Test–Driven Development: 15 years later

Aalborg University
Department of Computer Science
Group SWD103F14 • Spring 2014

AALBORG UNIVERSITY

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
http://www.cs.aau.dk

Student project

Title: Test–Driven Development: 15 years later

Topic: Master's Thesis

Project period:
SWD10, Spring 2014

Project group:
SWD103F14

Group members:
Petr Jasek

Supervisor:
Ivan Aaen

Number of pages: 51
Total pages: 59
Appendices: 1
Finished on: 10-09-2014

Abstract:

The topic of the thesis is research of Test-Driven Development (TDD) and how it changed in last 15 years. It was done using descriptive and later exploratory research.
The thesis aims to explain TDD with all its benefits and shortcomings. It focuses on detail facts that are sometimes misunderstood and it lists some of the most widely spread myths and misconceptions and tries to explain or disprove them.
The thesis also examines the idea of "TDD is dead". Which was created by David Heinemeier Hansson and stirred somehow quiet discussion about TDD.
In conclusion, the thesis lists the valid critiques of TDD and proposes some ways how could TDD address the criticized problems.

Preface

This report was written by group SWD103F14 in the Spring 2014 as a master's thesis project in the 10th semester of the Software Development program at the Aalborg University.

The motivation for the thesis was author's interest in the topic of Test–Driven Development (TDD), popular yet quite controversial software development process, and its usefulness and applicability in software development. The main purpose of the thesis was to research the current state of TDD and understand why it people have such diverse opinion on it.

The thesis describes the TDD and its properties, discusses benefits and shortcomings, gives an overview of different opinions based on experience and presents some facts from empirical researches on TDD. The thesis also describes the relationship of TDD and processes like refactoring and modifiability in software design.

The research was based on available literature, including Internet articles, discussions, videos and also on personal experience with TDD.

I hope the thesis can be useful to anyone interested in TDD whether for the purpose of introduction to TDD, better understanding or as a pool for both positives data and drawbacks regarding TDD.

Author:

Petr Jasek

Table of Contents

Summary

The thesis aims to give a solid overview of Test-Driven Development (TDD). How it was (re)discovered and how it evolved through the time until this point. It shows and explains some of the burdens that TDD involuntarily picked up on its way in form of myths and misunderstandings that gave rise to variety of criticism.

The thesis hopes to serve as an reference for the correct way how to practice TDD together with the rationale for it.

It uses descriptive research to create the current definitions (there are more definitions) of TDD and then it uses exploratory research to find what the correct one. Based on this, justified criticism can be assessed.

Some of the most important yet neglected facts about TDD are explained thoroughly in the thesis.

The myths and misconception are explained and their validity hopefully disproved throughout the thesis. The thesis also lists benefits, shortcomings and challenges of TDD based on current academic research.

The thesis also analyses the debate about TDD being dead that was started by David Heinemeier Hansson and resulted into series of video discussions between Hansson, Kent Beck and Martin Fowler.

In the conclusion the thesis lists the critiques that I consider eligible. Some proposes for addressing the critiques are also made in the conclusion.

Chapter 1

Introduction

It is 15 years since Kent Beck released his book Extreme Programming Explained: Embrace Change [BECK, 1999]. In the book he Beck presents his methodology eXtreme Programming. On of the things the methodology suggests is that we should write our tests before we write production code. This is considered the starting point of (re)discovering of Test–Driven Development (TDD).

TDD is a software development process that has been both highly praised and criticized in the past years. It is very controversial topic and often results into heated discussions between its promoters and adversaries. Sadly, reasonable portion of the argument is based on misinterpreted or misunderstood facts and also myths about TDD.

Unfortunately, it is not only the critics who understand TDD incorrectly. Often TDD proponents presents TDD incorrectly. The critics then denounce TDD because of poor understanding of some TDD promoters. This can be partly seen as TDD's fault as its most common definition is summed up as this [BECK, 2002]:

1. Write a test. Think about how you would like the operation in your mind to appear in your code. Invent the interface you wish you had.

2. Make it run. Quickly getting that bar to go to green dominates everything else. If a clean, simple solution is obvious, then type it in. If the clean, simple solution is obvious but it will take you a minute, then make a note of it and get back to the main problem, which is getting the bar green in seconds.

3. Make it right. Now that the system is behaving, put the sinful ways of the recent past behind you. Step back onto the straight and narrow

> path of software righteousness. Remove the duplication that you have introduced, and get to green quickly.

This definition is very simple yet its vagueness leaves a lot of space for own interpretation. Therefore TDD implementations vary from developer to developer and unfortunately those variations are not always cosmetic. Some of the incorrect interpretations of TDD will be discussed in chapter 7.

In the thesis I will try to present my view of the correct way to practice TDD. I will try to give the rationale for it through a descriptive research together with an exploratory research.

In the second chapter I explain how I did the research and why I chose to do it this way. In chapter 3, I describe what is TDD, how it was discovered by Kent Beck and what was going on in research areas after TDD started to be used. Chapter 4 looks more closely at details of TDD. In chapter 5, I take a look at the myths and misconceptions about TDD. Chapter 6 talks about benefits and shortcomings of TDD and explains challenges that TDD faces. Chapter 7 is analysis of recent debate that spread through the Internet on the topic of "TDD is dead".

The research question I give myself is: **Which criticism of TDD is justified and is there a way TDD can eliminate it?**. My main goal, though, is to gain better understanding of TDD and all that has relation to it but the thesis should also help others to achieve the same.

Chapter 2

Research methods

In the beginning of working on the thesis, I first did a small informal exploratory research in academic papers to see what are the topics discussed regarding TDD.

I did the research in the IEEE Xplore Digital Library[1]. I tried to find all papers on TDD to see what has been researcher. I used the following pattern for the search:

```
("test driven" or "test-driven") AND (development OR design)
```

I found 224 papers since year 2003 (figure 2.1). I ordered them by the year and than by the number of citations of the paper so I can find the most cited papers each year. My goal was to discover some patters that can be later examined more closely. It should be said that not all of the papers focus directly on TDD.

Moreover when I started read through the papers (starting from the most cited ones) I found that there is slight diversity in understanding what TDD means. And that was not only problem I had with portion of the researches. I discuss the main problems in the end of the next chapter.

When I was thinking how to best tackle the ambiguity of TDD (or at least in the understanding of it) I concluded that first, I have to understand it myself. I also need to understand the rationale behind every part of TDD.

2.1 Descriptive research

I used descriptive research to gain better understanding of TDD. I was doing it absentmindedly in an informal way, driven only by the desire to

[1] http://ieeexplore.ieee.org/

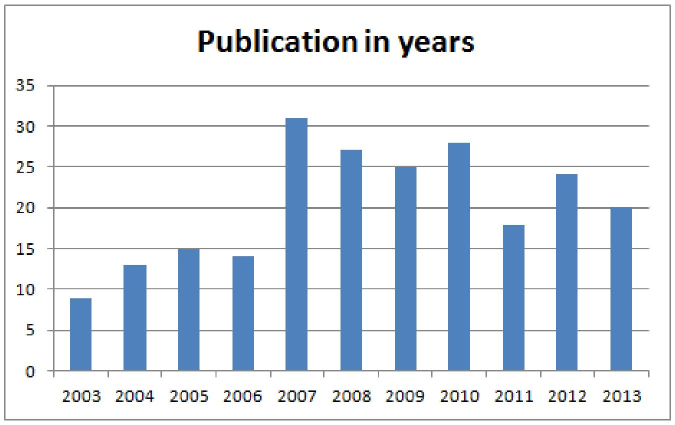


Figure 2.1: Publications in years that consider TDD in some way

find out what are the different views on TDD. I collected various opinions from literature (including Internet).

I started from the roots. I read books written by the author of TDD, Kent Beck. I was looking for his opinions and his rationale for it. This way I got pointed towards other resources but those were mainly TDD supportive resources.

Therefore, I searched the Internet for opinions against TDD to have comprehensive set of data from both TDD promoters and adversaries. At this point I saw that the interpretations of some parts of TDD differ not only among developers but also same people through time refine their definition of TDD. At that point, I could see why TDD is such a problematic topic.

My goal at that moment was to state the different opinions and conclude how TDD should look like in the present time. On top of that I wanted to research the rationale behind it. What are the reasons TDD should be that way.

2.2 Exploratory research

I had collected various data on TDD. Many different opinions from many different people. What I needed to do was find the correct definition of TDD so I can assess which critique of TDD is justifiable.

From this point I was not looking at TDD as a property of Kent Beck. I considered it to be so public and widely used that the definition given by Kent Beck don't have to be necessary the right ones. I was looking for the claims and proves that will make TDD more viable development process.

This way, I hoped, the definition of TDD will become simpler and more understandable. I wanted to find the right questions to ask. Such questions, that the answers for should silence the critics.

Looking for these questions lead me to myths and misconceptions about TDD. I've discovered that a portion of the critiques arises from myths that spread both between TDD critiques and practitioners. I also found out that some of the TDD claims, like increased quality or productivity, are examined in papers and some of the papers confirm them. This made me check academic research again for empirical studies with some conclusive results. I drew from it some of the benefits, shortcomings and challenges of TDD.

Lastly, when I discovered the "Is TDD dead?" debate [HANSSON et al., 2014], I knew I have to analyze it and compare it to my findings because it was debate between people that greatly affect the TDD topic. The debate had probably the biggest impact on my understanding TDD.

Chapter 3

What is TDD

In this chapter we will talk about TDD as it is today, about the aspects that define TDD, about its history and also about the research that was done with regards to TDD. The history is not discussed at first because it needs some explanations and definition of TDD, first.

3.1 TDD cycle

TDD is a software development process that relies on the repetition of a very short development cycle. The cycle is sometimes called the red, green, refactor cycle 3.1. You repeat it until you have tests for all the functionality your software needs and all of the tests pass. The three steps of the cycle were shortly described in the first chapter. Now we will look at the cycle more closely. The description is based on [BECK, 2002; MARTIN, 2007; JEFFRIES & MELNIK, 2007] and personal experience.

