

Applying Extreme Programming in Object Oriented Programming Lessons in Classroom

Kimmo Lomperi

Abstract

When programming in classroom in programming courses students tend to write schoolboy or schoolgirl code i.e. the products of their work have very little resemblance to products in workplaces. The assignments of their programs are first coded, then tested vaguely and superficially and finally discarded. Reliability of software is disregarded. However, after having been graduated these fellows should code reliable software to real customers.

Extreme programming could not only be used in assessing the quality of students' work but that it's use would also assist those students who appeared to have difficulty both in mastering the proper coding methods and then adhering to them.

The purpose of this study was gaining experience of extreme programming in an unbiased fashion, without having to come up with any preordained results being on the side of XP or against it.

A case study of one of the cornerstones of extreme programming, pair-programming, was made.

Index Terms

Classroom Programming, Extreme Programming, Object Oriented Programming, Pair Programming

1. INTRODUCTION

1.1 Background

THIS paper is done for finding whether extreme programming would solve the problem of the poor quality of programs of college and polytechnics level students. The heads of students are not empty. The problem is not the assumed emptiness of students' heads. The real problem is the assumption of emptiness of students' heads as a basis of teaching (Saarinen E. et al 2000). I try to keep this in mind in this paper.

To learn the ability to test programs is important for students. Software houses take that for granted that students know how to test programs; thus students will be grabbed to real software coding projects quickly after getting employed.

As being a teacher of basic courses of object oriented programming several questions dealing with coding classes and methods have arisen. Most familiar languages for me are Java and C++; thus Java code is the frame of this study.

Young eighteen years old students do not understand customers' point of views when receiving releases from software producers. Customers want to get properly functioning products.

Doing testing means doing three things; designing tests, implementing the tests you have designed, and evaluating those tests. Extreme methods are being discussed here. Also critical comments on extreme programming including testing will be proclaimed.

The most immediate difference between agile methods and engineering methods is that agile methods are less document-oriented, usually emphasizing a smaller amount of documentation for a given task. In many ways they are rather code-oriented: following a route that says that the key part of documentation is source code (Fowler M. 2003).

Agile methods have much in common with students in programming lessons. Students much rather concentrate on coding and nothing but coding. Designing methods do not share pupils' interest. However, designing methods will still be worth learning for anybody who do not know the basics of them.

Students sometimes think that testing is boring and of no use. On the other hand sometimes students try to amend a program which was executing apparently correctly (Thompson 1991). If there is any chance to get a little joy to testing it would be worth trying extreme testing methods. Also from a professor's point of view a quick and rewarding testing method would be attractive.

1.2 Research Problem

Is extreme programming a suitable method for use in object-oriented programming courses?

1.3 Objectives

The first objective is to find out whether extreme programming improves the quality of student made programs in an object-oriented programming course.

Milestone 1 is introduction of the circumstances where object-oriented coding in classroom is implemented. This milestone is included in subobjectives of this study.

Milestone 2 is to use pair-programming in classroom. This milestone is included in main objectives of this study.

Milestone 3 is to use extreme programming in classroom to improve the quality of code. This is the main objective of this study.

1.4 Scope

A literature study of the subject concerned will be made. In addition practical and empirical tests will be executed in classroom.

A scope is learning from experience (Beck, K. 2000). By not trying to do too much, we preserve our ability to produce required quality on time (Beck, K. 2000).

All the details of extreme programming will not be dealt with. The main concern is pair-programming, the heart of extreme programming, and other main issues of extreme programming. In classroom some of the minor details of XP are not so relevant, that is why they are excluded from this study.

1.5 Methodology

One tries to search from literature and web etc. what has been studied concerning this subject. Information found will be reported in this paper. A qualitative analysis of using extreme programming in an object-oriented programming course will be made. Agile object-oriented programming in classroom hasn't been studied widely. That is why decent research papers are not found adequately.

The most promising methods will be not found necessarily from literature. It is possible that one can invent them during the reflection of these problems.

It would be desirable to test the most promising method in classroom. A case study will be implemented.

1.6 Structure of This Report

In chapter one is the introduction part. The next couple chapters will deal with object-oriented programming and extreme programming and testing.

Quality of code is discussed in chapter four. Chapter five presents a case study executed in object-oriented programming course on 26th of March, 2004. The discussion and conclusion chapters discuss these matters and aspects also for future respects. References are at the end of this paper.

