limited.
This paper comprises a literature survey study, which intends to categorize and examine the current existing testing frameworks in agile methodology. The comparative study is done with the Test Driven Development (TDD) and Agile Model Driven Development (AMDD) techniques. The results show the benefits and drawbacks of the current existing testing frameworks in agile methodology in general. However many techniques still make every effort to get the general solutions. Basing on our results general ideas are suggested regarding the testing frameworks in agile methodology

Keywords: XP/Scrum, Integration Testing, Top Down Integration and Bottom Up, Techniques in Integration Testing, Hybrid Testing, Bugs by Integration Testing, Integration Testing in V&V and Quality Assurance, Conversing People, Agile Testing, IEEE Standard for Software Testing, XP Vs. IEEE, TDD, TDD & Traditional Testing ,TDD & Documentation, TDD Development, TDD & AMDD, Why TDD, Test Deliverables, Release Criteria, Suspension Criteria for Failed Smoke Test, Resumption Requirements, Release to User Acceptance Test Criteria, Release to Production Criteria.

1. Introduction

Since many years we are adopted to different testing methods while developing software. In general, software development mainly focuses on time and cost and then we choose the best approach to develop the software. But here is the problem regarding to the testing that which kind of testing we should use.

Now days, testing is done in different type of languages like in Java we have testing frameworks for writing the test cases. These test cases are normally written for checking our code which we already developed for our software. This kind of test cases shows that what kind of development we are doing? And are we doing our development in the right track or not? And also how much time it's consuming i.e. how much it is efficient regarding the coding efficiency and speed.

In general, testing done in software development field is in three ways. In which we have the Unit testing which is helpful for the checking of the class level testing codes [1]. The second is on integration level, helpful for the integration between the different class levels. And in the end we have third testing level in which we checked our code as well as the efficiency of the system during the system level testing. Testing is totally based on the nature of the development as well as the outcome of the software [2]. Some projects / software's are text based and mostly are like the websites or GUI based. In the GUI we have already well developed components / sources which we can reuse in our software developments [4, 5]. And also the reusability of these components are beneficial in the sense of time saving and also the cost saving [2, 3].

The main goal of our thesis is to collect information regarding the current existing testing systems in the agile methodology, but these testing methods hardly integrate to form a unified picture. This problem area needs generality so that all these frameworks could interact and work together in future. Testing framework that will guide users build their tests in the general all purpose agile world. These guidelines will help bridge the gap of different test routines available today. Studying the testing frameworks available today for the agile methods to identify their similarities and differences and classifying them with respect to their structure. Then we talked about XP/IEEE, Unit & Integration Testing and Test Driven Development (TDD), which are discussed in detail in later sections.

This paper is based on our general observations and thinking of the expert personnel's who are working in this software development and testing field. The overall paper is described in the different sections for better understanding for the reader. The background Section (2) is giving the idea about object oriented software development and testing. The section 2 is further divided into the different sub sections where we are discussing about Integration testing that what is integration and components, what are the techniques of Integration testing, also about top down, bottom up and hybrid integration testing methods. Further in this section, Integration in Verification & Validation (V&V) as well as Integration testing in V&V and quality assurance and also it is explaining about that where integration testing cannot help in V&V. The agile testing is all about the IEEE standards for software testing & their sequence which normally IEEE followers are following. Further more; it is giving the idea about XP Vs IEEE and which better or worst in XP & IEEE.

Section 3 is giving the details of the methods we implemented in finding the material regarding the testing methods and frameworks existing in present agile methodology. Section 4, is giving the overall results of the research work where we explained about the general ideas that are being used in the present testing frameworks and also the Myths & Misconceptions regarding to TDD [6]. In this section we also explained a different type of testing that is the smoke testing. In Smoke testing we have the concept of working with the developer and conduct a code review. The Discussion is also a major part of our paper which is in Section 5 where we discussed all about that what TDD is and in that TDD what is TFD with refactoring and how X unit frame work is working. After this Test Driven Database Development about the data bases that how the Database Administrators (DBA's) can improve and work in testing field. The last part in discussion section we are explaining about the TDD & AMDD where we discussed about the Agile Model Driven Development & Life Cycle of that.

The method which we adopted for solving our argument is the literature study, where we collected the material regarding the testing methods in Agile from the professional books, journals of the conferences and also from the search engines like yahoo, Google, answers.com etc.

2. Background

Object oriented testing is required for object oriented code. In this approach we are testing classes and methods which we already developed. Other software testing techniques are based on functional testing [7, 8]. In those testing techniques we usually take the system as a black box and we check the system against the requirements and see whether the system performs well for each requirement. On the other hand object oriented testing is a white box testing and we go into the code and classify a unit to test as class or method and then we individually test those units [7, 9]. Then we proceed further by integrating those units with other relevant units and see are they functioning correct with them. Object oriented testing also focuses on testing Classes, subclasses, inheritance, polymorphism etc., which is not a requirement in traditional software testing [6, 7, 10]. The main difference when it comes to object oriented testing is that, testing starts from the design phase because the designs are very close to the actual implementation e.g. class diagram. If we can validate the designs and prevent business and functional faults from propagating, testing the actual application would take a very little time. As implementation is handled through objects and if the object is used more than once there is no need to test them both as error in one would guarantee a fault in the other uses of the object. We think that it is easier to do object oriented tests because we have developed the program and we know all steps how to proceed in testing then as we carry object oriented testing through whole the project life cycle we have UML diagrams to verify the code with [1]. In case of other testing techniques we have to come up with a testing model and layout a strategy on how we can cover most of the requirements [6, 11]. In General, testing can be described as shown in Figure 1, which shoes the test case work flow [12]:

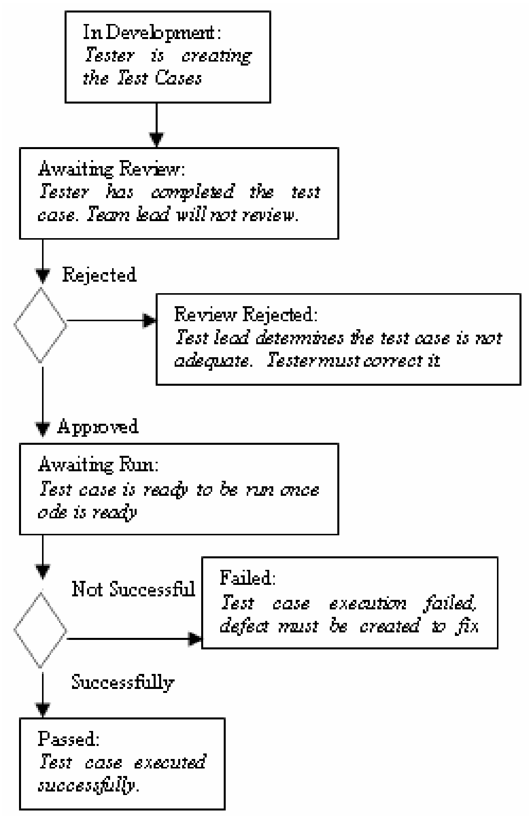
In Development:
Tester is creating the Test Cases
Awaiting Review:
Tester has completed the test case. Team lead will not review.
Rejected
Review Rejected:
Test lead determines the test case is not adequate. Tester must correct it
Approved
Awaiting Run:
Test case is ready to be run once ode is ready
Not Successful
Failed:
Test case execution failed, defect must be created to fix
Successfully
Passed:
Test case executed successfully.