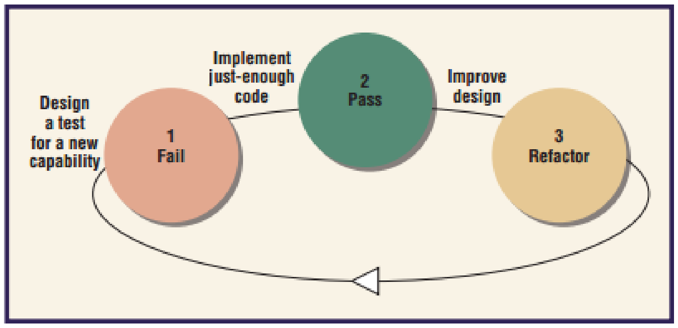

Design a test for a new capability
1 Fail
Implement just-enough code
2 Pass
Improve design
3 Refactor

Figure 3.1: Red, green, refactor cycle

3.1.1 Step 1 – Red

In the first step, your task is to write a test that should be focused on a new capability of your software. The test should fail showing that this capability is not implemented yet. This doesn't always happen and sometimes you write a test that should fail but it passes. There can be two main reasons for it and they are both considered as positive feedback.

First, probably the most common reason, is that you made a mistake while writing the test. This is an example of how the tests work as double checking (discussed in the next chapter). Your tests checks your code as well as the code checks your tests. Here the code discovered that you made a mistake in your test because it passed even though it shouldn't.

Second reason, why a test might pass, is that you are testing functionality that is already implemented. Maybe the functionality was implemented as a side effect of something else or it was implemented without your knowledge. In either case, you gain a better understanding of your software when you have to examine why test passes when it shouldn't.

Of course, there are also situations where you get false positives or false negatives results of tests. You might, for example, try to test a feature that is already implemented but make a mistake in the test. The test then fails even though it should pass.

You should never write more of a unit test than is sufficient to fail and you should never write more than one failing test. When you write a failing test you should move to the next phase.

3.1.2 Step 2 – Green

In the second phase, you write production code. The key is that you should write only enough code that is sufficient for the test to pass. This is sometimes very hard because you are forced to write a code that, you know, will change. You often already know what you want to test next and how will you make the next test pass. I will give simplified example.

Imagine you are making a function that calculates factorial. You already know the algorithm how to calculate factorial but to follow TDD you first write a test that factorial of 0 is 1. To pass this test you should implement the function in a most simple way possible. So its only line of code can be `return 1`. This will make your test pass. Kent Beck calls this practice *Fake It ('Til You Make It)* [BECK, 2002].

Your second test could be that factorial of 1 is 1. When you write the test it will actually pass because your factorial function already returns 1. The frustrating thing is that you know that the function shouldn't just return 1. You know it will change and you even know how it should change because the algorithm to calculate factorial is well known. There is a reason why TDD is so strict about doing it this way.

It is hard do set a boundary what is a well known algorithm that doesn't have to be tested. The solution for calculating factorial is well known and writing tests for it might seem unnecessary. But we don't want to test only the validity of the algorithm. We should also test for programmer errors like typos and such. In practice, most programmers, who follow TDD, would write test then implement the known algorithm, then write all other tests where. They will check after every test whether it really passes. Then they can write tests for special cases like exceptions and those will fail once again returning them tu classical TDD cycle. This approach can turn against you because you are not coding 'test–first' and you lose some advantages of it. Your tests suite can be incomplete because it is usually more demanding to write tests when your implementation is already done.

3.1.3 Step 3 – Refactor

Last step of the TDD cycle is refactoring. Once your test runs you should clean up after yourself. Because the TDD in the green phase suggests to implement things the most simple way to only past their test, the solution is often 'dirty'. In the refactoring phase you should deal with this problem.

While refactoring you should never add new functionality. Sometimes, you might be tempted to do it because refactoring shows you some smart way how to accomplish something but you shouldn't.

Focus should be put on removing duplications, increasing readability and expressiveness of your code but also refactor to reduce coupling and increase cohesion. To sum up, you should improve the design of your code. That relates both to the tests and implementation. Sometimes people forget that they should also refactor tests.

Refactoring doesn't address only the test and production code written in the last iteration of the cycle. You should refactor any part of the code that needs it. There is an unwritten suggestion that you don't need to fully refactor your code in each iteration. You proceed so rapidly that to refactor something that you know will change in the next iteration is wasting. This should, however, not be excuse to postpone refactoring when it is truly needed.

Once you refactor your code or even during the refactoring you should run your test suite. The tests should all pass. This way you know that you didn't brake anything. If you broke something and you don't know how you can always revert all the changes that you did during the refactoring. Once you done refactoring, you can proceed to next iteration (test) of the red, green, refactor cycle.

3.1.4 Three laws of TDD

The cycle is sometimes summed up into the three laws of TDD [MARTIN, 2007] that the TDD practitioners follow. This laws make sure that you are locked in the red, green, refactor cycle:

1. You may not write production code unless you've first written a failing unit test.
2. You may not write more of a unit test than is sufficient to fail.
3. You may not write more production code than is sufficient to make the failing unit test pass.

Following these laws perfectly doesn't always make sense. Sometimes you'll write a larger test. Sometimes you'll write extra production code. Sometimes you'll write tests after you've written the code to make them pass. Sometimes it'll take more than two minutes to go around the loop. The goal isn't perfect adherence to the laws — it's to decrease the interval between writing tests and production code to a matter of minutes. [MARTIN, 2007]

3.2 Other aspects of TDD

There are plenty other aspects to using TDD than red, green, refactor cycle. We will talk about the ones I consider most important or not enough understood.

3.2.1 TDD is more than Test-First programing

There is a notation problem going on in the TDD territory. TDD used to be called Test-First development. Kent Beck changed the name while writing the book Test-Driven Development: By Example [BECK, 2002]. It was to address the fact that the tests in TDD are not for the purpose of testing itself but more for the purpose of driving the design.

The problem from my point of view is that you can practice Test-First without doing TDD. You can have the whole design of your system done but during the implementation you will write your tests first. That is not TDD. Although TDD uses Test-First, it adds on top of it. In TDD your tests should drive the design of your system. I mention this problem because I have witnessed teams claiming that they are doing TDD but all they were doing was just writing their tests first ignoring the design.

I talk more about the design part of TDD in chapter 5.

3.2.2 Keep it small and simple

There are several principles in TDD (sometimes coming from XP) that express the idea of keeping things small and as simple as possible.

'Keep it simple stupid' (KISS) principle states that most systems work best if they are kept simple rather than made complicated; therefore simplicity should be a key goal in design and unnecessary complexity should be avoided [WIKI, 2014a].

'You aren't gonna need it' (YAGNI) is a principle of eXtreme Programming (XP) that states that a programmer should not add functionality until deemed necessary. XP co-founder Ron Jeffries has written: "Always implement things when you actually need them, never when you just foresee that you need them." [JEFFRIES et al., 2000]

TDD makes great use of unit testing. By unit is usually meant a piece of code that can be tested separately from its surrounding. Keeping units relatively small is claimed to provide critical benefits, including [Pathfinder Solutions, 2012]:

- Reduced debugging effort – When test failures are detected, having smaller units aids in tracking down errors.
- Self-documenting tests – Small test cases have improved readability and facilitate rapid understandability.

3.2.3 TDD is not only about unit testing

Even thought TDD literature mainly uses unit tests for demonstrations, those are not the only tests used in TDD. TDD can incorporate all kinds of tests you need to use. Integration tests, functional tests, acceptance tests[1], system tests and other.

[1] Acceptance Test-Driven Development (ATDD) is related to TDD but the main design driver are the acceptance tests. The tests are more oriented on communication with users whereas in TDD the tests are oriented on developers.[POGH, 2011]

The high focus on unit tests affects TDD in negative way because the use of unit tests is criticized among some developers [COPLIEN, COPLIEN].

3.3 History of TDD

Lets look shortly at the history of Test–Driven Development and the things that affected it. I will mention talk about things that had some effect on TDD.

In the 1960's, programmers for the Mercury Space Program use a form of TDD while programming punched cards. With punched cards, people had to be very sure that what they write actually does what it should because they had only few runs on the actual machine. Therefore they first wrote the expected output and then they wrote the 'code' [FEATHERS & FREEMAN, 2009].

Kent Beck read about this in one of his father's books (his father was a programmer). He tried it out and found out it is a great idea. [BECK, 2012] Beck says about it:

> *I was finally able to separate logical from physical design. I'd always been told to do that but no one ever explained how.*

You are able to make such a separation because tests specify what you want before you think how you want to do it. And so the modern TDD was rediscovered by Kent Beck. It started as a part of the eXtreme Programing (XP) methodology in 1999 that was also created by Kent Beck. In the book eXtreme Programing Explained: Embrace Change [BECK, 2004] Beck doesn't mention TDD yet but the XP methodology puts great focus on testing. It even encourages to practice Test–First development:

> *Code and tests can be written in either order. In XP, when possible, tests are written in advance of implementation.*

The book also strongly emphasize the importance of testing by quoting one of the most controversial mottoes of TDD:

> *Testing is as important as programming.*

In year 2000 the JUnit.org website is launched. JUnit is a unit testing framework for Java programming language. It was developed by Kent Beck and Erich Gamma. It is part of the xUnit[2] family of unit testing frameworks.

[2] Another xUnit framework, the Nunit, was registered also in year 2000

Unit testing plays significant role in TDD and easy to use frameworks thus made TDD more accessible to programmers.

The same year Beck and Gamma write article Test Infected: Programmers love writing tests [BECK & GAMMA, 2000]. In the article they show high enthusiasm for testing and they use it to promote JUnit:

> *Testing is not closely integrated with development. This prevents you from measuring the progress of development – you can't tell when something starts working or when something stops working. Using JUnit you can cheaply and incrementally build a test suite that will help you measure your progress, spot unintended side effects, and focus your development efforts.*

They proceed to build on the fact that testing is important and they show and example of a programming task that they solve while using testing and JUnit. They also touch the subject of Test–First development:

> *The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run.*

In the same year another topic closely related to TDD was introduced. At the XP 2000 conference a first paper regarding Mocks was presented [MACKINNON et al., 2000]. Mocks are one of the controversial topics that are related to TDD (discussed in chapter 7). From this point frameworks for mocking are created.

