ARCADA

Test-driven development

Weikko Aejmelaeus

Arcada University of Applied Sciences
Information technology

Helsinki 2009

Utlåtande (Skrivs av handledaren och granskaren)

DEGREE THESIS	
Arcada	
Degree Programme:	Information technology
Identification number:	2513
Author:	Weikko Aejmelaeus
Title:	Test-driven development
Supervisor:	M.Sc. Magnus Westerlund
Commissioned by:	Suunto Oy

Abstract:

In this thesis, a literature survey of TDD, and an evaluation of two different unit test frameworks for standard C++ was done.

The theoretical part examined test-driven development (TDD) from a developer's point of view with the objective to learn how the method works in practice and what difficulties there are to adopt it. The approach is to write the test before the code to be tested. The programmer writes the test and is using a unit test framework. It appeared that tests are written and executed frequently. Adaptation of the method proved to be difficult, since it is radically different from the traditional methods, where tests are written after the implementation.

In the practical part of the thesis two unit test frameworks for standard C++ was evaluated, CppUnit and Boost.Test. The learning from the theoretical part formulated the evaluation criteria to contain minimal amount of steps to add new tests and easy test execution. xUnit is an architecture description of unit test frameworks, which is implemented in most modern programming languages. In the evaluation it appeared that the evaluated framework that implements the xUnit architecture is not optimal for C++.

Keywords:	Test-driven development (TDD), Test design, C++ unit test framework, TDD adoption, CppUnit, Boost.Test
Number of pages:	47
Language:	English
Date of acceptance:	15.5.2009

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informationsteknik
Identifikationsnummer:	2513
Författare:	Weikko Aejmelaeus
Arbetets namn:	Testdriven utveckling
Handledare:	M.Sc. Magnus Westerlund
Uppdragsgivare:	Suunto Oy

Sammandrag:

I detta slutarbete gjordes en litteraturstudie av testdriven utveckling (TDU) och en utvärdering av två olika testramverk för standard C++.

Den teoretiska delen undersökte testdriven utveckling (TDU) från en programmerares synvinkel med målet att lära sig hur metoden fungerar i praktiken och vilka svårigheter det finns att anamma den. Metoden går ut på att testen skrivs innan koden som ska testas. Programmeraren skriver testerna och har till sin hjälp ett testramverk. Det framkom att man frekvent skriver nya tester och att de körs ofta. Att anamma metoden är svårt eftersom den skiljer sig radikalt från de traditionella metoderna, där testerna skrivs efter implementationen.

I den praktiska delen av slutarbetet utvärderades två testramverk för standard C++, CppUnit och Boost.Test. Kunskaperna från den teoretiska delen användes till att skapa utvärderingskriterierna. De fokuserar på att det skall vara så enkelt som möjligt att lägga till nya test samt att de skall vara enkla att köra. xUnit är en arkitekturbeskrivning för testramverk som implementerats till de flesta moderna programmeringsspråken. Evalueringen visade att testramverket som implementerade xUnit arkitekturen inte är optimalt för C++.

Nyckelord:	Test-driven development (TDD), Test design, C++ unit test framework, TDD adoption, CppUnit, Boost.Test
Sidantal:	47
Språk:	Engelska
Datum för godkännande:	15.5.2009

TABLE OF CONTENTS

ABBREVATIONS

TDD	Test-driven development
GUI	Graphical user interface
IDE	Integrated development environment
C++	A general purpose programming language
STL	Standard Template Library
XP	Extreme Programming

ACKNOWLEDGEMENTS

I would like to thank my teachers at Arcada, thesis supervisor M.Sc. Magnus Westerlund for help and guidance in the writing process and M.Sc. Hanne Karlsson for final polishing and valuable feedback. I would also like to thank my colleagues at Suunto for help and reviewing, especially M.Sc. Jyrki Uusitalo for making it possible to write about this topic. Last but not least I would like to thank my fiancé for her support.

Weikko Aejmelaeus

Vantaa 15.5.2009

1. INTRODUCTION

Test-driven development (TDD) or test-first programming is a software development method that is often used in agile software development processes. The idea is that the tests are written before the code that is tested. The developer works in small steps, using a unit test framework that provides rapid response to small changes (Beck, 2003:x).

A unit test framework provides mechanisms for running tests. Thus the test writer can concentrate on writing the test specific logic (Meszaros, 2007:298). The xUnit architecture is a unit test framework architecture that is ported to most current programming languages (Hamill, 2005:18).

Scrum both is an iterative and incremental development process that includes a skeleton which defines roles and routines within a project (ScrumAlliance, 2009). When it is applied to software development, agile methods are used. In scrum the customer becomes a part of the development team (Wikipedia, 2009c).

1.1. History

Agile methods started to evolve in the mid-1990s. Before that software was developed with focus on careful project planning and extensive design and documentation. In small and medium sized projects this approach leads to a large overhead and bureaucracy. In agile development methods two to four week iterations are used. The goal for each iteration is to deliver working software. Each iteration is carefully planned, including analysis, design, coding and testing. (Sommerville, 2004:396).

Specifying inputs and outputs before programming the specific implementation is not new in software development. TDD is based on this idea. TDD has its origin in Extreme Programming (XP) that is an agile software development process created by Kent Beck in year 2000 (Sommerville, 2004:398).

1.2. Background

This thesis is written for Suunto Oy. Suunto is a leading Finnish manufacturer of sports instruments for a variety of training, diving and outdoor sports. Suunto's team for PC - and Web software has introduced Scrum as their process for software development. TDD is one method that is planned to be used in the software development process.

1.3. Objectives

This thesis is divided into two main parts, one theoretical and one practical. The theoretical part is a literature survey of the topic where literature and scientific articles are used. The practical part is to evaluate unit test frameworks for the C++ programming language. The gained knowledge from the theoretical part will be used as a base for creating the evaluation criteria.

1.3.1. Theoretical

One objective is to learn how TDD is done in practice, with focus on how the method can increase the quality of written code. This includes understanding of how test are written, how unit test frameworks are functioning and what patterns can be used when practicing TDD.

The TDD method is radically different from the traditional way to create software; therefore yet another objective is to illustrate with the literature survey what is required of the team and the individual developer to practice TDD. What are the most common pitfalls? How can these be avoided?

1.3.2. Practical

There are many different unit test frameworks for standard C++. But none of them is a de facto standard, Java has JUnit or C# .NET has NUnit. The xUnit architecture is built on an object oriented programming paradigm, but C++ is a multi-paradigm language that allows usage of free functions that are not members of a class. Is the xUnit architecture optimal for an automated C++ unit test framework, or is a solution using C-style free functions a better alternative? Two different unit test frameworks will be selected for evaluation, one xUnit oriented and one that utilize free functions. The evaluation result will be a recommendation for a standard C++ unit test framework selection. The gained knowledge from the thesis theoretical part will be used to create evaluation criteria for the comparison.