Figure 1, the test case works flow [12]

2.1 Integration Testing

Even if a software component is successfully unit tested, in a distributed application it has a less or no value if that component cannot be successfully integrated with the rest of the application. Once unit tested components are delivered then we integrate those together [13]. These integrated components are tested to weed out errors and bugs caused due to the integration [14]. It is very important because, it is possible that a different programmer develops different components and lot of bugs emerges during the integration step [14, 15].

2.1.1 Techniques in Integration Testing

Before we begin Integration Testing it is important that all the components must successfully unit tested. The Steps involved in integration testing are the following [9, 15]:

1. Make a Test Plan
2. Generate Test Cases
3. Generate Test Data
4. Create scripts to run test cases
5. Execute the test cases (After components integration)
6. Fix the bugs if any and again test the code
7. Repeat the test cycle until the components have been successfully integrated

2.1.2 Top down Integration testing

We can start testing in different manners, top down (Breadth first and depth first), bottom up and hybrid [14, 16, 19]. In top down integration testing high level control routines are tested first. While performing top down integration testing, we can move in breadth first manner and in depth first manner as well [14, 19]. By performing the tests in depth first manner means that each module will be tested first then we move into details and test the modules that relates to the top module going in details. In breadth first integration testing we test the modules at same level first and then moves to the next level [14, 19].

2.1.3 Bottom up Integration Testing

In bottom up integration testing we combine some unit tested components and perform tests. Then we move in this manner incrementally and combine some modules further. While performing bottom up testing, individuals or sub-teams first test the module (units combined) and then deliver the tested modules to the integration team [14, 16].

2.1.4 Hybrid Testing

We can perform integration testing in hybrid manner, in this approach we will test some modules first, as in start modules do not have complete functionality, so after testing it when functionality is complete or more detail is available we move in depth first manner. So first we take start in breadth first manner and then moves according to depth first strategy.

Normally we are using in our software development i.e. Bottom up approach but for bottom up approach we have some techniques for integration testing which are given below:

1) We will use bottom up approach for integration testing.
2) In bottom up approach we will combine unit tested components and perform test on it. This test will be performed by individuals and sub-teams.
3) After combining some of the components, other components are also added to it and process will be repeated.

2.1.5 Motivation for Bottom up testing

Integration testing strategy focuses on lower layers first because they have least external dependencies. Testing the lower layers first also reduces the amount of uncertainty when testing the upper layers. If an integration test case fails in the upper layers, it is unlikely that a reason is a faulty lower layer component. Since, these components have already been tested. So we narrow our scope of debugging [15, 16, 18]:

1. Programmers at out organization doesn't have knowledge about integration testing, so bottom up approach can be started by combining individual components, which can be tested by these programmers [10, 17].
2. We have programmers have experience with unit tests, so they can use bottom up approach, so units are combined to make a small module [5, 19, 20].
3. This approach can be started with out any external team, because the developers can run unit tests and it is easy for them to make small integrations and move incrementally [17, 19, 20].
4. Incrementally combining more and more modules will be a good experience for the present individuals.
5. We will hire external experts also for integration testing, but they will not be involved at basic steps. These externals will also be a valuable source for training of present experts [16, 18].
6. We can start integration testing as soon as unit test is applied on different components, so we don't have to arrange external teams immediately [13, 14].

2.1.6 Making a test plan

We have to consider following things for making a test plan [18]:

1) How to execute the tests
2) Identify units/components and features to be tested

1) Review the unit testing outcomes, in case of any problem, repeat unit tests
2) Make necessary assumptions after consensus of developers and customers
3) Define post conditions in case of successful execution of tests and actions to be taken
4) Define actions to be taken in case of test failure.

2.1.7 Bugs Identified By Integration Testing

There can be many problems in units, which give raise to bugs while integrating. Such as [5, 18]:
1) Improper call or return sequence, unit test will not identify this problem.
2) Inconsistent data validation criteria, if this data is not used as results of independent units, then it will produce bug while integrating the units.
3) Inconsistent handling of data objects.

2.1.8 Integration in Verification & Validation (V&V)

As mentioned above software testing is dynamic V&V, so if we consider software testing paradigm alone, there are many stages of testing. Software integration testing comes after unit tests and when basic functional units have been created and tested [5, 21]. Components may independently works well and fulfill all the requirements, because individual components do not have many concerns across unit boundaries or interaction patterns with other units. It is necessary to verify that the system as a whole works well. So V&V cannot be completed until system as a whole fulfill all the requirements, and obviously system is an integration of different modules, so without integration testing we cannot perform verification and validation. Its beyond the scope of inspections and walk through [5, 6, 21].

2.1.9 Integration Testing In Verification & Validation (V&V) and Quality assurance

As software Verification and Validation (V&V) is the process of ensuring that software being developed or changed will satisfy functional and other requirements (validation) and each step in the process of building the software yields the right products (verification) [22]. V&V can be static and dynamic. Software testing lies in dynamic V&V because it tests the system by execution [22].

2.1.10 Where Integration testing cannot help in verification and validation?

Integration testing cannot help in the process of V&V in certain cases, i.e. [13, 14]:

1) It cannot help in verification of non-functional requirements and some functional requirements also.
2) It cannot guarantee us that the system will produce correct results; it is only concerned that the units will interact in correct manner.
3) It cannot guarantee that the system will provide all the functionalities as per requirements; it can be achieved by inspections and walk through.

Generally, agile projects understand that, they deliver working software (or perhaps executable prototypes) as quickly as possible and as frequently as practical. Development is organized into a rapid series of functionally complete releases, each one made available for the user to try. Since each such release is really the first chance the user's had to think about the new features, rework is just part of the job, not a crisis.

2.2 Conversing with people

It is very hard to do less documentation during the agile development, and mostly the people are trying to get the new ways and techniques. But it is very important to contact with your other relative persons like face to face conversation and collaboration. XP has people program in pairs and tries hard to have a customer representative working every day in the same bullpen as the developers [23]. One of Scrum's practices is carefully crafted daily standup meetings that create and preserve group understanding. Crystal, perhaps the least dogmatic process conceivable, nevertheless insists on frequent retrospectives [24]. These techniques advance the communication that documents cannot replace. In all agile methods, it is necessary to take customers with you in the development time. But if you have suitable customer then you will not need any detailed documentation. By asking some question from customer and also showing him daily work will be helpful to achieve you goals in the development. It is very important to design the meetings with customer in the agile development and also talk with the customer on practical issues as well as concentrating on some object of work. But there'll be a background of unspoken understanding of the essentials, so that different people will automatically make consistent choices when confronted with similar problems.

2.3 Agile testing

Agile Testing would most obviously apply to agile development projects, but it should work - perhaps less well - on conventional projects too [25]. The first step is to abandon the notion that others communicate at us with requirements and design documents, and that we communicate back at them with test plans and bug reports. We've always realized that the documents based on our tests are inconsistent - incomplete, incorrect, and ambiguous but our reaction has been to insist, in our usually powerless way, that the document producers do better [15, 22]. But now we can see that "better" will never be good enough [22]. Documents can't be an adequate representation of working code. So we can let free of the illusion that documents will save us. We can view them as they are: interesting texts, partly fictional, often useful.