In 2002, Kent Beck publishes the TDD book – Test Driven Development: By Example [BECK, 2002]. Obviously one of the most important milestones regarding TDD. In the book, Beck presents TDD on two examples. He also dedicates part of the book to patterns for TDD where he presents solutions for various testing problems that can occur when practicing TDD.

From this point TDD starts, in a way, its own journey and research papers on the topic of TDD start to emerge.

3.4 Research of TDD

In this section I will talk about TDD from the point of view of research papers. I went through reasonable portion of papers but the amount of things I could conclude from them was disappointing.

I will start with problems I found in academic papers. First of all, in some papers the understanding of TDD was not sufficient for me to accept their results. There were a lot of studies that used students and/or very small scale projects. Even further in some of the studies people were using TDD for the first time. This facts can radically skew the results.

Some papers were confusing Test-First development and TDD. Because of the fact that Test-First is part of TDD, the results of such a paper could still be used but not as result for TDD but result for Test-First. In one such a paper authors write:

> *Our qualitative survey indicates that developers may favor a modified TDD in which some upfront designing is done before developing the code using the test first style.* [GUPTA & JALOTE, 2007]

TDD doesn't dictate no design up front. I talk more about this in chapter 5.

Another problem with academic papers is that drawing general conclusions from empirical studies in software engineering is difficult because any process depends to a large degree on a potentially large number of relevant context variables [WRIGHT et al., 2010].

Also, the results of empirical studies are not sparingly in opposition, at least the studies regarding TDD, I did examine. As an example can serve following extract from [SINIAALTO & ABRAHAMSSON, 2008]:

> *The results provide some warning that the benefits of TDD are not automatic and as self-evident as expected.Some of the findings imply that TDD may produce a less complex code while other findings indicate the opposite. The existing empirical evidence supports the claim that TDD yields improve external quality, especially when employed in an industrial context. This finding clearly conflicts with the case study which identifies certain risks in the adoption of TDD.*

From articles from an issue of journal IEEE Software [IEEE Software, 2007] that focused on TDD, some conclusion could be made:

- TDD developers apply more effort but achieve better quality.
- Effects of TDD are more visible on real projects.
- It is very hard to get meaningful results.
- There is some dissent about the results.

It should be acknowledged that even though the results in many articles cannot be fully trusted, most of the articles mention this in the “Threats to validity” section. These were the main reasons that concerned me about empirical research in the TDD area.

Chapter 4

TDD under microscope

In this chapter we will look at TDD more in detail. There are some parts and effects of TDD that are under–emphasized or not enough understood yet very important. I tried to pick the most important ones.

4.1 Importance of testing

In the eXtreme Programing book [BECK, 2004], Beck emphasize (among other things) how important it is to test. He makes several statements regarding testing. For example, he expresses that it is important to test early. He claims:

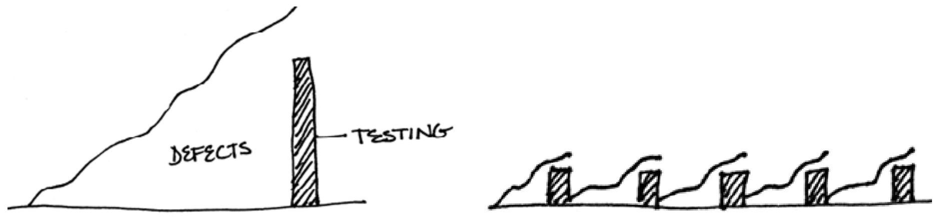
> *Defects in software are expensive but eliminating them is also expensive. However, most defects end up costing more than it would cost to prevent them.*

He introduces two principles for testing in XP. In the first, he explains that software testing is double-checking. The test says what you want to achieve and the implementation should fulfill it. And he follows:

> *If the two expressions of the computation match, the code and the tests are in harmony and likely to be both correct.*

As a second principle, Beck mentions the Defect Cost Increase (DCI) which is according to him one of the few empirically verified truths about software development. He expresses it in a drawing (figure 4.1 and says:

> *The sooner you find a defect, the cheaper it is to fix. Catch a defect the minute it is created and the cost to fix it will be minimal.*



(a) Late testing **(b)** Frequent testing

Figure 4.1: Difference between late and frequent testing

XP says that testing should be done early, often and the tests should be automated. People sometimes test on the fly while debugging by making some assertions in their code to see where the actual bug occurs. Problem with such testing is that it is done after you have discovered a bug, it is not automated and it will be most likely removed after the bug is found and fixed. If it will be removed it is not repeatable and if such a testing is left in the code it pollutes it because it is not really part of the code. It is a test. You might as well write a unit tests. You can keep them and it will test your software continuously whenever you run your test suite.

In TDD you write your tests first so in theory you should avoid defects completely. That is of course not possible. Also the approach to fixing bugs is different from the traditional. In TDD, when bug is found, you write a test or tests that should reproduce this bug. This test must fail, showing that they really simulate the bug. Then you change the code for the test to pass. This should fix the bug.

Convince the development team to add new tests every time a problem is found, no matter when the problem is found. By doing so, the unit test suites improve during the development and test phases. [NAGAPPAN et al., 2008]

TDD considers tests equally important as the code. This point will be further discussed in chapter 7 as one of the controversial topics. Tests are not just tests in TDD. They are considered also the API documentation and designing 'assistants' [MARTIN, 2008; BECK, 2014].

The importance of tests does not mean that they cannot be removed. If you find you have tests that test the same functionality or tests that become invalid (maybe the interface has to change) you can remove them. Nevertheless, it is often difficult to be sure that you can remove tests safely. The rule is, if you are not 100% sure the test is worthless, don't delete it [BECK, 2002].

One last thing about tests is that they can be a way how to measure your progress. Whenever you make a test pass you made a progress. Whenever you have tests for all functionalities of a feature and all of those tests pass, the feature is implemented.

4.2 Cost of defects

As mentioned earlier in this chapter, one of the reasons why people want to test early is that they are aware of the increase of the cost of defect per time. That is, later discovered bugs are more costly to fix. But is it true? And if it is true, how big is the increase in the cost.

The original idea probably comes from Barry Boehm. In his paper, Understanding and Controlling Software Costs [BOEHM & PAPACCIO, 1988], he suggests that the defects found 'in the field' cost 50–200 times more to correct than those corrected earlier. The 50–200 increase is staggering but it must be taken into account that this was 25 years ago. In that time, most software products were developed with waterfall–like processes. Therefore fixing a defect found in the testing phase caused by incorrect analysis of problem domain could indeed present large cost increase.

In paper from 2001 [BOEHM & BASILI, BOEHM & BASILI] Boehm indicates that for small non-critical systems, the cost increase ratio is much smaller. But does it apply to all defects?

This question is mentioned by critics of the Defect Cost Increase (DCI) idea. Naturally there are some types of bugs whose fixing cost might seem invariant. I would use an anecdotal example from blog article The Tyranny of Always [BOLTON, 2009]. In it, the author writes:

> *This has happened dozens of times for me, and I rather suspect that this has happened at least once for large numbers of testers: just before release, you find a bug, you show it to a developer, and she says, "Oh. Cool. Just a second. [type, type, type]. Fixed." That didn't cost 10,000 to one.*

I agree with the author in this but I have a thought to add and I think it is quite important thought. The DCI is not the same for all defects. The defect might be easy to fix as long as you don't build on top of them. So maybe you can fix the defect now with the same cost as you would a year before. That doesn't mean that the DCI doesn't apply. The cost just might increase later in the development if the bug is not fixed. The DCI is also tightly coupled with the 'quality of code'. If the code is clean and objects, modules, etc. are well isolated the bugs will also be isolated and will most likely affect less code. DCI won't be so high in such a software.

The main problem I see with DCI is that it is viewed too much as a dogma and there has to be some number that defines how big is the Defect Cost Increase instead of just accepting that if you are not fixing your bugs they might cost you much more later.

DCI is also used as a metric for a software but as such it has many faults. One issue connected to DCI as a metric is that it penalizes quality [JONES, 2012]). Considering that you spend 10 hours for writing tests. Lets imagine two situations. In the first one your code is bad and your tests discover 10 bugs and you spend 10 hours on fixing them. That is you spend two hours for a finding and fixing a bug. In the second situation your code quality is very high. Your tests discover one bug and you spend one hour on fixing this bug. But now you spend 11 hours to find and fix one bug. The better the quality of your code is the more costly your tests might look.

People then might argue that good programmers could write less tests because they are less likely to make mistakes but there is another reason why you should have tests for your code and that is, to lower the fear of changing the code.

4.3 Code refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written. [FOWLER et al., 1999]

The code should be 'clean' because by continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code. [KERIEVSKY, 2004]

A study on refactoring [BAVOTA et al., 2012] examines its relation to newly induced bugs into the software on three Java software systems: Apache Ant, Xerces, and ArgoUML.

The study finds that in general, there is no significant difference between classes involved or not in refactoring in the proportion of bug–fixing. Such differences occur in some specific cases, and in all of them classes involved in refactoring have a higher chance of being involved in bug-inducing changes.

The study also finds that some kinds of refactoring, namely pulling-up method or inlining a temporary variable, are much more likely to induce bug fixes. The table from the study can be found in Appendix A.

In another study, from 2014 [KIM et al., 2014], developers were asked about the benefits they observe from refactoring. Results (figure 4.2) show that developers indeed associate refactoring with some improvements in their software of which the most beneficial are improved readability and maintainability. Refactoring also decreases the number of bugs and makes adding new features easier.