1.4. Limitations

This thesis will not describe agile processes in general; it focuses on TDD and its methods. The tests are described from a unit test perspective which verifies that all different elements of the system work as the developer intended.

2. TEST-DRIVEN DEVELOPMENT

In an overview of the literature in the field of agile testing, the terms TDD and test-first programming stands out. TDD used to be called test-first programming as an opposite of traditional development were tests usually were written after the code it is testing (Beck, 2003:203).

In traditional software development models the tests are written after the code is implemented, in other words test-last. This does not drive the design of the code to be testable. Even if the code is designed with testability in mind, the chance is low that it will be, without modifying the production code (Meszaros, 2007:32). When tests are written before the code it tests the development is driven by the software requirements and specifications. If test are written after the implementation there is a risk that tests are written to satisfy the implementation, not the requirements (Elssamadisy, 2008:176).

When practicing TDD the tests are written by the developer. The tests are so called white, or glass box tests, which mean that the developer knows the internal logic of the tested component (Wohlin, 2005:159).

The goal with TDD is to get clean code that works and to get a predictable development process. When the TDD method is used new code can only be written if it has an automated test that has failed. Duplication must be eliminated. This gives a cyclic process that Beck describes as the TDD mantra – red/green/refactor. (Beck, 2003:ix-x).

- Red – Write a little test that doesn't work, and perhaps doesn't even compile at first.
- Green – Make the test work quickly, committing whatever sins necessary in the process.
- Refactor – Eliminate all of the duplication created in merely getting the test to work.

The first step is to write a new test and see it fail. The failure is important, since in that way it can be verified that the test actually works when the correct implementation is done. In step two when getting the test to pass the test itself and the implementation is verified. In the third step the new code is refactored, duplication and other possibly bad solutions are removed.

Refactoring is defined as "behavior-preserving transformation" which means that the code is restructured without changing its external functionality. This can be done with the test as support and verification that no functionality has changed during the refactoring. (Astels, 2003).

2.1. The TDD Cycle

This part will focus on explaining and describing the TDD cycle and working rhythm. It will be illustrated with a simple calculator application that can perform a multiplication operation with integers. TDD enables taking small steps when needed. In this simple example where the implementation is obvious the steps appear unnecessary small, but when complex functionality is developed, being able to take small steps can be essential (Beck, 2003:9).

The example will illustrate a failing test, a fake implementation and a gradually developed implementation. It will be programmed using C++ and the Boost.Test unit test framework, which is presented more in detail in chapter 3.

One of TDD's basic rules is that no code is allowed to be written before there is a failed automated unit test to test the functionality. The first step is to create a test for a calculator that supports multiplication (see example 2.1).

Example 2.1. Test of multiplication

CalculatorTest.cpp

```
#define BOOST_TEST_MODULE calculator test
#include <boost/test/unit_test.hpp>

#include "Calculator.h"

BOOST_AUTO_TEST_CASE(testMultiplication)
{
    Calculator c;
    BOOST_CHECK_EQUAL(c.multiply(3, 5), 15);
}
```

Since the calculator class is not yet implemented the test will not compile. The compiler informs that the class `Calculator` cannot be found and that the function `multiply` is undefined. The next step is to get the test to compile in the simplest way possible. To do this the `Calculator` class is added and the `multiply` function is implemented. The simplest implementation in this case is to return 0 (see example 2.2). With this implementation the test compiles but fails, which is a wished result since it gives a concrete way to measure failure and success (Beck, 2003:5).

Example 2.2. The calculator class implemented in the simplest way possible.

Calculator.h

```
class Calculator
{
public:
    int multiply(int term_lhs, int term_rhs)
    {
        return 0;
    }
};
```

The next step is to get to the green state, i.e. to get the test to pass. This should be done in the easiest way possible. In this case it would be to fake the result and return the hardcoded value 15 (see example 2.3). It is obvious that this is not the correct implementation, but in this way it is verified that the test works.

Example 2.3. The fake implementation for the calculator test

Calculator.h

```
...
    int multiply(int term_lhs, int term_rhs)
    {
        return 15;
    }
...
```

In the previous step the green state in the TDD cycle was reached. The next step is to refactor the code to get the correct implementation. This is done by gradually introducing variables to do the multiplication. In example 2.4 one of the terms are included in the calculation.

Example 2.4. Refactored implementation for the calculator test

Calculator.h

```
...
    int multiply(int term_lhs, int term_rhs)
    {
        return term_lhs * 5;
    }
...
```

In the previous example one hardcoded value was changed to a variable. When hardcoded values are driven out of the implementation it can be done by adding additional tests. This technique is called triangulation, see example 2.5. When the new test is added the test fails and the correct implementation has to be programmed.

Example 2.5. Additional calculator test

CalculatorTest.cpp

```
...
BOOST_AUTO_TEST_CASE(testMultiplication2)
{
    Calculator c;
    BOOST_CHECK_EQUAL(c.multiply(7, 7), 49);
}
...
```

This part described the TDD working cycle and rhythm. It was shown how the test can be written before the actual implementation. The example was very simple and the small steps would not be necessary for such a trivial problem. When there is a obvious implementation it can be implemented directly. Tests were frequently added and executed during the development in the TDD cycle. In the example several TDD patterns such as Fake It and Triangulation was used. TDD patterns are described in chapter 2.6.

The tests that the TDD process produces is a low-level design documentation of the system. It is written in a language that programmers understand. When tests are executed regulary it is assured that the documentation stays up-to-date (Martin, 2007:32).

2.2. Writing tests

By writing the tests first, the requirements are supported. The developer must analyze the requirements before any code can be written. When applying this practice the design is driven by the requirements and no unnecessary code is written (Elssamadisy, 2008:179).

Simple things such as plain set and get functions do not require explicit unit tests, they are considered robust enough. But if they contain any built in logic that for example needs initialization; a test is in its place (Hamill, 2005).

An automated unit test is supposed to be run in isolation, that means that the tests should not be dependent on other tests or the order the tests are executed. Tests should not require access to external components such as database communication, network communication or a computers file system. The tests should not require any manual configuration before they are executed, for example editing a configuration file (Feathers, 2007:3). Tests that are created with the isolation principle in mind tend to be more loosely coupled and cohesive (Beck, 2003:125).

Test should be written to verify one piece of functionality. These tests are called single-condition tests. When tests are written in this way defects are easier to pinpoint in the source code. Unit test frameworks aborts a test function if an assertion fails. If a test verifies several pieces of functionality the test will not be executed completely if a test fails in the beginning (Meszaros, 2007:353-357).

The tests should execute fast, otherwise they will not be run often enough. If the whole suite of tests takes longer than 10 minutes to execute, there is a risk that developers won't run all tests before checking in modifications (Beck, 2003:194; Elssamadisy, 2008:170).

Written tests should be easy to read and understand. The intent of the test should be as clear as possible. Therefore the tests should contain evident data. The expected and actual result should be included in the test case if possible and their relationship should be as apparent as possible (Beck, 2003:130). In example 2.6 two similar tests assertions are presented. In the later the expected result is made evident and the relationship between the actual and expected result is apparent.

