Can Extreme Programming Be Combined With Software Reliability?

Eila Pohjola

Abstract— **Software reliability is the probability of failure-free operation of a computer program for a specified time in a specified environment. Much research has focused on predicting software reliability at delivery based on the trend of failures encountered during testing. Software reliability engineering combines the use of quantitative reliability objectives and operational profiles.**

Extreme Programming is a light weight software development process. Testing is said by some to be at the heart of XP and there is nothing it cannot achieve, while others find the practice chaotic and uncontrollable.

There is lack of research regarding the quality of the end product produced by using XP processes. Another gap in the general knowledge is the use of Extreme Programming in critical projects. Can it be done?

The promoters of XP claim that reliability comes if you follow the procedure, others find the chaos of XP problematic. Some of these problems have been tried to solve by using XP with extensions.

This paper gives a literature based review on software reliability engineering and whether it is possible to achieve high reliability by using XP.

Index Terms—**Extreme Programming, Software Reliability, Software Reliability Engineering.**

1. INTRODUCTION

Extreme Programming (Beck 1999a; 1999b), or XP, is the most known of the agile software development processes measured by the number of papers and books written on it. XP is an incremental software process that is supposed to suit a fast changing environment. It aims at keeping the system design as simple and adaptable as possible by *refactoring* and always implementing the simplest thing that could work. *Pair programming*, *unit tests*, and the code itself help communicate the system structure and intent. Unit tests and continuous integration of increments give programmers instant feedback about the impact of the changes made.

Reliability (Musa et al. 1990) is probably the most important of the characteristics inherent in the concept "software quality". It is intimately connected with defects and defects represent the largest cost element in programming. Software reliability concerns itself with how well the software functions to meet the requirements of the customer.

One of the most common failings of XP teams is insufficient testing (Jeffries 2001). XP asks for more testing than many teams are used to. But what about projects that need reliability at a substantially higher level?

1.1 Research problem and questions

This paper is concentrating in software reliability and whether it is possible to achieve reliability by using Extreme Programming. The XP approach offers some guidance on quality assurance, but is very vague on achieving software reliability. The aim of this research is to explore how Extreme Programming is claimed to achieve software reliability and could the traditional software reliability methods be combined with it.

In order to find an answer to the research problem, we must first define the term reliability. Thus the first (or perhaps number zero) research question is: How to define software reliability?

As we come closer to the actual topic of this research we can define two main research questions:

What are the software reliability testing methods of traditional software development and Extreme Programming?

Means to answer this question is to describe the traditional software reliability methods. Then I focus on the quality assurance and reliability methods of Extreme Programming. The third research question is:

Is it possible to achieve software reliability by using Extreme testing methods?

The objective of this study is to answer the before mentioned research questions and offer some knowledge on how to achieve software reliability by using Extreme Programming practices.

1.2 Scope and definitions

The terms mentioned in the previous chapter are somewhat vague and offer a vast array of previous studies and literature, so the scope of this study must be restricted in order to meet the course objectives. Some of the key terms in this research are defined as follows:

eXtreme testing refers to testing practices of the Extreme Programming practise, where testing is performed mainly by developers who write unit tests using test-first approach (i.e. write test code before the actual implementation). In Extreme Programming practise also the customer provides functional tests. While the team is implementing, the customer is specifying functional tests, which are to convince the customer that the user stories being implemented in the iteration are ready and functional. (Beck 1999b)

Testing is a process of planning, preparation and measuring aimed at establishing the characteristics of an information

system and demonstrating the difference between the actual and required status (Pol et al. 2002). The term Testing includes here only unit testing and functional system testing.

Software reliability methods in this research are confined to only to those methods used in testing and other methods involved in other stages of the life-cycle are excluded. Also formal mathematical methods like logic proofing are excluded.

1.3 Methods

This paper is a literature review. The main sources of information will be the databases for scientific articles and books on software development in general and specially on agile methods. The material can be obtained from databases of scientific articles, libraries and also the World Wide Web.

The aim is to find relevant material by making thorough searches of databases and the Web with several keywords related to the topic and then narrow the material down to the most relevant.

1.4 Structure and outline of the study

The rest of the paper is structured as follows. Chapter two introduces the concept of software reliability and software reliability engineering. We will also look into some methods used in testing software reliability. Next chapter introduces Extreme Programming and its 12 practices. We will concentrate more into the quality assurance methods of Extreme Programming. Chapter four will attempt to match the XP quality assurance methods to the traditional software reliability methods and to assess the influence of XP methods. Another subject is to evaluate the possibility to use traditional software reliability methods in Extreme Programming. Summary and discussion is in chapter five.