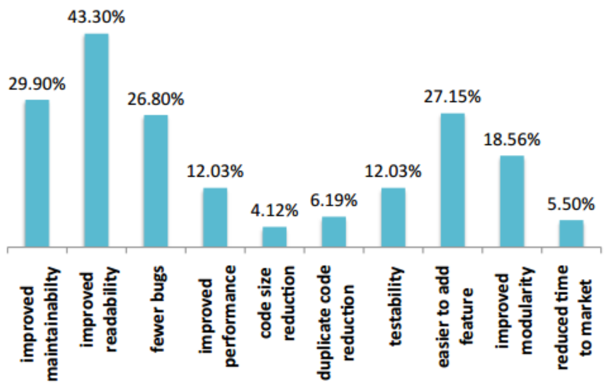


Figure 4.2: Benefits of refactoring

In the same study, the developers were asked about the risk factors that they associate with refactoring (figure 4.3). This result shows that most developers consider regression bugs and build breaks the biggest problem. To quote one developer's response from the study:

> *The primary risk is regression, mostly from misunderstanding subtle corner cases in the original code and not accounting for them in the refactored code.*

This point is understandable and it makes complete sense. If you are refactoring someone else's code you might not know all its implications. Therefore you can misunderstand it and break it while refactoring. There is a well known solution to this problem, though. Tests! Martin Fowler writes in his book on refactoring [FOWLER et al., 1999]:

> *Whenever I do refactoring, the first step is always the same. I need to build a solid set of tests for that section of code. The tests*

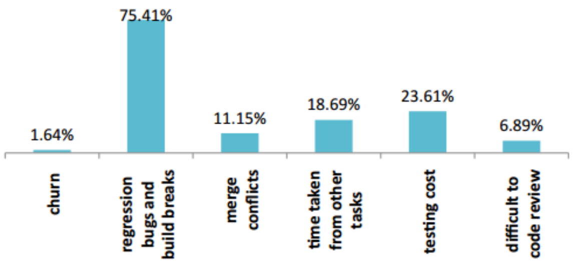


Figure 4.3: The risk factors associated with refactoring

are essential because even though I follow refactoring structures to avoid most of the opportunities for introducing bugs, I'm still human and still make mistakes.

Developers perceive that there is no safety net for checking the correctness of refactoring edits when the test adequacy is low [KIM et al., 2014]. The fear of changing the code is one of the strongest points for TDD. In theory, with TDD you should have 100% test code coverage because you cannot write production code without having a failing test. This is unrealistic assumption as there are always things slipping through that are not tested. Nevertheless, by using TDD you should achieve very high code coverage.

High code coverage means that your code is safe to refactor because if you break some functionality, some tests will fail and inform you about that. This increases the confidence of developers in refactoring the code. The fear of introducing bugs is lessened by the tests.

To sum up the refactoring should be the way to well designed system (regardless of TDD). There are two ways. First, you have a system and you want to improve the design. To do it you want to refactor but to refactor you should have tests. TDD goes from the other side. You write tests so you can refactor so you can improve the design of your system.

4.4 Code modifiability

Modifiability is a quality attribute of the software architecture that relates to the cost of change and refers to the ease with which a software system can accommodate changes [BACHMANN et al., 2007]. In other words it is a quality that reduces cost of changing (including adding) requirements of the software.

It is important quality because 40 to 80 percent of software cost is consumed by maintenance. Further, enhancements are responsible for roughly 60 percent of software maintenance costs. So, software maintenance is largely about adding new capability to old software, not about fixing it. [GLASS, 2001]

One of the reasons for the high cost of software maintenance is that productivity of changing existing code is at least an order of magnitude lower than developing new software. Incorporating anticipated changes, changes for which the software system has been prepared, requires generally considerably less effort than including unanticipated changes. [PerOlof BENGTSSON, 2000]

For reasons mentioned above, the software should be designed to be modifiable. Modifiability is connected to coupling. Less coupled system should be easier to modify because the changes you make should have only local impact. As discussed further (in chapter 7) there are always trade–offs. When you are designing you should be aware of it.

For example, when pushing low coupling to the boundaries you can introduce level of abstraction that might not be needed. It improves modifiability but for the cost of harder to understand, less cohesive code.

TDD supports modifiability in the way it supports modularity. By having tests first, you automatically have a user of your code. You have to think first how you will use your code. This, however, does not secure modifiability. Good design is very important for modifiability and that is not achieved by TDD itself. The developer, even when practicing TDD, has to take the design into account. More about this in chapter 5.

For the software to be modifiable, the developers also have to think ahead. You have to design for modifiability and you should put focus on it in parts of the software that are likely to change.

There are four concerns regarding modifiability [BACHMANN et al., 2007]:

1. Who makes the change?
2. When is the change made?
3. What can be changed?
4. How is the cost of change measured?

The advantage of TDD is reliable (to some extent) test suite. Because of that you don't have to worry to much if you are changing some else's software. There are tests that should warn you in case you misunderstand something and brake the code. Therefore, it should not matter who is making the changes.

The time in software life cycle, when the change is made, can be important to some extent depending on the severity of the change. Most software tends to deteriorate in terms of quality and therefore modifiability. Nevertheless, TDD is an incremental, iterative practice and it should not matter much how far in the development are you.

The answer to the question "what can be changed?" should be everything, especially in well designed systems. Nevertheless, I can see situation where software is working in cooperation with some hardware that has some physical limits. The software might be bound by these limits and therefore not modifiable in this way.

How can you say, how costly will some change be. To find the answer might be very complex problem that requires great amount of developers experience as well as knowledge of the code. I believe that tests can help as a documentation in a way that you can see the API of your software and you can better guess how much will need to be change to implement the new functionality.

To sum up, my opinion is that TDD fosters modifiability but cannot be considered as a way to achieve it because there is still requirement for experienced developers that can foresee areas in the software that are likely to change.

Chapter 5

Myths and misconceptions about TDD

When you browse the Internet discussions about TDD you can encounter many myths and misinterpretations of TDD. The myths are spread equally between followers and opponents of TDD. Let's look at the most harmful myths about TDD.

5.1 There is no design in TDD

There are different variation of this 'problem' and it is both myth and misconception. As written before, TDD is not only "write your tests first". Yes, TDD requires developers to write tests before the implementation but that doesn't mean there is no design.

TDD developers should design their software with the help of their tests. The test should make you think about how will you use the implementation before you write it. But that is not all.

TDD was never saying you cannot design before you start coding. It merely follows the path of agile practices that you shouldn't spend a lot of time on designing up front because your design is likely to change as you gain better understanding about your domain. That usually happens during coding as you discover issues with the design.

There is related misconception that a great design (architecture) will magically appear just from the tests. That is, naturally, not true. Even in TDD you should think about your design. But you should let the tests guide you by 'listening' to them. If something becomes hard to test it means it is hard to use. That usually means that there is a design problem and you should refactor. This is the way tests should improve your design.

TDD shortens the feedback loop on design decisions [BECK, 2002]. So it is you who makes design decisions. After or during that, by writing a test, you examine how it would the design decision affect your software. So before you write a test you should think about it in its context. You should always take the context into account. It would be foolish to think that standard design principles can be ignored just because you are doing TDD.

5.2 TDD wastes developers' time

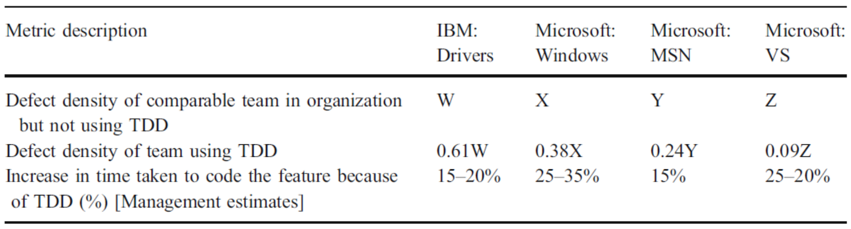
It is true that if you won't write any tests for you software, it will probably take you less time to write. That, however, depends highly on the amount of bugs you introduce into your software. More bugs you make, the more time it will take you to find them and fix them. If you write tests for your software it should not matter very much whether you write them first, last or as a combination of both ways.

A study about effectiveness of test-first programming [ERDOGMUS, 2005] concluded that Test-First programmers write more tests per unit of programming effort. In turn, a higher number of programmer tests lead to proportionally higher levels of productivity. Thus, through a chain effect, Test-First appears to improve productivity. The study also speculates that Test-First might give developers better understanding of the software:

> *We believe that advancing development one test at a time and writing tests before implementation encourages better decomposition, improves understanding of the underlying requirements, and reduces the scope of the tasks to be performed.*

Another study's [NAGAPPAN et al., 2008] findings suggest that even though TDD increases the time of development it lowers the defect density (figure 5.1). From an efficacy perspective the increase in development time is offset by the reduced maintenance costs due to the improvement in quality.

The study also mentions that further releases of a software developed with TDD experience low defect densities due to the presence of the tests. So whether TDD 'wastes' time depends on how long will the software's lifespan. Longer the software's lifetime the more useful the tests will be as they will lower the defect rate.



Metric description	IBM: Drivers	Microsoft: Windows	Microsoft: MSN	Microsoft: VS
Defect density of comparable team in organization but not using TDD	W	X	Y	Z
Defect density of team using TDD	0.61W	0.38X	0.24Y	0.09Z
Increase in time taken to code the feature because of TDD (%) [Management estimates]	15–20%	25–35%	15%	25–20%

Figure 5.1: TDD – Defect density and time increase

5.3 You have to test everything

This misconception is mainly represented by the argument whether or how you should test private members of your class but there is no easy answer for it. It has a lot to do with the design of your code or so called code visibility.

In TDD you shouldn't encounter a situation in which you write a test and cannot access what you want to test because the implementation shouldn't exist yet. To test a method you have to use it as some other part of your software or user of your API would use it. Therefore it cannot be private. You can still have private methods but as said in the book Pragmatic Unit Testing [HUNT & THOMAS, 2003]:

> *In general, you don't want to break any encapsulation for the sake of testing (or as Mom used to say, "don't expose your privates!"). Most of the time, you should be able to test a class by exercising its public methods. If there is significant functionality that is hidden behind private or protected access, that might be a warning sign that there's another class in there struggling to get out.*

A problem can occur if you want to decompose, for example, more complicated algorithm into smaller functions. While developing, you write tests for the individual parts to help you make sure that each part does what it should. Once you are done, you find out that you are exposing the details of the algorithm. You want to change the methods that are implementation details to be private but that would break your tests. The dilemma now is whether to keep your implementation details exposed or delete the tests for them. Removing the tests shouldn't lower the confidence about the correctness of the algorithm because it should be tested as a whole in the first place but it isn't always trivial decision whether tests can be removed.

