Test Driven Development Method in Software Development Process

Denis Duka, Lovre Hribar
Ericsson Nikola Tesla
Research & Development Center
Split, Croatia
denis.duka@ericsson.com; lovre.hribar@ericsson.com

Abstract: **Test-driven development (TDD) is an advanced technique of using automated unit tests to drive the design of software and force decoupling of dependencies. TDD does not replace traditional testing, instead it defines a proven way to ensure effective unit testing. The overall initiative results in continuous improvements and provides a mechanism for propagating the lessons learned between projects to further improve the organization. This paper gives an overview of TDD concept. Some initial results analysis as well as plans for further practice enhancement are also presented.**

1. INTRODUCTION

The pressure to improve software development process is not new, but in today's competitive environment there is even greater emphasis on delivering a better service at lower cost. Despite that, we still do not have reliable tools for ensuring that complicated software systems intended for high-confidence tasks are free from faults and operational failures. Faults may result from root causes that range from knowledge errors, to communication errors, to incomplete analysis errors, to transcription errors. Exacerbating the problem is the ever-growing expectations of the end-users and the growth in the complexity of the tasks. There are essentially three ways of dealing with faults [1]:

1. Fault-avoidance is achieved through appropriate specification, design**,** implementation and maintenance activities intended to avoid faults in the first place. This includes use of advanced software construction methods, formal methods and re-use of reliable self-describing software building blocks (objects), and active knowledge domain support.

2. **Fault-elimination** is the analytical compensation for errors committed during specification, design and implementation. It manifests as verification, validation and testing.

3. **Fault-tolerance** is the run-time compensation for any residual problems, out-of-specification changes in the operational environment, user errors, etc [2].

Absolute fault-avoidance may not be economical or feasible. The next best thing is to eliminate faults as soon as they occur, certainly before they propagate into the operational phase of the system [3]. Given the error proneness of humans, it is prudent to revisit software artifacts one or more times to ensure that our understanding of the issues, our designs, and our implementations are mutually conformant and correct. Process feed-back and feed-forward loops for problem detection and fault elimination [3, 4] are beneficial. These loops may be over different releases of the product, over individual phases of a single release, and/or over individual tasks. Reliable implementation of very tight fault-elimination loops, especially those that are not just reactive (i.e. that result from a problem that needs to be corrected), but also proactive (forward error correction – preventive activities and dynamic process improvement) are generally associated with high Capability Maturity Model (CMM) levels [5]. Additionally, the earlier one finds an error, the less expensive it is to fix [3, 6, 7].

Research and development of techniques for making software development easier and faster have been going on for as long as software has existed. Still, projects commonly spend at least 50 percent of their development effort on rework that could have been avoided or at least been fixed less expensively. That is, 20-80 percent depending on the maturity of the organization and the types of systems the organization develops. In a larger study on productivity improvement data, most of the effort savings generated by improving software process maturity, software architectures and software risk management came from reductions in avoidable rework [8]. A major reason for this is that faults are cheaper to find and remove earlier in the development cycle.

One approach to improve software development process and to reduce rework is Test Driven Development (TDD). It is a programming technique that relies on the repetition of a very short development cycle: First the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards. In other words, it is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring. It is also one way to think through your design before your write your functional code.

The concept of test-driven development has been sporadically used for a long time, e.g. one usage case from as

early as the late 1960's has been reported. However, it became popular with the emergence of the development practice eXtreme Programming (XP). Among the practices included in XP, TDD is considered as one of few that has standalone benefits. The main difference between TDD and a typical test process is that in TDD the developers write the tests before the code. A result of this is that the test cases drive the design of the product since it is the test cases that decide what is required of each unit [9].

This paper first describes the test driven development cycle highlihgting the needed steps. As further described, there are several approaches while implementing TDD model as a part of regular software development process. The following chapters provide comparison with traditional testing emphasizing also some advantages of this technique. TDD implementation in the field is presented in the sixth chapter. Case study results as well as comparision with some previous results are also given here.

2. TEST DRIVEN DEVELOPMENT CYCLE

The test cases can be seen as example-based specifications of the code. In short, a developer that uses traditional TDD works in the following way [10]:

1. Write the test case,
2. Execute the test case and verify that it fails as expected,
3. Implement code that makes the test case passes,
4. Refactor the code if necessary.