Example 2.6. Evident data in test cases

CalculatorTest.cpp

```
...
BOOST_AUTO_TEST_CASE(testMultiplication)
{
    Calculator c;
    BOOST_CHECK_EQUAL(c.multiply(39, 17), 663);
    // Expected data made evident
    BOOST_CHECK_EQUAL(c.multiply(39, 17), (39 * 17));
}
...
```

Test code should be treated as production code. The same rules of good design apply to both test code and to production code. Test code should be refactored when needed and duplication should be eliminated, just as production code (Elssamadisy, 2008:170).

2.3. Refactoring

Refactoring is defined as "behavior-preserving transformation". This process contains elimination of duplication, simplification of complex logic and clarification of unclear code. This covers everything from changing variable names to divide or unify a large hierarchy. Refactoring is done in small steps (Kerievsky, 2005:9).

Code that has symptoms of low quality or bad design is referred to as code that smells. Examples of this can be duplication, a class or function that is to large or a class that depends on the implementation details of another class (Wikipedia, 2009 b).

Refactoring is an important part of TDD. When a test is written and the green stage is achieved in the TDD cycle the next action is refactoring. To get to the green stage the simplest possible solution was used, this may contain duplication or other smells in the code. In the refactoring stage of the TDD cycle, duplication is the first thing to be eliminated. Duplication means that the same thing is expressed in two or more places in the code, and if one of them is changed, all others must change too (Hunt, Thomas, 2000:27).

The working process in TDD can be divided into two modes, coding and refactoring. When coding is practiced new functionality is added to the system. When refactoring is done, no new functionality is added. To get to the green stage in the TDD cycle the coding mode is active. After that it switches to refactoring (Astels, 2003:17).

Regression in a software perspective means that a working computer program is set to a non working state. Regression happens when changes are made that introduces defects in the functionality (Koskela, 2008:28). Refactoring can trigger regression. The test harness that is created when developing with TDD can be used to perform regression testing.

2.4. Design

With a waterfall software development approach each activity is a separate process. One activity is finished and then the development goes to the next stage (Sommerville, 2004:9). For a traditional software development method these activities usually are design, implementation and unit testing.

Upfront design means that complexity is built into the design to accommodate changes in the requirements. The agile community believes that the gained benefits are outweighed by the cost. The increased complexity in the design is harder to learn and understand. It is also more difficult to exercise maintenance on complex code. In addition to this it is hard to foresee what, and if any, changes on the requirement that is going to appear (Elssamadisy, 2008:199).

TDD is not a testing technique; it is a way of designing and developing software (Janzen, Saiedian, 2008:77). The process creates an exhaustive test harness that is supporting software design as an ongoing activity. When requirements change, the design is adopted to fit the new specifications and solve the current problem. One condition to have a changing design is to refactor the existing code, which means that the code is modified to fit the current requirements. The refactoring is enabled by having automated unit tests (Elssamadisy, 2008:272).

When a test is written before the implementation the developer is forced to think in a perspective of usability and testability of the program. By writing the tests first a program is designed to be easier to use and testable (Martin, Martin, 2006:32).

There are several factors that can be used to measure software design; these are code size, complexity, coupling and cohesion.

- Code size, measured as lines of code per module.
- Complexity, cyclomatic or conditional complexity of code is measured by the number of linearly independent paths through a program (Wikipedia, 2009d).
- Coupling, measured by the degree of which modules in a program is dependent on each other (Wikipedia, 2009 e).
- Cohesion, when software is designed with modules that have one single responsibility (Miles, Pilone, 2007:161).

A study (Janzen, Saiedan, 2008:77-84) suggests that TDD can improve the quality of software design. The code size is smaller when practicing TDD, therefore the code is easier to understand. Complexity is reduced and classes and methods are less complex when TDD is used, compared to a test-last development method. The study could not show how the impact of coupling is affected when using TDD, the conclusion was that this issue needs more studies. TDD produces smaller, but more coupled classes, though the coupling seems to be of a good kind with a high degree of abstraction and flexibility. Cohesion is hard to measure; the study could not show that TDD produces code with a higher degree of cohesion. It is worth to point out that the study is a quasi-experiment, which means that the development teams where not randomly assembled.

2.4.1. Test doubles

When a class is not dependent on other classes testing is generally a straight forward task. But when the tested class is dependent of other classes the options are to test the dependent classes together, or to isolate the test class with a so called test double (Elssamadisy, 2008:125).

Different types of test doubles are called stubs, fakes and mocks. A stub implementation is the simplest possible implementation of an interface. The methods return hardcoded values. A fake implementation is one step more advanced than a stub; the methods can have some logic built in. A fake can be seen as an alternative implementation of an interface. A mock object can be configured for interaction with other objects or to return certain values. Mock objects can be generated with a framework or manually (Koskela, 2008:144).

2.5. xUnit architecture

In this sub-chapter the xUnit architecture is described. The xUnit is a unit test framework architecture that supports automated tests. The framework architecture is a de facto standard for unit test tools and is ported to several programming languages. Different implementations share the same basic framework architecture.

The primary way to work with the framework is to create a test class that is derived from the framework’s `TestCase` class. The most important concepts of the architecture are assertion, test case, test fixture and test suite. An assertion is a true-false statement that is used in the actual test. The test case is a class that can contain one or more test methods.

A test fixture is used to handle initialization and cleanup of the test environment. The test suite is used to collect several test cases for execution.

2.5.1. Assertions

One goal with automated unit testing is that the tests are self checking, that means that no manual intervention is needed to interpret the test results. This is achieved with assertions (Meszaros, 2007:362). Assertions return success or failure, thus the result can be interpreted by a computer. The most common asserts are those that takes one expression and evaluates it, and those that takes two values and compares them (Beck, 2003:157).

2.5.2. Test case

The `TestCase` class is one of the most used classes when working with a xUnit framework. When a test class is created it is derived from the `TestCase`. In the test class test functions are implemented (Astels, 2003:66). Test functions are of void type, which means that no value is returned. The naming convention normally has test as pre- or postfix, for example `testSomething` or `someTest`.

2.5.3. Test fixture

A test fixture is used is to create a test environment where tests can be run in a well known repeatable way (Wikipedia, 2009 a). The xUnit framework architecture uses `setup` and `teardown` functions that are implemented in the test class.

An in-line setup can be used in each test method to initialize a needed precondition. But when there are several tests that use the same initialization the setup code can be moved to a fixture. The needed variables are set as members in the test class. With this approach variable initialization does not have to be copy pasted from one function to another. If a used interface changes, the change is made to only one place (Beck, 2003:158-159).

Test fixtures supports that tests are run in isolation. For each test method the setup function is called upon before execution and teardown afterwards. In this way each test is guaranteed to be run from a clean table (Hammil, 2005:28).

