Experiment about Test-first programming

Matthias M. Müller and Oliver Hagner
Computer Science Department
University of Karlsruhe
Am Fasanengarten 5, 76 128 Karlsruhe, Germany
{muellerm|hagner}@ipd.uka.de

Abstract

Test-first programming is one of the central techniques of Extreme Programming. Programming test-first means (1) write down a test-case before coding and (2) make all the tests executable for regression testing. Thus far, knowledge about test-first programming is limited to experience reports. Nothing is known about the benefits of test-first compared to traditional programming (design, implementation, test). This paper reports about an experiment comparing test-first to traditional programming. It turns out that test-first does not accelerate the implementation and the resulting programs are not more reliable, but test-first seems to support better program understanding.

1 Introduction

Test-first programming is one of the central techniques of Extreme Programming (XP). Test-first combines two general principles: first, write down the test-cases before coding, and second, make them executable for regression testing. The whole process including a small design, nanoincrements, and ongoing testing is also called test-driven development.
Jeffries describes test-first programming in four steps [1]:

1. Find out what has to be done.
2. Write a unit test for the desired new functionality. Pick the smallest increment from the new functionality.
3. Run the unit test. The new functionality is implemented, if the test succeeds. If there are still more unimplemented functionalities, go to step 1. If the test fails, go to step 4. Otherwise, all is done.
4. Fix the immediate problem: maybe it's the fact that the new method wasn't written yet. Maybe the method doesn't quite work. Fix whatever it is. Go to step 3.

The tasks of test-first are manifold within the framework of XP rules and practices: to ensure that programmed features cannot be lost, to force the developers to think about testable code and to write down unit tests, to automate test execution, to prevent the program from regressing by ongoing retesting, and to provide a test-suite as a basis for refactoring.
The list of goals to attain with test-first is not less important: to develop programs that are more capable of accepting changes, to program faster, to increase confidence of the developer

and the customer, by seeing all the tests run correctly, to reduce defect rates in such a way that the successive test-cycle overhead becomes neglectable, and to understand the program better.

While experience shows that it is difficult for the beginner to adopt to test-first [6, 10], the above formulated goals of test-first are still left for evaluation. One of the challenges of studying test-first is its embedding within XP. This embedding makes it difficult to show the effects of test-first without being blurred by other practices such as pairprogramming or a simple design. A solution to this problem would be an experiment in which XP is applied twice: with test-first and without test-first. The result would be an indirect evaluation of test-first. But this kind of experiment is too difficult and too expensive. To solve this problem, test-first was extracted from XP and evaluated on its own. Now, the experiment focuses on a single programmer with his traditional development process (design, implementation, test). The authors wanted to know if there are any advantages or disadvantages for a single programmer when switching from the traditional development process to test-first.

Concerning test-first, this paper focuses on (1) the programming efficiency (how fast someone obtains a solution), (2) the reliability of the resultant code (how many failures can be observed), and (3) program understanding (measured as proper calls of existing methods). The experiment focused neither on design aspects of the delivered solution (e.g. how changeable is the design) nor on the benefits of test-first in the long run (e.g. shorter test-cycles, shorter time to market, or defect rates in the production code).

The experiment was conducted as part of an XP course held with CS graduate students. The participants were divided into two groups: the experiment group which used test-first and the control group which followed the traditional development process. Both groups had to implement the main class of a graph library containing only the method declarations but not the method bodies. The subjects' work was divided into two phases. During the first phase, the implementation-phase, the subjects implemented the solution up to the point where they thought that their implementation was correct. The second phase, the acceptance-test phase, involved passing an acceptance-test. If the acceptance-test failed, the subject got the output of the test and had to fix the faults. This was iterated until the acceptance-test succeeded. Only then, was the solution accepted. The goal of the acceptance-test was to ensure a minimum of code quality of the final solutions.

The results show no difference between the two groups concerning the overall problem solving time and the final reliability of the produced results. But the test-first group had less errors when reusing an existing method more than once. The last observation is statistically significant with $p = 0.09$.

We also compared both groups after the first phase, that is, before the acceptance-test took place. Again, we measured no difference in problem solving time but the programs of the test-first group were less reliable (significance $p = 0.03$).

Looking at these results, we drew the following conclusions. Writing programs with test-first neither leads earlier to a solution nor provides more reliable results. On the other hand, using test-first increases program understanding measured as a proper reuse of existing interfaces.

An open question remains from our study. Why are the programs of the test-first group less reliable than those of the control group at the end of the first phase? Possible explanations are the following. (1) Were the subjects insufficiently experienced in the use of the test-first approach? That is, was their experience of test-first too small to see that a bit more testing was needed? (2) Did they lull themselves in a false sense of security? Did the ongoing execution of the tests suggest a code quality that did not exist? Or (3) did they not have

any respect for the acceptance-test as a result of the ongoing testing? This could be possible because they knew an acceptance-test was to come and took it into account as an additional quality measure at the end of the development process.
So far, knowledge about XP is limited to experience reports. Only pair programming has been investigated to a certain extend [2, 11, 4, 12]. This paper starts an evaluation of test-first. In fact, it isolates test-first from the other techniques of XP, but later, when we have an understanding of all techniques of XP, we can combine them and study their combined behavior.
The reminder of this paper presents the experimental settings in Section 2 and the measured results and their discussion in Section 3. A summary of the paper is outlined in Section 4.

