

On the Test Driven Development in the Small Development Teams

Aleksandar Bulajic*, Radoslav Stojic**

* IBM Denmark, Copenhagen, LANB@45.dk, aleksandar.bulajic.1145@fit.edu.rs

** Metropolitan University, Belgrade, Radoslav.stojic@fit.edu.rs

Abstract: While traditional testing methods are based on testing of already existing code and functionality, First -Tests approach, or Test Driven Development (TDD) approach writes a test before implementation code is available. Small project teams are introducing different kind of challenges and most important are limited resources and time constraints. TDD methodology offers better resources utilization, and claims that quality is improved even a total time used for development is shorter. Results of rresearch projects on TDD differ significantly, what draws conclusion that involved IT professionals skills and experience could be a crucial factor. That what can be concluded is that TDD approach provides better test coverage of implementation code and introduces fewer software defects, and better software quality, than software delivered by traditional approach. However, this approach uses approximately more than 15% time for development.

INTRODUCTION

The test-first concept has been introduced in the Extreme Programming, an Agile development methodology created by Kent Beck. Extreme Programming development method originate from the experience that Kent Beck collected by working as project manager 1996 on the Chrysler C3 project. In 1999 he wrote a book “Extreme Programming Explained”, where he presented this methodology that is combined from already known best practice principles. This methodology best fits to the small and mid-size teams.

Kent Beck is also known as designer of the automatic unit test driving framework known as xUnit framework, or today even better known as JUnit ,automatic unit test framework, used by Java programmers or NUnit, automatic unit test framework, used by .Microsoft, .NET programmers (C# programmers for example).

In 2003 Addison-Wesley published another Kent Beck book “Test Driven Development by Example”, that attracts more attention to the test-first concept and replaced test-first programming concept name by today widely used Test Driven Development (TDD) name.

The basic idea behind the Test Driven Development is developing a test before implementation of requirement. The first Section, “Test Driven Development”, describes this approach.

The next Section, ”TDD and a Small Development Teams”, discuss benefits and drawbacks of using TDD methodology, as well as pre-conditions and other tools that automate development tasks and improves team productivity and final product quality.

The Section “Conclusion” is a short discussion and conclusion.

I. TEST DRIVEN DEVELOPMENT

Test Driven Development (TDD) rules defined by Kent Beck are very simple:

1. Never write a single line of code unless you have a failing automated test,
2. Eliminate duplications.

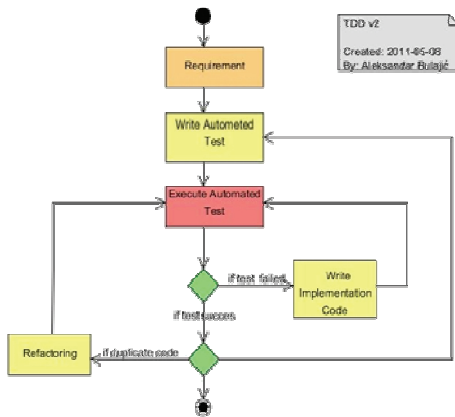
The first principle is fundamental for TDD approach because this principle introduces technique where a developer first writes a test and then implementation code.

Another important consequence of this rule is that test development is driving implementation. Implemented requirements are by default testable; otherwise, it will not be possible to develop a test case.

Second principle is today called a Refactoring, or improving a design of existing code. Refactoring also means implementing a modular design, encapsulation, and loose coupling, the most important principles of Object-Oriented Design, by continues code reorganization and without changing existing functionality.

A. Test Driven Development Work-flow

The following picture illustrates a Test Driven Development process:



Picture 1: TDD workflow diagram

1) Requirement

Requirements in case of the Extreme Programming and Agile development are managed differently than in a case of the traditional approach, where can be used a lot of time and huge communication overhead to specify requirements and requirements relations and interactions. Extreme programming splits development in the short iterations that can last from one to three or four weeks. Each iteration delivers fully functional software that adds new functionality or improves existing.

A short iteration requires that requirements complexity is reduced by dividing in the parts that can be completed in the short development time. A short iteration requires also that development starts very early, and there is no time to wait that whole requirement specification is completed.

Even many Agile and Extreme Programming sites are offering idealized pictures of the requirement management based on the user stories, the time necessary to understand requirement cannot be avoided. This means that even it is possible to start development from the short user story, communication overhead related to the understanding of the requirement details cannot be avoid. The differences are in the time used for specifying requirement and also in the requirement size and complexity. TDD requires less time and starts implementation as soon as some parts of requirements are known

2) Write Automated Test

This is implementation of the first TDD rule:

Never write a single line of code unless you have a failing automated test.

