Chapter 1

Test Driven Development

Stephan Wels

This paper discusses the test driven development (TDD) approach as an emerging alternative to traditional software development processes.

1.1 Introduction

Development processes play a central role in software engineering. Traditional process models such as the waterfall model have revealed major draw backs. Since the waterfall model prescribes the entire analysis must almost be done before implementation, customers can barely take part in the development of their application. The first time they see results might be at a point where the project already consumed lots of money. If the customer notices at that point, that the specification of the product doesn't meet the requirements, the specifications have to be changed. At that point, changes can easily get very expensive and even lead the project to fail, especially if those changes affect the architecture of the software.
Not only adjustment of requirements costs much, but also errors in implementation. These errors tend to be revealed in the late testing phase. Correcting these errors is expen-

sive, since the whole system may have to be refactored. Integration tests may reveal that even the architecture of the system must be refactored, which jeopardizes the whole project, since there might not be enough budget left for that at this point.
These drawbacks of traditional programming models are tried to be solved by using agile programming models instead. Agile programming models use an iterative approach, where in each iteration only few features are introduced, implemented and tested. Especially the high costs on repairing errors in old and large code fragments lead to a focus on tests. This paper introduces a development process where tests are not only used to reveal bugs, but in the first place utilized to design, document and specify features.
The rest of the paper is structured as follows: After an introduction into software tests in section 1.2, the test driven development approach is defined in section 1.3. Section 1.4 gives a few hints on how TDD should actually be applied in practice and section 1.5 shortly introduces some of the available software tools for practicing TDD. Section 1.6 deals with the question how development processes can be compared and section 1.7 explores large parts of the empirical evidence available in the literature. Section 1.8 concludes this paper.

1.2 Software tests

Software tests are mainly used to show that a system runs without errors. An error is a behavior of the system that is different from what is specified. This already implies that a system can only be tested meaningfully, if it's behavior has been specified before, because without a specification of what is right, it cannot be identified what is wrong.
Software tests are written on different levels of granularity. This paper differentiates between three levels:

1. unit test
2. integration test

3. application test.

The *unit test* is the finest grained test. Subject to this test is the correctness of a single unit. In object-oriented languages, each class is seen as a single unit. For the sake of separation of concerns, a unit test should always test only the functions of this single class.
Dependencies to other classes are solved by inserting *mock objects*. A mock object is an object that mimes the behavior of a real instance of a certain class. It implements all interfaces of the mimed class and can therefor be used instead of a *real* object during testing. Using mock objects instead of real instances has some advantages. Mock objects can be modeled before the mocked class has been implemented. This makes development more flexible, since the implementation of a certain module A might depend on the presence of another module B, which is not completely implemented yet. With mock objects module B can easily be mimed and the implementation of module A can start immediately.
Mock objects also enforce separation of concerns. Since the behavior of mocked objects doesn't depend on the implementation of the respective class, the unit test will not fail due to an erroneous implementation of the mocked class. Errors that raise from the mocked class are subject to this class' unit tests.
However, mocking also introduces some overhead. Every time the specification for a mocked class changes, the mock objects have to be adjusted. In a test suite, where many tests depend on mocked objects, this overhead may be significant.
Integration tests focus on the interaction between units. These tests ensure that interfaces have been designed and implemented such that all units which need to work together can communicate properly.
The *application test* investigates the whole system and is therefore the coarsest grained kind of test considered here. This test ensures that the whole system meets the requirements of the customer. When the product is handed off to the customer, acceptance tests will be performed at the customer's site. This acceptance test is also an application tests, since they are not really interested in how single units work, but only in whether the application works the way

the planned it or not. In order to increase the likelihood that the customer is satisfied by the software - and in the end this is all that matters for a software house - application tests are also run at the developers' site beforehand.