2.5.4. Test suite

A test suite is used to execute several tests in the same test run. When a test class derived from the `TestCase` class is executed all its test method are invoked. Test classes and suites can be added to suites. In this way larger hierarchies of tests can be built and executed (Hammil, 2005:29).

2.5.5. Test organization

When using a xUnit-architecture based unit test framework tests can be organized in different ways. One common way is to use one test case class per tested class. Then all test methods belongs to the test class. With this approach the test methods share the same fixture. This can be a problem if the tests require different fixtures. One solution is to use an in-line fixture setup for each test method. But this can result in test code duplication and complicated tests. Another solution is to use the one test case per fixture approach (Meszaros, 2007:618).

The one test case per fixture pattern is used when test are organized based on commonality of the desired fixture. When this way of organizing test cases is used the tests methods can be made simpler and focus on the test since they are using the same fixture setup method (Meszaros, 2007:632).

Tests can also be organized into one test case class per feature. This can be used if a test case class has many test methods. Rearranging the tests into one test case class per feature can give an better overview of the tests. This approach is also used for system testing with automated tests. One variation of this test organization pattern is to have one test case class per method (Meszaros, 2007:625).

2.6. TDD patterns

This chapter presents patterns that are used when writing tests and practicing TDD. A pattern is a way to describe best practices and solutions in a way that others can reuse these experiences (Sommerville, 2004:421).

2.6.1. Four-Phase Test

The Four-Phase Test pattern is used to structure a test in a way that it is obvious what is tested.

- Setup, the tested component is set in the state that is required to make the test.
- Exercise, the actual interaction is done.
- Verify, measures if the desired outcome was achieved in the exercise phase.
- Teardown, cleanup is done if needed.

When tests are composed with these phases identified it is easier for a reader to determine what behavior the test is verifying (Meszaros, 2007:358). A variation of this pattern is the 3A pattern. The three steps are: Arrange – create some objects, Act – stimulate them and Assert – check the results (Beck, 2003:97).

2.6.2. Faking It

When TDD is practiced, the green state for the tests is preferred, i.e. the tests should pass. When an implementation is hard to find the test can stay in the red state for a longer time. To prevent this state the Faking It pattern could be used. When this approach is used a hardcoded constant is returned. Then the correct implementation is gradually developed, changing hardcoded values to variables (Beck, 2003:151).

2.6.3. Obvious Implementation

When the implementation to get a written test to pass is clear and obvious the Obvious Implementation pattern can be used. That means that the correct implementation is written directly, instead of creating the implementation in several steps. When using this pattern a bigger step can be taken than usually. If the obvious implementation fails the code should immediately be removed and then be rebuilt taking small steps (Koskela, 2008:105).

2.6.4. Triangulation

The Triangulation pattern can be used to gradually narrow down the space for the correct solution. Constants that are introduced with the Fake It pattern are gradually replaced by writing tests that push the development towards the correct implementation (Koskela, 2008:104).

2.6.5. Delta Assertion

If an exact state of the tested component is hard to obtain, the Delta Assertion pattern can be used. When using a delta assertion, a snapshot of relevant parts is recorded from the tested components current state, after that the testing action is exercised. The verification is done based on the difference of the result and the taken snapshot. This approach allows the test to be less dependent on the data that already exist in the tested component (Meszaros, 2007:485).

2.6.6. Anti-patterns

An anti pattern is a commonly occurring solution to a problem that has negative consequences (Brown, et al. 1998:7). Indicators of bad tests (Beck, 2003:194).

- Long setup, if the setup to make an assertion is very long the used objects are probably too big and needs to be split.

- Setup duplication, if a common place can not be found for common setup code, then there are probably too many objects that are intertwined.

- Long running tests, when tests take a long time to execute it indicates that there is a problem with the design that needs to be corrected.

2.7. Tools

The most used tool when practicing TDD is the automated unit test framework. This sub-chapter will focus on what other programming tools there are available that can support the TDD process.

2.7.1. Test coverage

Test coverage or code coverage is a way to measure how thoroughly written automated unit tests exercise the production code's statements, branches and expressions (Koskela, 2008:40).

A test coverage tool can be used to measure how well the production code is covered by the tests. One thing to pay attention to when using this tool is that it does not tell anything about the quality of the written tests (Elssamadisy, 2008:171).

2.7.2. Mock objects

A mock object framework can be used to dynamically generate objects of a given interface. These objects can then be configured dynamically before usage. Functions can be set up to return a given value. The object can also be configured to observe the tested components behavior. This is done by registering expected functions calls from the tested component. If these function calls are not done, the test fails (Meszaros 2007:544).

An example of a mock object framework for Java is EasyMock. When creating a mock object with EasyMock it is in recording mode. In recording mode the object can be configured with return values for functions. Expected function calls including expected parameters from the tested component can also be configured. To get the mock in action the frameworks replay function is called with the mock as parameter. Then the mock starts to watch and record how the tested component interacts with it. When all test functionality is exercised the mock can be passed to the frameworks verify function that fails the test if the expected collaboration with the tested component was not achieved (Koskela, 2008:118).

2.8. Adopting TDD

This sub-chapter will focus on what is required from the individual developer and the team to adopt TDD. The most common pitfalls and how they are avoided are discussed in the TDD adaptation pitfalls sub-chapter. Taking on a new software development technique is not an easy task, especially when it radically differs from the current way of working (Koskela, 2008:436).

Writing the tests before the implementation is radically different than the traditional test-last method. To change the way a professional software engineer, with possibly decades of experience, work is not an easy process, it will take time (Koskela, 2008:443).

2.8.1. Adopting tools and techniques

To adopt TDD it is required from the developer to learn a new software development technique and at least one new tool. The new tool that is mandatory is the unit test framework.

Learning a new tool or technique reduces the productivity and quality in the beginning. There is a effort of learning and understanding new ideas and to see how they fit the current development environment. The first project where the new tool or technique is used the developers are less effective, which makes the project to take a longer time than usual. The actual benefit of a new idea is not gained until the learning curve is overcome. It is difficult to say in general terms how long the learning curve is going to be. But the greater benefit in the end, the longer the learning curve is. The length of the learning curve and the greatness of the benefit can be estimated by asking someone already using or performing the same tool or technique. The specialist’s answer has to be critically evaluated, since the person can be a zealot or salesman who wants to promote a certain product (Glass, 2003:23-24).

2.8.2. Adopting TDD

For adopting TDD Elssamadisy points out a significant reduction of development speed. The first two months the loss can be 50 percent for a new project and up to 75 percent for a project with legacy code that was not written with testing in mind. (Elssamadisy, 2008:180). It will take two to six months for the developers to learn writing automated unit tests in a way that it becomes a habit and the benefits are seen (Elssamadisy, 2008:167).

Pair programming is an agile software development method that has its origins in XP. It means that two persons work together on a single computer (ExtremeProgramming.org, 1999). This method can be used to ease the team's learning curve. It is easier to be disciplined with writing tests when working together with some one else (Elssamadisy, 2008:168).