There are other parts of software that you don't have to test or at least don't have to Test-First. The parts include external frameworks and libraries you are using, code that has non–deterministic results and code that deals only with user interfaces [KAPELONIS, 2013]. Also code that contains facts with no logic like files with constants or configuration files shouldn't be tested [HALL, 2010].

5.4 Code coverage and quality assurance myths

There is a common myth about test code coverage. If you do TDD right you will achieve 100% code coverage. That is not only unrealistic but it can be also dangerous. Code coverage is often considered a reasonable metrics for the quality of your software but mostly, it is misleading metric. It can even turn against you if you chase code coverage just for the sake of having it as high as possible.

Code coverage takes into account only different paths in code. If you have two tests of a function, each test with different input values, the second test might not increase the code coverage. It might, however, test whether your function gives correct results for two different inputs. Some execution paths should be tested literally dozens of times with different edge values, unusual cases, and so on [BINSTOCK, 2011a].

There are also parts of software that cannot be properly cover, for example, the user interfaces. Also, there are things you implemented without writing a test for it first (either knowingly or unknowingly). Also there might be hidden requirements you didn't consider so you don't have tests for it, therefore also no code for it. You can still have 100% code coverage but you are not testing something you should.

The fact is that you can never fully trust your tests. There is several reasons for that. First, as mention above, you might not have all tests that are needed. In reality, for complex software, you never do have all the tests. It is one of the problems of unit testing. People are lured into feeling false security. Because I have tests and all are green, my software has no defects.

When all your tests are passing, your code is not ready for production. It is ready for some kind of quality assurance. An extra layer of testing by people who don't have the same assumptions as the developer who wrote the code [Let's Code, 2014]. TDD does not replace quality assurance [PAMIO, 2012].

Chapter 6

Benefits, shortcomings and challenges of TDD

In this chapter I will talk about the benefits, shortcomings and challenges of TDD. TDD has many benefits and shortcomings. Some of them are proven by empirical studies (with varying level of reliability). Sometimes a benefit or shortcoming of TDD is more related to the tests and testing alone but as TDD is build around tests I consider it still relevant. Also some benefits when overused can be considered as a shortcoming.

6.1 Benefits

High code coverage is one of the biggest and easiest to prove benefits of TDD. If you cannot write code without having a test for it, all your code has to be tested (in theory). For example, for a TDD developer to add an else branch to an existing if statement, the developer would first have to write a failing test case that motivates the branch [WIKI, 2014b].

Thanks to high code coverage **TDD reduces the fear of changing the code**. Developers are more confident to play with their code if they know they have a robust test suite to support them.

In TDD, the developer can focus on the task at hand. He moves forward one small step at a time. TDD helps articulate next small step/problem. **You can focus on what you want before you know how to do it**. This kind of approach **helps to avoid 'analysis – paralysis'**.

In TDD you can implement only what you write test for and you should write tests only for functionalities that you need. This **helps avoid over–engineering**.

TDD can help to create more modularized, flexible, and extensible code. This comes from unit testing because you are thinking about the code in terms of small units that can be tested separately. This **leads to smaller, more focused classes, looser coupling, and cleaner interfaces**. BUT pushing this technique to extreme can lead to over–engineering, lost of cohesion and introduction of unnecessary layers of complexity [HANSSON, 2014]. It is very hard, however, to decide when is the point that you are over–engineering and it requires some experience. To tests a unit in a isolation also sometimes requires mocks and those are frowned upon by some people [OSHEROVE, 2008].

TDD can help drive the design of the software. That does not mean that TDD will design the software for you. Rather it can help you find design problems, for example, when you find some tests hard to write. [PAREKH, 2014]

Tests will give you feedback on your API while also documenting it. Moreover tests can be used to record what you are thinking during development. [BECK, 2014]

A 2013 study [SHAWETA & SANJEEV, 2013] found that **TDD improves quality of software** through unit testing and decreased defect rate. The **decrease in defect density** was also observed in [NAGAPPAN et al., 2008].

Several articles report **increased productivity** [ERDOGMUS, 2005; KAUFMANN & JANZEN, 2003; SHAWETA & SANJEEV, 2013; JANZEN & SAIEDIAN, 2006] According to [SHAWETA & SANJEEV, 2013] the relative productivity increases with time and at start, TDD can be less productive than traditional ways of developing.

Study from 2006 [MÜLLER, 2006] suggests that TDD produces **more testable software** than conventional projects.

Other study from 2007 suggest that **TDD approach is more efficient than conventional approach** because it requires less development effort [GUPTA & JALOTE, 2007].

TDD also tends to create **less complex modules, classes and methods** because they are usually smaller than when developed by conventional development processes [JANZEN & SAIEDIAN, 2008].

6.2 Shortcomings

The whole team has to practice TDD. With shared code–base if only some people practice TDD, is loses sense. Also, it is often hard to justify

TDD to the management. Consultants sometimes say that it is better not to explain TDD to management. Only tell them that you test the code thoroughly.

A high number of passing unit tests may bring a **false sense of security**, resulting in fewer additional software testing activities, such as integration testing and acceptance testing [WIKI, 2014b]. Your code can be valid from the tests' point of view but can be invalid from the users' point of view. The tests are written by the same person who writes the implementation. Therefore the tests and implementation can share the same blind spots.

TDD is not easy to adopt and master. People consider it to be a myth but it is true. Simple confirmation is the fact that so many developers do it wrong. You have to gain experience to be able to understand the design feedback from your tests. You are also more likely to be slowed down by TDD at first before you gain some experience. TDD also puts very high demands on code refactoring skills [BINSTOCK, 2011b]. Also, the impact of TDD on program design is not necessarily a positive one in the hands of less experienced developers [SINIAALTO & ABRAHAMSSON, 2007].

Another issue is that writing and maintaining an excessive number of **tests costs time** and this is magnified by the fact that **TDD developers tend to over–test their software**. This is however not problem only in TDD it is more general unit testing issue [COPLIEN, COPLIEN]. Moreover, overzealous testing can actually damage your design [HANSSON, 2014].

The tests are part of your code. If you design them poorly **tests maintenance cost could overweight their usefulness**. Developers often forget that the tests should be equally important as the code and they should be refactor as well as the code [BECK, 2002]. An example of a bad tests are fragile tests [MESZAROS, 2007]. Such a tests are tightly coupled to implementation details.

With TDD you don't have to arrive to an optimal solution. When trying to implement algorithm TDD can help you to do it but there is **no assurance of arriving to an optimal solution**.

6.3 Challenges

There are many challenges that are related to TDD. From industry point of view, where industry are people who really apply TDD in real big scale software development, from academic point of view, that concerns research, and also from the consultant point of view, where consultants are people who talk about TDD mainly from their experience.

There is need for some data from the industry on TDD but mainly for some large leading software project that was developed with TDD. I think people need to see that it actually can work in a large scale projects (if it can work).

Some studies use industry data about TDD but it is hard to make definite conclusion because the studies usually compare some metrics measured in the TDD team with metrics from not TDD team. Those teams, however, work on different projects. Also having two teams usually means having two sets of people. Different people equals different experience.

Prove or disprove some of the metrics like increased quality, decrease in complexity, higher maintainability might be close to impossible. I think that there are just too many uncontrollable variables to produce some significant results.

Another challenge for TDD is to drop some of the religiousness that is now connected with it. I think it is really hurting TDD not acknowledging that good code can be written even without TDD. Instead TDD should be viewed more as an optional path that you can choose. Even Kent Beck admits that he doesn't blindly always follow TDD.

Developers are reluctant to try TDD because they consider it difficult process to implement (and they are not wrong). The fact is that usually, programmers perceive TDD more positively after exposure to it, and particularly they are much more likely to adopt TDD after having tried it [JANZEN & SAIEDIAN, 2006].

There is also the fact that TDD is intertwined with other processes mainly from XP and it is sometimes hard to distinguish what is part of TDD and what is not. What is a problem of TDD and what is a problem of some of the techniques that TDD uses. There is lot of light that has to be shed on this. It is one of the things I'm trying to accomplish with this thesis.

Chapter 7

Is TDD dead?

On April 23, 2014 David Heinemeier Hansson, the creator of Ruby on Rails, wrote an article “TDD is dead. Long live testing.” [Hansson, 2014]. This article caused that TDD became very hot topic once again. It causing outburst of discussions and reactions both positive and negative. Supporters of TDD didn’t take nicely that Hansson’s starts his blog post with:

> *Test–first fundamentalism is like abstinence–only sex ed: An unrealistic, ineffective morality campaign for self–loathing and shaming.*

Even though he expresses his strong feeling against TDD, in the article Hansson acknowledges the contribution of TDD but also claims that it is not the right way and we should move on from TDD. He suggests that people should move from unit test oriented testing to system–wise testing.

One of the most interesting reactions that followed this article was a series of video conversations between Kent Beck, David Heinemeier Hansson and Martin Fowler [HANSSON et al., 2014]. In these conversations the participants discuss all aspects of TDD from very different points of view. The discussion consists of 5 sessions that altogether took 3 hours.

I believe it is not only very recent and therefore actual debate, but it also includes Kent Beck (the father of TDD), David Heinemeier Hansson (together with James Coplien and Rich Hickey one of the most respected critics of TDD), and Martin Fowler (the author of the most known book on refactoring).

In this chapter I will look in detail on the main points of the discussion. I will not go through the debate chronologically because some of the topics were scattered throughout the discussion. I will attempt to separate the discussion into topics and write all the important that was said about that specific topic.

7.1 Mocking

Hansson expresses that he has problem with mocks[1]. According to him, the mocks introduce extra layers of unnecessary complexity and that only for the reason to test units separately.