2.1 General

The features that make object-oriented programming appealing; however; cause testing of object-oriented programs demanding and complex. Testing of a feature should be done only in one place i.e. if features in child classes are tested in vain that will cause unwanted complexity to testing process (Alkadi et al 1998). One should pay attention to testability (Binder 1994). The metrics of object-oriented programming are presented in e.g. (Chidamber et al 1994, Briand et al 2001). The first one of the papers says that metrics measure e.g. coupling between objects, number of methods and children of classes, method calls and depth of inheritance tree.

Testing procedure should reduce unnecessary testing, uncover errors and help errors from recurring. Normally we code first and test then. This practice very often leads to a recode-retest and retest-recode cycle (Alkadi et al 1998). In a test-driven manner testing the functionality of a program should be planned and designed before coding. One can use different testing techniques. Effective and successful testing helps in maintenance of software. It is important that pupils adapt testing as a normal practice among designing and coding.

In classroom in object-oriented programming design patterns (Gamma et al 1995) can be used. Pupils will learn a wide variety of different programming abstractions and will use the same vocabulary.

Testing theories are methodologies to handle testing. In literature several methodologies are presented (Arnold et al 1994, Harold et al 1992, Korel et al 1996, Wong et al 1995). Implementation of testing will be presented e.g. in following references (Devanbu et al 1996, Kung et al 1995).

When one has tested enough one's code? The answer may lie in good enough testing. Good enough testing is a process of developing a sufficient assessment of quality, at a reasonable cost, to enable wise and timely decisions to be made concerning the product (Bach 1998).

The usage of a testing assistant is pondered in certain studies (Alkadi et al 1998). These kind of tools are widely used in classroom studies. Students normally adapt the usage of the helping tools quickly and appreciate the use of the tools because they provide the same kind of atmosphere that there is in working places. A testing assistant provides a framework that helps to ensure that appropriate components and interactions will be tested by generating code segments that drive the testing process (see an example of a testing assistant in Figure 1).

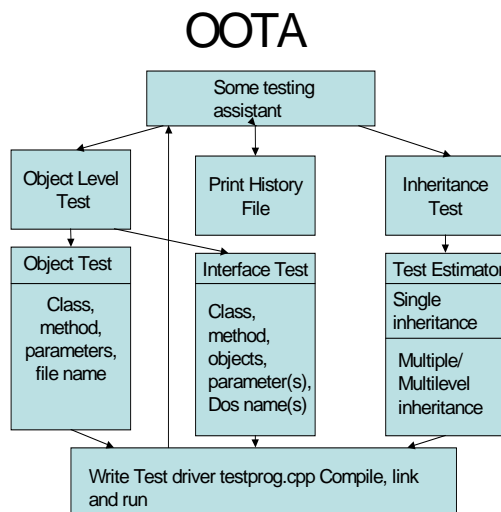


Figure1. Object-Oriented Testing Assistant (Alkadi et al 1998)

2.2 Special Features of Object Oriented Testing

Object-oriented paradigm (OOP) states that an object is an entity composed of data and procedures. The data is called attributes and the procedures are called methods in Java. The methods implement the operations on their object's data. Each object has a state, an identity and a behavior (Alkadi et al 1998). OOP has numerous other powerful features including inheritance, data abstraction and dynamic binding.

2.2.1 Inheritance

Inheritance enables subclasses use the data and methods of their superclasses excluded that attributes are not defined as a type private. Inheritance, which is either one to one (single) or one to many (multiple), facilitates and encourages reusability (Murphy et al 1994). Inheritance allows an object in a subclass to override the need of multiple instances of a method. In addition to overriding substituting is also used. With multiple inheritance a class is permitted to have more than one superclass (D'Souza et al 1994). Multiple inheritance increases the possibility of code sharing and increases also the risk of conflicts (Alkadi et al 1998).

2.2.2 Data Abstraction

An important feature of object-oriented programming is data abstraction which consists of encapsulation and information hiding. Data abstraction allows the encapsulation of the data and the procedures, that handles the data, together in one syntactical unit. Information hiding, a powerful feature used on components of the object to reduce unnecessary interfaces

of objects, uses the notation of data abstraction. Data abstraction improves usability of data structures and thus reduces writing of same modules in different systems. Therefore data abstraction reduces complexity (Alkadi et al 1998).