For each kind of test, a set of test cases has to be selected. This paper differentiates between three different approaches for generating test cases, namely *white box tests*, *black box tests* and *gray box tests*.
A white box test uses additional knowledge about the internal structure of a method to find suitable test cases. With this additional knowledge, the set of test cases can be chosen such that each line of code in the method is executed at least once (*line coverage*). Another approach is to select the test cases such that the execution reaches each path in the control flow at least once (*path coverage*). These coverage measures can be collected automatically using software tools that are introduced in chapter 1.5.
Opposed to white box tests, the black box tests are not selected by investigating the program itself, but only by the specification of the program.
Intuitively spoken, white box testing focuses on doing things right and black box testing focuses on doing the right things.
A third approach to test case selection is called gray box testing. This approach rises from test driven development and is a middle way between white and black box testing. It is similar to white box testing, as the tests are written by the developer of the module. On the other hand, the test is written before the actual implementation of the module, such that the test cannot yet depend completely on the code, but rather depends on the expected outer behavior.

In a test driven development process, the set of tests grows continuously. This test set can be used for *regression testing*. The idea of regression testing is to run all available test (or at least a larger relevant subset) each time after changing or adding code to verify correctness of changes. Changing code in one place, frequently causes erroneous behavior somewhere else. This experience causes stress for developers when they change a small part in a larger system, because this change might have broken something else in the system accidentally. This uncertainty about whether the change harmed the system or not is tried to be cleared

out by regression testing. The developer can get immediate feedback by starting the regression test at any point in time. If the outcome of the regression test reveals errors somewhere else in the system, the developer can identify the point of failure exactly by looking at the tests that failed. If the regression test doesn't reveal any errors, then the developer can expect the code change to be correct and can continue coding without the burden of potentially having harmed the system. This works of course only if the test case collection is reliable, otherwise the developer would probably not trust the regression test.
The reliability of the test case collection depends on the line and path coverage of the whole project. Achieving high test coverage is only possible if the whole development team keeps writing tests throughout the process. This is of course additional overhead, but it can pay off, since it also results in higher motivation of the developers, due to the reduced stress level, and on the other hand, the test case collection provides evidence, that the whole system is in a sane state. By putting tests into the focus of the development process, a reliable test collection is generated almost automatically and the team can profit from meaningful regression testing.

1.3 Test driven development

This section gives a definition for test driven development. The definition is taken from [1] and [9]. Test driven development (TDD) reverses the order of implementation and testing compared to a traditional test last approach. Where in a test last approach the feature is first implemented and then tested, the test first approach makes the programmer first write tests for the feature and then implement as much code, as is needed to satisfy these tests.
A more detailed definition can be taken from Beck [1]. The development process is described as a chain of tasks. There are three different types of a task, namely the *test task*, the *application task* and the *refactor task*, which are in an ideal TDD process executed in exactly this order.
Let the set of tests be denoted by C. The system is said to pass C if all tests in C run without failure. It is said to fail C if at least one test in C fails.

During a test task, a test case c_{new} is added to C where c_{new} must fail, therefore the system fails also $C_{new} = C \cup c_{new}$. A test task is only allowed to be executed, if the system passes C. Thus it always holds, that a test task starts with a C, that the system passes and ends with a C_{new} which the system fails.
During the application task, as much code is implemented, as is needed to make the system pass C_{new}. It always holds, that an application task starts with a system that fails C_{new} and ends with a system that passes C_{new}.
The refactoring task is responsible for cleaning up dirty code (e.g. code duplicates) that has been written during the preceding application task. A refactoring task always starts and ends with a system that passes C_{new}. The whole test driven development process is illustrated in Figure 1.1.