In the communication system with customer the testers and developers should sit in the same share offices or occupy alternate cubicles. Many testers should be assigned to help particular developers, rather than to test pieces of the product. The test plan should evolve through a series of what James Bach calls "drop-in meetings" - short, low-preparation, informal discussions of particular topics [26]. Test status should be reported via big, public, simple-to-read charts that answer specific development questions like "what parts of the product can we stop worrying about?". The conversation with the customer is much important with the developers. And also remember that the customer is trying to figure out what they need, want, and are getting, in large part by trying out the working code [27, 28]. Testers should sit down with them as they do that. Creating some tests together is an excellent way for both of you to learn what matters - and also to describe it to the developers in a clear and concrete way [29].

That's an instance of the important Hands-On that need to develop with developers as well. The normally strained relationship with them will be less so if they see you wanting to get started testing, even on something unfinished, especially if your expressed goal is to help them improve and complete it. They'll value tests they can run as they continue development.

2.4 IEEE Standard for Software Testing

The main goal of Software Testing is to evaluate product quality and improving it by identifying defects and problems [31]. The test plan goal is to prescribe the scope, approach, resources and schedule of the testing activities in a software project. These should be kept up to date because they are dynamic. The sections in the test plan should be ordered in the sequence specified are [11, 31]:

a) Test plan identifier: Specifies the identifier assigned to the particular test plan.
This helps us to identify whether this test plan is a master plan or a level plan it represents. Section specifies the good software test documentation.

b) Introduction: The software items and software features are summarized in this section and includes highest level test plan. Also includes the executive summary information briefly and to the point. Section specifies the software test documentation.
c) Test items: Making a list of what is to be tested including the version level the test items should be identified and also should specify the characteristics of their transmittal media. Making good testing of the all the require items gives the better quality of the product.
d) Features to be tested: Identifying all software features and combination of software features to be tested from user's point of view and what the system does. This gives up a better quality.
e) Features not to be tested: Identifying all features and significant combinations of features that will not be tested from user's point of view.
f) Approach: This describes the overall approach to testing i.e. the approach should describe in sufficient detail to permit identify the major testing tasks and estimation of the time required doing each one. This provides and helps to software test documentation.
g) Item pass/fail criteria: To determine whether each test item passed or failed testing. This gives better quality and documentation also.
h) Suspension criteria and resumption requirements: The criteria used to suspend a testing activity on the test items associated with the project plan. This used when for a good documentation and as well it gives the quality assurance.
I) Test deliverables: Identifying the deliverable documents (test plan documents, test cases ...). This helps for software test documentation.
j) Testing tasks: Gathering the set of tasks necessary to prepare for perform testing.
This helps when we have multi-phase process. This is for better testing documentation.
k) Environmental needs: Gathering necessary and desired properties of the test environment. This provides us for better quality assurance of the project.
l) Responsibilities: Pointing the groups for managing, designing, preparing, executing, checking and resolving. Making test documentation for better quality software product.
m) Staffing and training needs: Identify test staff by skill level. To get good quality of the software we have to assign good staff by skill level and document for future.
n) Schedule: Estimate the time required to do each testing task to achieve test milestones. Document this section based on realistic estimates to provide better quality in time.
o) Risks and contingencies: Identify the high risks in the test plan and contingency plans for each. This is for good documentation purpose.

p) Approvals: specify the names and titles of people who will approve the plan.

2.5 Agile Methodology like XP VS IEEE

Testing is a big issue because most of the software developments are not proper tested. It not the primary need of the customer, but it necessary to checked by the software providers [30]. In the software development the extreme programming is talking about that one cannot be uncertain that a function works unless one tests it. It's the basic question that what is the uncertain. If you are talking about the uncertainty of coding that what it meant. So when you will test the uncertainty of the XP, it uses the Unit Tests [20, 23].

The Unit tests are the automated tests and help to test the code. Most of the time the programmer when he is doing the programming, it is possible that he will break writing due to thinking but if all the test will run successfully then you will say that coding is complete now. Unit testing is also verifying about the particular module/part of the coding that is it working properly or not. In which we also write the different test cases for all of our functions and the methods [6, 19, 20]. And it will identify and fixed when ever any change comes out due to regression. Normally developers do this kind of testing not by the end user.

Sometimes we are facing the other uncertainty and in which you will find that what is your meaning and what you should have the meaning. So, to test this kind of the uncertainty, extreme programming is using acceptance tests based. The acceptance tests based are using on the basis of requirements provided by the customers side in the phase of exploration of release planning.

2.5.1 XP and IEEE - Which is better or worst?

In XP it is not necessary that we will find the uncertainty [23]. The good thing in this strategy is that, it makes the sense of long & short terms. The short terms are those in which we will do program overall faster but think first about the interfaces then implementation. But in the long term tests dramatically reduce the chance that some one will destroy or damage the code [32]. And the tests communicates too much of the info. That is it recorded in the documentation or not. The writing of the tests are providing the better view of the design, that is it easier to test/check the simple design more than a complex design. In the presence of the tests, it helps to reduce the over-engineering. It is only that we implement what we actually need for the testing. Some times programmers may miss some thing in the unit testing, and the customers are also writing the functional tests at the same time. And one more thing that when the defect will slip from the unit testing it is also caught in functional testing or the production [33]. The Unit testing overall provides us the better designs, code reliability, and less time debugging.

But in the IEEE the software testing plan is a Standard by the ANSI/IEEE. Test plan is a valuable to the extent that it helps us to manage our testing for finding the bugs [22]. And this is not a practical to follow the all Test Plan Standard. Because every technique is not valuable and relevant to our project so just pick and choose the valuable techniques.

Overall in our view that IEEE is better than the unit testing because that is just testing the modules or the parts of the testing but here in this it gives the facilities of the technical task of testing, improves the communication relevant to testing & process tasks, and also gives us the structure for managing and organizing, scheduling that how to organize and what will be the schedule and how we will manage the testing [25, 34].

But on the other hand if we are considering the development of short time then the unit testing is good because it will take less time for debugging, and it will provide us the better designs, and reliable codes. Because it's hard to find in IEEE that which are necessary and which are not necessary to do and it takes some time. So not too much beneficial that in short time project we will go in the detail documentation because our organization needs the out come for the customer and business aspects [35].

3. Method

Testing frame works will guide users in building their tests in general for all purpose agile worlds. These guidelines will help to bridge the gap of different test routine available today. The theory regarding this paper is coming from the existing material, which is available on the internet as well as in books. The paper is based on academic as well as industrial aspects in software development through agile methods. So the way which we choose for this topic is based on literature survey through web browsing/searching materials like research papers, tutorial, market news. And also the market analysis and requirements with the current position of agile methodologies are also important.