2. SOFTWARE RELIABILITY AND TESTING

Software reliability is often defined as the probability of failure-free operation of a computer program for a specified time in a specified environment. So what is a failure? Software failure is the departure of the external results of program operation from requirements. A fault is the defect in the program that, when executed under particular conditions, causes the failure. (Musa et al. 1990).

There are three principal reliability strategies: fault prevention, fault removal, and fault tolerance. Fault prevention uses requirements, design, and coding technologies and processes, as well as requirements and design reviews, to reduce the number of faults introduced in the first place. Fault removal uses code inspection and development testing to remove faults in the code once it is written. Fault tolerance reduces the number of failures that occur by detecting and countering deviations in program execution that may lead to failures. (Musa 1997)

Fault removal strategy is primary area of this study. The software reliability methods, including testing, that can be used in fault removal are discussed further in chapter 2.2.

It is said (Salasin 1989), though, that software reliability cannot be "tested" in to the product. Thus the reliability requirement must be taken into account during design and development. Testing can only provide measurement on how reliable the software is. Let's assume that reliable software causes a system to behave as we expect it to within the constraints imposed by physical devices. This definition implies that:

- The software as-built is same as software as-intended
- The software does not exhibit failures during operation
- The software is fault-tolerant

Much research has focused on predicting software reliability at delivery based on the trend of failures encountered during testing. The assumption underlying the use of reliability models for such prediction are: (Salasin 1990)

- The statistical distribution of test data matches the statistical distribution of data encountered in operation
- Failure rates decrease with "time on test" since errors are corrected when found
- A "reasonable number of errors" are encountered during testing, since we can't extrapolate failure rates from zero

Work by Musa (Musa 1990) has shown some success in predictions of software reliability based on test and operational experiences.

2.1 The concept of software reliability engineering

Software testing often results in delays to market and high cost without assuring product reliability. Software reliability engineering can be applied to carefully engineer testing to overcome these weaknesses. Software reliability engineering combines the use of quantitative reliability objectives and operational profiles (profiles of system use). The operational profiles guide developers in testing more realistically, which makes it possible to track the reliability actually being achieved. (Musa 1997)

There are two types of testing using software reliability engineering: development testing, in which you find and remove faults, and certification testing, in which you either accept or reject the software. Development testing precedes certification testing, which in turn servers as rehearsal for acceptance testing. (Musa 1997)

During development testing, you estimate and track failure intensity, which is the failures per unit execution time. Failure intensity is an alternative way of expressing software reliability. Testers use failure intensity information to determine any corrective actions that might need to be taken and to guide release. Development testing typically comprises feature, load, and regression testing. It is generally used for software developed in your own organization. (Musa 1997)

Certification testing does not involve debugging. There is no attempt to resolve failures you identify. Certification testing typically comprises only load testing. Certification testing is typically used on software acquired from outside of the own organization. (Musa 1997)

Musa introduces in his article (Musa 1997) the application steps of software reliability engineering.

- Determine which associated systems require separate

testing
- Decide which type(s) of testing is needed for each system to be tested
- Define necessary reliability in terms of severity classes and setting failure intensity objectives for the software
- Develop operational profiles (set of operations and their probability of occurrence)
- Prepare for testing
- Execute tests
- Interpret failure data

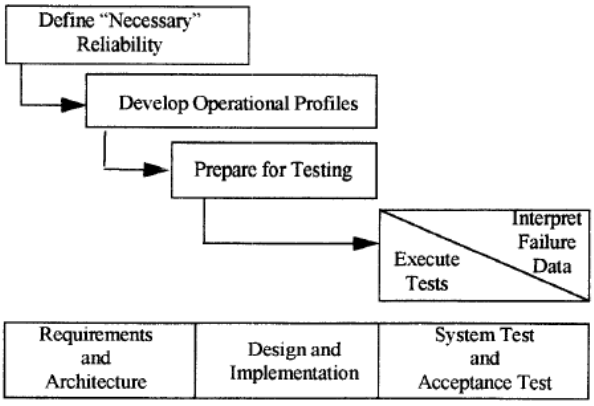

Figure 1. The core application steps of applying software reliability engineering to testing and corresponding development life-cycle stages.

Figure 1 shows how the application steps of software reliability engineering relate to corresponding development life-cycle stages. It does not take a stand on which life-cycle model is used, only to the stages where this method should be applied.

2.2 Software reliability methods

Previous chapter discussed the concept of software reliability on higher level. This chapter will introduce some methods used in software reliability testing.