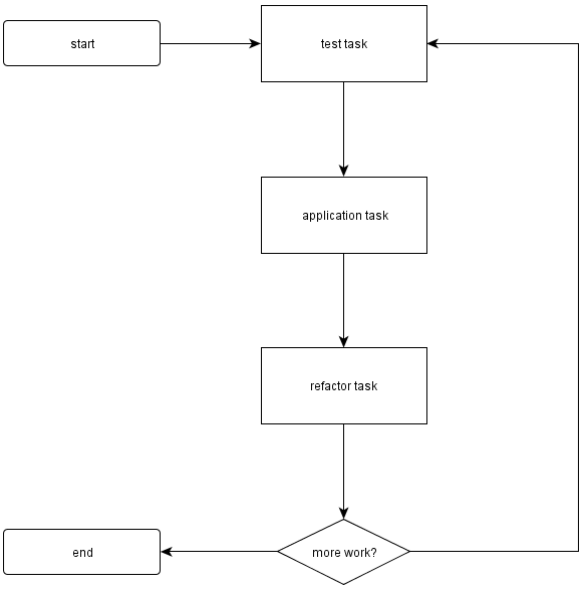

Figure 1.1: Illustration of the TDD process.

Testing in a traditional test last approach has only a single purpose, namely showing that a system works without errors. In TDD, tests play also a central role in design. Before starting the test-code-refactor cycle, a high level design is done upfront. The low level design is to be done during the test-code-refactor cycle. This is different from traditional development processes, where the complete design is done before implementation. In TDD, implementation can start

with the high level design done.
There are different interpretations of the abbreviation ”TDD” than ”test driven development”, such as ”test driven design” or ”test first design”, which point out the design goal of TDD more clearly. However, the term test driven development has become the common choice of interpreting TDD.

1.4 TDD - How To

The previous sections introduced the theoretical background and definition of TDD. Applying TDD is very practical though. This section provides a view on TDD from the practical perspective and points out common practices and pit falls during the TDD process. The ideas in this section are drawn from Kent Beck's book 'Test Driven Development By Example' [1].

1.4.1 When/How are tests written?

Test first
Consequent testing before implementing code is the basic rule in TDD. When writing a test for a new feature, the programmer gets to design this feature. Designing it by writing a test is much easier than designing it during implementation, since the programmer doesn't have to care about how the functionality is realized in detail, but only about what is the correct behavior and how is the feature's outer appearance supposed to look like. By reducing the number of tasks the programmer has to solve simultaneously, the stress level decreases and less errors are made.
Disregarding the test first rule, will lead to features that have not been designed thoughtfully yet and thus reduce software quality.

Test list
When a new feature is going to be tested, there are often many cases that have to be taken into account. Before starting with writing any test code, a test list of all cases that

the programmer can think of at that point is written down. Otherwise he would have to keep all these tests in mind while writing the first test and eventually forget at least one of the test cases. Since the TDD process builds upon a reliable test case collection, the programmer should never forget about a test case. On the other hand, TDD focuses on reducing the number of problems that have to be solved simultaneously and thereby reducing the stress level and increasing the quality of written code. Both goals can be achieved by simply managing such a test list.
When writing a test, there will often be additional test cases that come to the programmer's mind related to the test which is supposed to be implemented now. Of course, these additional test cases will be useful or even necessary, but they need to be suspended, until the programmer has finished his current test task. To achieve that, the test cases are simply added to the test list and can be covered by one of the next iterations.
When an item on the list has been finished, it is stroked out and the programmer picks a new item, where it is very clear, what the test case is good for and how it should be tested. When the behavior of a feature is totally clear, it is easy to write tests for it. When only things are picked that are easy to realize, the development process can be sped up. When an item on the list is too large or not clarified yet, this item should be divided in several easier subtasks.
Another psychological aspect of such a test list is the motivation that is gained from completing items on the list.

Assert first
Although the strict separation of testing and implementing code reduces the number of tasks to be done at once, writing a single test still means to solve several problems at the same time, e.g. whether the functionality is really new, or just an extension of an existing feature; how the new functions should be called; what the correct behavior is and how it is going to be checked whether the test succeeded. The last two points can be separated from all other issues, by simply writing asserts first. Writing the assert statement only required to know what should be achieved by the test. Which functions have to be called exactly and what these functions may be called is irrelevant at that point.