The literature study was done from the different books and research papers of the people available in the university library. Also we searched some material available on the websites to collect the subject related material. We also traced the material from the conferences regarding to agile testing methods which were held on different places in different timings. The discussion of different personnel's and companies which are implementing and criticizing on these issues that what's that best way for software development. The list of the references is given below in the references. Most of the research papers which we studied were published in the IEEE. Some material is taken from the general ideas which all available in different books and websites. And all the books and web sites list are available in the reference list. Most of my paper results are based on conferences and published papers. The search engines utilized for this topic includes yahoo.com, google.com, answers.com, scirus.com etc. By using these search engines we are able to track some of the published journals/papers by different organizations. Most of these journals and papers that we used are from the IEEE standards. Some books, material/tutorial was found by using the web search engines.

4. Results

In the User interface testing we will implement the Unit testing and the unit testing is based on the small parts of the system interfaces. Each interface is containing some data for example if we see the login screen on the different websites or the software then in that one we have the User name and password. Now on the unit testing we will check these two options in every aspect. In which like we will see that how long the text box of the user name means how many characters and which type of characters user can insert in the field. Same in the case of password field and also here we will check that which type of characters user can put as well as the type of data like numeric is allowed or only alphabets and how many we can insert. Then after that we will see that is this data is also going in out database or not. Now we will come on the testing of data bases where we will check and make the test cases for that database. After completion of test cases we will run the test cases and check the data and their relevant fields. If the outcome will correct then we have the surety that our data base is working well regarding to these fields which are user name and password.

In the above we are trying to explain the unit testing but when we will have other parts in that software and we will collect them. After collecting them we see their testing on the integration level. That their integration is having any problem or not. Also for checking their integration we will write their test cases. Here is the point that we already checked them before in the unit testing. But the problem is to make sure that they are interacting with each other efficiently or not. This kind of testing will be our integration type of testing for whom we will write some test cases like in J Unit, X Unit or many more.

4.1 Myths and Misconceptions

There are several common Ideas and their misunderstandings in the view of different people regarding to the TDD which we are explaining here for better understanding. These ideas list describes the realities which are given below [36, 37]:

1) Idea – We should create a 100% regression test collection.

In General – Its looks nice idea but it has several problems which are given below:

1) We may use some reusable components / frameworks etc... Which we have to download or purchase and also which do not come with a test suite, nor perhaps even with source code. Although mostly we can create black-box tests which validate the interface of the component these tests would not completely validate the component.

2) We have different type of user interfaces and it's very hard to test them although many user interface testing tools are available in the market some times these tools are difficult to use. Even if we will have some automated testing tools but its not necessary that we can use on all interfaces.

3) Some developers on the team may not have enough testing skills.

4) Database regression testing is a fairly new concept and not up till now it's not well supported by tools.

2) Idea - The unit tests form a good part of our design specification.

In General – Most people are claming that they are using agile development even thought they are not or perhaps people, who have never been involved with an actual agile project, will sometimes say this. In general the unit test form is a good for design specification, and for acceptance tests form is a good for our requirements specification, but there's more to it than this. Figure 3 shows this that the agilist's do in fact model as well as document for that matter. It's just that we were very good about this that how we do it. Because we thought this before going to write the code, and we effectively perform detailed design as we studied like: Single Source Information (SSI): "An Agile Practice for Effective Documentation article".

3) Idea - We only need to do unit test.

In General – In the simple system we don't need to do the unit testing but it's necessary for big and complicated applications. In the agile development methods we needs different type of testing like for acceptance testing, user testing, system integration testing, and a host of other testing techniques.

4) Idea – We can not extend the TDD

In General – Some thing is true here but some extensibility issues include:

1) Our tests collection takes too long to run: This is a common problem with equally common solutions. First, separate our test suite into two components. One test suite contains the tests for the new functionality that we are currently working on; the other test suite contains all tests. We run the first test suite regularly, migrating older tests for mature portions of your production code to the overall test suite as appropriate.

2) It doesn't mean that every developer knows how to test: This is mostly true that get them some appropriate training. And then make pairing of those people who have unit testing skills. Anybody who complains about this issue more often not seems to be looking for an excuse not to accept TDD.

3) Its not possible that every one will follow TDD approach: Every one is agree to work through TDD approach on the team level. If some people aren't doing so, then in order of preference they either need to start or need to be motivated to leave the team.

The above mentioned four ideas were talking about TDD in general concepts. Test-driven development (TDD) is a development technique where we must first write a test that fails before we write new functional code. Software developers can adopt TDD very quickly and also agile DBAs for databases development. TDD should be seen as complementary to Agile Model Driven Development (AMDD) [36]. The AMDD & TDD approaches can be used together. TDD does not replace traditional testing, instead it defines a proven way to ensure effective unit testing. A side effect of TDD is that the resulting tests are working examples for raising the code, so providing a working specification for the code. Our experience is that TDD works extremely well in practice and it is something that all software developers should consider adopting. The following is a representative list of TDD tools available to in the market in which cppUnit, csUnit (.Net), CUnit [38], DUnit [39] (Delphi) , DbUnit [40], JUnit [8,9] , NDbUnit [41], OUnit [42], PHPUnit [43], PyUnit (Python)[44], NUnit [45] and VbUnit [46].

4.2 Smoke Testing

On the other hand, after code reviews, smoke testing is the most cost effective method for identifying and fixing defects in software. Smoke tests are designed to confirm that changes in the code function as expected and do not destabilize an entire build. The following guidelines explain best practices for smoke testing. The effects of following the guidelines will vary widely, ranging from improving communication among team members to developing specific ways to use testing and debugging tools [47].

4.2.1 Working with Developer

Because smoke testing focuses on changed code, we must work with the developer who wrote the code. We will have to understand [48]:

1) We should understand the Code overall and also knows about where we made changes but it can be easier if that developer, who developed, will help.
2) How the change affects the functionality.
3) How the change affects the interdependencies of various components.

4.2.2 To Conduct a Code Review before Smo Testing

Conduct a code review that focuses on any changes in the code before we run a smoke test. Code reviews are the most effective and efficient method to validate code quality and ensure against code defects and faults of commission. Smoke tests ensure that the major critical or weak area identified either by code review or risk assessment is primarily validated, because if it fails the testing cannot continue [54].

4.2.3 To Install Private Binaries on a Clean Deb Build

The test must run on a clean test environment by using the debug binaries for the files being tested because a smoke test must focus on validating only the functional changes in updated binaries. We need not perform thorough tests. The purpose of smoke testing is not to ensure that the binary is completely error-free. This would require too much time. We perform smoke tests to validate the build at a high level. We want to ensure that changes in a binary do not weaken the general build in functionality [42, 54].

4.2.4 To Load and Web Testing

When we build our Load & Web tests, it is a good practice to run a smoke test before running any long or heavy test. In Web and in load testing, smoke testing is light test. We use a smoke test to validate that everything is correctly configured and running as expected before running your tests for performance [42, 54].

5. Discussion

In the discussion section mostly we are focusing here on the development on agile methodology that how we can do our development and then testing in agile way. So, for better understanding here we have some theory which is given below:

5.1 The TDD?

We can understand the different steps of test first development (TFD) by the given UML Diagram in figure 2. The initial step is to quickly add a test, basically just enough code to fail. Next step is to run our tests, often the complete test suite although for sake of speed you may decide to run only a division, to ensure that the new test does in fact fail. Then we update our functional code to make it pass the new tests. Fourth step is to run our tests again. If they fail we need to update our functional code and retest. Once the tests pass the next step is to start over. But here is a concept of refactoring means may be we will need to refactor any duplication out of your design as needed changing TFD into TDD [39, 41].