2.2.3 Dynamic Binding

Dynamic binding, which facilitates compilation of software, makes it possible to implement polymorphism. In polymorphism inheritance is implemented where the same message is sent to different objects of different classes at different times and responded to by different methods. Dynamic binding permits adding new objects of a certain class without having to modify existing code (Alkadi et al 1998). Biggest drawbacks of dynamic binding are the cost at runtime and the difficulty involved in implementation (Chidamber et al 1991).

2.3 Testing Techniques

Most common object-oriented testing techniques are object level testing (OLT) and testing of inheritance within classes (TIC). Naturally unit testing and features e.g. path, control, branch and statement testing can be used in object-oriented testing (Alkadi et al 1998). In classroom when programming object-oriented programs these tests will be implemented rather in an incremental fashion.

2.3.1 Object Level Tests

Methods of a class should be tested first. Every instance of an object in a class communicates properly with objects in the same class or in the class's scope of communication (Alkadi et al 1998). Object level tests include object testing and interface testing.

2.3.2 Inheritance Tests

Single, multiple and multilevel inheritance in an object of a class form the field of inheritance testing. Methods and data structures inherited should be carefully tested. A small addition of code in upper hierarchy of classes may cause unbelievable errors in subclasses. This is in contradiction to the encapsulation principle (Haikala 2001). Students use inheritance especially with graphical user interfaces e.g. a window class is a superclass and different window types will be inherited accordingly.

3. EXTREME PROGRAMMING AND TESTING

One should remember the principle "Work with human nature, not against it" (Beck, K. 2000).

3.1 General of Extreme Programming

Extreme programming (XP) is a lightweight software development method for relatively small teams dealing with changing requirements (Muller et al 2001). XP is different compared with traditional programming and designing methods for following respects.

No software specification exists in XP. Source code is the means of documentation in extreme programming.

Simplest possible solution is an important feature in extreme programming. Extra complexity is removed as soon as it is discovered (Beck, K. 2000). No separate coding and separate testing phases exist. Continuous auditing helps in this respect.

The code, both functionality and units, will be implemented in small increments. Changes, automated testing will be managed in a controlled fashion when this refactoring of code is used. All and all XP embraces changes.

Pair-programming, being a corner stone of XP, increases software quality without impacting time to deliver. It is counter intuitive, but two people working at a single computer will add as much functionality as two working separately, except that the results will be much higher in quality. With increased quality comes big savings later in the project (extremeprogramming.org. 2004). One sees unofficial pair-programming frequently when students are coding, what about pair-programming in testing? It is obvious that pair-testing is a bigger challenge than pair-programming. It is possible that pairs first program together and then testing will be implemented by the members of the pair separately.

3.2 Extreme Testing

Testing is a bet. The bet pays off when your expectations are violated. One way a test can pay off is when a test works that you didn't expect to work. Another way a test can pay off is when a test breaks when you expected it to work. In either case, you learn something (Beck, K. 2000).

It is impossible to test absolutely everything, without the tests being as complicated and error-prone as the code. It is suicide to test nothing (in this sense of isolated, automatic tests). You should test things that might break. If code is so simple that it can't possibly break, and you measure that the code in question doesn't actually break in practice, then you shouldn't write a test for it (Beck, K. 2000).

Here is what XP testing is like. Every time a programmer writes some code, they think it is going to work. So every time they think some code is going to work, they take that confidence out of the ether and turn it into an artifact that goes into the program (Beck, K. 2000).

The tests that you must write in XP are isolated and automatic. First, each test doesn't interact with the others you write. That way you avoid the problem that one test fails and causes a hundred other failures. Nothing discourages testing more than false negatives (Beck, K. 2000).

The tests should be automatic. Tests are most valuable when the stress level rises, when people are working too much, when human judgement starts to fail. So the tests must be automatic – returning an unqualified thumbs up/ thumbs down indication of whether the system is behaving (Beck, K. 2000).

In classroom usage XP's lack of documentation isn't a good practice. If development meets unexpected difficulties that make the code itself harder to understand. When a lot of people leave the project, the code just becomes nearly incomprehensible despite efforts to keep it simple. In addition testing will be difficult and inadequate then. In XP you are left on your own without e.g. documentation and explicit architecture to help (van Deursen 2001).

Regarding this paper extreme testing has proved to be a most promising one of agile testing methods. A combination of different tests is not out of question in this respect.