Mock Objects

Sometimes testing a simple unit involves other complex structures. A good example for such a complex structure is a database. There are probably many functions that rely on a working database connection, but creating a real working database connection may require significant effort. This additional effort for defining the database would have to be done in each test case that requires such a database connection, which is obviously not intended to be done by any test. The test should only focus on the unit that is being tested and just assume, that all other potentially complex external structures are available.
The solution to this problem is to define a mock object once that mimes the behavior of the respective complex structure and use this mock object in all cases where the respective structure is needed. Using mock objects instead of real objects separates unit tests more exactly. Assume that there is a unit that covers the database connection and several other units that do something completely different, but need a database connection. If now someone refactors the database unit and breaks part of its functionality, tests of the database will fail and additionally all other tests that build upon a working database connection, if they were using real objects. This makes locating the error more complicated than necessary. Additionally, mock objects are easier to read and understand than realistic objects.

1.4.2 What does a good test look like?

In general, tests should represent the most beautiful form of the new feature one can think of, i.e. easy and nice interfaces. The test is supposed to fail at first (since it is a new feature) - it may not even compile, since the interface for such a feature doesn't exist yet.
Tests are supposed to be small, to keep the test-application-refactor cycle (cf. 1.1) short and agile. Whether a test is good or not, doesn't depend only on it's length. Kent Beck proposes some other criteria in [1] that should be met by a good test, such as the following.

Isolated tests
Tests are not supposed to affect each other. An example for tests that depend on each other is mentioned above, where

breaking the database causes errors in completely different units, when a real database is used instead of a mock database. The main reason for separating tests is to ease up locating errors. If there is only one error in the system, it would be the easiest, if also only one test fails, which points exactly at where the error occurs. If a single error causes tens of tests to fail, it is harder to locate.
A test should never depend on another test to be run previously. At some point, the test case collection might be too large, to run all of them after each little change. This means that a subset must be selected. In order to simplify the selection of such a subset, it is important to know, that there is no test, that will fail just because another test has been removed from the test set.

Test data
Some functionalities need input arguments, this input is referred to as the test data. In most cases, not all possible argument combinations can be tested. The selection of a meaningful subset of input data isn't a trivial problem and should fulfill the following.
Since the test is supposed to be some kind of *complete*, all conceptual differing argument combinations should be tested. To improve the documentary character of a test, the data should be readable and thus as simple as possible. If the two inputs "231" and "1" are in the same equivalence class, then "1" is a better test data for the sake of readability.
Inputs can be divided into equivalence classes according to the control flow, they cause. For a method, which contains an if/else branch, two inputs should be tested where each branch should be tested once - these two inputs are then in two different equivalence classes since the control flow they cause differs. An optimal test collection would cover all these equivalence classes with one argument combination, which is easy to read. Additionally, there are often certain corner cases for methods e.g. NULL arguments. These cases should also be covered.

1.4.3 How are tests that fail treated?

When the regression tests are run and at least one test fails, the developer has to fix it. Knowing that there is something

that doesn't work is stressful, to keep the motivation high and stress low, the issue that caused the regression tests to fail, is meant to be fixed as quickly as possible. The failing test can often be turned into a succeeding test without having to implement the full functionality as the following patterns suggest.

Fake it('Til you make it)
The "Fake it" pattern helps fixing the failing test immediately. Instead of calculating the expected result and returning it, the expected result is just returned as a constant. Of course, this code will have to be replaced by something that actually does the calculations of the expected output, but there is an important advantage to fake the result first. The developer will be more calmed, if he knows that the regression tests run successfully. This gives him more time to think about a clean solution, instead of being pushed by the failing tests to a quick and dirty solution.
This pattern is meant for larger problems, where the implementation is not obvious yet. For some tests, there is an obvious implementation as described in the following.

Obvious implementation
Dividing large problems into several smaller problems and having to concentrate on as few problems simultaneously as possible is important in TDD, but of course, in some cases it is simply not necessary. If a test requires a simple functionality to be implemented and the developer is sure about how to implement it, then there is no need for faking the result first or dividing it into further subproblems. In this case, the obvious implementation should just be done directly.