It is required from the team that disbeliefs are set aside for at least two months. To learn TDD without external training is hard. The team can be sent on external seminars or training courses. It is also highly recommended to bring in external expertise. This person should have practical experience of the TDD method and should be able to teach the team with a hands-on approach for example through pair programming (Elssamadisy, 2008:180-181).

In a case study of an agile software development team (Cunningham et al, 2004:19) the difficulties of using TDD is discussed. The data was collected with a questionnaire that was a part of the study. There were major obstacles of unit test and TDD usage. Developers found it hard to practice TDD on the complex parts of the system and to write tests for all pieces of source code. Limitations of existing unit test frameworks were also a problem. GUI applications are hard to test. Developers tended to abandon writing unit tests during deadline pressure, it was more important to complete all features and to deliver the promised functionality.

2.8.3. TDD adaptation pitfalls

It is required that the whole team is committed when starting TDD. If one person breaks the tests it is much easier for other to do so also. The developers must be used to fix tests that are broken, even in code that some one other has written and in a part of the system that is not familiar. To do this the team must use collective code ownership (Elssamadisy, 2008:170).

Programmers prefer programming in favor of testing and tend to write incomplete test that only tests happy scenarios. Therefore critical parts of the system that is hard to test can get poorly written tests (Sommerville, 2004:404).

The adoption must be seen positive by the developers, the benefit must be clear. If the software development team does not realize why TDD can be useful the adaption will probably fail (Koskela, 2008:436).

3. TEST FRAMEWORK EVALUATION

One of the goals of this thesis is to evaluate unit test frameworks for the C++ programming language. There is no de facto standard unit test framework for standard C++ as with other programming languages; Java has JUnit and C# .NET has NUnit.

The evaluated unit test frameworks are CppUnit and Boost.Test. Both of them are licensed as free software. The CppUnit was chosen since it is presented as the main C++ port of the xUnit-architecture. At the company Boost libraries are already used and they are found robust and useful, therefore the Boost.Test unit test framework is also evaluated.

3.1. Requirements and evaluation criteria

The PC software projects in the company are divided into two parts, the business and application logic-layer and the GUI layer. The GUI layer is built with Borland C++ Builder 6. The business and application logic layer is built with Microsoft Visual Studio 2008.

It is the business and application logic layer that is going to be unit tested. The unit test framework shall be integrated into Microsoft Visual Studio 2008. The test should run automatically when a project is compiled or built. It should be possible to have a setting that makes the build fail if a unit test fails. The framework should be able to produce relevant output for the compilers output screen. If a test fails an error message should be reported to the output screen in the same way as a compile warning or error.

As shown in chapter 2.1 the TDD cycle requires that test are written frequently. The tests should be easy to write. When writing tests the focus should be on testing, not coding the tests (Meszaros, 2007:27). Thus it should be easy to setup a test environment for a unit and easy to add new tests.

Fixtures can help to reduce test code duplication and tests can be run in isolation. One evaluation criteria is fixture support.

The TDD cycle requires frequent test execution. If the tests are hard to run there is a risk that the developer is tempted to skip the tests and hoping that the changes did not break anything in the existing code (Koskela, 2008:29). None of the evaluated frameworks are commercial products with explicit built in support for Visual Studio integration. Therefore one evaluation criteria is that the unit test framework supports IDE integration and that the tests can easily be executed.

The evaluation criteria:

- Minimal amount of steps to setup tests for a unit.
- Minimal amount of steps to add a test.
- Fixture support.
- IDE integration and easy test execution.

The unit test framework evaluation will be done writing unit tests to a simple class representing a person, shown in example 3.1. The frameworks are presented with examples of writing the easiest possible test, use test fixtures, use test suites and how to configure IDE output and integration.

Example 3.1. The class Person

Person.h

```
class Person
{
public:
    Person(const std::string & name, const int age) :
        m_name(name), m_age(age) {}
    const std::string & getName() { return m_name; }
    const int getAge() { return m_age; }
private:
    const std::string m_name;
    const int m_age;
};
```

3.2. CppUnit

CppUnit was chosen for evaluation since it is presented as the C++ port of the xUnit-architecture in Hammil's Unit Test Frameworks. It also is the most downloaded C/C++ unit testing tool on opensourcetesting.org (Aberdour, 2009).

CppUnit is open source and licensed under GPL (GNU General Public License). The framework is port of Java's JUnit into C++. It uses the architecture of the xUnit model. The implementation details are C++ specific; it uses templates and STL (Standard Template Library). C macros are used for easing repetitive tasks. The framework supports output in an IDE friendly way and through a GUI component for usage in an MFC or Qt development environment. (Hammil, 2005:70-72).

3.2.1. CppUnit simple test

The easiest way to write a unit test with CppUnit is to create a unit test class that is derived from the framework's `TestCase` and override the `runTest()` function. The implementation will throw an exception if an assertion fail, see Example 3.2.

Example 3.2. Person simple test

PersonTest.cpp

```
#include "cppunit/TestCase.h"
#include "Person.h"

class PersonTest : public CppUnit::TestCase
{
public:
    PersonTest(const std::string & name) : CppUnit::TestCase(name)
    {
        //
    }
    void runTest()
    {
        Person person("Bjarne", 40);
        CPPUNIT_ASSERT(person.getName() == "Bjarne");
    }
};

int main()
{
    try
    {
        PersonTest test("PersonTest");
        test.runTest();
        std::cout << "Test success" << std::endl;
        return 0;
    }
    catch (...)
    {
        std::cout << "Test failure" << std::endl;
        return 1;
    }

}
```

3.2.2. CppUnit fixture usage

A test fixture can be used to setup a test environment for the different tests. This is done by deriving the test class from CppUnit's `TestFixture` class. The fixtures instance variables can be created and prepared in the `setUp` function. The `teardown` function cleans up after the tests are executed. To run the tests a `TestRunner` is used. The simplest way to run a test is to create a `TestCaller` object. The test caller objects parameters are the test name and the address of the test function to be executed. The test caller object is added to the runner with the `addTest` function. See example 3.3.

Example 3.3. Person fixture test

PersonFixtureTest.cpp

```
#include "cppunit/TestCase.h"
#include "cppunit/ui/text/TestRunner.h"
#include "cppunit/TestCaller.h"
#include "Person.h"

class PersonFixtureTest : public CppUnit::TestFixture
{
public:
    void setUp()
    {
        m_person = new Person("Bjarne", 40);
    }
    void teardown()
    {
        delete m_person;
    }
    void testName()
    {
        CPPUNIT_ASSERT("Bjarne" == m_person->getName());
    }
private:
    Person * m_person;
};

int main()
{
    CppUnit::TextUi::TestRunner runner;
    runner.addTest(
        new CppUnit::TestCaller<PersonFixtureTest>
            ("testName", &PersonFixtureTest::testName));
    if (runner.run())
        return 0;
    else
        return 1;
}
```

3.2.3. CppUnit test suite

A test suite can be used to run several tests at once. This is done by adding the suite method to the test class. In the suite, test callers are added. When the suite is added to the test runner, all tests in the suite are executed, see example 3.4. A test suite can also contain other suites, in that way larger tests can be managed. (Feathers & Lepilleur, 2008).