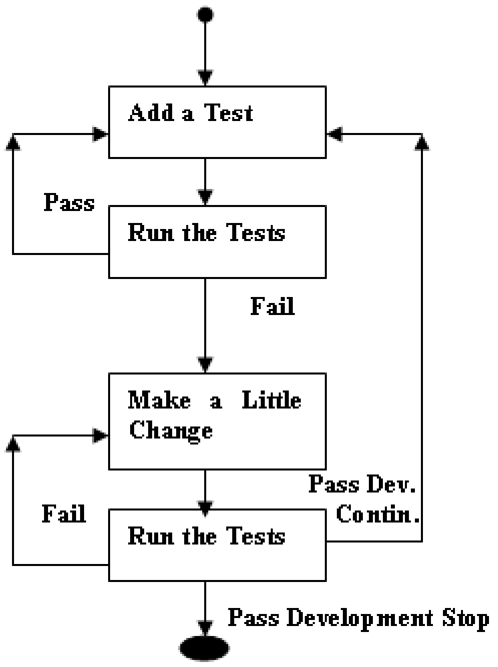

Figure 2, the Steps of Test-First Development (TFD) [41]

We like to describe TDD with this simple formula [41]:

TDD = TFD + refactoring.

TDD completely turns traditional development around [41]. In order to write functional code first and then the testing code as an addition. Then we will write our test code before your functional code. Also we do so in very less steps. One test and a less bit of related functional code at a time. A programmer taking a TDD approach rejects to write a new function until the first test that fails because, that function is not present. In reality, they refuse to add even a single line of code until a test exists for it [39]. Once the test is in place they then do the work required to guarantee that the test suite now passes. Here our new code may break several existing tests as well as the new one. Once your code works, you then refactor it to guarantee that it's remains of high quality. This looks simple in standard, but when we are learning to take a TDD approach initially it has to be proved the required great discipline because it is easy to write the functional code by skipping the new test. One of the advantages of pair programming (Williams and Kessler 2002) is that your pair helps you to stay on track [39]. A fundamental theory of TDD is that we have a unit-testing framework available to us. Agile software developers mostly use the XUnit group of open source tools like Java Unit or Visual Basic Unit, although commercial tools are also possible options. Without such tools TDD is virtually impossible [4]. The above figure 2 represents a UML state chart diagram for how people typically work with the XUnit tools [19, 20]. This following figure 3 is giving the overall idea of refactoring.

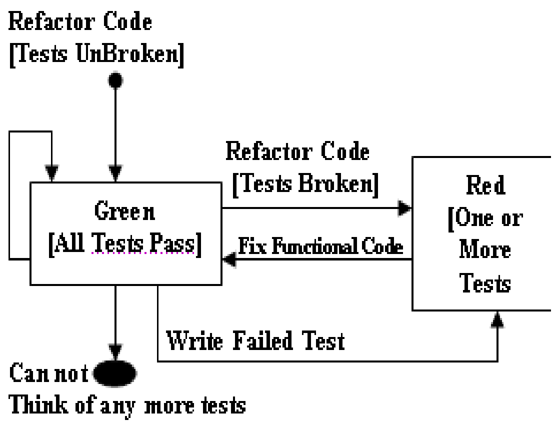
Refactor Code [Tests UnBroken]
Refactor Code [Tests Broken]
Green [All Tests Pass]
Fix Functional Code
Red [One or More Tests
Write Failed Test
Can not Think of any more tests

Figure 3, Testing via the XUnit Framework [20].

Kent Beck, who popularized TDD in eXtreme Programming (XP), defines two simple rules for TDD. In which initially we should write new business code only when an automated test has failed. And then we should remove any duplication that we find. Mr. Beck explains how these two simple rules generate complex individual and group behavior [47]:

1) We design organically, with the running code providing feedback between decisions [47].

2) You write your own tests because you can't wait 20 times per day for someone else to write them for you [47].

1) Your development environment must provide rapid response to small changes (e.g. you need a fast compiler and regression test suite) [47].

2) Your designs must consist of highly cohesive, loosely coupled components (e.g. your design is highly normalized) to make testing easier (this also makes evolution and maintenance of your system easier too) [47].

3) For developers, the implication is that they need to learn how to write effective unit tests. Mr. Beck's experience is that good unit tests [47]:

   1) Run fast (they have short setups, run times, and break downs).

   2) Run in isolation (you should be able to reorder them).

   3) Use data that makes them easy to read and to understand.

   4) Use real data (e.g. copies of production data) when they need to.

   5) Represent one step towards your overall goal.

5.1.1 TDD and Traditional Testing

The TDD is mainly a programming technique with a side effect of make sure that your source code is completely unit tested [40]. We still need to consider traditional testing techniques such as functional testing, user acceptance testing, and system integration testing, and so on. Maximum of this testing can also be done early in our project if we choose to do so we should. In fact, in XP the acceptance tests for a user story are specified by the project stakeholder either before or in simultaneously to the code being written. By giving to the stakeholders the confidence that the system does in fact meets their requirements [48].

In usual testing a successful test finds one or more defects and it is similar with TDD. When a test fails we have made progress because we now know that we need to resolve the problem. Most important is that we have to make a clear measure of success when the test no longer fails. The TDD increases our confidence that our system essentially meets the requirements which is already been defined for it. That our system actually works and therefore we can proceed with confidence [48, 49]. In the traditional testing the more the risk of system the greater detailed our tests need to be. In traditional testing and TDD we were not motivated for perfection; instead we are testing to the importance of the system. In agile modeling (AM), we should test with a cause and know why we are testing something. And also on which level it needs to be tested. An interesting side effect of TDD is that we achieve complete test where every single line of code is tested. And also something that traditional testing does not guarantee but although it does recommend it. In general we think it is practically safe to say that TDD results in much better code testing than do traditional techniques [48, 49].

5. 1.2 The TDD and Its Documentation

Most of the programmers do not like to read the documentation and they are focusing on just the code. To understand a class or operation most programmers will first look for sample code that already raises it. Good written unit tests do exactly this like to provide a working specification of our functional code. And also as a result unit tests effectively become a major portion of our technical documentation. The implication is that the outlook of the expert documentation host needs to reflect this reality [5, 48]. Likely acceptance tests can form an important part of our requirements documentation. It makes a lot of logic when we stop and think about it. Our acceptance tests define exactly what our stakeholders expect of our system. So, we specify our critical requirements. For example we are likely to find that we still need user, system overview, operations, and support documentation. We may even find that we require abstract documentation over viewing the business process that our system supports. When we approach documentation with an open mind, we should think that we will find that these two types of tests cover the major part of our documentation that needs for developers and business stakeholders [5, 48, 49].

5.1.3 Test-Driven Database Development

Here the main question arises while talking about the TDD. The question is "Can TDD work for data-oriented development?" [50]. as shown above in figure 1, it is important to note that none of the steps specify object programming languages like Java / C#. Even though those are the environments TDD is typically used in. it is necessary to write a test before making a change to our database schema. And we should also make changes, run the tests and refactor our schema as required [50, 51].