Formal methods are collection of notations and techniques for describing and analyzing systems. Formal analysis techniques can be used to verify that a system satisfies its specifications. Software testing is perhaps the most frequently used quality assurance method. Instead of trying to provide a comprehensive check of the system, testing is focused on sampling the executions, according to some coverage criteria, and comparing the actual behavior with the behavior that is expected according to specification. Testing does not guarantee to find all errors or even some. (Peled 2001)

Testing can be divided to White box testing and Black box testing. White box testing is more often used in unit testing and in other "lower" levels of testing. Black box testing in other hand is used more frequently in system and acceptance testing. (Peled 2001; Pressman 1987)

The following two chapters present the testing methods used for achieving software reliability (Peled 2001). Methods presented here are traditional software testing methods, in this context they are often called software reliability testing methods. These methods are used to achieve software reliability in a sense that all these testing methods fall under the fault removal strategy of the three principal software reliability strategies.

2.2.1 White box testing

White box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Using white box testing methods, the software engineer can derive test cases that 1) guarantee that all independent paths within a module have been exercised at least once, 2) exercise all logical decisions on their true and false sides, 3) execute all loops at their boundaries and within operational bounds, and 4) exercise internal data structures to assure their validity. (Pressman 1987)

I will start by presenting briefly the various white box testing techniques introduced by Peled (Peled 2001).

Dataflow analysis is often used within compilers to perform static analysis of the program.

Inspections and walkthroughs are manual testing methods, carried out by a small team of people during a meeting, which typically lasts one or three hours. Inspections and walkthroughs can besides code check also documents provided by the project.

In *unit testing*, a test case usually corresponds to the selection of an execution path. During testing, one can seldom check all the executions of a system in a comprehensive way. Thus testing is often done based on coverage criteria. Coverage criteria allow collecting sets of executions that are likely to present the same errors. The control flow coverage criteria include:

- Statement coverage: Each executable statement of the program appears in at least one test case.
- Edge coverage: Each executable edge of the flowchart appears in some test case.
- Condition coverage: Each Boolean combination that may appear in any decision predicate during some execution of the program must appear in some test case.
- Path coverage: Every executable path is covered by a test case.

Testing selfdom guarantees that all or even some of the design and programming errors will be found. One way of measuring the quality of a test suite is by performing *code coverage analysis*.

It is common practice in specification and verification to *partition the executions into sets* of executions that should not be distinguished from each other. Then, instead of taking care of all the executions, we take care of at least on sequence from each such set.

Large software is usually developed by different teams, each responsible for a part of the code. The same principle can be applied to software testing. Such a *compositional approach* has the additional advantage of better management. Tester is allowed to concentrate on only a part of the features. Another advantage is that finding an error in a small part of the code usually pinpoints the source of the error more accurately.

2.2.2 Black box testing

Black box testing methods focus on the functional requirements of the software. That, is black box testing

enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements of the program. Black box testing is not an alternative to white box techniques. It is a complementary approach that is likely to uncover a different class of errors than white box methods. (Pressman 1987)

Black box testing checks a system without considering its internal structure. Testing is often based on modeling the system as graph or an automaton, and using graph algorithms to construct the test suite. (Peled 2001)

3. Extreme Programming

The first know lifecycle model was the waterfall model (Pressman 1987; Scach 2002). The waterfall model didn't just appear (Beck 1999a). It was a rational reaction to the shocking measurement that the cost of changing a piece of software rose dramatically over time. Engineers wanted to make the biggest most far-reaching decisions at earliest possible stage. This did not seem to work correctly, and the next attempt to solve the problem of ever changing requirements was the iterative life-cycle model (Scach 2002).

Extreme Programming is a somewhat controversial new approach to software development based on the incremental model (Scach 2002). XP is said (Wake 2001) to be a programming discipline. Extreme Programming is a disciplined approach to software development that emphasizes customer satisfaction and teamwork (Beck 1999a; 1999b). Another definition is that it is a software development process designed for small to mid-size projects, has strong customer involvement, a simplified requirements gathering and prioritization practice, and an emphasis on testing (Beck 1999b; Williams et al. 2002).

3.1 Introduction to Extreme Programming practices

XP turns the conventional software process sideways (Beck 1999a). Rather than planning, analyzing, and designing for the far-flung future, XP exploits the reduction in the cost of changing software to do all of these activities a little at a time, throughout software development.

Figure 2 shows XP at timescales ranging from years to days. The customer picks the next release by choosing the most valuable features (called *stories* in XP) from among all the possible stories, as informed by the costs of the stories and the measured speed of the team in implementing stories.

