

Defect prevention and detection practices in XP

Päivi Savola

Abstract—This paper discusses the subject of defect prevention and detection practices in XP concentrating especially to Pair Programming and Test Driven Development. Evidence about XP practices' effectiveness is found to be inconclusive, although it points to the direction of slightly better quality with approximately the same developer effort. Parallels to reviews and unit testing after programming are drawn, and some possible improvement viewpoints suggested.

Index Terms—Software quality, Defect prevention, Defect detection, Extreme Programming

1. INTRODUCTION

Today there is virtually no branch of business that does not rely on software. Software is becoming ever more complex and ever more business-critical, and the ability to produce high quality software will be of an ever-increasing importance. Additionally, in the fast-paced corporate world the requirements placed on software can change quickly and the market windows are ever narrower.

For several decades much attention has been paid in improving software quality, and different means to prevent, detect and reduce defects in the code. One of the primary subjects of attention has been the development process, and development practices. It has been assumed that a more developed, “mature” process produces better software in a more predictable way. Different problems have each been addressed with methodological solutions.

Since the 1990's there has been an increased interest in so-called agile software development methodologies, which attempt to solve the problem of changing and emergent software requirements by adapting to change instead of spending large amount of resources in trying to predict and control it.

However, currently the agile software methodology movement is largely developer driven, and many of the methodologies address the problem of software quality assurance only indirectly, if at all. Furthermore, there is still very little research available about the efficiency and capability of agile development methodologies.

One of the most famous of agile methodologies is Extreme

Programming (XP). Despite being very people and communication-oriented, XP demands a perhaps surprising amount of rigor, and can be considered one of the more quality oriented agile software development methodologies.

1.1 Research problem and scope

This paper aims to study the effectiveness of the Extreme programming (XP) defect prevention and detection practices, concentrating specifically in Pair Programming (PP) and Test Driven Development (TDD).

We attempt to determine whether the quality claims made by the XP proponents are indeed supported with evidence, and if so, what degree of the benefits can be achieved.

The paper will be presented as a literature study from the available sources. Primarily we look for information about the quality effects of Pair programming and Test Driven Development compared to environments which do not use these practices. To shed more light into the differences we also compare Pair Programming with reviews and Test Driven Development with unit testing after development.

Because of the huge field covered by defect prevention and detection, there has been a need to limit the scope of this study. The XP practices to be studied are limited to Pair Programming and Test Driven Development only, whose effect on software quality is perhaps the most direct and therefore easiest to study.

Additionally this paper attempts to discuss only programming defects and verification of functionality and quality attributes. Defect detection and prevention procedures having to do with validation – the suitability of software for its intended use – is quite different from the defects caused by design and programming errors, and therefore left outside the scope of this paper.

Finally, discussion about tools is also avoided, because the field is simply too large to do it any justice in this paper.

1.2 Terminology

For the purposes of this paper it is necessary to define some terms.

Quality: In this paper, quality is understood to be primarily freedom from defects, although in general usage the term is understood to mean much more.

Defect: A fault in the software. A defect may make software behave in an undesirable way when executed. In this paper the word defect specifically refers to the results of programming errors, although in general usage the term is often used to refer to other undesirable behavior as well.

Defect prevention: Activities that aim to lower the amount of defects introduced in the code during programming. By definition these activities must precede or be simultaneous to programming.

Defect detection: Activities that aim to discover defects already in the software. Examples are reviews and unit testing.

Process: An agreed on way of working.

Process-driven or plan-driven: This term is used in this paper to refer to the “traditional” software development methodologies, whose basic approach to problems is to try to define and document a process to prevent the problem from happening.

Agile: A software development methodology can be said to be agile, if it welcomes change instead of trying to prevent it.

1.3 Structure of the paper

The background chapter discusses the ideology behind the agile software development methodologies, introduces XP as a methodology and glances briefly through the most important traditional defect prevention and detection practices.