The database TDD will not work as efficiently as application TDD. Even though unit-testing tools, like DataBase UNIT are now available, they are still a rising technology at the time of this writing [44]. Some of the DBAs are improving the quality of the testing that they are doing. But we have not yet seen anyone take a TDD approach to database development. On the other hand the unit testing tools are still not well accepted within the data community. Even though TDD is changing, so our expectation is that over the next few years database TDD will grow [44]. After this, the concept of evolutionary development is new to many data professionals. And also a result that motivates to take a TDD approach has a stable hold. This issue affects the nature of the tools available to data professionals because a serial approach still controls within the traditional data community. Most of the tools do not support evolutionary development. Our experience is that most people who do data-oriented work seem to prefer a model-driven architecture but they not like to work in test-driven approach because a test-driven approach has not been widely considered until now. Also another reason may be that most of the data professionals are likely visual thinkers. So generally they prefer a modeling-driven approach [52].

5.1.4 The TDD and Agile Model-Driven Development (AMDD)

We compare TDD with Model driven development (MDD) as most of the points are from the Agile model driven development (AMDD) and the life cycle of AMDD is shown in the below figure 4 [36]:

1) TDD shortens the programming feedback loop whereas AMDD shortens the modeling feedback loop [36].
2) TDD provides detailed specification (tests) whereas AMDD can provide traditional specifications [36].
3) TDD promotes the development of high-quality code whereas AMDD promotes high-quality communication with your stakeholders and other developers [36].
4) TDD provides concrete evidence that your software works whereas AMDD supports your team, including stakeholders, in working toward a common understanding [36].
5) TDD "speaks" to programmers whereas AMDD speaks to data professionals [36].
6) TDD provides very finely grained concrete feedback on the order of minutes whereas AMDD enables verbal feedback on the order minutes [36].
7) TDD helps to ensure that your design is clean by focusing on creation of operations that are callable and testable whereas AMDD provides an opportunity to think through larger design/architectural issues before you code [36].
8) TDD is non-visually oriented whereas AMDD is visually oriented [51].

So, these above mentioned TDD and AMDD Techniques support evolutionary development.

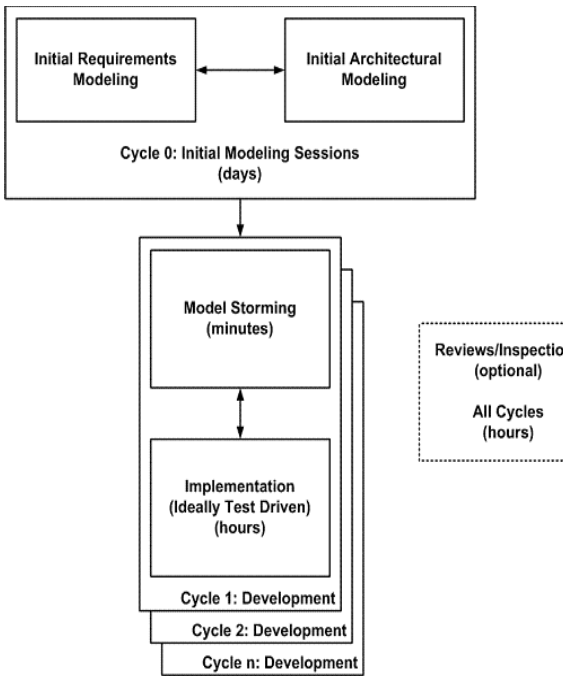
Initial Requirements Modeling
Initial Architectural Modeling
Cycle 0: Initial Modeling Sessions (days)
Model Storming (minutes)
Implementation (Ideally Test Driven) (hours)
Cycle 1: Development
Cycle 2: Development
Cycle n: Development
Reviews/Inspectio (optional)
All Cycles (hours)

Figure 4. The Agile Model Driven Development (AMDD) lifecycle [36].

So, here the question is raised that which approach should we follow? The answer depends on us and our group member's observations with their preferences. Most of the people are mainly visual thinkers and they prefer to think things through drawing. But some people do not like to work well with drawings and therefore they may prefer a TDD approach [42]. Most of the people work somewhere in the middle of these two limits. And for result they prefer to use each technique which makes more sense. By using both techniques together we will have more advantages.

So, here again a new question raised that how do you combine the two approaches? AMDD is used to create models with our project stakeholders to help to search their requirements, and then the search requirements are used sufficiently in architectural and design models [43]. TDD is used as a critical part of our build efforts to ensure that we develop an efficient working code. The final result will have a high-quality with working system that meets the all pre defined needs of our project stakeholders [41].

5.1.5 Why TDD is important?

Major advantage of TDD is that it enables us to take small steps when we are writing software code [41]. This is a practice that we have preferred mostly because it is more useful than attempting to code in large steps. For example, we assume that to add some new functional code then to compile the code and to test it. There are many chances that our tests will be broken by defects that exist in the new code. It will be much easier to find and then to fix those defects if we have written two new lines of code as compared to that of thousand lines of code. The conclusion is that the faster our compiler and regression test combine together the more important it is to continue in smaller and smaller steps. We generally prefer to add some new lines of functional code, somewhat less than ten, before we recompile and rerun my tests [49, 50].

Most of the people agree in the small projects by using agile techniques. It is also possible that people involving for several months and they will not work for real projects that are much larger. In general, Smalltalk system taking a complete test-driven approach was said that it took 4 years and 40 persons years of effort. That resulted in 250,000 lines of functional code and 250,000 lines of test code. So, there are 4000 tests running in under 20 minutes, with the full suite being run several times a day. Even if there were larger systems out there, generally in software houses where several hundred person years of effort were involved, it is clear that TDD works for good-sized systems [49, 50, 51].

5.2 Another Testing Approach

The development team will be responsible for installing the limited new builds into the existing structure of the system test environment. And also updating the client machines if that is necessary.

A series of scripts called the smoke tests, once the build is dropped by the development team. Smoke Test, will be run to ensure that the shipment from development is in a state that is ready for testing. The Smoke Test scripts will test the basic functionality of the system. These scripts may be automated once they are successfully performed manually. If an extreme number of Smoke Test items fail, the product will be shipped back to development and no testing will begin until the Smoke Test passes.

Once the first drop begins, group meetings will be held to discuss the bug list with the Project/Development Manager. Group meetings are used to prioritize, set priority and severity and assign bugs. Each week following the first drop, additional drops will be delivered to system test to test the bugs fixed from the prior drops.

To track the defect, we will use the defect tracker for analyzing the bugs. The defect tracker document will be distributed to the project and development manager to ensure that everyone understands how to use Defect Tracker and how to effectively enter bugs.

5.2.1 The Test Deliverables

Below are the deliverables for each phase of the project.