The customer picks the next iteration's stories by choosing the most valuable stories remaining in the release, again informed by the costs of the stories and the team's speed. The programmers turn the stories into smaller-grained tasks, which they individually accept responsibility for. (Beck 1999a)

Then the programmer turns a task into a set of test cases that will demonstrate that the task is finished. Working with a partner, the programmer makes the test cases run, evolving the design in the meantime to maintain the simplest possible design for the system as a whole. (Beck 1999a)

The individual practices in XP are not by any means new. Many people have come to similar conclusions about the best way to deliver software in environment where requirements change violently (Newkirk 2002). XP has 12 basic practices (Beck 1999a; Beck 1999b; Newkirk 2002; Wake 2001). None

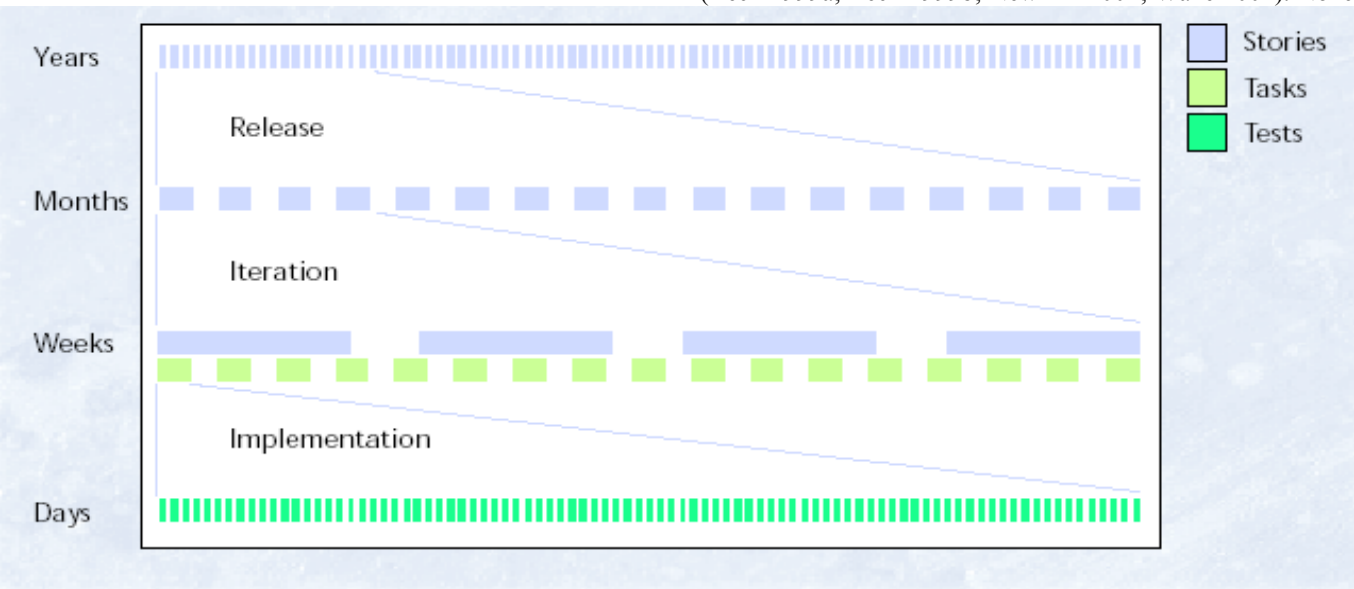

Figure 2. XP according to various timescales. At the scale of months and years, you have the stories in this release and then the stories in future releases. At the scale of weeks and months, you have stories in this iteration and then the stories remaining in this release. At the scale of days and weeks, you have the task you are working on now and then the rest of the tasks in the iteration. And at the scale of minutes and days, you have the test case you are working on now and then the rest of the test cases that you can imagine.

of the practices are unique or original. They all have been used for a long time (Beck 1999b).

Planning game. Customers decide the scope and timing of releases based on estimates provided by programmers. Programmers implement only the functionality demanded by the stories in this iteration.

Small releases. The system is put into production in a few months, before solving the whole problem. New releases are made often—anywhere from daily to monthly.

Metaphor. The shape of the system is defined by a metaphor or set of metaphors shared between the customer and programmers.

Simple design. At every moment, the design runs all the tests, communicates everything the programmers want to communicate, contains no duplicate code, and has the fewest possible classes and methods. This rule can be summarized as, "Say everything once and only once."

Tests. Programmers write unit tests minute by minute. These tests are collected and they must all run correctly. Customers write functional tests for the stories in iteration. These tests should also all run, although practically speaking, sometimes a business decision must be made comparing the cost of shipping a known defect and the cost of delay.

Refactoring. The design of the system is evolved through transformations of the existing design that keep all the tests running. Refactoring is the process of improving the design of code without affecting its external behaviour.

Pair programming. All production code is written by two people at one screen/keyboard/mouse.

Continuous integration. New code is integrated with the current system after no more than a few hours. When integrating, the system is built from scratch and all tests must pass or the changes are discarded.

Collective ownership. Every programmer improves any code anywhere in the system at any time if they see the opportunity.

On-site customer. A customer sits with the team full-time.

40-hour weeks. No one can work a second consecutive week of overtime. Even isolated overtime used too frequently is a sign of deeper problems that must be addressed.