Next we discuss the two most important defect detection and prevention practices in XP, pair programming and test driven development, and consequently compare them to two similar techniques, reviews and unit testing after the code has been written.

Finally we summarize the results, and suggest directions of further research.

2. BACKGROUND

In this chapter the ideology of agile software development methodologies is discussed, and the basic principles of Extreme Programming are introduced. We finish by presenting a brief overview of the defect prevention and detection

practices commonly recommended in process-driven environments.

2.1 Agile software development methodologies

Software is inherently complex and abstract, and developing software is as much – if not more – a creative process rather than production process. Because of its immaterial nature, software is inherently difficult to communicate about. Consequently software development is difficult to control and manage. The goals and objectives of a project are often hazy to everyone involved, and progress towards them difficult to measure.

Traditional software development methodologies emphasize the need for a process – an agreed way to do things – and planning as the keys for repeatable and consistent achievement of results in the software development field. Their approach is to “organize chaos” by dividing the problem into ever-smaller pieces and developing methods to cope with each subfacet of the problem.

Unfortunately, the traditional methods have so far met with only limited success. One of the greatest difficulties faced by software development is the problem of ever-changing requirements. The requirements change, because of changing environments, users, and the simple fact that a customer is often not able to articulate a solution to a problem they face until they have seen it. Iterative development has offered a partial solution to this problem by allowing the user quicker partial return for money and system developers faster feedback about what is and is not working.

Agile development methodologies break the traditional model. Instead of trying to predict, control and manage change they accept it as inevitable and try to adapt to it by emphasizing the importance of people and communication over processes and extensive planning and documentation. (Fowler, 2003) The agile methodologies propose that the effort used for initial design and the inevitable rework resulting from requirements changes can be avoided with a more modest approach that develops the system piece by piece and redesigns whenever necessary.

Agile development methodologies do not claim to be a “silver bullet” for the software industry. (Boehm and Turner, 2003) have analyzed the home grounds for agile and plan-driven development methodologies. Agile methodologies suit best projects where team size is small, and the team chafes with strict rules and processes. Additional important indicators are that the probability of changes in requirements is high and the criticality of the system low. Finally, agile development projects generally require more competent personnel than

traditional, plan-driven projects, since every member of the team is expected to be able to understand system architecture, and most should be able to do system design.

(Müller and Padberg, 2003) have developed a project model for evaluating the business advantages of using Extreme Programming methodology in a given situation. The cost of applying XP practices can be balanced by faster time to market and reduced quality assurance effort in the later stages of the project. In general they argue, that using XP is indicated if market pressure is high, and the development workforce allows for the use of near maximum number of pairs or the project programmers are much faster when working in pairs than alone.

2.2 Extreme Programming

Extreme Programming is perhaps the most famous of all the agile methodologies. It originally springs from the Smalltalk community and particularly the work of Kent Beck and Ward Cunningham in the late 1980's. (Fowler, 2003)

In XP, the code is viewed as the design, and working code is used define system development progress.

XP is both very informal and structured at the same time. It has a few key values and builds on a dozen key practices, which support and enhance each other. Adhering to the spirit of the XP values and most of the key practices is what makes an agile development project recognizable as an XP project.

XP is designed for fairly small teams of under 10 developers. The main reason for this is the lack of documentation – XP relies very heavily in interpersonal communication.

According to (Beck, 1999) the major XP practices are the following:

- Planning game, where customers and engineers agree on what will be included in the next release. The basic idea behind the game is that customers choose what the engineers will do, but only the engineers may determine how long it will take.
- Small releases, which means that new functionality will frequently be delivered and put into production before the whole system is ready. The purpose of this practice is to facilitate faster feedback about the system development direction.
- Metaphor, a common way of communicating about the system key idea to facilitate communication between customers and engineers.
- Simple Design means that at every moment of development the code should be working, and as simple and elegant as possible, with no duplicate code. Specifically, planning forward for complexity that may never come is forbidden, as are “kludge” fixes.
- Tests and specifically automated tests are a crucial part of XP. Programmers should write automated (and so, by definition, repeatable) unit tests for everything they code. In addition the tests must be written before the actual code that will make them run – no code should be written without a test it will satisfy. This practice is called Test Driven Development. Customers should write functional tests for the features they have chosen for this iteration. All these tests form an important part of the development code base, and must be maintained so that the system can be shown to be working by running all the tests at any time.
- Refactoring, thorough which the developers evolve the system design incrementally from the system features already implemented, not with forward planning at the beginning of development. Specifically, the system should be refactored whenever Simple Design (above) requires it.
- Pair Programming: All production code must be written in pairs – two people on front of a single screen, keyboard and mouse, thus subjecting all code to a continuous peer review.
- Continuous Integration, where all new code is integrated with the current system quickly to give visibility into the system development, since in XP the existing code is the current design. Whenever code is integrated, the system is rebuilt from scratch, and all tests must run.
- Collective Ownership means that no part of the system code is owned by specific programmers – instead every developer pair can change and refactor any part code as they see fit.
- On-Site Customer is perhaps one of the most difficult practices to adhere to. It means that a customer sits with the development team full-time, answering developer questions, writing functional tests and generally following that the system development progresses in the right direction.
- 40-hour Weeks. Continuously working overtime is considered a sign of trouble, and requires

addressing the cause, not symptoms.

- Open Workspace, where the developer team works together to facilitate easy communication.
- And finally, Just Rules means that if something in the XP process does not work for a team, they can change it.

While one of the main XP practices is Just Rules – meaning that changing the practices is allowed, the process itself on purpose does not set up an objective measurement system beyond project velocity. In fact, the system builds on positive, reinforcing feedback about the developer's work quality achieved through the continuous running of the unit tests, which is probably one of the reasons why XP developers report work satisfaction.

Improvement in the end products of the XP development process is therefore achieved through the personal learning and growth of the project developers. Unfortunately, this kind of improvement does not address systematic errors – problems with work habits that cause the same kind of defects to be injected again and again.

2.3 Defect prevention and detection practices

Traditional process-oriented software development methodologies try to prevent and detect defects in several ways. (Sommerville, 2000)

Process-oriented development methodologies typically spend quite a bit of time analyzing the problem domain, building models and gathering requirements. These requirements are then ideally distilled into a system architecture and design. While these activities certainly are important to avoiding requirements defects, they also have an important effect on programming defects as well.

The purpose of the requirements and design steps (whichever way they are named) is to make sure developers understand what they should be doing before they start actual coding. Plain common sense tells us, that a person who does not know what they are doing is probably not going to do it perfectly – and software developers are no exception.

Programmers familiar with the requirements and/or system overall architecture are also much more able to judge the importance and impact of their low-level design decisions, and/or to recognize a problem with the design when they face it.

Often process-oriented development projects also require

developers to adhere to a certain coding standard. The purpose of a coding standard is both to make deviations more visible, and to ensure the readability, clarity and maintainability of the code written by several different developers. There may also be checklists of certain most common errors to avoid while coding.

Defect detection can only be achieved by different kinds of testing. Testing is generally divided into two categories: Static and Dynamic testing. Static testing does not execute the code, and contains activities like reviews and using code analysis tools to find problems. Dynamic testing, on the other hand, does exercise the code, and is what is generally called testing. (Kaner et al., 2001)

The basic idea of a review is to look at what has been done with a critical eye. While anything that can be expressed can be reviewed, the most common associations of the word review in software development are reviews of the requirements and code.

An accepted truth in the software industry is that defects generally become more and more expensive to find and correct as the development process goes on, and that the costs associated to defects discovered by a customer are often orders of magnitude larger than those discovered during development. Requirements reviews are based on the idea that defects in requirements almost certainly find their way into defects in the code, and aim to weed out as many defects as early as possible. In code review the reviewer walks through the code trying to detect problems already in the code.