1.5 Software Tools

Applying TDD in real projects requires software tools to reduce the overhead of regression testing as far as possible. As TDD has drawn a lot of attention during the past decade, the community grew strong enough and came up with several tools that ease continuous testing. The tools mentioned in this chapter concentrate on the Java program-

ming language.
First of all, there is the xUnit project, which is a testing framework for several programming language. The corresponding testing framework for Java is called jUnit. It can be integrated easily into the standard IDE Eclipse. jUnit provides an easy environment for writing, executing and evaluating test suites. It is realized by a package of classes. A TestRunner class runs tests and test suites and reports results in a TestResult object. Each test case is encapsulated in a TestCase class. Several test cases can be summarized in a single TestSuite object. Assert statements are utilized to decide whether a test is successful or not. Running a test suite is done through the RCP view jUnit offers. It also visualizes the current state in form of a green or red bar, where a green bar represents a successful test run and a red bar indicates that there are still errors. The errors are listed including a the error trace, which makes evaluating the test results and debugging easy.
Second, a project named Cobertura is a tool that enables developers to check test coverage of the current test suite.
To reduce the refactoring effort of tests that make use of mock objects, the Java library Easy Mock is available. The mock objects defined through Easy Mock can adept automatically to most interface refactorings.
A last useful tool should automate the build and deploy process. Tools that can be used for this purpose are e.g. Ant and Maven.

1.6 Comparing software development approaches

With the TDD process defined, it is to be investigated whether the TDD approach is superior to traditional approaches or not. In order to tell whether the TDD approach is *better* than traditional approaches, it has to be defined first how the quality of a development approach is supposed to be measured. This paper restricts the quality to the two aspects *quality of product* and *price of product*. The quality of a development process is high, if the quality of the resulting product is high and the price is low.

quality of product
In case of software engineering, we can measure a product's quality by properties such as reliability, maintainability and reusability of components and of course satisfaction of the customer. Some of these quality properties are hard to measure, such as the customers satisfaction, however, properties like reliability can be measured easily when there is a proper test basis.
With these quality measurements in mind, it gets more obvious why a development approach would put tests into focus. When tests are written continuously throughout the development process, a steady feedback can be provided about the state of the product at any time in the process. Putting the tests even before implementation is assumed to increase the test coverage, as only features are implemented, for which tests exist. On the other hand, if tests are written afterwards, it is likely that some feature aspects are not covered.
A steady feedback on the current state of the product increases the satisfaction of the customer, because it provides realistic evidence on the progress of product development and whether the budget is going to be sufficient.
In section 1.7 empirical results are discussed, which assume that applying TDD has also a positive effect on the reusability of components.

price of product
The price for a piece of software is mostly determined by the hours developers have to work on it. Especially in software development it is important that work is done either correctly, or not at all. A wrong design decision may cause that all work that is based on this decision must be redone. It is assumed that a motivated developer is more focused and thus more efficient than one that is frustrated. To keep developers motivated, the psychological effects which occur during coding have to be addressed.
One source of frustration can be tasks that are too large. Every time a task is completed, the developer gets a feeling of success and can drop this task from his mind. Solving a large problem has two drawbacks. First of all, it will take more time until the task is done and thus the feeling of success, that motivates developers to move on, will take some time to appear. Second, until the task is solved, the developer is stressed more than someone who is solving a

smaller task, since he has to keep more things in mind, in order to solve a bigger task correctly.
The level of stress a developer feels may also have a bad influence on his results. If a developer feels the stress level rises, he will try to develop faster, the faster he develops, the more errors he will make, the more errors he makes, the higher the stress level will rise. This feedback loop has to be interrupted somewhere to keep the developer focused. What keeps pushing the stress level in the first place is the amount of errors a developer produces. Introducing tests will cause two things to happen. First, writing a small test and implementing as much as is needed until this test runs, is a quite small task. As mentioned above, completing such a small task raises the developer's motivation. Second, the number of errors will decrease, which gives the developer some confidence that things he implemented work as intended. The confidence a developer experiences, when running a passing test suite must be founded on a very good test suite. If the system has still errors and the test suite passes, because of low coverage, the developer is provided so-called *false confidence*. False confidence lets the developer leave the system in an erroneous state and ultimately either frustrate the developer, because errors will occur in a unit that has been successfully tested, or the error will not be discovered during the development process by chance and an erroneous product is delivered to the customer. If the developer notices errors in a successfully tested unit several times, the positive confidence effect of TDD vanishes. The false confidence effect is critical in TDD and is meant to be avoided by writing enough tests.

