Unit Testing for Agile Methods

Marika J. Östman

Abstract— **Agile methods and mainly Extreme Programming ("XP") has brought Unit testing to every software developer's awareness. Unit testing, as such, is not a new invention as it has been used in traditional software processes for decades. Since the invention is not a new one, it raises a question why has unit testing gained such an important role all of a sudden in at least some modern agile software processes.**

The aim of this study is to assess how unit testing is performed in agile methods, what kind of quality issues can be addressed by unit testing and what benefits unit testing might bring along.

This study does not cover all the agile methods but focuses on Extreme Programming, Dynamic Systems Development Method ("DSDM"), Feature Driven Development ("FDD") and Crystal methodologies. The agile methods were chosen for this paper by using agile software method comparison by Abrahamsson et al. (2003) as the main method selection criteria.

Index Terms—**agile methods, developer testing, unit testing**

1. INTRODUCTION

PRODUCING high quality software is one of the key objectives of software development (Leung & Wong, 1997). Software quality can be defined as ISO defines it: *the totality of features and characteristics of a product, process or a service that bear on its ability to satisfy stated or implied needs* (Stockman et al., 1990). Software testing is essentially used to improve software quality.

1.1 Unit Testing in Software Testing

Software testing is usually done at several levels and can be divided into different types of testing: unit testing, integration testing, system testing and user acceptance testing (Whittaker, 2000; Leung & Wong, 1997; Palmer & Felsing, 2001).

Unit testing can be defined to be the type of testing, where a software developer (usually the one who wrote the code) proves that a code module (or "unit") meets its requirements. The unit being tested may be called as unit, component or module.

Unit testing is the first type of testing that is performed to a certain software component and testing is done to the smallest granularity of code being meaningful to test (Stapleton, 2002).

Major purpose of unit testing is, thus, to find defects in a software component. Unit tests are performed straight after module implementation and are, accordingly, the earliest tests applied to the code. (Kung et al., 1998)

In unit testing, the components called by the unit are usually replaced with stubs, simulators or trusted components and calling components are replaced with drivers or trusted super components so that the component being tested is isolated (Rose, 2004).

Due to the fact that the developer of the component performs the unit tests (Younessi et al., 2002) unit testing can also be called developer level testing.

Integration testing is the second level of testing that is applied after unit testing to the developed software. The objective of integration testing is to test the integration of and communication between components. Additionally, it may include testing the integration of subsystems or communication with external systems. (Rose, 2004)

The system-testing phase comprises testing of the whole software developed as a whole, rather than testing the behavior of individual components as in unit testing. The goal of system testing is to test that the whole system functions correctly. This level of testing is different from integration testing in that the tests are concerned with the entire system and not just the interactions between components. System testing may, for example also have tests for system performance, resource utilization and security. (Rose, 2004)

User acceptance testing is the final stage of testing in software development. Once test results meet the acceptance criteria, the software system can be released for operational use (Leung & Wong, 1997).

Of all these different types of testing Unit testing, the first level of testing, and especially its use in agile methods are the objects of this paper.

1.2 Overview of Agile Software Methods

Agile methods aim to offer lighter software processes for faster and nimbler software development (Abrahamsson et al., 2003). Agile methods offer alternative software development model to traditional waterfall software development model.

Agile Alliance is the organization uniting different modern and agile software processes such as XP, DSDM, FDD and Scrum etc. The "agile methods"-catch originates from this name. Organization was founded in February 2001 in Utah in a meeting, where different Iterative and Incremental Development (IID) software development practitioners attended to discuss common ground (Larman & Basili, 2003). The result of the meeting was a statement of common values and principles in agile processes called "The Manifesto for Agile Software Development" (Fowler, 2003).

Agile processes can be generally described with the following attributes: incremental, adaptive, cooperative and straightforward (Abrahamsson et al., 2003). Incremental development refers to small continuos and rapid development cycles and adaptiveness of the processes makes such processes especially suitable for continuously changing requirements compared to traditional waterfall software development model that emphasizes on defining requirements early on (Boehm &Turner, 2003). Co-operative development refers to close relations between customer and developers.

Straightforward development refers to the easy usability of the method.

1.3 Research Problem

The goal of this study is to research unit testing in certain agile software processes and compare it to the component/module/unit testing performed in traditional waterfall software processes.

Research questions for this study are the following:

- How is unit testing performed in modern agile procedures compared to traditional module testing?
- What kind of software quality issues can be addressed by unit testing?
- What are the benefits of using unit testing?