There exist several proposed variations of the review process. Most reviews in the software industry are peer reviews, that is, the co-developers review each other's work output. Often there is also a review meeting, where the defects are logged and new ones possibly found.

Dynamic testing is generally divided into black box or functional testing and white-box or structural testing. Black box testing chooses the tests based on the system's expected behavior and interfaces, while white box system uses knowledge of the system's implementation and builds tests to exercise a certain part of the system. (Sommerville, 2000) White-box testing is often connected with different coverage figures, telling which percentage of code statements and logical branches the testing has gone through.

Another common classification of testing is based on the level the testing happens at. Unit testing tests a single module, unit or object of source code. Integration testing is performed to test the collaboration of a subset of modules, and system testing tests the system as a whole.

Often the more mature organizations have also implemented some kind of defect prevention improvement process, whose purpose is to lessen the amount of defects originally introduced to the code by correcting the root causes of defects in the development process.

Two such processes are outlined by (Card, 1998) and (Mays, 1990). Both methods suggest analyzing existing problem reports, to identify the root causes of problems in that specific development culture and environment. (Card, 1998) emphasizes recognizing situations that lead to the introduction of defects again and again, and lists three projects, in each of which had 20-40 % of the problem reports could be traced back to the same systematic error as the principal cause. (Mays, 1990) reports that the defect prevention process outlined in the paper has been used in several organizations, resulting in defect reductions of the order of 50%. However, there is no data of the average results.

3. EXTREME PROGRAMMING DEFECT PREVENTION AND DETECTION PRACTICES

Most of the XP practices, in particular Pair Programming, Tests, Metaphor, Simple Design, Refactoring, Continuous Integration and Collective Ownership can be argued to be programming defect prevention and detection practices as defined in this paper. We concentrate on the two: Pair Programming and Test Driven Development.

3.1 Pair Programming

Pair Programming is the practice of two persons writing code together sharing a single screen, keyboard and mouse, jointly producing the design and code required. One is called the driver, and controls the pencil, keyboard or mouse, and is responsible for actually producing the output. The other is called the navigator, and is responsible for thinking about what the pair is doing - continuously and actively observing and partaking in the work by pointing out defects, suggesting alternative solutions, finding resources and considering the context and the strategic implications of design/code just produced. The partners are expected to change roles periodically. (Williams, 2000)

Working intensively together is claimed to have several advantages: the developers are more reluctant to interrupt their work when they work together resulting in an increased productivity. Because of peer pressure, pairs are also less likely to take shortcuts with prescribed methodology. Problem solving is less likely to "get stuck" with a pair than with a single programmer. Knowledge about the system and problem domain is more efficiently distributed, and new developers can

be quickly brought up to speed in a project. Additionally, developers report that pair programming and problem solving enhances work satisfaction.

In several sources pair programming is claimed to improve code quality without actually significantly increasing development time. (Williams et al., 2000; Williams 2001)

In a university experiment (Williams et al., 2000) the students completed four programming assignments. Pairs produced better quality and more consistent results as measured in the proportion of unit test cases passed. Pairs also were found to be more punctual about handing in their assignments, probably because they had to pre-schedule their work time, and were less likely to make the decision to drop the assignment/class because they did not want to let their partner down.

The average amount of test cases passed by programs handed in by individuals varied between 70,4-78,1, while the programs handed in by the pairs varied from 86,4 – 94,4 with a rising trend towards the end. The difference was found to be statistically significant. After the first assignment the pairs were found to spend a total of slightly more work hours but less clock time (slightly under 60%) developing the programs.

(Williams, 2000) further argues, that because of today's situation where time-to-market pressure can be strong, and the costs of a low quality product can be high in terms of support requests, the increased speed, quality and customer satisfaction resulting from pair programming are more than worth the relatively small extra expenses.

(Jensen, 2003) reports a case of using pair programming in the industry, developing a product of approximately 50 000 Fortran source code lines. The results are overwhelmingly positive, with a 127 % gain in productivity as measured by source code lines, and an error rate three magnitudes less than expected based on previous project data.