1.7 Emperical evidence on the influence of TDD

There is only few reliable empirical evidence on whether TDD works better or worse than other approaches. In order to get statistically significant statements on this topic, expensive experiments would have to be made.
In an optimal setting two test groups are needed which program the exact same product. The programmers of the two groups should have very similar programming skills and

experience. The group that uses TDD must be used to the approach and be able to apply it correctly. As it is mentioned in previous chapters, TDD may have a negative effect on the product if it is not applied correctly. The negative effects mostly occur due to false confidence.
The comparison group needs to master the approach they are using as well. Despite test first or test last the two groups should use similar development models, i.e. both should utilize agile development models. The development task and subject groups should also be of considerable size, such that it mirrors a real world application.
Of course, there is no company, that wants to pay a product twice, so there will be no such optimal experiment as described above. The results of the experiments mentioned in the following should be noticed with that background. Most of them just provide gentle hints whether TDD works better or worse than traditional approaches.

Experiments are done to statistically proof certain hypotheses - e.g. that TDD works better than a more traditional approach. When analyzing the results of such an experiment, certain properties of the experiment must be taken into account.
First of all, there is the *power* of the experiment. The power of an experiment measures the probability that a true hypothesis will be confirmed by the results of the experiment. The likelihood for rejecting a true hypothesis by mistake is then given by $1-power$. This is also called the *type II* error.
Second, the probability for accepting the hypothesis by mistake is given by the so-called *significance level* α. This is called the *type I* error.
Obviously, the significance level and the power of an experiment depend on each other. If the probability for a type II error is meant to be decreased, it is of course less likely that the hypothesis is accepted.
Jacob Cohen states in [2], that an experiment should have at least a power of 80%, in order to have a real chance of proving the hypotheses. A common choice for the level of significance seems to be between 5% and 10% in the evaluation of experiments focusing on the effects of TDD.
Another quality measure for an experiment is the *external validity*. The external validity is the ability to transfer the result from the designed experiment to the real world. In many of the experiments, only students were taken as sub-

ject to reduce the costs of the experiment. This is a so called *thread* to external validity, since in the real world, the TDD is supposed to be applied by professionals, which might perform differently if they were given the same task in another experiment.

Matthias Müller et al. performed an experiment on test first programming in 2001 [8]. 19 students took part in the experiment, where ten of them utilized a test first approach and the nine other subjects used traditional test last programming. All of them had attended a one-semester graduate lab course, which introduced test first programming along with other XP techniques.
Both groups were given a main class of a graph library, where the code was deleted from all method bodies. The task was to re-implement the bodies of the methods in the main class. The median of the overall working time was about 10 hours.
The study could not find anything statistically significant, however, they noticed a slight increase in reliability of the programs. This means that the programs of the test first group passed slightly more black box tests than the programs of the test last group.
Beyond that, they noticed that the test first programmers got a deeper understanding of the existing methods faster than the test last group. This has been measured by the correct reuse of existing methods. The authors assume that this effect results from the consequent testing and rechecking failed tests. By checking the failed tests, they take another look at the methods they might have been used in a wrong way and thereby learn faster how to use them correctly.