1.4 Research method

The research method of this paper is a literature study.

This exact topic of unit testing in agile methods is not widely covered by the scientific literature and thus makes an interesting topic to research. The most material found of unit testing in agile methodologies from electronic databases is XP material regarding the concept of Test Driven Development in which unit testing plays a significant role.

The main resources for literature are electronic databases available from the library of the Helsinki University of Technology, although material from Internet and books describing agile software methods and testing matters have also been used as resources.

1.5 Scope

This study is focused on unit testing on software developer level in agile software processes. Unit testing tools are out of the scope of this paper.

All agile methods are not covered in this paper. This paper is focused on XP, DSDM, FDD and Crystal methods, which are all rather well known agile methods. Agile methods comparative analysis (Abrahamsson et al., 2003) was used as a selection criteria for choosing agile methods for this paper. Abrahamsson et al. (2003) present comparisons how different agile methods provide different levels of process descriptions and concrete guidance for unit testing phase in software development. Those agile methods that were taken under closer study in this paper have their unit testing process described and may or may not offer also concrete guidance to performing unit testing. Methods that fulfill this criterion are (Abrahamsson et al., 2003): FDD, XP, DSDM, Crystal family of methods and ASD i.e. adaptive software development. Material on ASD related to unit testing was not available on databases available from Technical University of Helsinki and, accordingly, said method was excluded from the scope of this paper. Crystal family of methodologies consists of many methods designed for different kind of software processes. Due to the reference deficiency from other Crystal methodologies than Crystal Clear, this paper focuses mainly on the Crystal Clear method. Said method is targeted for smaller developer teams.

Pragmatic programming i.e. PP, is not a genuine agile method (Abrahamsson et al., 2002:83) and, therefore, it is not covered by this paper, although it contains concrete guidance related to unit testing.

1.6 Structure

The second chapter describes unit testing principles focusing on the definition of the “unit” in unit testing and the phases of unit testing. All possible unit testing features are not presented in this paper, since the focus is more on the unit testing particularly in agile methods.

The third chapter describes briefly the software processes of traditional software development and the agile methods covered by this study and then presents how unit testing is done in these software processes.

The fourth chapter focuses on software quality issues that can be addressed by unit testing and the relevance of these to the quality of the software in general.

The fifth chapter discusses the possible benefits of using unit testing and also some possible limitations and disadvantages that may be involved in using unit testing as an integral part of software process.

Last chapter discusses the role of unit testing in agile and traditional methods and refers briefly the writer’s own experiences in unit testing.

2. UNIT TESTING PRINCIPLES

Virtually all software that is written is unit tested and unit testing is the most frequently used technique in software defect management (Younessi et al., 2002).

2.1 Software Units

The testable unit in unit testing may mean different things for different software processes and for different software projects.

In software development during the time prior to object oriented languages, software units or components were considered to be functional procedures or function modules (Kung et al., 1998).

In object oriented software development the “unit” to be tested are functions or methods defined in classes or modules that consist of collection of classes (Kung et al., 1998).

Unit may be defined in general as a module consisting of functions, small programs, classes etc. that is usually in code line length between 50-600 lines (Younessi et al., 2002).

2.2 Software Units in Agile Methods

In FDD software process unit testing is defined, as testing of the smallest granularity of function being meaningful to test. As regards object oriented languages units consist, according to FDD, of nonprivate functions of a class. (Palmer & Felsing, 2001)

In XP software process unit tests are written for every new class or class function prior the class or function in question is implemented (George & Williams, 2003).

In DSDM the definition of unit in unit testing is not given, but it is left for software projects to define an overall strategy

for testing.

In Crystal Clear the agile practice of Crystal methodologies for small teams i.e. for up to six developers (Cockburn, 2002a) suggests that unit testing be done for classes, functions or in some cases for compound classes (Cockburn, 2002b).

As a conclusion, there are no wide differences between the definition of unit in unit testing in various agile software processes and in traditional software programming. Although it might be thought, that in agile methods the unit would be a slightly smaller one than in waterfall software processes. This seems to be the case for FDD, XP and Crystal Clear agile methods at least, if one compares their definition of unit to the possible definitions of units in traditional programming: small programs or modules.

2.3 Unit Testing Phases

The ANSI/IEEE Std 1008-1987 divides unit-testing process into three phases and eight activities (ANSI/IEEE Std 1008, 1987).