Example 3.4. Person suite test

PersonSuiteTest.cpp

```
...
void testAge()
{
    CPPUNIT_ASSERT(40 == m_person->getAge());
}
static CppUnit::Test * suite()
{
    CppUnit::TestSuite * personSuite =
        new CppUnit::TestSuite("PersonTestSuite");
    personSuite->addTest(new CppUnit::TestCaller<PersonSuiteTest>(
        "testName", &PersonSuiteTest::testName));
    personSuite->addTest(new CppUnit::TestCaller<PersonSuiteTest>(
        "testAge", &PersonSuiteTest::testAge));
    return personSuite;
}
...
int main()
{
    CppUnit::TextUi::TestRunner runner;
    runner.addTest(PersonSuiteTest::suite());
    if (runner.run())
        return 0;
    else
        return 1;

}
```

3.2.4. CppUnit helper macros

To manually create suites is a highly repetitative task. The framework has helper macros that can be used to create the `suite` function. A test suite is created with the test class name as parameter to the `CPPUNIT_TEST_SUITE` macro. The test functions are added with the `CPPUNIT_TEST` macro. Test names are created from the test class name combined with the test function name. The test suite declaration is then ended with the `CPPUNIT_TEST_SUITE_END()` macro. See example 3.5

Example 3.5. Test suite helper macros.

PersonTestMacroHelper.cpp

```
...
#include "cppunit/extensions/HelperMacros.h"
...
CPPUNIT_TEST_SUITE(PersonTestMacroHelper);
CPPUNIT_TEST(testName);
CPPUNIT_TEST(testAge);
CPPUNIT_TEST_SUITE_END();
...
```

A test factory registry can be used to automatically register a test suite. The advantages of this is that the test suites do not have to be registered separatly to the runner, it also reduces compilation bottlenecks when all test file headers don’t need to be included by the file containing the main method (see example 3.6).

Example 3.6. Test factory registry

PersonTestMacroHelper.cpp

```
...
CPPUNIT_TEST_SUITE_REGISTRATION(PersonTestMacroHelper);
```

Runner.cpp

```
#include "cppunit/ui/text/TestRunner.h"
#include "cppunit/extensions/TestFactoryRegistry.h"

int main()
{
    CppUnit::TextUi::TestRunner runner;
    CppUnit::TestFactoryRegistry & registry =
        CppUnit::TestFactoryRegistry::getRegistry();
    runner.addTest(registry.makeTest());
    if (runner.run())
        return 0;
    else
        return 1;
}
```

3.2.5. CppUnit IDE integration

A post-build event can be added in the project settings. This makes the project build to fail if an executed test fails. In Microsoft Visual Studio 2008 a post-build event is added in the project configuration, `$(TargetPath)` variable is added to the Command Line property. In example 3.7 the test runner is configured with the frameworks

CompilerOutputter to create the output in a compiler compatible format. In that way it is possible to directly access a code line where a test failed, in the same way as with a warning or error from the compiler output (see example 3.8).

Example 3.7. Compiler compatible output configuration.

Runner.cpp

```
...
runner.setOutputter(
    new CppUnit::CompilerOutputter(&runner.result(), std::cout));
...
```

Example 3.8. Test output to the IDE output window

```
------ Build started: Project: CppUnitTest, Configuration: Debug Win32
Compiling...
PersonTestMacroHelper.cpp
Runner.cpp
Generating Code...
Linking...
Embedding manifest...
CppUnit
.F.
c:\thesis\cppunittest\persontestmacrohelper.cpp(21) : error : Assertion
Test name: PersonSuiteTest::testName
assertion failed
- Expression: "Kalle" == m_person->getName()
Failures !!!
Run: 2   Failure total: 1   Failures: 1   Errors: 0
Project : error PRJ0019: A tool returned an error code from "CppUnit"
Build log was saved at
"file://c:\Thesis\CppUnitTest\Debug\BuildLog.htm"
CppUnitTest - 2 error(s), 0 warning(s)

========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped
```

3.3. Boost.Test

The Boost.Test framework was chosen for evaluation since the development organization is currently using the Boost libraries and has found them very good and useful.

The Boost.Test unit test framework is a part of Boost C++ source libraries. It is licensed under the Boost License, which encourages commercial and non-commercial usage. The architecture does not follow the xUnit model that uses test classes that has their member functions as test functions. In Boost.Test the test functions are free functions that are registered to the unit test framework for execution. The framework uses macros to generate code needed for test execution. Assertions are used to perform the actual tests.

3.3.1. Boost.Test simple test

The simplest way to create a test is to use the `BOOST_AUTO_TEST_CASE` macro. It takes the test case name as parameter. The definition of `BOOST_TEST_MODULE` generates a test module initialization function and main function implementation, see example 3.9.

Example 3.9. Simple test

PersonTestSimple.cpp

```
#define BOOST_TEST_MODULE personExample
#include <boost/test/included/unit_test.hpp>

#include "Person.h"

BOOST_AUTO_TEST_CASE(test_personName)
{
    Person person("Bjarne", 40);
    BOOST_CHECK(person.getName() == "Bjarne");
}
```

3.3.2. Boost.Test fixture usage

In the xUnit architecture fixtures are implemented in the test class, variables are class members and the setup and teardown functions are used for initialization and cleanup. With Boost.Test a fixture is implemented as a separate fixture class or struct. The setup and teardown functionality is implemented in the constructor and destructor. This is a natural way to implement it in the C++ programming language. The test using a fixture has access to all of the fixtures public and protected members, a struct data type has all it’s members as public, so often it is used as the fixture data type. The fixture is implemented with the fixture variables as member fields. When using a fixture the `BOOST_FIXTURE_TEST_CASE` macro can be used. It takes the test name and the fixture name as parameters. Within the test the fixtures public and protected member variables are directly accessible, see example 3.10. (Rozental, 2007).

Example 3.10. Fixture test

PersonTestFixture.cpp

```
...
struct PersonFixture
{
    PersonFixture()
    {
        m_person = new Person("Bjarne", 40);
    }
    ~PersonFixture()
    {
        delete m_person;
    }
    Person * m_person;
};

BOOST_FIXTURE_TEST_CASE(test_personName, PersonFixture)
{
    BOOST_CHECK(m_person->getName() == "Bjarne");
}
```

3.3.3. Boost.Test test suite

Test suites are used to build hierarchies of tests. The `BOOST_TEST_MODULE` is the master suite when executing tests. A test suite is declared with the macro `BOOST_AUTO_TEST_SUITE` that takes the suite name as parameter. The suite is ended with `BOOST_AUTO_TEST_SUITE_END`. Tests that are not inside a suite are executed with the master test suite, see example 3.11. When a suite is declared a fixture can be assigned to the whole suite. This is done with the `BOOST_FIXTURE_TEST_SUITE` macro that takes the suite name and the fixture name as parameters. When this approach is used normal test cases have the suite fixture's public and protected members accessible.