Exploratory testing is exactly the kind of thing that comes before rigor and makes rigor possible. ET is primarily creative. It has value in terms of the problems we find while doing it, but it also has value because it is a way to create other kinds of tests (Bach, J. Exploratory Testing and the Planning Myth). Exploratory testing has been discarded for the reason that it needs deep proficiency. Beginners do not have that kind of properties.

Other tests can be executed in addition to XP testing e.g. parallel tests, stress tests and monkey tests (Beck, K. 2000).

One implements the customer's most important requirements first, so if further functionality has to be dropped it is less important than the functionality that is already running in the system (Beck, K. 2000).

If we want programmers and customers to write tests, we had better make the process as painless as possible, realizing that the tests are there as instrumentation, and it is the behavior of the system being instrumented that everyone cares about, not the tests themselves. If it was possible to develop without tests, we would dump all the tests in a minute (Beck, K. 2000).

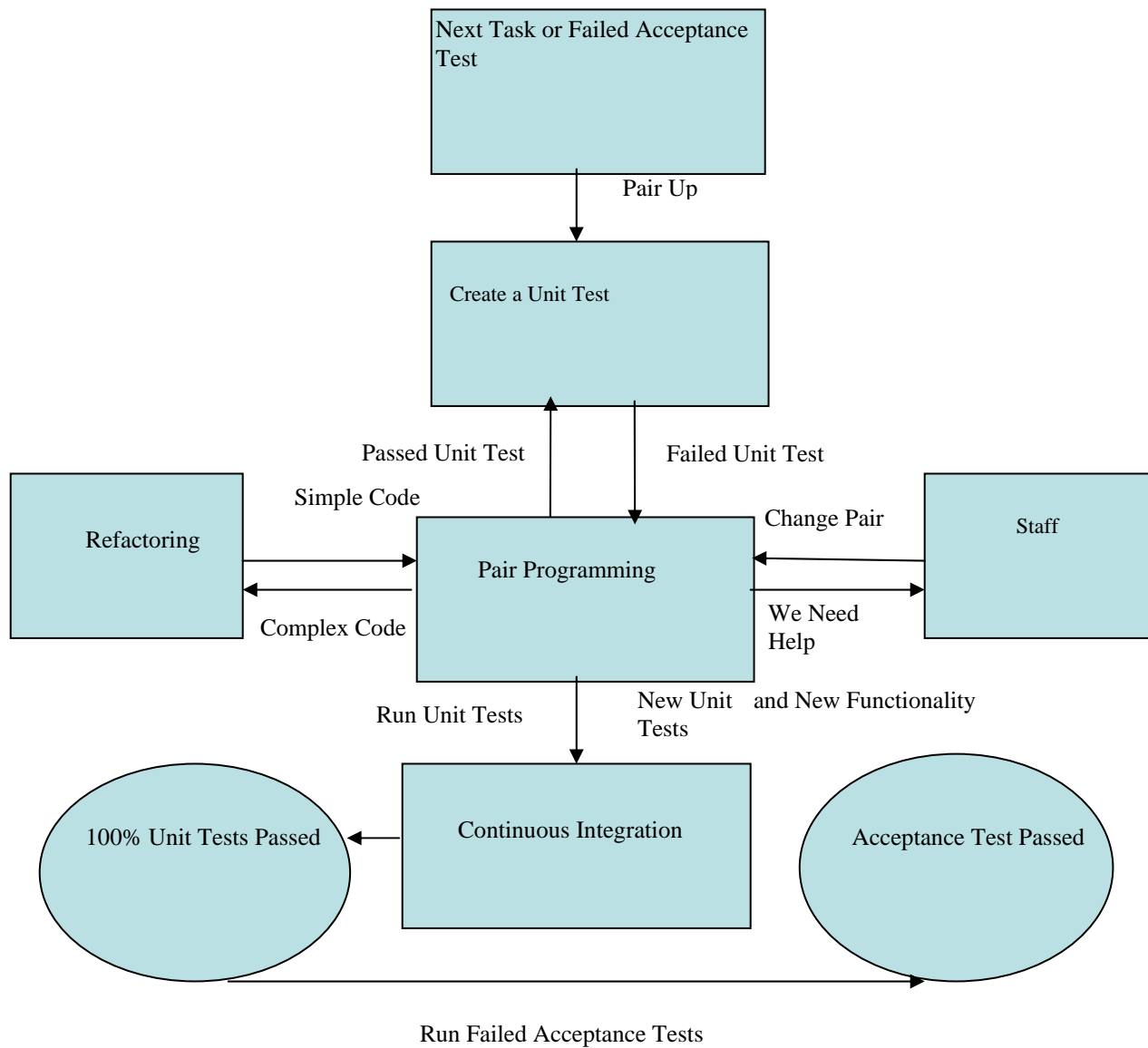


Figure 2. Collective Code Ownership
<http://www.extremeprogramming.org>

In Figure 2 a model is presented. All code must have unit tests. All the unit tests have to be passed 100% before code can be released. Acceptance tests are run often. Also databases including information of errors and situations concerned could be used when acceptance tests are being executed. A lightweight methodology has only a few rules and practices to follow. That kind of methodology suits well to extreme testing principles.

It is desired that both programmers and customers write tests. The programmer-written unit tests always run at 100%. The

functional tests (customer-written) don't necessarily run at 100% all the time. As you get close to a release, the customer will need to categorize the failing functional tests. Some will be more important to fix than others.

The tester's job is to translate the sometimes vague testing ideas of the customer into real, automatic, isolated tests (Beck, K. 2000). The role of tester in an XP team is really focused on the customer. You are responsible for helping the customer choose and write functional tests (Beck, K. 2000). An XP tester is not a separate person, dedicated to breaking the system and humiliating the programmers. However, someone has to run all tests regularly (if you can not run your unit and functional tests together), broadcast test results, and to make sure that the testing tools run well (Beck, K. 2000).

3.2.1 Unit Testing

If there is a technique at the heart of XP, it is unit testing (Beck, K.1999). If one wants to be sure that a new feature works one have to write unit tests for every feature (Jeffries, R. 1999). To be sure that new code hasn't broken anything else all the unit tests in the entire system must be running completely (Jeffries, R. 1999). One should continually write unit tests. Depending on their flawless operation further coding can take place (Beck, K. 2000).

3.2.2 Functional Testing