(i) Perform test planning
- Plan the approach, resources and schedule
- Determine features to be tested
- Refine the general plan

(ii) Acquire the test set
- Design the set of tests
- Implement the refined plan and design

(iii) Measure the test unit
- Execute the test procedures
- Check for termination
- Evaluate the test effort and unit

This standard was created for traditional unit testing and all the phases and activities may or may not be applicable for agile unit testing depending on the agile software process in question.

3. SOFTWARE PROCESSES AND UNIT TESTING

Agile software processes are generally IID software processes and all the processes covered by this study include unit testing as part of the process to partly ensure the quality of the software product developed. These iterative processes have different testing schemes, but the common thing is that testing is not left as the final step before the whole product or module is implemented as is the case in the traditional waterfall software process. This difference is partly due to the iterations implemented in agile methods. Traditional software process is covered in this chapter to provide comparison between unit testing in agile and traditional software processes.

3.1 Traditional Software Process

Traditional waterfall software process that has been used in software development for decades consists of phases that are represented in the figure below.

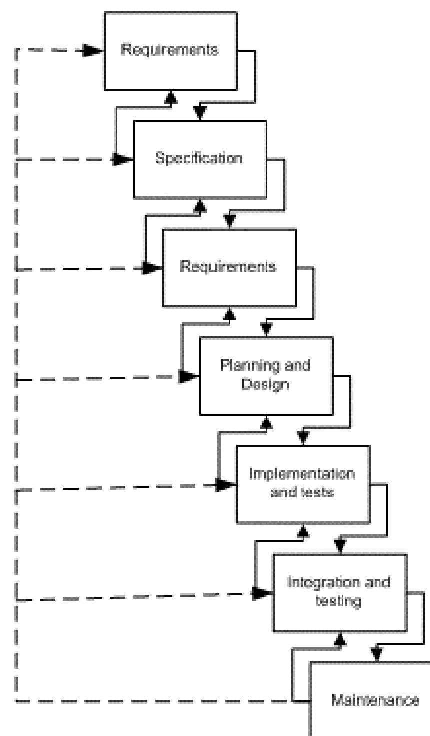

Figure 1 Waterfall Software Process Model (Schach, 1990)

The waterfall software process model is different from agile methods in that the process does not contain iterations and, thus, prototyping is hard to do in this model. The waterfall software process model tries to first specify all the requirements needed for the whole software product. Then specifications are written for the software product. Implementation of the product and its components follows design and planning phases. After components are implemented they are integrated. Testing occurs after implementation and integration phases (Haikala &Märijärvi, 1998)

3.2 Unit Testing in Traditional Software Process

Testing forms an important part of traditional software process. Generally three types of testing is performed: unit, integration and system testing. Unit testing, that could also be called as module testing, is performed after the implementation of a certain module. (Andriole, 1986)

In traditional software processes module developer implements the unit tests. The purpose in unit testing is to find typographical, syntax and logic errors in some module. Each of the modules of code are checked individually by the programmers, who wrote them, in order to ensure that each correctly implements its design and specified requirements (Andriole, 1986).

Unit tests in traditional software processes are usually not kept as regression tests for later on and partly for that reason are sometimes done in a more ad-hoc style to test functionality of some component. Unit tests may also not have been made as automated tests that can be run automatically. At least

Williams et al. (2003) states that even IBM developer's style to do unit tests was ad-hoc before adopting new unit testing standards.

3.3 XP Software Process

XP is a collection of existing software engineering practices that are used to enable successful software development despite of ever-changing requirements. The novelty of XP is the way these software-engineering practices are used to function with each other. (Abrahamsson et al., 2003)

Major practices in XP are: (Beck, 1999)

(i) Planning Game
(ii) Small releases
(iii) Metaphor
(iv) Simple Design
(v) Tests
(vi) Refactoring
(vii) Pair programming
(viii) Continuous integration
(ix) Collective ownership
(x) On-Site customer
(xi) Open workspace
(xii) Just rules

XP differs from traditional software process so that plans and designs are not made for the far future. These activities are done little by little throughout software development.

XP development lifecycle consists of five phases that are (Abrahamsson et al., 2002):

(i) Exploration
(ii) Planning
(iii) Iterations to release
(iv) Productionizing
(v) Maintenance and Death

In exploration and planning phases customer writes story cards, programmers determine the implementation effort for these requirements and requirements are prioritized. "Iterations to release" phase is the software development phase needed for first release divided into development efforts of one to four weeks. Productionizing phase is used for extra testing etc. to ensure that the software is ready to be handed out to the client. The last phases are related to maintenance of the software product after the first release (Abrahamsson et al., 2002).