This means that before any line of the requirement implementation code is written, an automated test code is written. Automated test will fail because there is not existing implementation code.

Writing an automated test is not a spontaneous process. A test is chosen from a tests list that is already created during brainstorm sessions. This list contains a list of tests that “verifies requirement and describes the completion criteria”. (Newkirk and Vorontsov 2004)

3) Execute Automated Test

Because implementation code is not yet written and there is existing only empty implementation method to avoid compiler errors, a test will always fail.

The automated testing tool is precondition to implement TDD. All tests or particular test shall be possible to execute automatically. This requirement is very important because developing small tests, step by step, can suddenly end by a test suite or test suites that contain hundreds of tests that shall be executed each time when implementation code is changed or new test is added to the test suite. This kind of testing is called Regression Test and this is very important step in the Quality Assurance process.

4) Write Implementation Code

If test fails then Write Implementation Code should provide implementation code that makes a corresponding test case to execute successfully. The TDD considers a passing test not useful and unnecessary. New test shall always fail and cause changes in the implementation code. These changes can introduce writing of new implementation code or changes in existing code.

5) Refactoring

“Although refactoring code has been done informally for years, [William Opdyke's](#) 1992 Ph.D. dissertation^[15] is the first known paper to specifically examine refactoring,^[16] although all the theory and machinery have long been available as [program transformation](#) systems.” (Wikipedia_Code_Refactoring 2011)

Removing duplicate code today is better known as code refactoring. Code refactoring has been widely popular in recent years and very popular book written by Martin Fowler book “Refactoring Improving the Design of Existing Code” (Fowler 1999) still can be used as a reference book that contains a long list of refactoring techniques.

There are available different definitions and all agree that refactoring is improving design of existing code. Refactoring does not add new functionality or change old one. Refactoring eliminates duplicates, makes code more readable and improves code internal structure.

6) When to Stop TDD Iterations

If all tests are green (successfully executed) then on the Figure “Test Driven Development workflow diagram” are available three paths::

1. Refactoring.
2. Write Automated Test,
3. Stop Further Development.

The first two are described in previous Sections.

The third possibility means that application development is completed. Some authors are calling a Red, Green and Refactor sequence a TDD pattern.

A decision to stop further development can be made when all test cases from a test case list are implemented. This means that all requirements are implemented and there is no reason to use more time and resources.

II. TDD AND A SMALL DEVELOPMENT TEAMS

When IT professional hears about a small team, the first association is a project size. And then come to his mind other key words such as Agile development, Extreme Programming (XP), SCRUM, Lean Methodology and cutting development expenses too. The never-ending dream is to create more by spending less money and time, and in the same time improve quality of the final product.

A project size, is not decisive factor, even does not make sense to create huge team to accomplish a small project. Small in this case is measured by amount of money and required resources. Today are huge project divided to the small project teams where each team responsibility is development of the particular component or even component parts. That depends of the requirements and of the chosen architecture, and project or component complexity. This approach shifts complexity to the integration phase where some or all components are assembled and created an application.

A. *Preconditions for Applying TDD*

Clients and software companies hurry up to transform requirements to the implementation and to see application running, even it is already clear that requirements are obscure as well as benefits of such implementation.

Too many times, clients and software companies are, forgetting to implement proper Quality Assurance procedures. Too often that is discovered when it is already too late to correct it, without introducing delays and huge expenses, because application software is already running in production. Many times development is starting without a proper test strategy, and even more often without sufficient resources to test application properly from the very first beginning.

Why? An answer is not easy to find. The most of involved people would probably justify decisions by expenses and available resources. Doing more by using less becomes a motto that created many issues.

From managers point of view it is not an issue. It is just a question how to force people to do more in the same amount of time. All right, force is not very popular word. Motivation is preferable word. And manager is not more manager, but rather a coach. His main issue is motivating developers to work hard and deliver perfect software for a fraction of time that has been used yesterday? All right, perfect is too strong word. No one is expecting perfect solution. Today a perfect

is replaced by good solution and even solution. Make something that works and then improve it in next iteration, if it is next iteration.

This philosophy is not in conflict by best practice. Software development is an iterative process and new iteration should deliver new functionality or improve existing. Delivery speed and shorten time to market become key words. Robustness and flexibility are also often used, but question is could it fit very well in case when speed is important and resources and time are very limited?