Open workspace. The team works in a large room with small cubicles around the periphery. Pair programmers work on computers set up in the centre.

The thirteenth practice of XP is the fact that rules are just rules. By being part of an Extreme team, you sign up to follow the rules. But they're just the rules. The team can change the rules at any time as long as they agree on how they will assess the effects of the change. (Beck 1999b)

3.2 Quality assurance methods in Extreme Programming

In Extreme Programming the test-first method is being used in unit testing (Wake 2001; Beck, 1999b). Unit testing is claimed to be at the heart of XP (Beck 1999a; Beck 1999b). In XP unit testing is part of every programmer's daily business. There are, however, two twists: Programmers write their own tests and they write these tests before they code. It is claimed (Beck 1999a; Beck 1999b) that XP primarily addresses the accepted wisdom that programmers can't possibly test their own code by having you write code in pairs.

The promoters of XP claim that XP testing strategy doesn't ask any more work than the usual bench testing strategies. It just changes the form of the tests. Instead of activities that evaporate into the ether as soon as they are finished, you record the tests in a permanent form. These tests will run automatically today, and this afternoon after integration and tomorrow. (Beck 1999a; Beck 1999b; Wake 2001)

The Test/Code cycle of XP goes as follows (Wake 2001):

- Write one test
- Compile the test. It should fail to compile, because the code that the test calls has not been implemented yet
- Implement just enough to compile
- Runt the test and see it fail
- Implement just enough to make the test pass
- Refactor for clarity and remove duplication
- Repeat from the top

Tests also come from the customers. At the beginning of iteration, the customers think about what would convince them that the stories for iteration are completed. These thoughts are converted into system wide tests, either directly by the customer using a textual or graphical scripting language or by the programmers using their own testing tools. These tests, too, accumulate confidence, but in this case they accumulate the customer's confidence of the correct operation of the system. (Beck 1999b; Wake 2001)

The XP process has some practices for testing but offers only rough guidance in practice. It has been said by the promoters of XP (Wake 2001; Beck 1999b) that the testing is a discipline unto itself.

Beck (Beck 1999b) introduces some forms of testing that the XP team might need when they are in trouble. These are:

- Parallel test – a test designed to prove that a new system works exactly like the old system
- Stress test - a test design to simulate the worst possible load
- Monkey test – a test designed to make sure the system acts sensibly in the face of nonsensical input

Pair programming is also considered (Beck 1999b) a quality assurance method by XP practioners. When pair programming one is continually inspecting the code. It has been claimed (Beck 1999b; Wake 2001) that the code resulting from pair programming is of better quality.

Refactoring is another quality assurance method in XP (Wake 2001). It is the process of improving the design of code without affecting its external behaviour. Refactoring is done, so that code would be as simple as possible, ready for any changes that come along and at the same time testable.

These two XP practices, even though quality assurance methods in XP, are not part of the testing methods. Thus this paper will not concentrate on them.

4. SOFTWARE RELIABILITY IN EXTREME PROGRAMMING

Some anecdotal evidence argues success of the Extreme Programming in producing higher quality in less time. Although precise information about benefits and costs of the Extreme Programming practice represents a critical guideline for improvement of software quality, there has been little

work on the subject beyond subjective reports and a study in academic environment. (Succi et al. 2001)

IT managers often view XP as a slightly chaotic methodology. Many even regard XP as dangerous and unpredictable because to them it appears to neglect planning and controlling in large-scale or long-term projects. (Cockburn 2002)

4.1 Extreme Programming vs. traditional methods

As mentioned before there are very few studies about Extreme Programming. Most of them have been made in an academic environment, which even at its best can not describe reality accurately. One of these studies compared Extreme Programming and traditional software development. The paper (Macias et al. 2003) describes an experiment carried out during the Spring/2002 academic semester with computer science students at the University of Sheffield. The study is set in academic environment and thus is not fully applicable in "the real world" but it does give a general picture on how things could be.

The objective of the experiment (Macias et al. 2003) was to assess Extreme Programming. With this purpose, it was compared with a traditional approach which played the role of a control treatment. The observable practices followed by the teams in Extreme Programming treatment were: planning game, testing, pair programming, simple design, coding standards, collective ownership, continuous integration, small releases, and some cases of metaphors and refactoring. They did not follow "40 hours week" nor "on site customer". The teams followed an additional practice: testing based on requirements.

Results (Macias et al. 2003) supported the fact that Extreme Programming teams produced as good results as the traditional approach. The implications of this result are very important. The most relevant one for the Software Engineering community is that a procedure free of design stage provides as good results as one including design stage. The lack of design resulted from applying Extreme Programming.