3.4 Unit Testing in XP

Testing plays an important role in XP. Testing in XP can be called as Extreme testing and constitutes of unit tests and functional tests.

Unit testing technique is at the heart of XP and unit testing is part of every programmer's daily practice. Unit testing is performed by the developers of the unit in question. XP unit testing process is different from traditional unit testing also in the way, in which order code implementation and testing are performed. In XP, unit tests are written before the code implementation. (Beck, 1999)

XP performs test-driven development ("TDD"). TDD is the name for the practice, in which unit test cases are incrementally written prior to code implementation (George & Williams, 2003). Test first technique is not just testing, because this kind of coding drives also design decisions when test writing means considering what the functions return and what they take as attributes etc. (Beck, 2001).

XP unit tests are recorded in permanent form so that they can be used as regression tests. When extreme programmers release code, all the unit tests must run (Jeffries, 1999).

3.5 DSDM Software Process

DSDM is essentially a quality centered software method that uses Rapid Application Development ("RAD") techniques (Coleman, 1998). RAD can be defined as building what the business needs, when it needs it (Stapleton, 1997). DSDM uses prototyping to ensure frequent deliveries to customers and delivers these prototypes within fixed timeslots called timeboxes. These timeslots are usually a short period of time a matter of days or a couple of weeks (Stapleton, 1997;Coleman, 1998).

DSDM development lifecycle consists of five phases that are (Stapleton, 1997):

(i) Feasibility study
(ii) Business study
(iii) Functional model iteration
(iv) System design and build iteration
(v) Implementation

The first two phases are sequential and the last three phases, during which the actual software development is done, are iterative and incremental (Abrahamsson et al., 2002).

The first two phases assess the suitability of DSDM for the software project in question and analyze the essential characteristics of the business and technology (Abrahamsson et al., 2002).

"The functional model iteration" phase is the first iteration phase where the contents and approach is planned for the iteration, prototypes are being built etc. The design and build iteration phase is where the system is mainly built and also unit testing is done. The implementation phase is where the system is transferred to production environment and to the client (Abrahamsson et al., 2002).

3.6 Unit Testing in DSDM

This section is based on the book by Stapleton (1997).

One of the important principles in DSDM is that testing is integrated into the software process throughout the lifecycle and all forms of testing, including unit testing, are performed throughout the project not forgetting even the regression testing that is vital in iterative processes.

Developers test their components after implementation with unit tests. Therefore, unit testing is done after some component has been "finished". This is much the same approach as with traditional waterfall software process except

that the components are tested before each timeslot and unit tests are also run for previously generated components as regression tests.

Unit testing is thought to be so important for the quality of components that there is a principle that no component should ever be delivered to customer without undergoing unit testing and also user testing at the very least.

DSDM does not give any tight rules how to apply unit testing. Instead the method advises that the overall strategy for testing should be produced in the initial stages of the project. DSDM promotes the use of testing tools in the testing process, since they can save a lot of developer time.

3.7 FDD Software Process

Feature Driven Development is a process-oriented software development method for developing business critical systems that focus on design and building phases (Abrahamsson et al., 2003).

FDD is a highly iterative software process that emphasizes quality at each step. Fundamental practices of FDD are: (Palmer & Felsing, 2001)

- (i) Domain object modeling
- (ii) Developing by feature
- (iii) Individual class (code) ownership
- (iv) Feature teams
- (v) Inspections
- (vi) Regular builds
- (vii) Reporting/Visibility of results

"The domain object model" used in designing is a form of object decomposition and serves as an overall framework, where to add functions feature by feature.

Features are designed to be small enough to be implemented in two weeks time and most are small enough to be implemented even in a few hours or days. Accessory methods such as attribute setting and getting are not thought to be features.

In FDD programming classes have owners who are designated responsible for the class. Also features have owners, but features usually consist of many classes and thus feature teams consist of class owners. Programmers that lead the development are called chief programmers in FDD.

Regular builds of the system ensure that integration errors are noticed early.