Test-driven development requires developers to create automated unit tests that define code requirements (immediately) before writing the code itself. The tests contain assertions that are either true or false. Passing the tests confirms correct behavior as developers evolve and refactor the code. Developers often use testing frameworks to create and automatically run sets of test cases.

The following figure presents the TDD cycle:

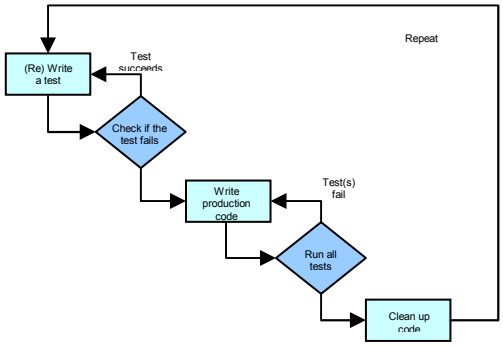

Figure 1 – A Graphical Representation of the TDD Cycle

2.1 Add a test

In test-driven development, each new feature begins with writing a test. This test must inevitably fail because it is written before the feature has been implemented (if it does not fail, then the proposed *new* feature is obviated.) To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through *use cases* and *user stories* that cover the requirements and exception conditions. This could also imply a variant, or modification of an existing test. This is a differentiating feature of test-driven development versus writing unit tests *after* the code is written: it makes the developer focus on the requirements *before* writing the code, a subtle but important difference.

2.2 Run all tests and see if the new one fails

This validates that the test harness is working correctly and that the new test does not mistakenly pass without requiring any new code. This step also tests the test itself, in the negative: it rules out the possibility that the new test will always pass, and therefore be worthless.

The new test should also fail for the expected reason. This increases confidence (although it does not entirely guarantee) that it is testing the right thing, and will pass only in intended cases.

2.3 Write some code

The next step is to write some code that will cause the test to pass. The new code written at this stage will not be perfect and may, for example, pass the test in an inelegant way. That is acceptable because later steps will improve and hone it.

It is important that the code written is only designed to pass the test; no further (and therefore untested) functionality should be predicted and allowed for at any stage.

2.4 Run the automated tests and see them succeed

If all test cases now pass, the programmer can be confident that the code meets all the tested requirements. This is a good point from which to begin the final step of the cycle.

2.5 Refactor code

Now the code can be cleaned up as necessary. By re-running the test cases, the developer can be confident that refactoring is not damaging any existing functionality. The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to removing any duplication between the test code and the production code.

2.6 Repeate

Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. *Continuous Integration* helps by providing revertible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself [11] unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the main program being written.

3. TDD DEVELOPMENT STYLE

There are various aspects to using Test Driven Development. By focusing on writing only the code necessary to pass tests, designs can be cleaner and clearer than is often achieved by other methods [10].

To achieve some advanced design concept (such as a *design pattern*), tests are written that will generate that design. The code may remain simpler than the target pattern, but still pass all required tests. This can be unsettling at first but it allows the developer to focus only on what is important.

Write the tests first. The tests should be written before the functionality that is being tested. This has been claimed to have two benefits. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset, rather than worrying about it later. It also ensures that tests for every feature will be written. When writing feature-first code, there is a tendency by developers and the development organizations to push the developer onto the next feature, neglecting testing entirely.

First fail the test cases. The idea is to ensure that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. This has been coined the *test-driven development mantra*, known as red/green/refactor where red means *fail* and green is *pass*.

Test-driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Receiving the expected test results at each stage reinforces the programmer's mental model of the code, boosts confidence and increases productivity.

Advanced practices of test-driven development can lead to Acceptance Test-Driven Development (ATDD) where the criteria specified by the customer are automated into acceptance tests, which then drive the traditional Unit Test-Driven Development (UTDD) process [12]. This process ensures the customer has an automated mechanism to decide whether the software meets their requirements. With ATDD, the development team now has a specific target to satisfy, the acceptance tests, which keeps them continuously focused on what the customer really wants from that user story.

4. TDD AND TRADITIONAL TESTING

TDD is primarily a design technique with a side effect of ensuring that your source code is thoroughly unit tested. However, there is more to testing than this. You'll still need to consider other *agile* testing techniques such as *agile acceptance testing* and *investigative testing*. Much of this testing can also be done early in your project. In fact, in XP the acceptance tests for a user story are specified by the project stakeholder(s) either before or in parallel to the code being written, giving stakeholders the confidence that the system does in fact meet their requirements.