Good estimates, how long will a feature take to implement, reduce the probability that one has to drop functionality (Beck, K. 2000). One implements the customer's most important requirements first, so if further functionality has to be dropped it is less important than the functionality that is already running in the system (Beck, K. 2000). If important functionality is left to the end of every release cycle, the customer would soon get upset. To avoid this two strategies can be used. Firstly, one can get a lot of practice by making estimates and feeding back the actual results. Better estimates reduce the probability that you will have to drop functionality. Secondly, customer's most important requirements should be implemented first. So if further functionality has to be dropped it is less important than the functionality that is already running in the system (Beck, K. 2000).

If we want programmers and customers to write tests, we had better make the process as painless as possible, realizing that the tests are there as instrumentation, and it is the behavior of the system being instrumented that everyone cares about, not the tests themselves. If it was possible to develop without tests, we would dump all the tests in a minute (Beck, K. 2000).

It is desired that both programmers and customers write tests. The programmer-written unit tests always run at 100%. The functional tests (customer-written) don't necessarily run at 100% all the time. As you get close to a release, the customer will need to categorize the failing functional tests. Some will

be more important to fix than others. The tester's job is to translate the sometimes vague testing ideas of the customer into real, automatic, isolated tests (Beck, K. 2000).

4. QUALITY OF (SCHOOLBOY/SCHOOLGIRL) CODE

Auditing is an effective method for improving the quality of code. Together with testing they should form the basis of a coding environment of good enough quality. In extreme programming especially pair-programming is suitable for verification and validation of code. An unofficial walkthrough is a method, where the coder explains to some other person(s) what he/she thinks the code does (Haikala 2001). This method is applicable when students code.

Extreme programming excesses the quality of code proves the following papers (Poole et al 2001) and (Wright 2002). The latter one expresses that XP doubles the productivity and drops the amount of defects to one fourth.

Collective code ownership is an important way to ensure quality in programs (see Figure 2) but it has certain perils (see chapter Discussion).

On the other hand when considering the drawbacks of extreme programming: XP disregards documentation and architectural design. That fact hinders XP's usage as a solitary method for improving the quality of schoolboy/-girl code. In school students should learn all the main methods concerning quality including extreme programming.

5. A CASE STUDY ON PAIR-PROGRAMMING

In the previous chapter the quality of code in classroom programming was talked about. One way to improve the situation is presented here. This case study must be considered a worth while exercise. Here is presented a case study executed in object-oriented programming course on 26th of March, 2004.