According to the study (Macias et al. 2003) internal quality and external quality are unrelated. The behavior of the internal quality factors was not related to the behavior of the external factors. This means that some systems could present good user characteristics and poor internal construction, or good internal construction and poor presentation for the user, or any other combination. But there was not any pattern, according to the data from the correlation coefficient.

Study by Paulk reviews XP in the light of CMM (Capability Maturity Model). XP advocates many good engineering practices, although some practices may be controversial and counter-productive outside a narrow domain. Paulk suggests that the ideas in XP should be carefully considered for adoption where appropriate in an organization's business environment since XP can be used to address many of the CMM Level 2 and 3 practices. In turn, organizations using XP should carefully consider the management and infrastructure issues described in the CMM. (Paulk 2001)

The risk in changing to XP is that the emergent properties providing value in its proper context may not emerge. Still, the emphasis in choosing and improving software processes should be to let common sense prevail - and to use data whenever possible to provide insight when answering challenging questions. (Paulk 2001)

4.2 Why XP is claimed to be reliable?

One of the most widespread criticisms of Agile methods in general is that they do not work for systems that have criticality, reliability and safety requirements. Paper by Lindvall et al. reports a disagreement amongst the developers about suitability of agile methods for these types of projects. Some developers feel that Agile Methods work if performance requirements are made explicit early, and if proper levels of testing can be planned for. Others argue that Agile best fits applications that can be built "bare bones" very quickly, especially applications that spend most of their lifetime in maintenance. (Lindvall et al. 2001)

What can XP offer for the projects that need high reliability? According to Jeffries (Jeffries 2001), one of the promoters of Extreme Programming, XP projects typically report higher reliability than the same teams had attained before doing XP. Jeffries claims that XP provides very good reliability because of the following reasons:

Unit Tests, ideally written before the code that is tested, cover "everything that could possibly break".

Acceptance Tests, independently defined by the customer, test all the requirements.

Whenever defects slip through the unit tests, to be detected by the acceptance tests, it is recommended that the programmers upgrade the unit tests, not only to show the existing defect, but to upgrade the testing practices in general based on what was learned about the "missing" tests.

Whenever defects slip through the acceptance tests and are caught by users, the same practice is used to upgrade both acceptance tests and unit tests, and the testing practices.

All production code is programmed by two programmers working together. This provides *one hundred percent inspection* by at least one other person.

In XP, code is owned by the team, not by individuals. This means that over the course of the project, essentially all the code is viewed and edited by even more programmers than the original pair who wrote it. This provides even higher levels of inspection.

XP teams release software to users very frequently, ideally every couple of weeks. This ensures that the software gets plenty of assessment in the real working environment. This enables the team to build an excellent sense of system quality.

Although this argumentation is pretty straightforward, Jeffries offers no proof in his article, that this is really the case. Arguments presented above could serve as hypothesis, for a more through study on the subject.

Williams et al. offer another point on the reliability. With TDD (test-driven development), software engineers write low-level, automated unit tests every time they create a new class/method, before they write the code. As a result, methods are "testable" (e.g. in the simplest case, at least have return values). Development cannot proceed until all the unit test cases for the new user story pass and all the unit test cases for the entire existing code base pass. (Williams et al. 2002)

"Test then code" is the phrase used to express XP's emphasis on testing. It captures the principle that testing should be planned early and test cases developed in parallel with requirements analysis, although the traditional emphasis is on black-box testing. Thinking about testing early in the life cycle is a well-known good software engineering practice, even if too infrequently practiced. (Paulk 2001)

The test-driven development practice of XP, is the key to working with critical projects. Because all of the tests have to be passed before release, projects developed with XP can adhere to strict (or safety) requirements. Customers can write acceptance tests that measure non-functional requirements, but they are more difficult and may require more sophisticated environments than unit tests. (Lindvall et al. 2002)

Usually the proofs of XP's reliability, like the ones mentioned above, concentrate on the methods and how by using them you can produce good quality software. But you can not trust the methodology on its own, when it is not used. Jeffries argues, that a very common failing in XP projects is insufficient testing, especially insufficient acceptance testing (Jeffries 2001).

4.3 XP with extensions

It is allowed and even desirable to change the XP process to fit individual needs (Beck 1999b). There have been (Lippert et al. 2003) reported experiences of retrofitting XP. According to a study by Lippert et al. when suitably adapted for use in projects with complex domains or limited resources, it has been found that XP offers a high degree of security and reliability without limiting the advantages of agile software development.