High quality of designs and code are ensured by inspections, which are an integral part of FDD. The primary purpose of these inspections is to find defects as with testing. (Palmer & Felsing, 2001)

FDD consists of five sequential phases, during which the software is designed and built (Abrahamsson et al., 2002):

- (i) Develop an overall model
- (ii) Build a Features List
- (iii) Plan by Feature
- (iv) Design by Feature
- (v) Build by Feature

The first three phases are sequential phases comprising essentially of the planning of the software and its division into features and assigning of such features for development teams.

The last two phases, "design by feature" and "build by feature", are iterative phases, during which the software is developed i.e. the features are implemented. (Abrahamsson et al., 2002)

3.8 Unit Testing in FDD

This section is based on the book by Palmer and Felsing (2001).

Quality is emphasized at each step of FDD. Quality is ensured by testing and inspections. Nevertheless, testing is not mentioned as part of the fundamental principles of FDD and, therefore, takes a lower profile in comparison to, for example, testing in Extreme programming.

Unit testing in FDD is most often performed by individual developers without formal intervention by others and is done after implementation of some testable code. Class owners handle unit testing of their own code and chief programmers determine, whether features utilizing many components from different class owners need to be unit tested. Unit testing is thought to be useful since it is much quicker and cheaper for developers to find defects at the earliest stage of development.

The mechanism or level of formality of unit testing is not defined in the process, but is left for the chief programmer of the developer team to determine. Although the FDD method advises that, XP techniques for unit testing may be adopted.

In FDD projects unit testing is complemented by code inspections mentioned in the previous chapter, since inspections are thought to be even more effective in finding defects than testing alone. FDD practitioners think that inspections are capable of finding more types of defects than can be found by testing. The feature team responsible for some software feature conducts this inspection before or after unit testing. The developed item may be integrated to the build only after unit tests and inspections have been carried out.

Unit tests form a test framework that can be executed in every build so unit tests can also act as regression tests.

3.9 Crystal Methodologies Software Process

Crystal is a family of methods targeted for different kinds of software development projects. The main criteria for selecting one method from the pool of Crystal methods are the size of the developer team and the severity of losses due to possible failures in the software being developed. (Cockburn, 2002a)

The methods that have been constructed and tested in Crystal family are Crystal Clear and Crystal Orange. All the family members are denoted with color that indicates the "heaviness" of the method, i.e. the darker the color the heavier the method (Abrahamsson et al., 2002). Thus Crystal Clear, the method under survey in this paper, is the lightest of these methods.

Common principles for all Crystal methods are that all

projects use incremental development cycles with maximum increment length of four months, but preferably between one to three months. Important issues are communication and co-operation between people. Development practices, tools and work products are not limited to certain ones and methodologies also allow the adoption of practices from other agile methods like XP or Scrum (Cockburn, 2002a).

Crystal Clear is designed for very small projects comprising up to six developers (Abrahamsson et al., 2002). The method may be extended with some effort on communication and testing practices to more demanding projects.

Crystal methodologies contain policy standards that need to be applied in software development. These standards for Crystal Clear are (Cockburn, 2002a):

- (i) Software is delivered incrementally on a regular basis
- (ii) Progress is tracked by milestones based on software deliveries and major decisions rather than written documents
- (iii) Some amount of automated regression testing of application functionality
- (iv) Direct user involvement
- (v) Two user viewings per release
- (vi) Workshops are held for product and methodology tuning at the beginning and in the middle of each increment

Policy standards are mandatory, but may be substituted by equivalent practices from XP, Scrum or ASD (Cockburn, 2002a). "(Cockburn, 2002a)" states that Crystal Clear provides a place to fall, if one tries and gives up on XP, since any part of XP can be substituted with Clear.

3.10 Unit Testing in Crystal Clear

One of the policies for Crystal Clear is that there should be automated regression test cases from application functionality. Also one of the work products that should be delivered by Crystal Clear are test cases (Cockburn et al., 2002). Therefore, testing plays an important role in Crystal Clear method even though Clear is the lightest of the Crystal methodologies.

The main roles of development people in Crystal Clear are the designer-programmer and the senior designer-programmer. The designer-programmer role can be also divided into sub-roles that are, for example, the unit tester, the programmer, the business class designer etc. (Abrahamsson et al., 2002).

Unit testing can be done using TDD as in XP or it may be done in a more traditional way (Cockburn, 2002b).