Unfortunately there were only six pupils present when pair-programming was tested in classroom. However, all of the students were eager on testing pair-programming when coding object-oriented programs. The students got a short introduction to the roles of pair members. They were also told that normally better quality of code can be reached by using pair-programming.

The problem dealt with inheritance and a maintaining application of tabeling five objects of a class. Every second value of the table was supposed to be an integer from a superclass and every second value a character from a subclass. Data was asked from user and the table was finally output on the screen.

1. Question: How did you like pair-programming

unpleasant					pleasant
1	2	3	4	5	
		x			
		x			
			x		
			x		
			x		
		x			

Table 1

2. Question: Was the product i.e. program of better quality than when coding alone?

poor quality					good quality
1	2	3	4	5	
		x			
		x			
			x		
			x		
				x	
		x			

Table 2

In analyzing the results one should notice that those students who put their 'x' in the column number 3 actually meant that the pleasure level was number three. Concerning Table 2 it means that the quality level was number 3. The results show that students like pair-programming and think that pair-programming, being the heart of extreme programming, brought joy to their programming.

Comments on the matter:

One group thought that pair-programming is a good thing, because neither of the students felt that she/he was a strong programmer so far. Pair-programming gave them confidence to program. This result has been obtained in other experiments also (Muller et al, 2001) and (Cockburn et al 2000). Of the roles students thought that it should be agreed in advance how long certain roles are valid and when is the time of switching the roles.

One advanced group had strength and energy enough to implement a separate testing program to debug their code. Separate test cases, isolated tests and small increments were used. In their code a checking program with checksums and testing of types assured the quality of the code.

5.1 Comparing The Results to Previous Experiments

In 1999 at the University of Utah, senior software engineering students took part in a similar pair-programming experiment (Williams et al 2000a). In my experiment, on March 2004, the course where students are programming is their second programming course. In the experiment in Utah 85% were satisfied with pair programming i.e. indicated their preference for pair-programming (Williams et al 2000a). Of the time used for coding in Utah the time dropped drastically from 60% to 15% of extra time compared to single programming. In my experiment the group being so small no control group was established. Though, the fastest of the groups completed their work approximately in the same time than previously fastest single programmers had completed with their similar programming work.

When dealing with the quality of code Bisant (Bisant et al 1989) experimented inspection of code and design with 29 students. A pair tried to find errors in a twenty minutes period. Saving of time was significant compared to time lost in normal inspection steps. A real difference in the quality of results was found compared to a control group.

In experiments (Cockburn et al 2000) (Williams et al 2000a) a significant reduction of errors was found out when pair-programming instead of single programming ($p < 0.01$).

In Nosek's experiment (Nosek 1998) five pairs of professionals tested pair-programming. Pairs liked more the programming and trusted their results more than when doing single programming. A single programmer used 41% more time than a pair. However, two single programmers were together 30% more efficient than a pair, if the quality of code is left out of inspection.

Muller (Muller et al 2001) tested pair-programming with six pairs of university students. One pair had big problems, because the first one wanted to design and the other one wanted to get the task done. The pair didn't enjoy pair-programming. That is obvious that there are prima-ballerinas also among programmers. There are social rules of pair-programming (Williams et al 2000b), but organisation of work is unclear.

6. DISCUSSION

Advantages of extreme programming include the following aspects. Firstly pair programming allows knowledge to transfer

among participants. Continuous auditing ensures good quality of programs. Secondly programs are coded in small well-defined increments. Thirdly most unit tests should be written in advance before coding. This helps with the specifications of programs. This also enforces a discipline of providing comprehensive unit tests (Allen et al 2003). When structure of program should be simplified or stream-lined refactoring of code could be used (Fowler et al 1999). If one writes one's code again and again, just like Ernest Hemingway wrote his books, duplication will be omitted; thus communication, simplicity and flexibility will be added to the construction of the code.

Restructuring is also a method of interest in the contest of this paper.

Of collective ownership: Anyone can change any code anywhere in the system at any time (Beck, K. 2000). Does this add value to quality prospects?

Using extreme programming in classroom poses some logistical and managemental problems (Wege et al 2001). Effective software management may cause problems because students can devote only part of their time of their work-week to work on the project. Students often have varying schedules and even conflicts, this affects their ability to program in pairs. How to ensure that intelligent and maintainable code will be produced by these newcomers in software branch? (Wege et al 2001). What about estimates – how long will a feature take to implement – do students have the ability to estimate?