Some methodological extensions have been developed to XP for use a number of areas in which questions and problems frequently occur. Lippert et al. apply these extensions in cyclic and iterative approach that emphasizes constant feedback and project preparation. According to the study, unlike system presentations, using early system versions helps to address questions relating to features such as stability, load behaviour, and performance. (Lippert et al. 2003)

Problems can occur if the customer does not follow through on promises to extensively test the prototypes and subsequent versions. Approach emphasizes the importance of holding a frank conversation at the beginning of the project that explains what is expected of the customer, particularly with respect to regular involvement in testing. This conversation should make clear how the customer can integrate this effort into a normal work routine and that the testing will yield tangible benefits. Responding to objections about testing provides an opportunity to initiate an early search for solutions that foster the XP goal of encouraging the customer's close involvement in the development process. (Lippert et al. 2003)

Even if the customer does carry out testing, the results from manual acceptance tests can be interpreted in different ways. Developers must be able to assess the quality of the customer testing, but doing so is a difficult undertaking, especially when the feedback is positive. (Lippert et al. 2003)

One solution is to both document the test setup and results and to evaluate them in a feedback cycle. Feedback can point to remaining weaknesses in the tests, providing a basis for more meaningful tests. While endorsed test results offer more protection against unwarranted additional requirements, the feedback cycle is beneficial because high-quality acceptance tests avoid the need for such requirements. (Lippert et al. 2003)

The study by Lippert et al. reports experiences from several projects using extensions. These extensions have been developed to solve a particular problem the project is facing. Developers' experiences from these projects have been good, but there is not quantified evidence on how well these extensions did. In generally there seems to a lack of evidence in the literature on how well the XP practice could answer to the need of software reliability.

Williams et al. address this problem in their study. According to them basic characteristics of XP enable an extension of XP to encompass a measure of reliability. They examine the enhancement of XP practices to include explicit estimation of the probability that the software system performs according to its requirements based on aspecified usage profile. (Williams et al. 2002)

They (Williams et al. 2002) propose the composition of XP acceptance testing and Software Reliability Engineering (SRE) in order to obtain quantifiable measures of reliability.

Operational profiles are at the heart of SRE (Musa et al. 1997). Two additional requirements for creating an operational profile are required of the customers (Williams et al. 2002). These added steps are necessary for estimation of the reliability range for a system developed using XP.

In the first step the customer must quantify the fraction of usage of each of the "m" user stories, us_j, $0< j \leq m$. User stories are numbered (j) between zero and m, which is the total amount of user stories. The usage of a user story j (us_j), can be anything between 0-100%. It is assumed that the attempted story coverage is 100% from the user perspective.

In the second step customer specifies acceptance test cases. For each user story j, the customer specifies n_j acceptance test cases that cover this story, n being the number of test cases. Let the fraction of the story that is covered by test-case be at_{ij}, $0 < i < n_j$. Here i is the test case between zero and n_j, which is the total number of test cases for this user story. However, the coverage for all acceptance test cases for a story may not total 100%. Part of it may be due the economics of software testing.

Williams et al. are developing an open source "Good Enough" Reliability Tool (GERT) to support the composition of XP acceptance testing and software reliability engineering. GERT estimates the reliability range for the system. Currently, the tool plugs in with JUnit. An upper bound on reliability is estimated using a Nelson-style model:

$$\hat{R} = \sum_{j=1}^{m} \sum_{i=1}^{n_j} us_j at_{ij} x_{ij}$$

where the acceptance test execution score is x_{ij} . If Acceptance Test Case i of User Story j passes, the score is 1, otherwise it is 0. Proper lower bound is still under investigation. One option is to weight successful test cases proportionally to their number in a particular coverage category, and inversely proportion to the coverage they are intended to offer.

The group (Williams et al. 2002) is working on a co-

requisite confidence interval model. This model will indicate the upper and lower bounds of the reliability estimate; a prime determinant of the confidence interval model will be the number of test cases written, particularly for critical, high usage user stories. According to the paper, other issues, such as coverage correlation and sampling issues also need to be taken into account.

There are some limitations to the model described. XP culture, needing agility, operates under resource constrained conditions using as little as one test case for each space an acceptance scenario covers. That alone creates a reliability over-estimation problem unless it is accounted for. (Williams et al. 2002) Additionally:

- The model is highly reliant on good input from a customer regarding the operational profile, and acceptance test cases.
- In XP, the dependencies and overlap between user stories is not identified.
- It is very important that customers specify as many acceptance test cases as possible or reliability will be overstated.

5. SUMMARY AND DISCUSSION

The literature and studies available regarding software reliability is vast. This field of study has been part of the software development scene for long time and the sources and their recommendations could be considered as trustworthy. It is harder to find unbiased information on Extreme Programming and especially on the subject of quantifying the quality of products produced by using XP. One should always read the studies about quality of XP with some skepticism as the studies quite often lack rigorous research methods and are no more than accounts of success stories.