(Gehring, 2003) describes another experiment in pair programming on university students, with three programming assignments. The students were not allowed to pair with each other more than once. Students reported great satisfaction with the pair programming experience, average of 3,17 on a scale of 1-4. Students also generally felt that the co-operation and communication in the partnership were excellent (averaged 3,48 and 3,36 on the previous scale). The quality of student work was evaluated based on the student grades. The results of pair programming were significantly better in the first than the second and third assignments, although the author notes that statistical significance is hard to achieve because of the limited student numbers and small grade variance in the assignments.

(Williams, 2001) describes also another study with 41 university students. The students were divided into pairs and individuals based on expressed preference and grade point averages. Individuals used a variant of PSP (Personal Software Process, a highly disciplined process for improving one's personal software development skills and effectiveness based on rigorous measurement) and the pairs CSP (Collaborative Software Process, which is an extension to personal software process). In the study it was found that after the initial adjustment period programmers working in pairs had consistently about 15% less defects, and spent approximately 15% extra working hours in developing the programs, although the latter was not statistically significant.

There are a few common weaknesses with the studies. Almost all of the studies reported have been done with university students in university course environment where general assignment size is small. Pair programming has also been a voluntary experience; there is as yet very little information on how successful pair programming is if one or both partners are reluctant to use it.

Most of the studies report an 15-20% increase of program quality (as measured in loss of defects) as a result of pair programming, and only a slightly longer development time. However, the results of the single industry study differ from the university ones by an order of magnitude.

3.2 Comparing Pair Programming with reviews

Reviews are the most efficient known defect detection technique known in the traditional industry. Code inspection has been found to have 40-60 % defect detection rate depending on the used reading technique. (Laitenberger, 1998) In addition inspections also have a preventive effect; if developers know their work is going to be reviewed, they often check for the problems they know might surface in the review.

In a university experiment (Biffel and Halling, 2003) 169 students were trained in different review techniques, and then were instructed to review a 35-page design document of moderate complexity where 86 defects (an average industry rate) were seeded. Each student worked alone, and the found defects and their discovery times were recorded. Perhaps surprisingly, the effectiveness of students in finding defects was more dependent on the total effort spent than the number of students. In the study it was found that the first 12 hours of review effort discovered about 40 % of the defects. 20 % more defects (totaling 60 % of the defects in the document) were found with another 12 hours of review effort, and an additional 10% (totaling a detection rate of 70 %) by the addition of 12 more work hours. Students using the same review reading technique found similar defects, and best total detection rates

were achieved with combining multiple review reading techniques.

(Porter and Votta, 1997) have conducted a series of studies on the effect of different review variables on the effectiveness, effort and time interval of reviews. Surprisingly, the review process (including the presence or absence of review meeting, and the type of the procedures observed at the review meeting) mostly contributed to the review calendar time interval – in fact, a process where meetings were not held at all, but code was reviewed twice by the same person with a time interval in between was found to be slightly better than the others. The effectiveness of review was very dependent on review techniques used by the reviewers, with different perspective- and scenario-based reading techniques clearly superior to checklists and ad-hoc methods. Finally, the main driver of spent effort was the number of reviewers, and beyond two reviewers the effectiveness benefit increased only very slowly. This would fit in fairly well with the findings of the previous study.

In Pair Programming the design and all code written is essentially subjected to a continuous review. The navigator is responsible of strategic thinking and bringing up different issues and viewpoints.

The programmers report that this greatly enhances their confidence in their design and code quality (Williams et al., 2000). The amount of actual truth behind this hypothesis is difficult to determine due to the interlinked nature of XP practices. Since the XP practices enhance each other, and activities do not often have clear-cut boundaries, measurement of the XP process and evaluating the efficiency and individual contributions of its components can be extremely difficult.