If you ask a question is it possible to deliver robust, flexible and good quality software quickly, you will today get a positive answer as quickly too. Agile development is an answer and too many people really believe that it is a right answer.

I would not say that it is not possible in a small project, where requirements are well understand, and implementation team works together for a while, and role and responsibilities inside of the team are well known. In case of complex projects that involves remote components, more layers, complex interfaces and complex data model and business logic, it is not possible. To make it possible, complex project shall be first divided to a number of a small and middle sized projects or components. But complexity in this case is moved from implementation to integration and it can be very irresponsible to expect that integration will be a piece of cake and works smoothly. In case of complex projects divided to small projects are involved too many people and too many small teams and very often these teams are spread all around a world or are a part of different organization or cultural environment.

In the small teams, there are not available sufficient resources for implementing proper QA, and resources for testing purpose are very limited too. In these teams, it becomes even more important to automate testing and ensure that by introducing new functionality the old one is not broken. This kind of testing is called Regression Testing. Regression testing has to be automated otherwise even in the less complex project it will not be possible manually execute all old tests and check all test results. In case of Agile development method and TDD, iterations are very short and can take just a few weeks.

Productivity and effectiveness of the small development team hardly depend of the available tools and frameworks, and process automation. Important advantage of the small development teams are fast communications. The huge development team introduces huge communication overhead. In the small teams, this communication overhead is set to a minimum. Disadvantage is limited resources, so the same team member has to execute more different roles, for example he act as developer and write a code and in the same time test application and write a test code. In this case there are basically two approaches, one called classic approach where implementation code is written first and then written a number of test cases to test implementation and TDD approach, where first is written a test case and then

implementation code. In case of TDD the same person is writing test case and implementation code.

Successful implementation of TDD methodology is based on fulfillment of certain set of preconditions, and availability of additional components and tools, and frameworks for automate development process.

Preconditions can be summarized as:

1. Developers are able to act as testers
2. If there are testers available then testers should be able to write a code and simple programs (Beck 1996),
3. Project size is a small or middle-size project,
4. Requirements are known even it is not necessary that are fully completed,
5. Developers are familiar by tools and frameworks

Applying TDD in the small development teams cannot be effective without tools that automate development process. Without tools, applying this methodology would probably create ineffective and cumbersome development environment where benefits would be changed to pitfalls and affect a quality of final product.

Requirement for automate development process are:

1. Development Framework and Automated test framework that is able to execute test cases and easily collaborate with development framework,
2. Automatic build procedures,
3. Continuous integration,
4. Development and Test environments for each developer – each developer should have own development environment where is executed his own test. When test and code are ready it is merged to main project branch. Otherwise team would introduce communication overhead to find which changes corrupted test environment and even more time to correct test cases.

In the small development teams time and resources are always most critical factors. This shall be always considered when existing framework or a tool want be replaced by new one.

B. TDD and Standards and Practices

The TDD development approach, as well as any other Agile development approach is based on short development iterations. Iteration can last just a week or two and usually no longer than 3-4 weeks. In the small development teams, where developer and tester and QA manager is the same person, it can be difficult to deliver good quality code on both sides. Short time constraints are introducing additional pressure on developers to deliver planned functionality and

developers many times forget that testing is a first priority task too.

In such case it is important to have standards, procedures and supervision that ensures that every development team member respect and implement it. TDD promotes a Test-First approach and forces developers to write a test first, and then implementation code.

Continuous refactoring and improving of code design removes duplicates and enforces modular design. These are two most important TDD premises

But how many test cases will be developed and how existing code will be refactored depends of a particular developer, project team or company where development process is completed. If company has well described procedures and implementation standards, as well as control mechanisms, then delivered code quality will probably satisfy quality requirements. If any of these not exists, or is a partially implemented, code quality will be unknown, and depend mostly of developer skills and experience, as well as of his personal motivation.

Processes and method purpose is to become independent from human beings behavior and even skills. Processes and methods are designed to provide standard quality despite differences in human beings skills and experiences. Of course, that this is only a theory. In praxes, delivered quality can depends of a person and his skills, and experiences and his personal motivation.

Tools and frameworks are important part of each development and good tools and frameworks can significantly increase developer productivity and effectiveness, as well as a quality of final product.

For TDD critical tool is automated testing framework. It is a mandatory tool, and this tool should be able easy

integrate to an Integrated Development Environment (IDE) framework. Today, this is not an issue, and on the market are available different automated testing frameworks known as xUnit family frameworks and these frameworks are open sources and can be used free of charge. There are also available different open-source IDEs where are automated testing frameworks already integrated or can be very easy integrated.