He ‘accuses’ TDD of nudging people in this direction because isolation is kind of a goal in unit testing because you want to test each unit separately. This can be wrong when overused. He acknowledges that TDD is good for cases where you have very clear input and very clear output but in real work that is not always the case. You can achieve it, however, by mocking but that is too big of a sacrifice for the sake of testing, thinks Hansson.

Fowler replies that it is not TDD neither unit testing that forces you to isolate your classes. You might end up with the same design even with no tests at all. Beck adds that it is a design decision the developer makes that specifies how isolated a class should be. There are many good reasons for making your classes isolated (decoupled) but applying some design ideas every time and to everything is obviously wrong. Beck then makes one strong point:

> *Design (and TDD) is a question of trade–offs. They should never be ignored. You should always be aware of the trade–offs you make. You have to be a good designer to make the right ones.*

Beck further says that if you have mocks returning mocks, returning mocks, you are coupled to exact implementation “three streets away”. Then, of course, if you change something you will break the tests. But this is just bad design and has nothing to do with TDD.

Hansson then expresses that it is the attempt to isolate from outer things like database that bothers him most. He talks about the examples where people explain that if you are isolated from your database (via interfaces and mocks) you can later switch the database for a file system. This is a dream, Hansson says. For most cases such a transition is never trivial because the

[1]Meaning all kinds of objects that should mimic behavior of a real object (mocks, fakes, stubs, etc.). http://en.wikipedia.org/wiki/Mock_object

subjects in question usually have completely different specifications. Moreover according to his experience, it rarely happens that you need to make such a transition.

Hansson also says that people always think about cohesion and coupling as two separated things. In some cases, however, they are opposite to each other. If you are having better (lower) coupling you have worse (lower) cohesion. He gives the isolation of classes as an example. To isolate, you separate the related code into more classes making it harder to understand for the sake of testing. All this he considers to be a test–induced damage because the desire to test drives you this way.

Fowler acknowledges that isolation can cause problems if it introduces extra layers that are unnecessary. It can be considered as over–engineering but Beck ads that decoupling is sometimes worth the hassle.

Another problem related to mocking is the requirement often associated with TDD and XP that your tests should run fast. If they do not run fast, you will be less likely to run them often. To achieve this, you have to mock everything that can slow down your tests e.g. database or file system. This annoys Hansson because creating additional interfaces and mocks for the sake of running your tests fast makes no sense to him.

Hansson also thinks that some of the effort to make the code testable is wasted because easily testable code doesn't make it better designed. It also usually means that to isolate your classes you have to write more but more code doing the same thing is bad code (not always, though), Hansson thinks. The unnecessary changes to design for the sake of testing he considers yet another example of test–induced damage.

People, according to Hansson, cannot comprehend the test–induced damage. They are saying: "How could the tests cause any damage? The higher the number of tests, the better. The faster the tests run, the better." Hansson disagrees with this opinion. He thinks that TDD, on some scale, encourages you to do things that don't have to be done and even things that shouldn't be done.

Beck understands the argument but doesn't consider it so much to be argument against TDD. It is not TDD but developer's design decision that can induce the damage into your software, he says.

7.2 Red, Green, Refactor

The red, green, refactor cycle is more like a mandate than an option and that is wrong, says Hansson. He explains that the cycle is not bad on its own and he used it several times and even now sometimes uses it when he feels it is helpful. He is, however, strongly against the "you have to use it all the time" point of view that is promoted by most of the TDD supporters.

Beck answers by a question, whether a programmer deserves to feel confident. He explains his own relation to doing red, green, refactor. He says that it makes him confident and happy to move forward by doing the small incremental steps with tests and therefore he applies this practice wherever he can. He adds however, that there are some parts of software he cannot write tests for first but it doesn't bother him. He says:

> *If I can play both classical music and jazz, I have a better chance to be a good musician.*

Hansson agrees with this point of view. That TDD is more like a tool that you should have in your toolbox and you can use it if you need it. His problem is more connected to the fact that TDD purists want you to do it all the time and he does not have the same feelings about it as Beck.

Seeing failure is not pleasurable for him. He feels like he is doing something wrong when he cannot think about how to write a test but has some kind of implementation in his head. He explains that he doesn't think through his code by proposing hypothesis and then filling it in. He likes to work his way to the test rather than from the test. He thinks that it should be all right as long as both ways arrive to the same destination.

Fowler response to it was, that the main goal should be self–testing code. System should be able to test itself. TDD is one way to approach this problem. He also acknowledges, however, that some context might be more suitable for TDD than other.

Another problem of TDD is that people tend to under–refactor their code, says Hansson. Fowler replies that the red, green refactor cycle is matter of skill. More skilled developers usually refactor more because they can find a better design. It is not easy. When you refactor it can take a long time but you are not making any 'real' progress. It should not be overlooked, he adds, that your tests make it easier for you to refactor, whether you are skilled programmer or beginner.

7.3 Tests

During the debate both Beck and Fowler bring up a feedback as an important topic. TDD gives you frequent feedback. How else will you get the feedback, asks Fowler and presents three types of feedback:

- Is the software doing what the user wants/needs?
- Have I broken anything?
- Is my code base healthy?

Hansson acknowledges usefulness of such a feedback but says that it comes with a cost and this cost is often not well understood. As said before, TDD comes with the trade–offs and sometimes it just not pays back to follow TDD. Whenever you try to achieve ideal in some statistics you encounter following issue, Hansson says.

He gives an example. If you achieve 95% uptime of your system but you want to achieve 99% or even 99.99%, the effort to move to the next step grows exponentially. Of course it can make sense for critical systems to go for the highest possible uptime but we don't build such systems most of the time.

From this point of view is the approach in TDD weird for Hansson because you are suppose to write tests before production code which should in theory mean near 100% coverage. Yet, the 100% coverage is much more costly than say 99% and drastically more costly for 95% coverage. TDD developers don't take this into account much, concludes Hansson. Beck agrees on the fact that it is trade–off how much are you willing to invest into fidelity of your tests.

Another problem with testing that Hansson sees is that they cause overconfidence. There are teams that discarded quality assurance because they don't need it. They have tests. They don't see value in exploratory tests or anything that quality assurance can offer. They are blinded by their 'green' tests but the fact is that the real world feedback loop is missing.

This is the complete opposite of what was happening before programmers started to write tests themselves. They just wrote the code and then threw it to the quality assurance team. Neither of these extremes is good. With this statement both Beck and Fowler agree. Beck adds:

> *If you think you are not making mistakes anymore, you are making mistake.*

Next issue Hansson brings to the fore is over–testing. He claims that TDD pushes you towards over–testing. He can see it when you have to change four lines of the test code because you need to change one line of the implementation code. He then asks Beck and Fowler when do they think the code suffers from over–testing or under–testing.

Beck gives as an example of well tested software developed with TDD his JUnit[2]. The code coverage of JUnit is 100% and he doesn't consider it over–tested. He considers the test–to–code ratio as an incorrect measurement. Some complex algorithm may require a lots of test, even though it takes up ten lines of code. On the other hand some class with no logic can have the exact opposite ratio.

A better metric is delta coverage, says Beck. Delta coverage reports how much of unique code (code that is not covered by other tests) does a test cover. 0% delta coverage can be indicator of unnecessary test. These tests can be deleted if they don't have any documentary purpose.

Fowler says that he sometimes experiments with the code to see if a test will fail. For example, he will comment out a line of code. You don't have enough tests or good enough tests, if you cannot confidently change the code. You have too many tests if you have feeling you spend much more time changing tests over code. You can find the right amount from experience, says Fowler.

As the last thing that bothers him about tests, Hansson mentions that test should be the documentation and the top authority. Some TDD advocates even puts tests above the code. This he thinks is also a reason why some developers neglect refactoring. It is more important to have the next test running so the design becomes less important and code becomes under–refactored.

Beck replies to it with statement that if you throw away the implementation and you have a good test suite you can rebuild the implementation with confidence. On the other hand, if you throw away the tests you have much less confidence that you will recreate all the tests you had before.

For Fowler the tests are equally valuable as implementation because they are pair used for double–checking.

7.4 Effects of TDD

During the debate Hansson acknowledges one thing. It is very hard to find out whether a software development technique is really working. You cannot repeat experiments because there are too many incontrollable variables. You have to combine experience and arrive to some conclusion.

For Hansson, the most interesting things on TDD come from the regression suite not the designing driven part. As said before, he just considers it as a tool. Fowler replies, that this actually is how people should be thinking

[2] http://en.wikipedia.org/wiki/JUnit

about it but Beck adds that TDD also solves some problems for him [BECK, 2014].

One effect of TDD is that people get sucked into it, thinks Hansson. It is addictive and if people see it work they try to forcefully apply it everywhere. Positive thing about TDD, says Hansson, is that it is 'gateway drug' for self–testing regression code, so people will see the usefulness of regression testing.

In the discussion, Fowler answers question what effect has TDD on inexperienced developers in comparison with experienced developers. Fowler answers that the question shouldn't be aimed at inexperienced vs. experienced developers but rather on (in)experienced developers that use TDD and not use TDD.

He assumes that for inexperienced developer TDD helps by having very small increments. You can think about the pieces separately. That does not guarantee good software design. It takes some experience to make good design, regardless of TDD. Experienced developers will most likely arrive at good design, regardless of TDD.

Hansson's another issue is the atmosphere around TDD. People are way too moralistic about TDD, he says. There is almost religious approach to TDD by some of the promoters and they tie it to professionalism [He probably has Robert C. Martin in mind [MARTIN, 2007, 2014]].

Hansson says that one of the reason why he wrote the article TDD is dead was that he has seen people who reported that TDD doesn't work for them but they were just told that they are not doing it right and they hadn't learn it right. They didn't feel confident with TDD but they were still pressured to develop that way because "that's how we do it". Because "TDD is the right way how to do it", these people were usually 'scared' of saying out loud that TDD doesn't work for them. If doing TDD hurts when it shouldn't don't do it. Don't feel bad about yourself, concludes Hansson.