Unit testing can be done by writing the tests so that they can be driven automatically as regression tests i.e. there does not have to be any interaction with the tester after putting the tests running. It is neither fully optional nor mandatory to use these fully automated regression test suites for unit testing in Crystal Clear (Cockburn, 2002b). However, Cockburn (2002b:54) is of the opinion that a fully automated unit regression test suite is not a critical factor for project success.

It is recommended for Clear development teams to experiment with different types of unit testing practices such as test first design (Cockburn, 2002b). But getting the tests automated is found more desirable than using, for example TDD. Cockburn (2002b:59) suggests that readymade unit testing tools like Junit, CppUnit etc. should be used as part of unit testing.

4. SOFTWARE QUALITY AND UNIT TESTING

Unit testing has an important role in all of the agile methods covered by this paper. Accordingly, it is important to find out what kind of quality issues can be addressed by unit testing.

Unit testing is executed by the developer for a software module using a range of inputs selected randomly from an input domain in order to identify the defects, which cause it to fail (Younessi et al., 2002).

In traditional software processes unit testing of modules is used to find typographical, syntax and logic errors (Andriole, 1986).

Defects that are found from software can be divided to different types. Dromey (1995) and Younessi et al. (2002) present defect type division to reliability type ("R-type"), functional type ("F-type"), usability type ("U-type") and those that detract from maintainability ("M-type"). An example of R-type defect could be division by zero that adversely affects the reliability of the software. In the same way F-type defect contributes to problems in functionality and M-type to problems in maintainability. This kind of defect classification is not the final truth since one defect might have properties so that it actually belongs to multiple defect types.

4.1 Unit Testing Experiments and Effect on Software Defects

Younessi et al. (2002) present a general model of unit testing efficacy, where they have made empirical findings of which of the above mentioned types of defects can be found by unit testing. The findings are that hardly any structural defects, i.e. M-type defects, are uncovered by unit testing. M-type defect might be a poor variable name, missing comment etc. Testing does well in uncovering a moderate percentage of F- and R-type defects. Yet the tests failed to uncover as many defects as could have been excepted by the experimenters. The best percentage of defects found were U-type-defects, i.e. the defects detracting usability of the software component.

There also exists another experiment regarding unit testing by Solheim and Rowland (1993). They used unit testing as one defect reduction practice while focusing mainly on different integration testing practices in artificial software systems, where faults were seeded. They found in their studies that unit testing of components is worthwhile for defect reduction and especially so, if the cost of unit testing is lower than that of conducting other types of tests. The types of defects found by unit testing were not covered by the study.

It can be thought that, since unit tests usually cover only some small code component such as a function in agile methods, and are implemented by the developer of the program that unit tests are not capable of finding logic errors

that a certain developer might have about business logic. These misunderstandings cannot be found from the code. Accordingly and since the developer codes the tests, also functional and system testing is important for the quality of the program.

Fault avoidance and consequently also higher software quality is achieved through appropriate specification, design, implementation and maintenance activities used to prevent faults in the first place (Williams et al., 2003). If these are not done properly, unit testing is naturally not able, as such, to save the situation.

4.2 TDD Testing Experiments

There does not exist much scientific research made of the benefits of unit testing in agile methods. But some research results do exist of TDD used by XP, which, in turn, uses unit testing as an integral part so that the effects of TDD on quality can be used to assess the effects of unit testing on quality. Most of the research done on TDD is not very reliable since students or only small groups of professionals have been used as "guinea-pigs" and, accordingly, the tests have not been done on large scale industrial settings (Williams et al., 2003).

George and Williams (2003) state that software components developed with TDD combined with pair programming passed 18% more black-box test cases than code written with control pairs, who did not use TDD. The control group failed to make any automated tests after implementation even if they were instructed to do so. This result does not actually mean that making unit tests would improve the software quality since the first group used test first type coding, which may aid to make the software design better and less coupled. But one could draw the conclusion that also the unit tests, as such, had a positive impact on the software component quality.

Williams et al. (2003) present a case where IBM programmers changed from ad-hoc unit testing to TDD and they experienced a major decrease of 40% in defect density compared to the ad-hoc unit testing group. This result does not say much about unit testing efficacy since both control groups used unit testing. Only the other group made a regression testable unit testing pool in a more disciplined manner and used test-driven design also. It could be thought that the reason why ad-hoc unit testing did not yield such good results was that the developers did the tests as they used to do and the TDD control group focused more energy on them. But this is just guesswork.

To conclude, the studies related to TDD do not give any trustworthy results that could be used to justify the role of unit testing in agile methods.