Chapter two discussed the concept of software reliability and testing. Three principal reliability strategies could be identified: fault prevention, fault removal and fault tolerance. The fault removal strategy was the main target of interest in this study. The traditional way to execute this strategy is by testing, thus the methods used for software reliability testing were introduced. Another issue introduced regarding the reliability of software was software reliability engineering (SRE). SRE can be applied to engineer testing to overcome its weaknesses, delays to market and high cost without assuring product reliability.

Following chapter introduced the concept of Extreme Programming and the 12 practices used in it. Special attention was given to the testing practices, unit testing and functional tests provided by the customer. Even though XP has a practice for testing it offers only rough guidance in how practice it. It has even been said by the promoters of XP that the testing is a discipline unto itself.

Chapter four discussed software reliability in Extreme Programming. There has been a lot of talk about the quality of software that XP practices produce. Some state out their concern regarding the lack of control, others promote it saying that quality comes if you follow the procedure.

There has been very little or none useful studies outside the academic environment regarding XP and especially reliability in XP. Clearly more research in this area is needed. One of the studies (Paulk 2001) reviewed XP in the light of Capability Maturity Model and came to the conclusion that both of these practices could benefit from another. One could conclude that on its own XP is not adequate to solve the problems of software reliability. Thus we come to the subject of XP with extensions.

Some extensions of XP introduce the concept of software reliability engineering to it, some try to solve the quality problems that the XP projects sometimes face problem by problem. Once again there is very little research reports regarding the outcomes of these extensions.

Introducing the software reliability methods covered in chapter two to Extreme Programming would be interesting. Currently there is very little guidance on this subject from the XP point of view. At present the responsibility of producing tests belongs to developers and customers. Methods presented require through knowledge of testing practices, equivalence portioning, coverage criteria etc. This double role of customer-tester expert or developer-tester expert will most likely be tough to combine.

REFERENCES

Beck, K. 1999a, “Embracing Change With Extreme Programming”, *IEEE Computer,* Volume: 32 Issue: 10, Oct. 1999, pp. 70-77.

Beck, K. 1999b, “Extreme Programming Explained, Embrace Change”, Addison-Wesley.

Cockburn, A. 2002, “Agile Software Development”, Addison-Wesley.

Jeffries, R. 2001, “XP and Reliability”, http://www.xprogramming.com/xpmag/Reliability.htm, [Modified 8.10.2001], [Referenced 7.3.2004].

Lindvall, M., Basili v., Boehm B., Costa p., Dangle K., Shull F., Tesoriero R., Williams L., Zelkowitz, M. 2002, “Empirical Findings in Agile Methods”, Proceedings of Extreme Programming and Agile Methods – XP/Agile Universe, pp. 197-207.

Lippert, M., Becker.Pechau, P., Breitling, H., Koch, J., Kornstädt, A., Roock, S., Schmolitzky, A., Wolf, H., Züllighoven, H. 2003, “Developing Complex Projects Using XP With Extensions”, Computer, Volume 36 Issue 6, pp 67-73.

Macias, F., Holcombe, M., Gheorghe, M. 2003, “A Formal Experiment Comparing Extreme Programming with Traditional Software Construction”, Proceedings of the fourth Mexican International Conference on Computer Science, pp 73-80.

Musa, J. 1997, “Introduction to Software Reliability Engineering and Testing”, Proceedings of the Eight International Symposium on Software Reliability Engineering, pp 334-337.

Musa, J., Iannino, A., Okumoto, K. 1990, “Software Reliability: Measurement, Prediction, Application”, McGraw-Hill.

Newkirk, J. 2002, “Introduction to Agile Processes and Extreme Programming”, Proceedings of the 24th International Conference on Software Engineering, pp 695-696.

Paulk, M. 2001, “Extreme Programming from a CMM Perspective”, IEEE Software, Volume 18 No 6, pp. 19-26.

Peled, D. 2001, “Software Reliability Methods”, Springer.

Pol, M., Teunissen, R., van Veenendaal, E. 2002, “Software Testing - A guide to the TMAP Approach”, Addison-Wesley.

Pressman, R. 1987, “Software Engineering, A Practitioner’s Approach”, McGraw-Hill

Salasin, J. 1989, “Building Reliable Systems: Software Testing and Analysis”, Proceedings of the 13th Annual International Computer Software and Applications Conference, pp 517-520.

Schach, S. 2002, “Object-Oriented and Classical Software Engineering”, McGraw-Hill.

Succi, G., Stefanovic, M., Pedrycz, W. 2001, “Quantitative Assessment of Extreme Programming Practices”, Canadian Conference on Electrical and Computer Engineering, Volume 1, pp 78-80.

Wake, W. 2001, “Extreme Programming Explored”, Addison-Wesley.

Williams, L. Wang, L., Vouk, M. 2002, “Good Enough” Reliability for Extreme Programming, Fast Abstract at the International Symposium on Software Reliability Engineering.