There are also available automatic build tools and continuous integration tools that are able to execute build procedure in a predefined time intervals and execute automated tests (Unit Tests) and report any errors to developers by sending a mail notification.

A question is what is wrong and why delivered software quality still vary and introduces number of defects that are suppose to be discovered during development process?

An answer is that human beings are not machines and cannot be programmed to repeat exactly the same task again and again and keep motivation and hard work at highest level every time and still be able to think creatively. Even motivation is high, lack of knowledge and experience can

cause a software defect. Internal project organization, procedures, applied standards and processes management can be critical factor and affect significantly final product quality.

Different organizations developed standards that are specific to software products and customers often require that software product meet particularly standards as for example government organizations can require that software product meet ISO 9000 or CMMI standards. The most popular standards for software development are:

1. ISO 9000 standard family,
2. Capability Maturity Model Integration (CMMI),
3. IEEE 1298.

CMMI is very popular and many customers when ordering a software project require that Software Company is certified at least to CMMI level 3. The CMMI defines 5 different levels of maturity:

1. Level 1 Initial – poor process management, quality and schedule unpredictable,
2. Level 2 Managed - established process management based on experiences from previous projects and is by nature reactive ,
3. Level 3 Defined – project defines processes from existing standards and is able to track a progress, ; by nature is proactive and product quality is predictable,
4. Level 4 Quantitatively managed – processes are measured and quality goals are established and measured during project duration,,
5. Level 5 Optimized. – continues process improvement based on evaluation and measurements.

Each of these levels defines a set of documents and procedures and processes that are applied during software development process.

There is also important question what is testing purpose? Verify that software functionality is working as expected or try to break software and discover as many bugs as possible? Test cases written by using TDD are in the first category and usually further testing stops when all test cases are executed successfully. The simplest reason is that further testing can be expensive and reduce time for developing new features.

TDD is testing implementation code by using Unit Tests and this kind of testing is a white box testing approach, what means that tester, in this case the same person who developed implementation code, knows very well internal program structure. This is one of very often criticized TDD approach, because quality assurance standards recommend that developer and tester are different persons and that other person than developer who write a code should test that code.

Unit testing is just one of the tests that should be executed before software is deployed. There are also Integration Tests, System Tests, non-functional tests, such as stress and performance testing and also test of build procedures and deployment procedures, and these are not a part of the TDD..

Writing a test code, as well as writing implementation code, is error prone, and it takes time to design a test and write a code. Test case code can also contain errors and wrongly report test results. This can be caused by changing implementation code or even refactoring, as a side effect kind of error. The worst case scenario is when test case reports successful execution, but should fail. This can be caused by hard-coded values, for example.

Test maintenance can be daunting and time consuming tasks in case when hundred of tests should be corrected because requirement or design has been changed. Separating test case from test data in this case can help a lot and reduce test maintenance time significantly.

Even test tools are open source, that can be used free of charge, time to use it on the real IT project is not free and customers or IT company shall pay for it. Time spend on development of the test cases cannot be used on development of application functionality. These two, even close related processes, and even simultaneously executed, are executed in the separate time frames. This means that more time spent on test development is less time spent on application functionality implementation, and opposite.

Small development team means also a small number of human resources. A size of a small development team is from 3 to 5 people in average and one is a Project or Team Leader. If only one is dedicated to a test activity, it means in average 25% percentage fewer resources that can be used for other activities. If two of peoples are used only for testing activities it means that 50% less resources are used for development activities.

Another important precondition for successful testing implementation in the small development teams is availability of automatic testing tools. Even test development can be done by using simple test editor, test execution shall be automatic.

The simple reason is that manual testing, especially in case of full regression testing, is not possible to execute and successfully complete under time constraints that are defined as milestones for each software application.

Even many would advocate that software quality delivered by Agile development methodology, is better or the same quality as a quality of software developed by using traditional development methodology, I suggest to use common sense and estimate your chances in advance. Important parameters that should be considered are:

1. Project complexity,
2. Target platform (it is suppose that development and deployment under Windows will be easier

than under Unix or, for example under Mainframe computer),

3. Technology requirements (an old technology, emerging technology, mature technology, etc.),
4. Available tools (collaboration, development, build, continues integration tool, deployment and testing tools, frameworks, automatic testing tools, builds, deployment tools),
5. Team size,
6. Team competence (age, experience, are they already know each other or it is first time they work together)
7. Project management,(competence, experience, knowledge related to project requirements, knowledge and previous experience by using Agile development methodology, available procedures and methodologies, available tools).
8. Client (it is an old customer or brand new customer, how much support you can expect from such customer, competence of the people assigned as contact persons, testers and customer representatives),
9. Motivation of all above mentioned and dedication to project success. End user motivation and dedication can be crucial, but if it is not at good level, project management and project team has important task to increase it in positive direction.
10. Available time and resources,
11. Education and career development possibilities.