Fowler agrees that the dogmatism is hurting TDD and compares TDD to Agile development. Agile has won the 'label'. Everyone claims being Agile but people rarely practice true Agile development. That is similar to what is happening to TDD but TDD has smaller scale. Successful ideas are very likely to be misused, concludes Fowler.

Hansson adds that people cannot leave good ideas alone. They always want to build on top of it at any cost. That's why TDD's current version accumulated some bad stuff on it.

Beck replies that he never thought about it that way because he reinvents the TDD every time he programs. He still thinks, though, that portion of the problems connected with TDD are related more to poor design decisions.

He concludes:

> *TDD is not dead for me but maybe, like phoenix, it has to raise from the ashes. So thanks for setting fire to it, David [Hansson].*

7.5 Conclusion on the debate

My description of the debate might seem as it was a heated debate with a lot of arguments but it was not. I think it gave all the participants and viewers great insight into different mindsets. You could see some TDD myths to be mentioned, explained and also torn apart.

Problem of TDD is that there is more interpretations of it. Even though it is wrong to think that design will magically appear from the tests, there are many people who practice TDD this way. They understand it this way and maybe it even works for them in their projects. Hansson himself talked about TDD as it was suppose to be responsible for the design. It is important to understand what Beck said several times during the debate: "TDD does not make the design decisions. Developer does."

However, from my point of view Hansson's main frustration came from the people who were abusing TDD. Using TDD as a panacea and talking about it as it is the only way to write a code. That is obviously wrong. The notion, that TDD is a tool that does not have to be used for all tasks, should be put into awareness.

Hansson's argument that not everyone thought process is the same and some people might have problems with red, green, refactor, seems to be valid arguments as well. People have different mindsets so some can have troubles when thinking first about test that should fail.

Overall, I think the debate significantly increased my understanding of TDD, its benefits and pitfalls and whoever is interested in TDD should watch it.

Chapter 8

Conclusion

So 15 years has passed but the heart of TDD didn't change much. We still need to write the tests first, we still should follow the red, green, refactor cycle and we still should let the tests to drive our design. What changed about TDD is the number of people that apply it.

With increased number of people that use TDD, ambiguity has spread between developers about what TDD is really about. Many myths has polluted the area of TDD. Critics start to denounce TDD based on false assumptions. For example, TDD was incorrectly accused of ignoring design while it was actually the developers who ignored the design but still presented their practice as TDD.

The problem of TDD is that it is intertwined with other process and it is sometimes hard to detach it from them. Nevertheless, from my point of view, Kent Beck managed to defend TDD from 'being dead'. The debate between him David Hansson and Martin Fowler showed, though, that it might be necessary to spread the word how people should really practice TDD.

The research question I asked was: **Which criticism of TDD is justified and is there a way TDD can eliminate it?** Now I will try to answer it based on my findings.

A thing that harmed TDD quite significantly was dogmatism of some of its proponents. Programming is highly intellectual discipline. You cannot expect everyone to be able to work certain way. Moreover, you cannot force people into something they are not comfortable with. From my research, the thing that had possibly the strongest impact on me was the thought that TDD is more like a tool that you can use when you need it.

In my mind, I imagined it as a hammer. Hammer is very useful tool but there are jobs that you cannot do with hammer properly. Maybe you need, say, screwdriver. It is wrong to use hammer when you should use screwdriver. Some people use it because they don't have a screwdriver in their toolbox. That can be understandable. Some use it, however, because they've been told that if they cannot do it with hammer, it's because they are not banging with it hard enough. That is definitely wrong approach.

I would propose that this is an area to which researchers could pay some attention. What types of software tasks are suited for TDD? I think the results of such study could be actually really useful for better understanding of the strengths and weaknesses of TDD.

Another finding of my thesis was that it is extremely hard to put a number (metric) on TDD (or for that matter any software development technique) and try to evaluate it that way. The results will be almost always questionable because they are just too many incontrollable variables. Taking this into account, the critics of the strong claims that TDD produces better code had a point. Nevertheless, I still see value in this kinds of research, though, I just think that it has to be very carefully and thoroughly thought through.

TDD is also party responsible for giving false sense of security. When developers see their tests pass they forget that the tests are not bulletproof. Some quality assurance is still needed because sometimes it takes a person from outside of developer team to break your code. You never know what the user will do with your software. TDD should try to raise awareness about it.

The thesis aimed to examine Test-Driven Development. I hope the readers will find it useful and it will help them in understanding TDD. I also hope that it can be used as a reference to the right way TDD should be practiced, for instance, for future research studies.

Bibliography

BACHMANN, F., BASS, L., & NORD, R. (2007). Modifiability tactics. *Software Engineering Institute*.

BAVOTA, G., CARLUCCIO, B. D., LUCIA, A. D., PENTA, M. D., OLIVETO, R., & STROLLO, O. (2012). When does a refactoring induce bugs? an empirical study. *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 104–113.

BECK, K. (1999). *Extreme Programming Explained: Embrace Change* (1st ed.). Addison-Wesley Professional.

BECK, K. (2002). *Test-Driven Development By Example*. Addison-Wesley.

BECK, K. (2004). *Extreme Programming Explained: Embrace Change* (2st ed.). Addison-Wesley Professional.

BECK, K. (2012). Test-driven development: Why does kent beck refer to the 'rediscovery' of test-driven development? http://www.quora.com/Test-Driven-Development/Why-does-Kent-Beck-refer-to-the-rediscovery-of-test-driven-development.

BECK, K. (2014). Rip tdd. https://www.facebook.com/notes/kent-beck/rip-tdd/750840194948847.

BECK, K. & GAMMA, E. (2000). Test-infected: programmers love writing tests.

BINSTOCK, A. (2011a). Chasing code-coverage baubles. http://www.drdobbs.com/architecture-and-design/chasing-code-coverage-baubles/231601779.

BINSTOCK, A. (2011b). Tdd is about design, not testing. http://www.drdobbs.com/tdd-is-about-design-not-testing/229218691.

BOEHM, B. & BASILI, V. Software defect reduction top 10 list. *Computer*, *Vol. 34:1*.

BOEHM, B. & PAPACCIO, P. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, *Vol. 14:10*, 1462–1477.

BOLTON, M. (2009). The tyranny of always. http://www.developsense.com/blog/2009/08/tyranny-of-always/.

COPLIEN, J. O. Why most unit testing is waste. *RBCS-US*. http://www.rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf.

ERDOGMUS, H. (2005). On the effectiveness of test-first approach to programming. *IEEE Transactions on Software Engineering*, *Vol. 31:3*, 226–237.

FEATHERS, M. & FREEMAN, S. (2009). Test driven development: Ten years later. http://www.infoq.com/presentations/tdd-ten-years-later.

FOWLER, M., BECK, K., JohnBRANT, OPDYKE, W., & ROBERTS, D. (1999). *Refactoring: Improving the Design of Existing Code* (1st ed.). Addison-Wesley Professional.

GLASS, R. L. (2001). Frequently forgotten fundamental facts about software engineering. *IEEE Software*, *Vol. 18:3*.

GUPTA, A. & JALOTE, P. (2007). An experimental evaluation of the effectiveness and efficiency of the test driven development. *First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007*, 285–294.

HALL, A. (2010). Anti-test-driven development arguments and myths. http://compositecode.com/2010/06/18/anti-test-driven-development-arguments-and-myths/.

Hansson, D. H. (2014). Tdd is dead. long live testing. http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html.

HANSSON, D. H. (2014). Test-induced design damage. http://david.heinemeierhansson.com/2014/test-induced-design-damage.html.

HANSSON, D. H., BECK, K., & FOWLER, M. (2014). Is tdd dead? http://martinfowler.com/articles/is-tdd-dead/.

HUNT, A. & THOMAS, D. (2003). *Pragmatic Unit Testing in Java with JUnit* (1st ed.). The Pragmatic Programmers.

IEEE Software (2007). Ieee software. *Vol. 24: 3*.

JANZEN, D. S. & SAIEDIAN, H. (2006). On the influence of test-driven development on software design. *Proceedings of 19th Conference on Software Engineering Education and Training, 2006*, 141–148.

JANZEN, D. S. & SAIEDIAN, H. (2008). Does test-driven development really improve software design quality? *IEEE SOFTWARE*, *Vol. 25:2*, 77–84.

JEFFRIES, R., ANDERSON, A., & HENDRICKSON, C. (2000). *Extreme Programming Installed* (1st ed.). Addison-Wesley Professional.

JEFFRIES, R. & MELNIK, G. (2007). Guest editors' introduction: Tdd–the art of fearless programming. *IEEE Software*, *Vol. 24:3*, 24–30.

JONES, C. (2012). A short history of the cost per defect metric. version 4.0. *David Consulting Group*. http://www.davidconsultinggroup.com/insights/publications/a-short-history-of-the-cost-per-defect-metric/.

KAPELONIS, K. (2013). Don't test blindly: The right methods for unit testing your java apps. http://zeroturnaround.com/rebellabs/dont-test-blindly-the-right-methods-for-unit-testing-your-java-apps/.

KAUFMANN, R. & JANZEN, D. (2003). Implications of test-driven development: a pilot study. *Proceeding OOPSLA '03 Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 298–299.

KERIEVSKY, J. (2004). *Refactoring to Patterns* (1st ed.). Addison-Wesley Professional.

KIM, M., ZIMMERMANN, T., & NAGAPPAN, N. (2014). An empirical study of refactoring challenges and benefits at microsoft. *IEEE transactions on software engineering*, *Vol. 40:7*. http://www.computer.org/csdl/trans/ts/2014/07/06802406-abs.html.

Let's Code (2014). How does tdd change qa? http://www.letscodejavascript.com/v3/blog/2014/02/how_does_tdd_change_qa.

MACKINNON, T., FREEMAN, S., & CRAIG, P. (2000). Endo-testing: Unit testing with mock objects. *XP Two Thousand Conference*. http://www.ccs.neu.edu/research/demeter/related-work/extreme-programming/MockObjectsFinal.PDF.