Do programmers in pair-programming need two displays? I didn't find anything like this. Muller (Muller et al, 2001) insists that pairs must have two displays. The first one for programming and the other one for testing or debugging purposes. Muller also says that is very difficult for students to build or test in small increments. They see the things as a whole. I didn't find anything like that. On the contrary a prematured youngster often pays attention to some unimportant details without understanding the context. At least one of my pupils, however, in spite of Muller's experiences is clever with coding and testing in small increments.

7. CONCLUSIONS

Agile way of thinking surely has something to give for majority of IT professionals including those who study in IT branch.

The iterative and incremental models have been used in classroom and should be emphasized in the future. One can start from a scratch e.g. test some single extreme method in one's own daily work. Studying is the work of students – or at least it should be. Even a single successive method may gain profit in workplace or give a feeling of success to students.

Pair programming can be used for quality assurance and studying purposes. Refactoring helps in maintaining software. As a co-product students will learn the importance of testing! That is not so obvious for students around eighteen years; whether to test or not. For students it is good to reflect the values of extreme testing and apply these studied methods elsewhere. When one learns new interesting models of thinking one will be able to change one's way of thinking and one is open to test new paths.

7.1 Own Comments

It was interesting to get accustomed to agile systems and methods. In addition writing a technical paper with instructions was a demanding task. I have reflected these matters with my previous knowledge of programming and testing. The reflection will keep on going.

Personally I was delighted that especially one student has adapted well testing in small increments. That is little amazing because she was absent from the lesson when pair-programming was practised.

7.2 Future Aspects

Extreme programming and other agile methods and combinations of agile methods and test-driven methods should be used in classroom studies.

Especially interesting testing method would be crosstesting in groups i.e. the code, produced e.g. using pair-programming of a group, is given to some other group to be tested and so on.

Also in inspection of code unofficial walkthroughs among students should be encouraged.

REFERENCES

- [1] Alkadi, I.S., Carver D.L. 1998. A Testing Assistant for Object-Oriented Programs, IEEE Software 1998 pp. 149-158.
- [2] Allen E., Bannet J. and Cartwright R. 2003. A First-Class Approach to Java Generity. Submitted to POPL.
- [3] Allen E., Cartwright R. and Stoler B. 2002. Efficient Implementation of Run-time Generic Types for Java. IFIP WG2.1 Working Conference on Generic Programming, July 2002.
- [4] Allen E., Cartwright R. The Case for Run-Time Types in Generic Java. Principles and Practice of Programming in Java, June 2002.