Even a TDD approach can deliver better software quality, it cannot be achieved only by using TDD, without implementing quality assurance and standards and procedures that ensures that TDD methodology is properly implemented.

C. TDD Benefits and Drawbacks

Wikipedia (Wikipedia_TDD 2011) provides a list of benefits and drawbacks that pretty well summarizes what different authors present as TDD benefits. Following is a short summary of benefits without long explanations (Wikipedia_TDD 2011):

- Writing more tests,
- Debugging is rarely used because development is using small steps and tests are developed before implementation,
- TDD affects implementation code design,
- Implementation code is by definition testable, because test is written before implementation,

- Even TDD requires more code to be written; total development time (test and implementation) is shorter,
- Automated test usually covers every code path,
- More modularized design and loose coupling,
- Number of defects is lower because of high number of tests,
- Improved maintainability and further development. Drawbacks of TDD are (Wikipedia_TDD 2011):
- TDD is difficult to use in case of User Interface (UI) testing, database or service testing. In these cases Unit Tests are often replaced by Integration Tests,
- Management support is essential,
- There is not separation of duties, the same developer is writing a test cases and implementation code. This can affect both, test and implementation code quality,
- Maintenance of tests cases overhead,
- Test coverage during initial development and any next iteration can differ.

But to draw any conclusion about any kind of methodology, it is necessary first to apply methodology on real project and collect experience and then compare to previous experience where is used another methodology.

III. CONCLUSION

The first principle of TDD, “never write a single line of code unless you have a failing automated test” is a fundamental for TDD approach, because this principle introduces technique where test is written before implementation is available and is driving implementation, and requirements are testable.

The second rule is requirement for continues code refactoring. Eliminating duplication of code requires continues source code inspection and changes that should improve internal code design. Developer learns about code internal structure and becomes confident to make changes. If changes are necessary, than developer impact analysis and estimations are in this case much more reliable.

Short development iterations discover requirements misunderstandings very early and reduce expenses for correcting these misunderstandings.

This approach raised very important question about final software product quality, especially because a TDD approach claims that total time used for development is shorter. This means that cutting of development expenses and shortening development time, would not affect final software product quality.

Test-first approach leads to development of a small portion of code and errors are discovered quickly without

using debugger. Implementations code has fewer defects what means that more time can be used for development of new features and functionalities.

Conclusion is that TDD approach provides better test coverage of implementation code and introduces fewer software defects, what means also that delivered software

has better quality, than software delivered by traditional approach.

Claim that TDD approach is using the same amount or less time for project development cannot be confirmed and according to research papers TDD approach uses approximately 15% more time for development

Claim that TDD improves internal software design and makes further changes and maintenance easier cannot be confirmed. It seems that design primary depends of developer skills and experience, as well as of implementation of best practice and internal standards.

REFERENCES

- [1] Beck, Kent (1989), "Simple Smalltalk Testing: With Patterns", available at Internet (<http://www.xprogramming.com/testfram.htm>) (17-03-2011)
- [2] Fowler, Martin (1999), "Refactoring Improving the Design of Existing Code", Addison-Wesley
- [3] Hunt, Andrew, Thomas, David (2005), "The Pragmatic Programmer", Addison-Wesley
- [4] Meszaros, Gerard (2007), "JUnit Test Patterns: Refactoring Test Code", Addison-Wesley
- [5] Meszaros, Gerard , Bohnet, Ralph, Andrea, Jennitta (2003), "Agile Regression Testing Using Record & Playback", OOPSLA '03 Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (<http://agileregressiontestpaper.gerardmeszaros.com/>) (24-05-2011)
- [6] Newkirk, James W., Vorontsov, Alexei A. (2004), "Test-Driven Development in Microsoft .NET", Microsoft Press
- [7] Wikipedia_Code_Refactoring (2011), "Code refactoring", available at Internet (http://en.wikipedia.org/wiki/Code_refactoring) (13-05-2011)
- [8] Wikipedia_TDD (2011), "Test-driven development", available at Internet (http://en.wikipedia.org/wiki/Test-driven_development#cite_note-Beck-0) (02-05-2011)