The closest we can get is research information about the effectiveness of a similar review reading technique, called perspective-based reading, with a designer viewpoint to the system. In the study of (Biffel and Halling, 2003) this viewpoint was found to be less time-efficient than other viewpoints (user, tester) and checklist-based reading techniques and found defects, especially major defects, more slowly. This would suggest that training pair programmers in using different perspective-based review reading techniques might be useful, if the knowledge can be utilized in pair programming sessions. On the other hand, the fact that XP uses Test Driven Development together with pair programming may already provide a testing viewpoint to the programming process.

Additionally, if Pair Programming indeed did incorporate a review of the code, then the quality improvement from Pair Programming would be expected to be better. We speculate that since in XP the navigator needs to think also about design issues, the Pair Programming review process is actually much

less efficient than a structured, focused review.

(Kaner et al., 2001) reports the widely observed truth from process-driven environments, that almost regardless of the application environment, 20% of all produced software modules contain 80% of all the defects, and as many as 50% of the modules are completely defect-free. This would lead us to suggest that in XP, despite Pair Programming, the produced system quality might significantly increase if the most critical parts of the code were separately reviewed with more rigor.

3.3 Test Driven Development

Test Driven Development is one of the most important and distinguishable practices used by Extreme Programming. The key principle of Test Driven Development is that system code can be written only to refactor existing code or to make a failed test run – basically developers must write the unit tests for their code before writing the code itself.

Test Driven Development (TDD) has many names; Test First Coding, Test First Programming (TFP), Test First Design (TFD) and finally Test Driven Design (also TDD). (George and Williams, 2003)

This practice is claimed to have many advantages. According to (Beck, 2001; Jeffries, 1999) it forces the developers to think very precisely and exactly about what they want the next piece of code to accomplish, and to define criteria for when that goal has been reached. If rigorously practiced it also produces comprehensive regression test sets that can be used to ensure that all code is still working after any new change.

Test Driven Development practice results in relatively quick feedback to the developer about whether something is working or not, moving much of the debugging activity to occur during and just after coding, when the design and implementation is still fresh in the developer's memory. Test Driven Development also encourages developers to write code that is easy to test, further contributing to its quality. Automated test cases create an ever-growing regression test set that shows broken functionality quickly and contributes to the reliability of the system.

(Jeffries, 1999) describes the XP approach to testing, Extreme Testing, where all test cases are automated. The test cases are divided into two subcategories: unit tests and functional tests, which test a specific functionality of the whole system. Unit tests are produced by practicing Test Driven Development, and all unit tests must all run before any code is integrated. This increases confidence that the system is working and that a change does not break anything. Unit tests are not reported

on, since all of them should run all the time. Functional tests demonstrate that a specific new increment of functionality works. These functional tests should be customer-owned, comprehensive, repeatable, automatic, timely and public. They must provide value to the customer, and be available whenever the developers need them.

(Beck, 2001) has argued, that Test Driven Development (or Test First Coding, as he calls TDD) as practiced in XP is actually a design technique, not a testing technique. It has a twofold purpose: help with analysis by making the programmer figure out what the correct behavior is before implementing something, and help with design by defining the module's interface and scope with the outside world. (Beck, 2001) also claims that this results in a code that is more cohesive and less coupled since unnecessary structures are often not created, and that Test First Coding helps to separate logical design from physical design from implementation.

Another paper (Steinberg) describes that the author, as a teacher of and introductory Java programming course has found this approach quite useful in achieving more cohesion and less coupling in student programs by using Test Driven Development approach to avoid introducing the main method as long as possible.

A problem with TDD is that it trusts the test sets fairly implicitly – yet tests are just code, and prone to the same kind of error real system code is. Additionally there exists practically no information available about XP-style functional testing and its effectiveness.

There may also be problems with test case coverage, as described by (Elssamadisy and Schalliol, 2002) who were part of a very large and very long XP project – about 50 people and well over 2 years. Their problem was that eventually the system to be developed got so complex, that they themselves were no longer able to remember all dependencies – which, in the XP way, were never documented. As a result it was very easy (in fact almost impossible not to) to just forget to modify all the necessary test cases before starting coding, or keep track of the overall dependencies of the design while coding. This resulted in parts of the system not being covered by the comprehensive unit tests. The modules they were changing always had complete unit tests, but those modules depended on other modules, which depended on still other modules and so on.