5. WHY TO USE UNIT TESTING?

Unit testing is laborious and takes time, but there must be some benefits in unit testing since it is used so widely in modern software processes.

5.1 Possible Benefits in Unit Testing

Unit testing gives developers the confidence about newly developed implementation as the system grows (Beck, 2001; Jeffries, 1999). This confidence may be especially beneficial when building large and complex systems and aids to the overall debug process, since the developer may be able to "trust" sub components of the system better, if the tests have been done well.

Unit testing gives developers continuous feedback, which is a positive factor in implementing unit tests (Williams et al., 2003).

With the use of unit tests in development the developers make the code automatically more testable by making functions to return values etc., which can be beneficial for later maintenance of code. (Williams et al., 2003)

Unit testing is the first stage of testing and usually defect correction is cheaper and easier at this stage and this can be counted as a benefit to using unit testing (Stapleton, 1997).

Tests that are saved for later and kept up to date may also be useful for the maintenance of the software component since tests contain information on how to call the component and how the developer has intended it to be used.

Last but not least, an important benefit as presented in the chapter 4.1 above is that unit testing is able to find defects and make improvements to the quality of software.

5.2 Possible Limitations, Problems and Disadvantages of Unit Testing

Unit testing and keeping all the tests up to date over iterations takes time and resources. Developers who create unit tests also have to be motivated to make the tests. The testing phase should not be left to less attention in situations where the software project starts to be late from its schedule (Elssamadisy & Schalliol, 2002).

Unit tests are thought to be beneficial since they form a regression testing pool that can be beneficial when incorporating changes to the code base. But there are situations where unit tests cannot find anything wrong even though the system is broken. This situation may occur, for example, when changes are made to the functionality of the software and units involved are not prepared for this kind of functionality (Elssamadisy & Schalliol, 2002). Thus, the tests created for the units do not cover this new functionality. The problem could be solved by also incorporating an automated testing pool for functional testing as is usually done for unit testing only.

Unit testing of certain types of components may be laborious and error-prone due to the generation of throw away stubs or drivers needed to test the component as unit testing is supposed to be done by ignoring the rest of the system (Whittaker, 2000).

In some agile methods unit testing has taken such an important that the general role of developers in testing may have become more important than it has been in traditional software development. In traditional unit testing the unit tests may have been more add-hoc style tests supposed to test some operation, where the test would not have been updated as the program evolves. In agile methods and especially in XP the

tests are such an important part of the process that the requirements for developer's knowledge as regards all aspects of testing have increased. This may be thought to be a disadvantage for developers that are not up to the task. But on the other hand, testing can be learned.

6. DISCUSSION

Unit testing in both traditional and agile processes aims to make software quality better by decreasing the amount of defects in the software. Unit testing seems to be a good practice for the reduction of software defects and appears to affect the software quality positively even though there is no wide research conducted on its effects on different types of defects in software.

If unit testing is done in an organized manner and the tests are kept up to date it may aid software developers especially when software requirements change during the process. Changing requirements are also the reason for using agile software process. It can be thought that this is one of the reasons why almost all of the agile methods use unit testing and mention unit testing as part of their software development process.

The role of unit testing has been quite strong also in traditional software processes since it is the most common type of testing used in traditional software development. That is why it cannot be said that the role of unit testing in all of the modern agile methods would have changed significantly from the role unit testing has in traditional software development. For example, in FDD method also inspections are used to find defects in the developed code. Of the agile methods covered by this article, XP seems to put most emphasis on unit testing. But XP also generally seems to have the strictest practices to be obeyed.

It can be thought that all of these agile methods and especially XP have generally improved the status of unit testing so that it is performed nowadays in a more organized and less ad-hoc manner than it traditionally has been performed. Also all kinds of unit testing tools have been developed for software developers in the recent years due to the new popularity that unit testing has gained probably due to its role in agile methods. These tools aid developers in running unit tests as regression tests and make testing more automated.

The definition of "unit" may have also decreased slightly in moving towards agile methods from traditional software process so that testing has become more frequent and, correspondingly, also more organized.

There are similar challenges in unit testing as with general testing and software developers need to adopt new ways to function in order to get the full benefits that unit testing practices might bring along when tests are done in an organized manner.

The writer of this paper also shares some recent personal experience in using unit testing in a software development project. The experiences have thus far been positive even though frequent test writing does not always seem to take things forward and may be arduous at times, especially if the unit being tested uses databases and other external interfaces, which it may need to be isolated from. As described in the paper, my personal trust for the quality of unit tested components and functions has improved and has aided in the debugging of the project.