Example 3.11. Test suite usage

PersonTestSuite.cpp

```
...

BOOST_FIXTURE_TEST_SUITE(personTestSuite, PersonFixture)

BOOST_AUTO_TEST_CASE(test_name)
{
    BOOST_CHECK_EQUAL(m_person->getName(), "Bjarne");
}
BOOST_AUTO_TEST_SUITE_END()

BOOST_AUTO_TEST_CASE(test_random)
{
    BOOST_MESSAGE("Random person test");
}

BOOST_FIXTURE_TEST_SUITE(personTestSuite, PersonFixture)

BOOST_AUTO_TEST_CASE(test_age)
{
    BOOST_CHECK_EQUAL(m_person->getAge(), 40);
}

BOOST_AUTO_TEST_SUITE_END()
```

The tests in a suite can be divided to several locations in the source code. When they are executed the order is the same as they are implemented, except if a test case belongs to an earlier declared suite, see example 3.12 for execution output details.

Example 3.12. Suite hierarchy output

```
Test suite "masterPersonSuite" passed with:
  2 assertions out of 2 passed
  3 test cases out of 3 passed
  Test suite "personTestSuite" passed with:
    2 assertions out of 2 passed
    2 test cases out of 2 passed
    Test case "test_name" passed with:
      1 assertion out of 1 passed
    Test case "test_age" passed with:
      1 assertion out of 1 passed
  Test case "test_random" passed
```

3.3.4. Boost.Test IDE integration

In Microsoft Visual Studio 2008 the test execution results can be directed to the compiler output window. This is done by adding a post build event to the project configuration.

The failing tests' error messages are formatted for the compiler output, thus the error messages can be used to find the failing tests in the source code such as with compiler warnings and errors (See example 3.13)

Example 3.13. Boost test framework compiler output

```
Build started: Project: BoostTestEvaluation, Configuration: Debug Win32
Compiling...
PersonTestSuite.cpp
Linking...
Embedding manifest...
Performing Post-Build Event...
Running 3 test cases...
c:/thesis/boosttestevaluation/persontestsuite.cpp(24): error in
"test_name": check m_person->getName() ==
"Kent" failed [Bjarne != Kent]
Build log was saved at
"file://c:\Thesis\BoostTestEvaluation\Debug\BuildLog.htm"
BoostTestEvaluation - 1 error(s), 0 warning(s)

======= Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =======
```

3.4. Test framework evaluation conclusion

The result is presented in the table below (see table 1) and as text where each evaluation criteria and result is discussed more in detail. When a decision is made it is important to be able to aggregate different qualities and properties of the evaluated object. When a table is used the information that is the base for the decision is easily foreseen (Höst et al, 2006:74).

Table 1. Unit test frame work evaluation result

Unit test framework evaluation	CppUnit	Boost.Test
Minimal amount of steps to setup tests for a unit	Implement test class, test function, main and instantiate test runner. Steps: 4	Implement free test function and define main. Steps: 2
Minimal amount of steps to add a test	Implement test function to test class, function registration to test suite. Steps: 2	Implement free test function. Steps: 1
Fixture support	Yes, xUnit architecture style, test class as fixture	Yes, Implemented as class or struct, can be used from any test
IDE integration and easy test execution	CompilerOutputter has to be added to runner to get clickable compiler output	Yes, by default

Minimal amount of steps to setup tests for a unit

To setup tests with CppUnit a test class is created and the tests are implemented as member functions. Then a main function has to be added and a test runner instantiated. With the Boost.Test framework a free test function can be written and the runner that declares a main function can be created with a define statement.

Minimal amount of steps to add a test

CppUnit includes two steps. The test function is added to the test class, and then the function is registered to the test suite. With Boost.Test it is only one step, adding the test function.

Fixture support

Both CppUnit and Boost.Test supports fixtures. With CppUnit a fixture is implemented following the xUnit architecture. The test class acts as the fixture, initialization and cleanup are done by implementing setup and teardown functions in the test class. With the Boost.Test framework a fixture is created with a fixture class, the initialization and cleanup is done in the constructor and destructor. The fixture class can then be assigned to separate test functions, or all test functions within a test suite. The Boost.Test framework has a key advantage when using fixtures. Within a test unit several fixtures can be declared and used by the test functions. With CppUnit the test class is the fixture, when a new fixture is needed a new test class has to be created.

IDE integration and easy test execution

IDE integration with Microsoft Visual Studio 2008 is supported by both unit test frameworks. A build can be set to fail if a test fails by adding a post build event. The test result output can be directed to the compiler output window with failing tests reported in the same way as compiler warnings and errors. With CppUnit a CompilerOutputter has to be added to the test runner for correct compiler output.

3.4.1. Recommendation

The recommendation based on evaluation results is that Boost.Test unit test framework is more suitable for testing in a C++ development environment than CppUnit. The Boost.Test unit test framework requires less steps to setup a testing environment. The usage of fixtures in combination with free test functions is more flexible than using the test class as a fixture.

4. DISCUSSION

Test-driven development is one of the hardest agile methods to adopt (Elssamadisy, 2008:181). The way of working is radically different from the traditional test-last approach. It is hard to adopt the technique to write the tests before the actual implementation. The tests themselves are not easy to write either; it requires a lot of practice. The question “what are good test?” is as hard to answer as “what is good code?”

4.1. Criticism

When studying literature about TDD the method is praised and presented as a highly working solution. The method is presented in theory with fairly simple examples, but there is a question of how it works in practice in a real software development project.

The TDD method presents a way to develop and design software. The design is described as an ongoing activity that is developed incrementally. This could work in a relatively small project that uses a well known platform. But it does not solve large scale design for a project with many modules divided into several layers. In this case some degree of upfront design and ahead planning is needed.

4.2. Software quality

The test harness that is created when developing with TDD enables refactoring. When it is possible to refactor, new features and functionality can be added to a software product. Once a software project with poorly written unit tests is extended with new functionality without changing the design to fit the new requirements the code base will become fragile over time. This will result in that the developer does not dare to make bigger changes with the risk that something else in the system breaks. Quick and dirty bandage solutions are added to the system when implementing new functionality and fixing bugs, increasing the design debt. A stock phrase in software development circles is that "If it ain't broke, don't fix it". This approach to software development is the beginning of the end to a software product. When developing with TDD the test harness can be used for regression testing. This means that changes can be made in the design to adopt new features, which lengthens a software products lifetime. This an important business value in means of quality to market and costs.

Feedback is important when developing software. The earlier the development team can get feedback on the developed software the better it can react. When testing is made manually by a testing department the cycle for finding and repairing a bug becomes unnecessary long. The TDD process creates and maintains an exhaustive test harness which can help to find bugs early in the development cycle. This does not mean that testing by a testing department is not needed. The software needs system testing in a real environment. Professional tester's tests are focused on trying to break the software, which is needed, since developers tend to write tests that are verifying that the software works. Unit tests will never find all the bugs, but it certainly can help to discover many of them in a early stage of the development. In this way the tests performed by the testing department can focus on finding the bugs related to system testing. The testing cycle can be shortened, which has a clear business value in means of time to market.