3.4 Comparing Test Driven Development with unit testing after development

(George and Williams, 2003) ran three sets of experiments with 24 professional pair programmers to investigate claims

about Test Driven Development. One group developed code with traditional waterfall-type process; the others used TDD approach in developing a small program. The TDD group was found to produce more error-free code that passed 18% more functional black box test cases. The TDD pairs also took 16% more time for development. The developers following the “traditional” process often, despite instructions to the contrary, did not write the required automated unit test cases at all, which may account for both the time discrepancy and quality difference.

(George and Williams, 2003) found a fairly good correlation with development time and code quality. The coverage of test cases written by TDD developers was much above traditional industry average – a mean of 98% method, 92% statement and 97% branch coverage, while unit testing in industry reaches 80-90% coverage levels. The programmers in the experiment generally felt that Test Driven Development improves code quality and the lack of up-front design is not a problem, but that TDD also increases development time and that it may be a difficult approach to adopt. (George and Williams, 2003) noted that getting developers to write good unit test cases after writing the code proved difficult, and concluded that TDD approach encourages good unit testing coverage and also enhances code quality.

Both (Williams L., et al., 2003) and (George and Williams, 2003) also describe a university experiment of 19 students (Müller and Hagner, 2002), where Test Driven Development effectiveness was measured in development time, reliability and understandability. Both TDD and control groups were given a specification of the task to be accomplished, and the control group was asked to produce unit test cases after coding. There was found to be no significant difference in development speed or quality between the two groups.

In yet another study (Williams L., et al., 2003) explore the use of Test Driven Development in an industrial setting as a defect reduction practice combined with black box functional tests. The project studied was a device driver of about 73 600 lines of code, most of it new. A stable maintenance release device driver project of 56 500 lines of code was used as a control project. The control project had more platforms, and therefore was run thorough about twice as many functional test cases as the TDD project.

The developer groups were not fully comparable either – the control project had 5 experienced, collocated engineers, while the TDD project had 9 somewhat inexperienced and physically distributed engineers. The control project unit testing was fairly informal and “ad hoc”. It is important to note that the TDD project still did design up-front and not during development as per XP practices.

(Williams L., et al., 2003) found that the defect density of the Test Driven Development project code when it entered the functional verification testing was 40 % lower than with the control project. The severity of the observed faults, however, was essentially equal. TDD impact on developer productivity was insignificant.

In general, Test Driven Development seems to produce somewhat higher quality code, and much more comprehensive test sets than the more traditional approach of unit testing only after programming. Test Driven Development seems to take slightly more time, which is understandable given that there is more code (the test sets) to produce.

While modifying the automated test sets after every change in the code is certainly effort-intensive, up-to-date, comprehensive test sets might prove extremely valuable once the system reaches maintenance state. Unfortunately, there are no studies on this.

(Laitenberger, 1998) examines the defect detection effectiveness of combining reviews with different unit testing techniques. Code is first reviewed, and then tested with structural (white-box) methodology using different coverage criteria. In the study it was found that white-box testing techniques generally focus on and find the same kinds of defects as inspection does.

The results of the study suggest that since the XP practice of Pair Programming can already be considered a partial review, XP unit testing might be best to be functionally oriented using black-box techniques for maximum defect detection efficiency.

4. SUMMARY

In the previous chapters we have discussed the defect prevention and detection practices used by the Extreme Programming agile software development process. The most distinguishing and important defect prevention and detection practices in XP are Pair Programming and Test Driven Development. We discussed some of the existing knowledge about these practices, and compared them to reviews and traditional unit testing.