Phase	Deliverable	Respons-ible
Pre-baseline	Project Initiation will be performed on the basis of functional specification. This includes finding a test lead for the project and setting up a project in Defect Tracker.	Test Lead
Pre-baseline	Kick off meeting is done to know the project manager with the test methodology and test deliverables, set expectations, and identify the next steps.	Test Lead
Pre-baseline	Attend meetings to create functional specifications. Offer suggestions if anything is not testable or poorly designed.	Test Lead
Pre-baseline	The Test Plan will break functionality into logical areas like mostly specified in the functional specification. After completion, project manager, development manager, user project manager and the production support manager will review it. Once reviewed and amended, it must be approved and signed by the Test Lead, Project Manager, Development Manager, Production Support Manager, and User Project Manager. Before completing the project plan, the development project plan must be completed as to know what dates we are being asked to hold. This also guides us in determining if the test estimates are reasonable. The project plan will be detailed, relating back to the functional specification and test Plan.	Test Lead
Pre-baseline	Once the detailed test plan has been created and reviewed by the test and development teams, test cases are created. Test Cases are stored in Defect Tracker. Each test case includes the steps necessary to perform the test, expected results and contains any data needed to perform the test and to verify that how it works?	Tester, guided by the Test Lead
Pre-baseline	To review the Test Plan is to ensure all points of the Functional Specification are accounted for. Just like to ensure that the Test Cases have traceability with the Test Plan and Functional Specification. Finally, that the Project Plan has traceability with the Test Plan.	Test Lead
Once testing begins	Once testing begins, group meetings will be held to prioritize and assign bugs. This is conducted by the Test Lead and will include the Project Manager and Development Lead. Once user testing begins, the User	Test Lead

Once testing begins	Once testing begins, group meetings will be held to prioritize and assign bugs. This is conducted by the Test Lead and will include the Project Manager and Development Lead. Once user testing begins, the User Project Manager will also attend. Group meetings are usually held two to five times per week, depending on the need.	Test Lead
Bi-Weekly	Update the project plan with percentage complete for each task and enter notes regarding critical issues that came. Also determine if the test effort is on budget.	Test Lead
Weekly	The Project Manager will specify that who is to receive the weekly status report. This report will identify the percentage complete of all tasks. That should be due by that week and the tasks to be worked on in the next week. Also the metrics indicating the testing statistics, budgeting information, and any risks that need to be addressed. This information can be generated from Defect Tracker.	Test Lead
Before sending to UAT	The Test Lead will create a report that summarizes the activities and outlines any areas of risk once the System Testing is done and the code is ready for User Acceptance Testing (UAT). It will be created based on a template and all assumptions will be listed.	Test Lead
Project Comple tion	Post Mortem Analysis is done to analyze how well our testing process worked.	Test Lead, Develop ment Team, User Team

5.2.2 Setup for Defect Tracker

Bugs can be traced if the Test Lead will create a project for Defect Tracker. The project name in Defect Tracker will be [Project Name].

5.2.3 The Release Criteria

Some of the release criteria are given below:

5.2.3.1 Test Case Pass/Fail Criteria

The feature will pass or fail depending upon the results of testing actions. The action passes if the actual output from an action is equal to the expected output specified by a test case. Any action within a test case should fail the entire feature or sub-features. The specific criteria for test case failure will be documented in Defect Tracker. It is not necessary that code is defective if the test case fails. A failure can only be understand as a difference between expected results, which is derived from project documentation, and actual results. There is always the possibility that expected results can be in error because of misunderstanding and incomplete project documentation.

5.2.3.1.1 The Pass criteria

All processes will execute with no unexpected errors. And also all processes will finish or update / execute within the expected time frame which was given by the business analysts and documented by the development team [53, 54].

5.2.3.2 Delayed Criteria for failed Smoke Test

The system test team may delay some or full-testing activities on a given project if any of the following occurs [45]:

1) Files are missing from the new updated project.
2) The development team cannot install the new updated project or a component.
3) The development team cannot configure the updated project or a component.
4) There is a fault with a feature that stops its testing.
5) Item does not contain the specified change.
6) A severe problem has occurred that does not allow testing to continue.
7) Development has not corrected the problems that previously suspended testing.
8) A new version of the software is available to test.

5.2.3.3 The Resumption Requirements

The resumption requirements are given below:

1) First to clean previous code from machines.
2) Then to re-install the items.
3) The problems that come across results in suspension are corrected.
4) When the following is delivered to the system test team, resumption of testing will begin:
    1) A new updated project via Visual Source Safe.
    2) A list of all bugs fixed in the new updated version.
    3) A list of all the changes to the modules in the new updated version and what functionalities it affects.

5.2.3.4 Release to User Acceptance Test Criteria

The release criteria necessary to allow the code to transfer to user acceptance testing are as follows [48, 49, 53]:

1) The test cases were scheduled for both Integration and system test phases which have passed.
2) Final regression testing is successfully passed.

5.2.3.5 Release to Production Criteria

The release standard necessary to allow the code to transfer to Production is as follows [48, 49, and 54]:

1) The test cases scheduled for both Integration and system test phases have passed.
2) Final regression testing is successfully passed.
3) There are no difference between the original setup and the version used during the final regression testing.
4) The User Acceptance Test was successfully completed
5) The User Acceptance Criteria was met.

6. Conclusion

Finally, the original goals of this thesis have been achieved where we have examined in detail the benefits and drawbacks of the testing frameworks that are currently available. By examining the testing frameworks available today we also classified them with respect to their structure. We also explained the complete ideas regarding the XP/IEEE, Unit & Integration Testing and TDD. By our analysis we chose only the relevant things which are helpful for the extreme programming. But on the other hand we are using IEEE in which IEEE uses the detail of the each step. We then choose the relevant data that we need for our project from the IEEE for our XP approach.

In the hybrid approach, we defined about the tasks that may be in pairs or individual. So in the testing of XP it is mandatory to test automation and only the XP mandates it. In XP first we write the Unit Test Code and then the real code. Writing unit test code, then the real code, involves additional work but it is inevitable. In XP, the testing code is too much. When we are using the XP Testing by using the Hybrid Approach then first we have to separate the Test Data from the Test Code, Secondly make the test code, generic enough, Third is to Data Driven Approach for the Automated Testing and in the last we have the users setup Test Data.

XP's very strong on programming techniques for rapid deployment and is, conversely, very weak in management techniques and project controls [35]. So we like to keep management techniques in test plan i.e. project plans, requirement specification, high-level design document, detail design document, development and test process standards, risk issues and testing the milestones. So, we derived our own results which already given in results section where we gave some instructions.

7. Future Work

Agile testing methods will be better as this work helps new agile testing methods. All the relevant fields of computers will also find information for the development and research purposes. Errors in the agile testing methods have been identified and the solutions been proposed. The laid down foundations in the agile approach create new basis of agile methods and diminishes chances of error prone aftermaths.

This paper is focusing on a new plan for agile testing. But any other organization, who want to play a vital role in the field of software development as well as testing. And better security for the software engineering field will be implemented and new rules will be introduced and implemented with the help of different software testing with the help of agile methodology as well as any other organization who wants to make their own standards.

8. Acknowledgements

We are really very thankful to our University who gave me chance for this Masters in Software Engineering. We are also very thankful to our Supervisor Dr. Steven Kirk (Department of Technology, Mathematics & CS) and Examiner Dr. Samantha Jenkins (Associate Professor, Department of Technology, Mathematics & CS), who helped me a lot to do this.

9. Dedications

To my parents, who care and support me in every field of life and also my sisters, cousins and friends.

10. References

[1] Andrew Hunt, David Thomas, *"Pragmatic Unit Testing in C# with NUnit"*, The Pragmatic Programmers; 1 edition (May 2004).