- [5] Allen E., Cartwright R. and Reis C., Rice University, "Production Programming in the Classroom", <http://www.cs.rice.edu/~eallen/papers/sigcse9-6-5.pdf> 23.3.2004.
- [6] Arnold T., Fuson W., "Testing 'In a Perfect World'", Communication of the ACM, September 1994, Vol. 37, No.9, pp. 79-86.
- [7] Bach, J. "A Framework for Good Enough Testing" Computer, October 1998, pp. 124-126.
- [8] Bach, J. Exploratory Testing and the Planning Myth
- [9] Beck, K. Embracing Change with Extreme Programming. Computer 1999, pp. 70-77.
- [10] Beck, K. 2000. Extreme Programming Explained. Embrace Change. Addison Wesley.
- [11] Binder R., "Design for testability in Object-Oriented Systems", Communication of the ACM, September 1994, Vol. 37, No.9, pp. 87-101.
- [12] Bisant D., Lyle J., "A two-person inspection method to improve programming productivity. IEEE Transaction on Software Engineering, 15, pp. 1294-1304, October 1989.
- [13] Briand, L.C., Bunse, C. and Daly, J.W., "A Controlled Experiment for Evaluating Quality Guidelines on The Maintainability of Object-Oriented Designs". IEEE Transactions on Software Engineering. 2001, pp. 513-530.
- [14] Cartwright R., Steele G. 1998. Compatible generality with run-time types for the Java programming language. In OOPSLA '98, October 1998.
- [15] Chidamber S.R., Kemerer C.F. "Towards a Metric Suite for Object Design", OOPSLA '91, pp. 197-211.
- [16] Chidamber, S.R., Kemerer C.F. "A Metrics Suite for Object-Oriented Design". IEEE Transactions on Software Engineering. June, 1994, pp. 476-493.
- [17] Cockburn A., Williams L. The costs and benefits of pair programming. In eXtreme Programming and Flexible Processes in Software Engineering, XP2000, Cagliari, Italy, June 2000.
- [18] Devanbu P.T., Rosenblum D.S. and Wolf A., "Generating Testing and Analysis tools with Aria" ACM Transaction on Software Engineering and Methodology, Vol. 5, No. 1, January 1996, pp. 42-62.
- [19] D'Sousa R.J. and LeBlanc R. Jr., "Class Testing by Examining Pointers", Journal of Object Oriented Programming, July-August 1994, pp. 33-39.
- [20] <http://www.extremeprogramming.org> 23.3.2004.
- [21] Fowler M. The New Methodology. April 2003. <http://www.martinfowler.com/articles/newMethodology.html#N401>.
- [22] Fowler M, Beck K., Brant J. "Refactoring: Improving the Design of Existing Code". Addison Wesley, 1999.
- [23] Gamma E., Helm R., Johnson R. and Vlissides J. "Design Patterns: Elements and Reusable Object-Oriented Software. Addison-Wesley 1995.
- [24] Haikala I., Märijärvi J. Ohjelmistotuotanto. ISBN 951-762-769-6. Talentum Media Oy 2001, pp. 249-285.
- [25] Harold M.J., McGregor J.D. and Fitzpatrick K.J., "Incremental Testing of Object-Oriented Class Structure", In Proceedings of the 1992 ACM Twentieth Annual Computer Science Conference Proceedings, ACM, February 1992, pp. 68-79.
- [26] Jeffries R.E., Extreme Testing. www.stoemagazine.com Software Testing & Quality Engineering. March/April 1999. pp. 23-26.
- [27] Korel B., Al-Yami A.M., "Assertion-Oriented Test Data Generation", 1996 IEEE 18th International Conference of Software Engineering, pp. 71-80.
- [28] Kung D., Gao J., Hsia P., Toyoshima Y. and Chen C. "A Test Strategy for Object-Oriented Programs", 1995 IEEE 19th Annual International Computer Software and Application Conference, pp. 239-243.
- [29] Kung D., Gao J., Hsia P., Toyoshima Y. and Chen C. "Object State Testing for Object-Oriented Programs", 1995 IEEE 19th Annual International Computer Software and Application Conference, pp. 232-238.
- [30] Muller, M.M., Tichy W.F. Case Study: Extreme Programming in a University Environment. Proceedings of the 23rd International Conference on Software Engineering. IEEE Computer Society, 2001, pp. 537-544.
- [31] Murphy G., Townsend P. and Wong P., "Experiences with Cluster and Class Testing", Communication of the ACM, September 1994, Vol. 37, No 9, pp. 39-47.
- [32] Nosek, J. The case for collaborative programming. Communications of the ACM, 41(3) pp. 105-108, March, 1998.
- [33] Poole, C., Huisman J.W., "Using Extreme Programming in a Maintenance Environment", IEEE Software. November/December 2001, pp. 42-50.
- [34] Saarinen E., Lonka K. Muodonmuutos. ISBN 951-0-24977-7. WSOY 2000, pp. 203-204.
- [35] Thompson, J.B. "The use of a quality assurance tool in the teaching of software engineering principles" Teaching of Software Engineering – Progress Reports, IEE 22 Oct, 1991, pp. 1-6.
- [36] van Deursen, A. "Program Comprehension Risks and Opportunities in Extreme Programming". 8th Working Conference in Reverse Engineering. IEEE Computer Society, 2001, pp. 176-185.
- [37] Wege C., Gerhardt F.. Learn XP: Host a Bootcamp Extreme Programming Examined. Addison-Wesley, 2001.

[38] Williams L., Kessler R., Cunningham W. and Jeffries R. Strengthening the Case for Pair Programming. IEEE Software. July/August 2000a.

[39] Williams L., Kessler R. "All I really need to know about pair programming I learned in kindergarten." Communications of the ACM, 43 pp. 108-114, May 2000b.

[40] Wong W.E., Horgan J.R., London S., Mathur A.P., "Effect of Test Set Minimization on Fault Detection Effectiveness", 1995 IEEE 17th International Conference on Software Engineering, pp. 41-50.

[41] Wright, G. "eXtreme Programming in a Hostile Environment". Third International Conference on Extreme Programming and Flexible Processes in Software Engineering. 2002, pp.48-51.