MARTIN, R. C. (2007). Professionalism and test-driven development. *IEEE Software*, *Vol. 24:3*, 32–36.

MARTIN, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship* (1st ed.). Prentice Hall.

MARTIN, R. C. (2014). Professionalism and tdd (reprise). http://blog.8thlight.com/uncle-bob/2014/05/02/ProfessionalismAndTDD.html.

MESZAROS, G. (2007). *xUnit Test Patterns: Refactoring Test Code* (1st ed.). Addison-Wesley.

MÜLLER, M. M. (2006). The effect of test-driven development on program code. *Extreme Programming and Agile Processes in Software Engineering, XP 2006*, 94–103.

NAGAPPAN, N., MAXIMILIEN, M. E., BHAT, T., & WILLIAMS, L. (2008). Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, *Vol. 13:3*, 289–302.

OSHEROVE, R. (2008). Over-testable systems, and mocks as bad test smells. http://osherove.com/blog/2008/7/3/over-testable-systems-and-mocks-as-bad-test-smells.html.

PAMIO, D. (2012). Is test driven development a replacement for qa? http://experiencesonsoftwaretesting.blogspot.dk/2012/02/is-test-driven-development-replacement.html.

PAREKH, S. (2014). Using tdd to influence design. http://www.thoughtworks.com/insights/blog/using-tdd-influence-design.

Pathfinder Solutions (2012). Effective tdd for complex embedded systems. http://www.pathfindersolns.com/wp-content/uploads/2012/05/Effective-TDD-Executive-Summary.pdf.

PerOlof BENGTSSON, Nico LASSING, J. B. H. v. V. (2000). Analyzing software architectures for modifiability.

POGH, K. (2011). *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration*. Addison-Wesley Professional.

SHAWETA, K. & SANJEEV, B. (2013). Comparative study of test driven development with traditional techniques. *International Journal of Soft Computing and Engineering*, *Vol. 3:1*, 352–360.

SINIAALTO, M. & ABRAHAMSSON, P. (2007). A comparative case study on the impact of test-driven development on program design and test coverage. *First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007*, 275–284.

SINIAALTO, M. & ABRAHAMSSON, P. (2008). Does test-driven development improve the program code? alarming results from a comparative case study. *Balancing Agility and Formalism in Software Engineering*, 143–156.

WIKI (2014a). Kiss principle. http://en.wikipedia.org/w/index.php?title=KISS_principle&oldid=620990465.

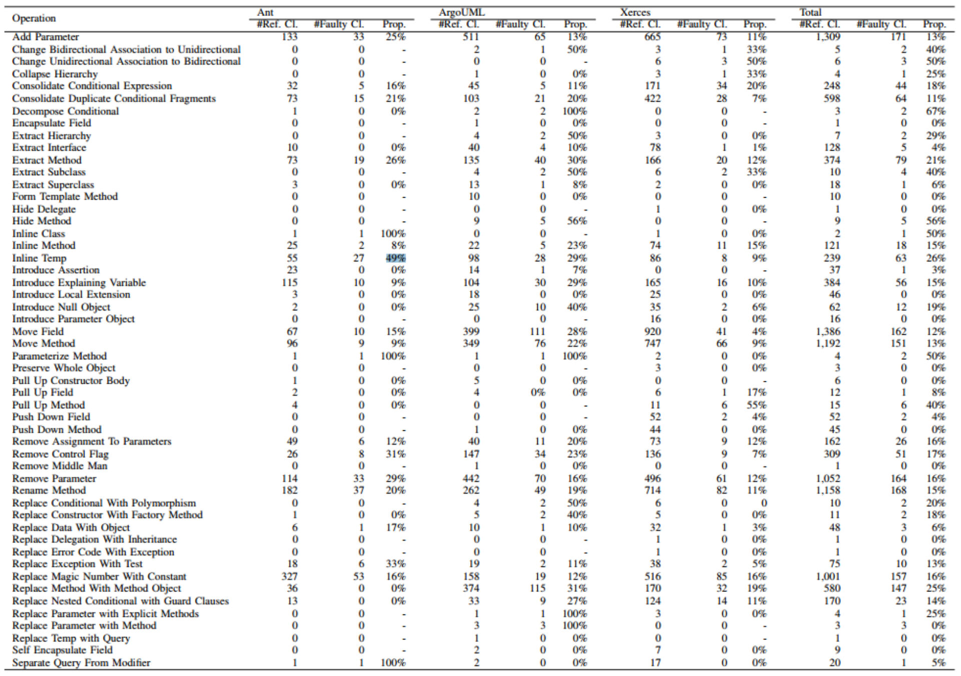
WIKI (2014b). Test-driven development. http://en.wikipedia.org/w/index.php?title=Test-driven_development&oldid=623040492.

WRIGHT, H. K., KIM, M., & PERRY, D. E. (2010). Validity concerns in software engineering research. *Proceedings of the FSE/SDP workshop on Future of software engineering research. FoSER '10*, 411–414.

Appendices

Appendix A

Refactored classes and fault–prone classes (among those subject to refactoring)



Operation	Ant			ArgoUML			Xerces			Total		
	#Ref. Cl.	#Faulty Cl.	Prop.	#Ref. Cl.	#Faulty Cl.	Prop.	#Ref. Cl.	#Faulty Cl.	Prop.	#Ref. Cl.	#Faulty Cl.	Prop.
Add Parameter	133	33	25%	511	65	13%	665	73	11%	1,309	171	13%
Change Bidirectional Association to Unidirectional	0	0	-	2	1	50%	3	1	33%	5	2	40%
Change Unidirectional Association to Bidirectional	0	0	-	0	0	-	6	3	50%	6	3	50%
Collapse Hierarchy	0	0	-	1	0	0%	3	1	33%	4	1	25%
Consolidate Conditional Expression	32	5	16%	45	5	11%	171	34	20%	248	44	18%
Consolidate Duplicate Conditional Fragments	73	15	21%	103	21	20%	422	28	7%	598	64	11%
Decompose Conditional	1	0	0%	2	2	100%	0	0	-	3	2	67%
Encapsulate Field	0	0	-	1	0	0%	0	0	-	1	0	0%
Extract Hierarchy	0	0	-	4	2	50%	3	0	0%	7	2	29%
Extract Interface	10	0	0%	40	4	10%	78	1	1%	128	5	4%
Extract Method	73	19	26%	135	40	30%	166	20	12%	374	79	21%
Extract Subclass	0	0	-	4	2	50%	6	2	33%	10	4	40%
Extract Superclass	3	0	0%	13	1	8%	2	0	0%	18	1	6%
Form Template Method	0	0	-	10	0	0%	0	0	-	10	0	0%
Hide Delegate	0	0	-	0	0	-	1	0	0%	1	0	0%
Hide Method	0	0	-	9	5	56%	0	0	-	9	5	56%
Inline Class	1	1	100%	0	0	-	1	0	0%	2	1	50%
Inline Method	25	2	8%	22	5	23%	74	11	15%	121	18	15%
Inline Temp	55	27	49%	98	28	29%	86	8	9%	239	63	26%
Introduce Assertion	23	0	0%	14	1	7%	0	0	-	37	1	3%
Introduce Explaining Variable	115	10	9%	104	30	29%	165	16	10%	384	56	15%
Introduce Local Extension	3	0	0%	18	0	0%	25	0	0%	46	0	0%
Introduce Null Object	2	0	0%	25	10	40%	35	2	6%	62	12	19%
Introduce Parameter Object	0	0	-	0	0	-	16	0	0%	16	0	0%
Move Field	67	10	15%	399	111	28%	920	41	4%	1,386	162	12%
Move Method	96	9	9%	349	76	22%	747	66	9%	1,192	151	13%
Parameterize Method	1	1	100%	1	1	100%	2	0	0%	4	2	50%
Preserve Whole Object	0	0	-	0	0	-	3	0	0%	3	0	0%
Pull Up Constructor Body	1	0	0%	5	0	0%	0	0	-	6	0	0%
Pull Up Field	2	0	0%	4	0%	0%	6	1	17%	12	1	8%
Pull Up Method	4	0	0%	0	0	-	11	6	55%	15	6	40%
Push Down Field	0	0	-	0	0	-	52	2	4%	52	2	4%
Push Down Method	0	0	-	1	0	0%	44	0	0%	45	0	0%
Remove Assignment To Parameters	49	6	12%	40	11	20%	73	9	12%	162	26	16%
Remove Control Flag	26	8	31%	147	34	23%	136	9	7%	309	51	17%
Remove Middle Man	0	0	-	1	0	0%	0	0	-	1	0	0%
Remove Parameter	114	33	29%	442	70	16%	496	61	12%	1,052	164	16%
Rename Method	182	37	20%	262	49	19%	714	82	11%	1,158	168	15%
Replace Conditional With Polymorphism	0	0	-	4	2	50%	6	0	0	10	2	20%
Replace Constructor With Factory Method	1	0	0%	5	2	40%	5	0	0%	11	2	18%
Replace Data With Object	6	1	17%	10	1	10%	32	1	3%	48	3	6%
Replace Delegation With Inheritance	0	0	-	0	0	-	1	0	0%	1	0	0%
Replace Error Code With Exception	0	0	-	0	0	-	1	0	0%	1	0	0%
Replace Exception With Test	18	6	33%	19	2	11%	38	2	5%	75	10	13%
Replace Magic Number With Constant	327	53	16%	158	19	12%	516	85	16%	1,001	157	16%
Replace Method With Method Object	36	0	0%	374	115	31%	170	32	19%	580	147	25%
Replace Nested Conditional with Guard Clauses	13	0	0%	33	9	27%	124	14	11%	170	23	14%
Replace Parameter with Explicit Methods	0	0	-	1	1	100%	3	0	0%	4	1	25%
Replace Parameter with Method	0	0	-	3	3	100%	0	0	-	3	3	0%
Replace Temp with Query	0	0	-	1	0	0%	0	0	-	1	0	0%
Self Encapsulate Field	0	0	-	2	0	0%	7	0	0%	9	0	0%
Separate Query From Modifier	1	1	100%	2	0	0%	17	0	0%	20	1	5%