Boby George et al. investigated in [5] the TDD approach in an industrial context. 24 professional developers from three different companies participated in the experiment. The subjects were randomly divided into two equally large groups, where one group used the TDD process and the other was asked to use the traditional water-fall approach (design-develop-test-debug). Both groups used pair-programming throughout the experiment. Pair-programming is a XP-technique, where two programmers develop one task on the same computer. The experiment

was performed at each company individually, such that always two pairs programmed the task. The task was to develop a small bowling game with about 200 lines of code. Additionally to evaluating the code quality and development time, all participants answered a survey, to give additional evidence on how the developers judge the different approaches.
They concentrated on two hypotheses. The first hypothesis H_q was that the TDD group produces code with higher external quality, i.e. the code will pass more black box tests. The second hypothesis H_p was that the TDD group has a higher productivity, which means that the group needs less time to solution.
By evaluating the programs of the two groups, they found that the external quality of the TDD programs was about 18% higher compared to the control group. On the other hand, the TDD group needed about 16% more time to get to the result.
An interesting finding concerns the developers' subjective impressions of the current state of the project. The groups were always asked to deliver their products, when they considered it done. When taking also the higher external quality and the greater developing time of the TDD groups into account, this indicates that TDD developers are not really more productive, i.e. they don't come quicker to the solution, but they seem to have a better awareness of the product's quality, as they released it with significantly less errors than the control groups..
The surveys reveal some additional insights. 87.5% of the developers thought that TDD helps understanding the requirements better and even 95.8% think that the TDD approach reduces the time spend on debugging. Roughly half of the developers considered learning the TDD process difficult.

Erdogmus et al. conducted an experiment on the test first approach and evaluated it in [4]. Their subjects were a group of 24 undergraduate computer science students, where 11 of them utilized the test first approach and the other 13 a test last approach. The subjects were asked to solve the same bowling game task as in the previous mentioned experiment by Boby George [5].
The hypotheses in this experiment were that the test first group writes more tests (H_T), produces programs with

higher external quality (H_Q) and are more productive than the test last group (H_P). Hypothesis H_T appeared to be valid at a significance level of 10%. For the other two hypothesis H_Q and H_T, there are no statistically significant results available.
Besides the statistical significant results, the authors believe that the test first subjects were slightly more productive and that this effect was due to the following. The authors assume that by testing first, a better task understanding can be achieved, because the developer himself has to express the requirements explicitly in the tests. Also a better task focus is assumed due to the reduced number of tasks that have to be solved simultaneously when working with the test first approach. Third, the authors mention a lower rework effort. This has to do with the limited scope of a test and the thereby easily detected cause of error.

Janzen et al. summarize several experiments on the effects of the test first approach in [6]. Their test subjects range from undergraduate students over graduated students up to professional developers with at least six years of professional experience. The professionals didn't develop the same task in different groups, but at least similar tasks, so that the company didn't have to pay twice for the same product. Overall six student groups of three students each and five professionals took part in the experiment. The student developers solved the same project in different groups. Additional to the projects performed during the experiment, they conducted a case study containing 10 further projects, which were all developed in a test last manner.
The study performed in [6] concentrates on the internal quality of the product. The internal quality of a program is considered higher, when the code is easier to understand and single modules of the program are reusable, i.e. the modules are only loosely coupled.
The evaluation of the case study revealed at the 5% significance level, that programs developed in a test first manner have fewer lines of coder per module.
By evaluating several complexity measures, they revealed a statistically significant difference between the complexity in terms of the number of branches per method and the number of methods per class between the test first and test last appraoch. The authors state that test first programmers

develop less complex programs.
In order to make the code easier testable, the test first developers made higher use of interfaces and abstract classes. According to the authors, the increased abstraction also improves reusability of the code.

Lisa Crispin shares her experiences with TDD in [3]. The conclusions drawn in this contribution don't result from formal experiments, but rather from personal experience as a tester in industrial development teams. The author is convinced that TDD increases the external quality of the code and refers to studies, such as [5] for statistical evidence. From her personal experience she gained from three different development teams that utilized TDD and several others that didn't, she concludes that code produced by the TDD teams have higher external quality and meet the customers requirements more often than the code produced by other teams.
Besides this positive effect of TDD, she discusses the challenging learning process of TDD. The author assumes that learning how to apply TDD takes a several months, or even years. After passing the learning phase, all programmers she worked with, sticked to the TDD approach.
Her experiences on internal code quality agree with Janzen's results in [6], where the internal code quality of TDD projects is considered superior to the non-TDD projects.