The amount of actual studies with quantitative results instead of qualitative opinions only is found to be small, and evidence inconclusive. Additionally only a few studies deal with programs of a realistic size, and therefore the results may not be transferable. There seems to be a tendency for a slightly higher product quality at the same or a slightly slower development speed with both individual XP practices than without. Whether the quality difference can be achieved by a comparable cost by process-driven methodologies remains

open. Future shall also show whether the comprehensive regression test sets produced by the Test Driven Development prove to be a valuable asset once the software reaches maintenance phase of its lifecycle.

Based on the existing research data on reviews, we speculate that the review process benefits claimed by Pair Programming are not in actuality fully realized, and suggest that introducing a separate focused review of the most critical parts of the system might improve XP output quality. We also suggest that XP programmers would benefit of being aware that functional, back-box type unit tests probably would prove much more efficient in finding defects than structural, white-box style tests.

In the future it would be interesting to have more data on using Extreme Programming practices on projects of medium to larger size to see if the results achieved in academic environment scale up.

Another extremely interesting course of study would examine the defects found in XP projects, and determine whether XP project defect causes are differently distributed from those of the process-driven project defects, and try to determine if there are certain types of defects XP techniques seldom detect.

REFERENCES

- Beck, K., Embracing Change with Extreme Programming. IEEE Computer, 1999, vol. 32, no 10. pp. 70-77.
- Beck, K., Aim, fire. IEEE Software, 2001, vol. 18, no 5. pp. 87 - 89.
- Biffi, S. and Halling M., Investigating the defect detection effectiveness and cost benefit of nominal inspection teams. IEEE transactions on software engineering, vol 29, no 5, 2003.
- Boehm, B. and R. Turner, Using risk to balance agile and plan-driven methods. IEEE Computer, 2003, vol. 36, no 6. pp. 57-66.
- Card D., Learning from our mistakes with defect causal analysis. IEEE software, 1998.
- Elssamadisy A. and Schalliol G., recognizing and responding to "bad smells" in extreme programming. Proceedings of the 24th International Conference on Software Engineering (ICSE), May 2002. pp. 617-622.
- Fowler, M. The New Methodology. <http://martinfowler.com/articles/newMethodology.html>. Referenced 21.3.2004
- Gehringer, E., A pair-programming experiment in a non-programming course. OOPSLA 2003.
- George, B. and L. Williams, An Initial Investigation of Test-Driven development in Industry. Proceedings of the ACM symposium on Applied computing, March 2003.
- Jeffries, R. E., eXtreme Testing. Software Testing and Quality Engineering. 1999, March/April.
- Jensen, R. W., A Pair Programming Experience. CrossTalk, The Journal of Defense Software Engineering, March, 2003. <http://www.stsc.hill.af.mil/crosstalk/2003/03/jensen.html>
- Kaner C., et al., Lessons Learned in Software Testing - A context-driven approach.. Wiley Computer Publishing, 2001.
- Laitenberger, O., Studying the effects of code inspections and structural testing on software quality. ISERN-Report ISERN-98-10.
- Leszak M., Perry D., Stoll D., A Case Study in Root Cause Defect Analysis. ICSE 2000.
- Mays, R., Applications of Defect prevention in software development. IEEE journal on selected areas in communications, 1990
- Müller M. and Hagner O., Experiment about test-first Programming. Conference on empirical assessment in software engineering (EASE), 2002.
- Müller, M and Padberg F, On the economic evaluation of XP projects. ESEC/FSE 2003
- Porter, A and Votta, L, What makes inspections work? IEEE Software, 1997
- Sommerville, I, Software Engineering, 6th edition, Addison Wesley Professional 2000
- Steinberg, D.; The effect of unit tests on entry points, coupling and cohesion in an introductory java programming course.
- Williams, L., et al., Strengthening the case for pair programming. IEEE Software, 2000, vol. 17, no 4. pp. 19-25.
- Williams, L., Integrating Pair programming into a Software Development process. IEEE 2001
- Williams, L., et al., Test-driven development as a defect-reduction practice. 14th International Symposium on Software Reliability Engineering, 2003, 17-20 Nov. pp. 34 - 45.