With traditional testing a successful test finds one or more defects. It is the same with TDD; when a test fails you have made progress because you now know that you need to resolve the problem. More importantly, you have a clear measure of success when the test no longer fails. TDD increases your confidence that your system actually meets the requirements defined for it, that your system actually works and therefore you can proceed with confidence [13].

As with traditional testing, the greater the risk profile of the system the more thorough your tests need to be. With both traditional testing and TDD you are not striving for perfection, instead you are testing to the importance of the system. In other words, you should *test with a purpose* and know why you are testing something and to what level it needs to be tested. An interesting side effect of TDD is that you achieve 100% coverage test – every single line of code is tested – something that traditional testing doesn't guarantee.

In general, TDD is a specification technique, a valuable side effect is that it results in significantly better code testing than do traditional techniques.

5. ADVANTAGES OF TDD

Developers list many advantages to working in this fast, incremental manner. Highest on the list of advantages to practicing TDD are:

- **Quick results:** Developers can see the effect of design decisions within minutes.
- **Flexibility:** Changes are easy because of the short distance between commits.
- **Automatic catalog of regression tests:** If something developed six months ago suddenly breaks under today's code, it is known immediately.

- **Good, clean code that works**

There are also following advantages:

- The suite of unit tests provides constant feedback that each component is still working.
- The unit tests act as documentation that cannot go out-of-date, unlike separate documentation, which can and frequently does.
- When the test passes and the production code is refactored to remove duplication, it is clear that the code is finished, and the developer can move on to a new test.
- Test-driven development forces critical analysis and design because the developer cannot create the production code without truly understanding what the desired result should be and how to test it.
- The software tends to be better designed, that is, loosely coupled and easily maintainable, because the developer is free to make design decisions and refactor at any time with confidence that the software is still working. This confidence is gained by running the tests. The need for a design pattern may emerge, and the code can be changed at that time.
- The test suite acts as a regression safety net on bugs: If a bug is found, the developer should create a test to reveal the bug and then modify the production code so that the bug goes away and all other tests still pass. On each successive test run, all previous bug fixes are verified.
- Reduced debugging time.

Test-driven development offers more than just simple validation of correctness, but can also drive the design of a program. By focusing on the test cases first, one must imagine how the functionality will be used by clients (in the first case, the test cases). So, the programmer is only concerned with the interface and not the implementation. This benefit is complementary to *Design by Contract* as it approaches code through test cases rather than through mathematical assertions or preconceptions [1].

The power test-driven development offers is the ability to take small steps when required. It allows a programmer to focus on the task at hand as the first goal is to make the test pass. Exceptional cases and error handling are not considered initially. Tests to create these extraneous circumstances are implemented separately. Another advantage is, as already mentioned, that test-driven development, when used properly, ensures that all written code is covered by a test. This can give the programming team, and subsequent users, a greater level of confidence in the code.

While it is true that more code is required with TDD than without TDD because of the unit test code, total code implementation time is typically shorter [14]. Large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the tests helps to catch defects early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project.

TDD can lead to more modularized, flexible, and extensible code. This effect often comes about because the methodology requires that the developers think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces.

Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. For example, in order for a TDD developer to add an *else* branch off an existing *if* statement, the developer would first have to write a failing test case that motivates the branch. As a result, the automated tests resulting from TDD tend to be very thorough: they will detect any unexpected changes in the code's behavior. This detects problems that can arise where fixing something later in the development cycle unexpectedly alters other functionality.

6. TDD RESULTS ANALYSIS

TDD programming method was applied on SW development project at Ericsson. The main reason for that was the conclusion that insufficient component testing before delivery to integration tests was caused by deadline pressure that commonly occur shortly before the deliveries. During such time pressure, people tend to deliver the code with lower quality assurance. To address this problem, a central part of the process change was to introduce component-level test-driven development. The reason why TDD could make developers test more efficient is that when writing the test cases before the code it is more likely that the written tests are executed before delivery. From this analysis, a proposal for a concept consisting of component-level test automation and TDD was made.