The contribution of Lisa Crispin in [3] raises another issue concerning the external validity of many experiments on TDD mentioned above. If TDD takes years to adopt correctly and only pays off when applied correctly, then the experimental results that used novices as subjects are probably not valid in the sense that they can easily be transferred to the real world.
Müller et al. conduct an experiment on TDD in [9], where they focus on the effect of experience on the results of TDD projects. To investigate the effect, they compare the code produced by a novice group of 11 computer science students, whose only experience on TDD was based on a one-semester XP lab course to the code produced by seven professional TDD programmers which had on average over three years industrial experience with TDD. All subjects solved the same programming task on their own, namely

a control system for an elevator. The task was assumed to be solvable in four to five hours in a single programming session.
By logging all their programming activities very carefully, their analysis of the experiments outcome led to the following conclusions which are statistically significant at the 5% level. The expert team achieved a higher conformance to the TDD process as defined in section 1.3. The experts worked on average in shorter test-code-refactor cycles and also had less varying cycle durations. The tests written by the expert team were of higher quality in terms of both, statement and block coverage.
In conclusion, the results of experiments done with subjects that are quite new to TDD, are not easily transferable to the professional application of TDD.

Opposed to most studies in the literature where TDD is concerned at least as good as traditional development processes, or even better, Madeyski contributes in [7] contradicting results. The author conducted an experiment to compare two central XP methodologies, namely pair programming and TDD, to the traditional process. He found statistical significant evidence at the 5% significance level that TDD has a negative effect on the external code quality compared to traditional test last approaches.
The subjects of the study were 188 computer science students. Most of them were in the second or third year. The author of [7] considers the thread to external validity caused by the kind of subjects small. According to Lisa Crispin [3] and Müller [9], this thread is not small but significant and thus, the external validity of the results in [7] are doubtable. Nevertheless, the study is worth mentioning to show, that there are not only studies that favor the TDD approach.

Since there are statistical significant results for TDD being both worse and better than the traditional approach it is clear, that evaluating TDD is very hard and that there is no reliable evidence yet.

1.8 Conclusion

TDD is a central component of the XP methodology which has gained a lot of attention during the last decade. It is not only a testing technique, but rather a design approach and has thus not only influence on the external quality of a product, but also on the internal code quality. Although there exists a clear definition of how TDD works, it seems to be very difficult to apply it. Learning how to apply TDD correctly may take several months and until then, the erroneous application of TDD may even cause decrease in product quality due to false confidence.
Some common techniques and tools have been introduced to ease the introduction into TDD practice.
The literature provides contradictory results on whether TDD is superior to traditional approaches regarding the external quality of the product or not. The statistical results regarding the internal code quality point towards a positive effect of TDD.

Bibliography

[1] K. Beck. *Test-Driven Development By Example*. Addison-Wesley Longman, 2008.

[2] J. Cohen. *Statistical power analysis for the behavioral sciences : Jacob Cohen.* Lawrence Erlbaum, 2 edition, Jan. 1988.

[3] L. Crispin. Driving software quality: How test-driven development impacts software quality. *IEEE Software*, 23:70–71, 2006.

[4] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31:226–237, 2005.

[5] B. George. An initial investigation of test driven development in industry. In *ACM Sympoium on Applied Computing*, pages 1135–1139. ACM Press, 2003.

[6] D. Janzen and H. Saiedian. Does test-driven development really improve software design quality? *IEEE Software*, March 2008.

[7] L. Madeyski. Preliminary analysis of the effects of pair programming and test-driven development on the external code quality. In *Proceedings of the 2005 conference on Software Engineering: Evolution and Emerging Technologies*, 2005.

[8] M. Müller and O. Hagner. Experiment about test-first programming, 2001.

[9] M. Müller and A. Höfer. The effect of experience on the test-driven development process. *Empirical Software Engineering*, 12(6):593–615, Dec. 2007.

Typeset August 3, 2012