2 Description of the experiment

2.1 Design of the experiment

The experiment uses a single-factor, posttest-only, inter-subject design [3]. The controlled independent variable was whether the experimental subjects program test-first (experiment or test group, subsequently called "TFG") or use the traditional development process (control group, subsequently called "CG"). Each subject of either group solved the same task and worked under the same conditions. The observed dependent variables for each subject were a variety of measurements of the development process (in particular total time), and various measurements of the delivered product (in particular program reliability and number of reused methods).

2.2 Subjects

Overall, 19 persons participated in the experiment, 10 in the TFG and 9 in the CG. All of them were male Computer Science graduate students who had just previously participated in a one-semester graduate lab course introducing the XP methodology, along with a larger programming assignment. This course covered technics from XP such as pair programming, test-first, refactoring and planning.
On average, these 19 students were in their 6th semester at the university, they had a median programming experience of 8 years total and estimated that they need a median of 3 months to program their largest program of about 5000 LOC. None of these measures was significantly different between the two groups. During the experiment all of the participants used Java with jUnit [9] as they did in the XP course.
None of the participants dropped out of the experiment so that all work was available for the evaluation.

2.3 Experiment task

The task to be solved in this experiment is called "GraphBase". It consists of implementing the main class of a given graph library [7] containing only the method declarations and method comments but not the method bodies. There are methods to add vertices and edges and to clear and to clone a whole graph. Other methods are only accessor methods, e.g. to show the number of vertices or edges, to find an edge between two given vertices or to test if the graph is empty, weighted or directed.

Each subject is told that the original code of GraphBase was lost and, because there is no backup, that it should be reimplemented by using the rest of the given graph library. The requirements for this task were described thoroughly in natural language. The subjects were expected to work and to test on their own until they thought, they had finished. They were also told that they had to pass an acceptance-test to ensure some quality of their solutions.

2.4 Experimental procedure

The experiment was run between July 2001 and August 2001, mostly during the semester breaks. Most of the subjects started around 9:30 in the morning. The experiment materials were printed on paper and consisted of two parts. Part one was issued at the start of the experiment and contained a task description. The second part was handed out at the end of the experiment. It contained questions about understandability of the documentation and asked for personal ratings concerning program understanding and reliability of the resultant program.
The subjects worked on the task using their specific Unix account from the XP course. The account was changed for the experiment to provide the automatic monitoring infrastructure. It nonintrusively recorded login/logout times, all compiled source versions and all output from each program run. The recorded source code versions included the GraphBase-class, all written test-classes, and all other Java-classes the subjects wrote in order to solve the experiment task. The subjects could modify the account setup as necessary. The source code of the graph library except for the GraphBase-method-bodies was provided to the subjects.
Subject's work was divided into two phases.

Implementation phase (IP), during which the subjects solved their assignment until they thought that their program would run correctly. This phase finished with their call for the acceptance-test.

Acceptance-test phase (AP), during which the subjects had to fix the faults that caused the acceptance-test to fail.

The acceptance-test itself is programmed using jUnit. It consists of 20 test cases with 522 assertions which build some graphs and check their structure for correctness. If an assertion fails it generates an output with the expected and the actual value, aborts the actual test and continues with the next one.

2.5 Power analysis

Cohen [5] stresses the importance of power analysis to get a closer look at the quality of a statistical hypotheses test.
The power of a statistical test of a null hypotheses is the probability that it will yield statistically significant results. It is defined as the probability that it will lead to the rejection of the null hypotheses, i.e., the probability that it will result in the conclusion that the phenomenom exists under the premise that the phenomenom is really existent. Statistically speaking, $1 - power$ is the probability for an error of the second kind.
In our experiment, we used groups with $n_{CG} = 9$ and $n_{TFG} = 10$ subjects. Due to this small number of data-points, we restrict our analysis to find only large effects. In this case, Cohen suggests an effect size of $ES = 0.8$. We set the significance level of the one-sided test to

$\alpha = 0.1$. Thus, the power analysis with a t-distribution yields a power of 0.645 [8]. That is, we have only a 64.5% chance to find a difference between the groups!

According to Cohen, the experiment has a poor power. He argues that only experiments with a power of more than 0.8 have a real chance to reveal any effect. Therefore, it is quite reasonable, that according to this poor power, our experiment has not the chance to show an effect, even if there is one. But, as we could not acquire any more subjects for the experiment, we had to live with this drawback.

2.6 Threats to internal validity

The control of the independent variable is threatened by the fact that it was not technically controlled. The subjects in the TFG were told in the requirements to use test-first programming as they did during the whole XP course. During the experiment, the subjects were asked by the experimentator several times if they got along with the test-first process. This question was affirmed by all subjects of the TFG.