REFERENCES

Abrahamsson, P., Salo, O., Ronkainen, J. and Warsta, J. (2002), *Agile software developmenent methods: review and analysis*. VTT publications, 478. Otamedia Oy, Espoo, Finland.

Abrahamsson, P., Warsta, J., Siponen, M.T. and Ronkainen, J. (2003). *New Directions on Agile Methods: A Comparative Analysis*. IN: Proceeding of the 25th International Conference on Software Engineering, 3-10 May, Oregon, USA. IEEE Computer Society. pp. 244-254.

Andriole, S.J. (1986). *Software Validation Verification Testing and Documentation*, Petrocelli books, New Jersey, USA.

Beck, K. (1999). Embracing Change with Extreme Programming, *IEEE Computer*, 32 (10) October, pp.70-77.

Beck, K. (2001). Aim, Fire. *IEEE Software*, 18 (5) September/October, pp.87-89.

Cockburn, A. (2002 a). *Agile software development*, Addison-Wesley, Reading, MA, USA.

Cocburn, A. (2002 b). *Crystal Clear: A Human Powered Methodology for Small Teams,* Addison-Wesley, Reading, MA, USA.

Dromey, R.G. (1995). A model for software product quality, *IEEE Transactions on Software Engineering*, 21 (2) February, pp.146–162.

Elssamadisy, A. and Schalliol, G. (2002). *Recognizing and responding to "bad smells" in extreme programming.* IN: Proceedings of the 24rd International Conference on Software Engineering, 19-25 May, Orlando, USA. IEEE Computer Society. pp. 617-622.

Fowler, M., (2003). *The New Methodology* [Internet]. Available from: <http://martinfowler.com/articles/newMethodology.html> [Accessed 14 April, 2004].

George, B. and Williams, L. (2003). *An Initial Investigation of Test-Driven development in Industry*. IN: Proceedings of the ACM symposium on Applied computing, 9-12 March, Florida, USA.

Haikala, I. and Märijärvi, J. (1998). *Ohjelmistotuotanto*, Gummerus, Jyväskylä, Finland.

IEEE Std 1008-1987 (1987), IEEE Standard for software unit testing

Jeffries, R.E. (1999). Extreme testing, *Software Testing and Quality Engineering*, 1 (2) March/April.

Kung, D., Hsia, P. and Gao, J. (1998). *Testing object-oriented software*, IEEE Computer society, California, USA.

Larman, C. and Basili, V.R. (2003). Iterative and incremental development: A brief history, *IEEE Computer*, 36 (6), pp 47-56.

Leung, H.K.N. and Wong, P.W.L. (1997). A study of user acceptance tests, *Software Quality Journal,* 6 (2) January, pp-137-149.

Palmer, S.R. and Felsing J.M. (2001). *A Practical Guide to Feature-Driven Development*, Prentice Hall, New Jersey, USA.

Rose, T. (2004). *Software Testing for Programmers* [Internet]. Available from: <http://www.developer.com/java/other/article.php/2217461> [Accessed 20 April 2004].

Schach, S.R. (1990). *Software Engineering*, Aksen Associates Incorporated Publishers, 499pp.

Solheim, J.A. and Rowland, J.H. (1993). An empirical study of testing and integration strategies using artificial software systems, *IEEE Transactions on Software Engineering,* 19 (10) October, pp. 941-949.

Stapleton, J. (1997). *Dynamic systems development method, The method in practise*, Addison-Wesley, Great Britain.

Stockman, S.G., Todd, A.R. and Robinson, G.A. (1990). A Framework for software quality measurement, *IEEE Journal on Selected Areas in Communication,* 8(2) February, pp. 224-233.

Whittaker, J.A. (2000). What is software testing? And Why Is It So Hard? *IEEE Software*, 17 (1) January/February, pp.70-79.

Williams, L., Maximilien, E.M. and Vouk M. (2003).*Test-driven development as a defect-reduction practice*., IN: Proceedings of the 14th International Symposium on Software Reliability Engineering, 2003, 17-20 November, pp. 34 - 45.

Younessi, H., Zeephongsekul, P. and Bodhisuwan, W. (2002). A General Model of Unit Testing Efficacy, *Software Quality Journal,* 10 (1) July, pp. 69-92.