4.3. Unit test framework evaluation

The objectives for the practical part were to evaluate unit test frameworks for C++ and to examine if the xUnit architecture is optimal for a C++ environment. To create the evaluation criteria the theoretical part of the thesis was used. The TDD cycle presented in chapter 2.1 showed that tests where frequently added and executed. Therefore one important measurement in the evaluation was that tests could be added with as few steps as possible. It was showed in chapter 3 that the framework using free functions had fewer steps to setup tests for a unit and adding new tests. It also had more flexible usage of fixtures than the xUnit port framework.

One advantage of using a xUnit-architecture based unit test framework is that it can be easier to adopt for developers that has knowledge of automated unit testing from other programming languages, where a xUnit-architecture based test framework is used.

4.4. Further development

This thesis focused on the TDD process from a unit test point of view, i.e. to verify that software modules work as the programmer intends. The next step would be to write automated acceptance tests. Those are written from a customer point of view and are intended to verify that the system works as specified. This is black box testing, which means that the tests does not know, and does not care about the underlying implementation. The acceptance tests could be written in collaboration with the testing team and the product owner. This requires a software design that is built in layers, where the business logic parts of the system could be accessed without the GUI. There are tools that support automated acceptance tests.

FitNesse is an acceptance testing framework. The software development team including process owner, testing department and software developers can collaboratively define acceptance tests. The tests are written in a non technical language in table format, specifying the input data and expected output. The link between the table data and system under test is implemented by the software developers (FitNesse, 2009).

The automated unit tests could be included in a nightly build server or a continuous integration server. In that way the tests would always be executed when the software is built.

4.5. Conclusion

The conclusion is that the unit tests created as a part of TDD is a powerful tool that should be used. It increases software quality in a way that has concrete business value. To fully adopt TDD and to use it as a design and development method is proved to be difficult since it is radically different from the traditional way to develop software. The adoption would require the will and understanding from each individual developer. It would also need support from the management since the developers would be less effective during the time it takes to overcome the learning curve.

TDD has potential and should be seriously evaluated in practice in a way that the possible benefit of the method could be measured. In chapter 2.8.2 the time to adopt the method is pointed out to be between two to six months. The evaluation time should be at least within this scope. It could be arranged as part of a project and be done by developers that want to test the method. The results of the evaluation could then be discussed with the development team and software project stakeholders on a seminar or workshop like event.

REFERENCES

Aberdour, Mark. 2009. Opensourcetesting.org. Retrieved 23.1.2009 [www] http://www.opensourcetesting.org/unit_ada.php

Astels, David. 2003. Test-driven development, a practical guide. Prentice Hall. ISBN: 0-13-101649-0

Beck, Kent. 2003. Test-driven development by example. Addison-Wesley. ISBN: 0-321-14653-0

Brown, William J. Malveau, Raphael C. McCormic III, Hays W. Mowbray, Thomas J. 1998. AntiPatterns, Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc. ISBN: 0-471-19713-0

Cunningham, Lynn. Layman, Lucas. Williams, Laurie. 2004. Motivations and Measurements in an Agile Case Study.

Elssamadisy, Amr. 2008. Agile Adoption Patterns, A roadmap to Organizational Success. Addison-Wesley. ISBN: 0-321-51452-1

ExtremeProgramming.org. 1999. Pair programming. Retrieved 18.3.2009 [www] http://www.extremeprogramming.org/rules/pair.html

Feathers, Michael. 2007. API Design as ifUnit Testing Mattered. Object Mentor, Inc. Retrieved 26.2.2009 [www] http://www.objectmentor.com/resources/articles/as_if_unit_testing_mattered.pdf

Feathers, Michael & Lepilleur, Baptiste. 2009. CppUnit Cookbook. Retrieved 27.1.2009 [www] http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html

FitNesse. 2008. Retrieved 1.4.2009 [www] http://fitnesse.org/

Glass, Robert L. 2003. Facts and Fallacies of Software Engineering. Addison Wesley. ISBN: 0-321-11742-5

Hamill, Paul. 2005. Unit Test Frameworks. O'Reilly Media Inc. ISBN: 0-596-00689-6

Hunt, Andrew. Thomas, David. 2000. Addison Wesley Longman, Inc. ISBN: 0-201-61622-X

Höst, Martin. Regnell, Björn. Runeson, Per. 2006. Att genomföra examensarbete. Studentlitteratur. ISBN: 91-44-00521-0

Janzen, David S. Saiedan, Hossein. 2008. Does Test-Driven Development Really Improve Software Design Quality? I: IEEE Software, March/April. p. 77-84.

Koskela, Lasse. 2008. Test Driven, Practical TDD and Acceptance TDD for Java Developers. Manning Publications Co. ISBN: 1-932394-85-0

Kerievsky, Joshua. 2004. Refactoring to Patterns. Addison-Wesley. ISBN: 0-321-21335-1

Martin, Robert C. 2007. Professionalism and Test-driven Development. I: IEEE Software, May/June. p. 32-36.

Martin, Robert C. Martin, Micah. 2006. Agile principles, patters, and practices in C#. Pearson Education, Inc. ISBN: 0-13-185725-8

Meszaros, Gerard. 2007.xUnit Test Patterns, Refactoring Test Code. ISBN: 0-13-149505-4

ScrumAlliance. 2009. What Is Scrum? Retrieved 13.3.2009 [www] http://www.scrumalliance.org/learn_about_scrum

Sommerville, Ian. 2004. Software Engineering, seventh edition. Addison-Wesley. ISBN: 0-321-21026

Rozental, Gennadiy. 2007 a. Per test case fixture, Boost.Test documentation. Retrieved 19.2.2009 [www] http://www.boost.org/doc/libs/1_38_0/libs/test/doc/html/utf/user-guide/fixture/per-test-case.html

Wikipedia. 2009 a. Test fixture. Retrieved 10.2.2009 [www] http://en.wikipedia.org/wiki/Test_fixture

Wikipedia. 2009 b. Code smell. Retrieved 11.2.2009 [www] http://en.wikipedia.org/wiki/Code_smell

Wikipedia. 2009 c. Scrum. Retrieved 10.3.2009 [www] http://en.wikipedia.org/wiki/Scrum_(development)

Wikipedia. 2009 d. Cyclomatic complexity. Retrieved 26.3.2009 [www] http://en.wikipedia.org/wiki/Cyclomatic_complexity

Wikipedia. 2009 e. Coupling. Retrieved 26.3.2009 [www] http://en.wikipedia.org/wiki/Coupling_(computer_science)

Wohlin, Claes. 2005. Introduktion till programvaruutveckling. Studentlitteratur. ISBN: 91-44-02861-X