The concept was implemented into a new product release and results were compared with its predecessors. During both projects personnel turnover was very low and the development processes were stable.

The chosen metric was Fault Slip Through i.e. measuring the number of faults per specific phase especially highlighting the fault slippage from different development phases. FST provides information regarding the number of faults not detected in a certain activity. These faults have instead been detected in a later activity.

In this particular case slippage from previous project phases to Function and System tests was measured. Test results are shown in the following table:

Table 1: Fault Statistics – Project Comparison

	Without TDD	With TDD	Improvement
Product FST to Function Test	73%	67%	8.2%
Product FST to System Test	57%	46%	19.3%

As can be seen from the table, slippage from Unit to Function test and from Function to System test was reduced. Lower result in first case can be explained by fact that one feature in Unit test was not fully following the new approach (due to the specific test environment it was tested off-site) causing the higher fault slippage than expected.

Our future goal is further enhancing the TDD practice in a away to include explicit estimation of the probability that the software system performs according to its requirements. Some productivity impact of such approach will also be examined.

7. CONCLUSION

Test-driven development is a software development practice that has been used sporadically for decades. With this practice, test cases (preferably automated) are incrementally written before production code is implemented.

The most obvious advantage of TDD is the same as for test automation in general, i.e. the possibility to do continuous quality assurance of the code. This gives both instant feedback to the developers about the state of their code and most likely, a significantly lower percentage of faults left to be found in later testing and at customer sites. Further, with early quality assurance, a common problem with test automation is avoided. That is, when an organization introduces automated testing late in the development cycle, it becomes a catch for all faults just before delivery to the customer. The corrections of found faults lead to a spiral of testing and re-testing which delays the delivery of the product.

A side effect of TDD is that the resulting tests are working examples for invoking the code, thereby providing a working specification for the code.

The main disadvantage with TDD is that in the worst case, the test cases duplicate the amount of code to write and maintain. However, this is the same problem as for all kinds of test automation. Nevertheless, to what extent the amount of code increases depends on the granularity of the test cases and what module level the test cases encapsulates, e.g. class level or component level.

Although gathering data for more accurate statistical analysis will require longer period of time, initial analysis that was performed during case study presented in this paper is pointing to some potential areas of improvements in product life cycle.

REFERENCES

[1] L. Williams, E. M. Maximilien, M. Vouk, "Test-Driven Development as a Defect-Reduction Practice", Proceedings of International Symposium on Software Reliability Engineering, pp 66-75, October 1996

[2] J. D. Musa, "Software Reliability Engineering", New York: McGraw-Hill, 1998.

[3] B. W. Boehm, "Software Engineering Economics", Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[4] S. E. Elmaghraby, E. I. Baxter, M. A. Vouk, "An Approach to the Modeling and Analysis of Software Production Processes", Intl. Trans. Operations Res, vol.2, pp. 117-135, 1995.

[5] M. C. Paulk, B. Curtis, M. B. Chrisis, "Capability Maturity Model for Software Version 1.1", Software Engineering Institute CMU/SEI-93-TR, February 1993.

[6] W. S. Humprey, "A Discipline for Software Engineering", Massachusetts: Addison Wesley Longman, Inc. 1995.

[7] I. Sommerville, "Software Engineering", Sixth ed. Harlow, England: Addison-Wesley, 2001.

[8] L.-O. Damm, L. Lundberg, "Results from introducing component-level test automation and Test-driven Development", Journal of Systems and Software, Volume 79, Issue 7, July 2006.

[9] L.-O. Damm, L. Lundberg, "Early and Cost-Effective Software Fault Detection", International Conference on Software Engineering, Proceedings of the 29th international conference on Software Engineering, pp. 560–570.

[10] K. Beck, "Test-Driven Development by Example", Addison Wesley, 2003

[11] JW. Newkirk and AA. Vorontsov , "Test-Driven Development in Microsoft. NET, Microsoft Press, 2004.

[12] L. Koskela, "Test Driven: TDD and Acceptance TDD for Java Developers", Manning Publications, 2007

[13] ***, "Techniques for successful Evolutionary/Agile Database Development", http://www.agiledata.org

[14] M. Muller, F. Padberg, "About the Return on Investment of Test-Driven Development", Universitat Kalsruhe, pp. 6. http://www.ipd.uka.de