[2] Somerville, Ian, *"Software Engineering"*, 7th edition, Addison-Wesley, Boston 2004.

[3] S.L P fleeger, *"Software Engineering Theory and Practice"*, *Prentice Hall*, Upper saddle River, 2001.

[4] W.S Humpyrey, *"A Discipline for Software Engineering"*, First edition Addison- *Wesley* Company, Reading, 1995.
[5] Dave Scherb Dscherb, SEI Areas of Work: Management, 2nd July……………………………2006, http://www.sei.cmu.edu/managing/managing.html

[6] David Astels, *"Test Driven Development: A Practical Guide"*, Prentice Hall PTR; 1st edition (August 10, 2003)

[7] Mirsolow Staron, Software Engineering – *"An Object Oriented Approach"*, 2nd edition, Addison-Wesley

[8] Craig Larman, *"Applying UML & Patterns – An introduction to Object Oriented Analysis & Design &*

[9] Paul Hamill, *"Unit Test Frameworks"*, O'Reilly Media, 1st edition (October 2004).

[10]Glenford J. Myers and revised by Tom Badgett and Todd M. Thomas with Corey Sandler, *"The Art of Software Testing"*, 2nd edition, John Wiley & Sons, Inc.

[11]Ilene Burnstein, "*Practical Software Testing, Springer";* 1st edition (June 24, 2003).

[12]Cem Kaner, Jack Falk, Hung Q. Nguyen, "*Testing Computer Software"*, Wiley; 2nd edition (April 12, 1999)

[13] Http Unit, A testing frameworks for HTML user interfaces: http://JUnit.org

[14] Cunningham, Ward. *FIT: Functional Integrated Test*. http://fit.c2.com

[15]Rick Mugridge, Ward Cunningham, "*Fit for Developing Software: Framework for Integrated Tests (Robert C. Martin)"*, Prentice Hall PTR (June 29, 2005)

[16]William E. Perry, "*Effective Methods for Software Testing, Wiley";* 3rd edition (May 1, 2006)

[17] Ammann, Paul, Paul E. Black, *Model Checkers in Software Testing* http://xsun.sdct.itl.nist.gov/~black/Papers/ir6777.pdf

[18]Tim Koomen, Martin Pol, "*Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing"*, Addison-Wesley Professional; 1st edition (May 28, 1999).

[19] *XUnit family of testing frameworks*: http://www.xprogramming.com/software.htm

[20] Gerard Meszaros, *"X-Unit Test Patterns: Refactoring Test Code"*, Addison-Wesley Professional (December 8, 2006).

[21]M Smolarova, P Navrat – "*Journal of Computing and Information Technology"*, 1997 – www.dcs.elf.stuba, visited on 1st July, 2006.

[22]Jerry Zeyu Gao, H.-S. Jacob Tsao, Ye Wu, "*Testing and Quality Assurance for Component-Based Software*", Artech House Publishers (September 2003)

[23] John Wiley & Sons , Agile Modeling, *"Effective Practices for Extreme Programming and the Unified Process"*, Scott W. Ambler, http://www.agilemodeling.com/essays/agileDocumentation.htm

[24] Ken Schwaber and Mike Beedle, "*Agile Software Development with Scrum*" Prentice Hall, 2001.

[25] Addicam.V.Sanja, info_agile_programming.pdf, *"Overview of Agile Management & Development"*, Visited on 15th June, 2006, www.projectperfect.com.au

[26]http://en.wikipedia.org/wiki/Agile_software_development, Visited on 27th June, 2006.

[27] JfcUnit, *A testing framework for Java Swing user interface*: http://JUnit.org

[28]*Mercury Interactive's WinRunner functional testing tool*: http://www-svca.mercuryinteractive.com/products

[29] www.inf.vtt.fi/pdf/publications/2002/P478.pdf

[30]http://www.agilemodeling.com/essays *JUnittesting framework*: http://JUnit.org

[31] *Richard E. Fairley, Mary Jane Willshire, "Iterative Rework: The Good, the Bad, and the Ugly*", IEEE Computer Society, 2005.

[32] Fowler, Martin. "*Patterns of Enterprise Application Architectures"*, Addison-Wesley (2002)

[33] C.Jorgensen, *"Software Testing – A Craftsman's Approach"*, 2nd Edition, CRC Press 2002.

[34]Rex Black, "*Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing"*, Wiley; 2nd edition (July 19, 2002).

[35] Integrating agile software development into stage-gate managed product development, Daniel Karlstrom, Per Runeson, Empirical Software Engineering, 2006, DOI10.1007/s10664-006-6402-8

[36] Agile model driven development is good enough, Ambler, Scott W. IEEE Software, 2003

[37]Frederick P. Brooks, "*The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition"*, Addison-Wesley Professional; 1st edition (August 2, 1995)

[38]"*CUnit, a Unit testing framework for C*", http://cunit.sourceforge.net, visited on 20th July, 2006.

[39]"*DUnit, an Xtreme testing framework for Borlar Delphi programs*", Source forg http://dunit.sourceforge.net, visited on 20th Jul 2006.

[40*"DbUnit, Db Unit framework"*, http://www.dbunit.org , visited on 22nd July, 2006.

[41]"*NDbUnit, NDb Unit Framework"*, http://www.ndbunit.org, visited on 1st Aug, 2006.

[42]*"OUnit, the Oracle Unit tester"*, http://www.ounit.com, PL Solutions, visited on 29th Jul, 2006.

[43] *"PHPUnit, a unit testing framework for PHP"*, http://phpunit.sourceforge.net, visited on 29th Jul, 2006.

[44]Steve Purcell*"PyUnit, The standard unit testing framework for Python"*, Sourceforge, http://pyunit.sourceforge.net, visited on 25th July, 2006.

[45]*"NUnit, Unit framework for all .Net languages"*, Sourceforge, http://www.nunit.org, visited on 1st Aug, 2006.

[46]*"VbUnit, Unit testing framework against Visual Basic"*, vbunit.com, http://www.vbunit.org, visited on 1st Aug, 2006.

[47]Robert Culbertson, Chris Brown, Gary Cobb, *"Rapid Testing"*, Prentice Hall PTR; 1st edition (December 29, 2001)

[48]Jeff Langr, *"Agile Java(TM): Crafting Code with Test-Driven Development (Robert C. Martin Series)"*, Prentice Hall PTR (February 14, 2005)

[49]James W. Newkirk, Alexei A. Vorontsov, *"Test-Driven Development in Microsoft .NET"*, Microsoft Press (April 14, 2004)

[50]Kent Beck, *"Test Driven Development: By Example"*, Addison-Wesley Professional; 1st edition (November 8, 2002).

[51]Scott W. Ambler, *"Agile Modeling - Effective Practices for Extreme Programming and the Unified Process"*, John Wiley & Sons 2005.

[52]Scott Ambler, *"Agile Database Techniques"*, Wiley (October 17, 2003)

[53]Joshua Kerievsky, *"Refactoring to Patterns (Addison-Wesley Signature Series)"*, Addison-Wesley Professional (August 5, 2004)

[54]Cem Kaner, James Bach, Bret Pettichord, *"Lessons Learned in Software Testing"*, Wiley; 1st edition (December 15, 2001)