2.7 Threats to external validity

There are two important problems for the external validity (generalizability) of the experiment. First, professional software engineers may have different levels of skill and experience than the participants, which might make the results too optimistic or too pessimistic: both higher and lower levels will occur, because the students are more skilled than most of the non-computer-scientists that frequently start working as programmers. Better skilled subjects might leave less room for improvement which might reduce the difference between the groups, but higher experience may also sharpen the eye as to where improvements are most desirable or most easily achieved. Conversely, lower skill may leave more room for improvement but may also impede applying test-first correctly at all. Second, the XP education of the subjects occurred only a short time ago. It is conceivable that the test-first usage of these persons had not yet stabilized and the mid-term benefits would be higher than observed in the experiment. Futhermore, work conditions different than those found in the experiment may positively or negatively influence the effectiveness of test-first.

3 Results and Discussion

This section investigates the following topics: problem solving time, reliability of the produced results, code reuse, and tester quality.

Box plots are used to show the results of the measurements. The filled boxes within a plot contain 50% of the data points. The lower (upper) border of the box marks the 25% (75%) quantile. The left (right) t-bar shows the 10% (90%) quantile. The median is marked with a thick dot (•). The M with the dashed line mark the mean value within a range of one standard error on each side. The variance of a data-distribution is measured as fraction of the 75%- to the 25%-quantile.

Significance was calculated with the Wilcoxon-Test where the significance p denotes the probability that the observed difference is due by chance.

3.1 Problem solving time

For the evaluation of the problem solving time, the following times and fractions were compared: times spent for the whole task, figure 1, the time spent in the implementation phase, figure 2, and the portion of the implementation phase to the whole assignment, figure 3.

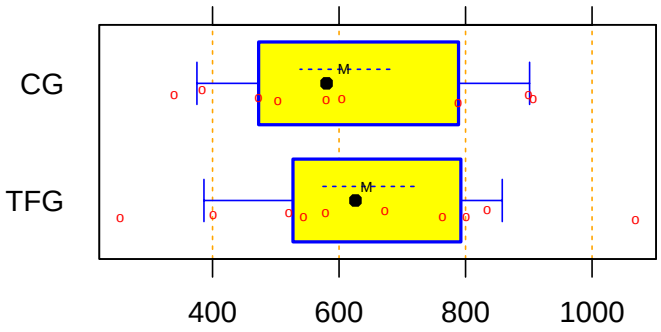

Figure 1: Overall working time in minutes.

Figure 1 shows only a small difference between both groups. This means the programming effort increases slightly when switching from traditional programming to test-first.
But how is the behavior of the two groups prior to the first acceptance-test? Figure 2 shows the time spent in the IP.

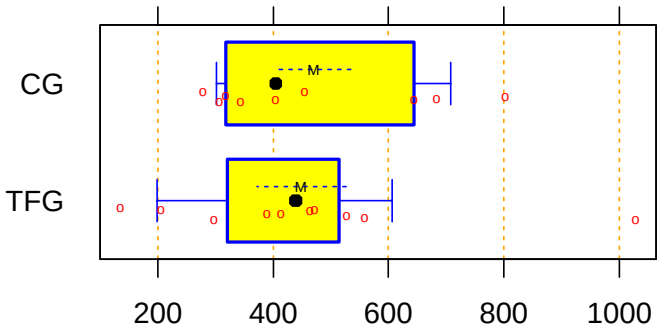

Figure 2: Working time in minutes for the implementation-phase.

And again, the box plots show only a slight difference between the medians. But when we look at the portion of the IP related to the whole assignment, see figure 3, we see that the TFG spent relatively less time in the IP. While there is no statistical evidence ($p = 0.158$) for this observation, the effect is quite visible.
To sum up, the TFG was not more efficient than the CG, as we originally expected. But the TFG tended to spend less time for the IP related to the overall working time.

3.2 Reliability

In this experiment, reliability was measured as portion of the passed assertions related to all possible executable assertions in the test. The initial behavior of jUnit had to be adjusted to

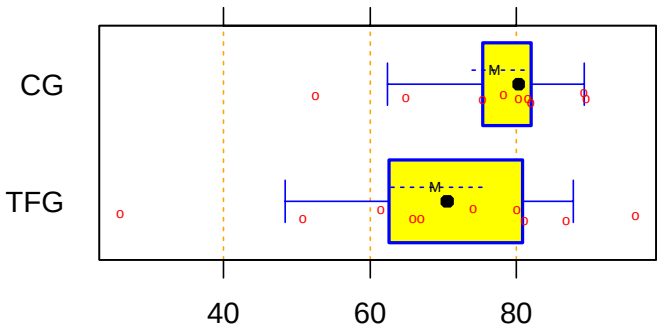

Figure 3: Portion of implementation-phase in percent to the overall working time.

count all failed assertions. That is, jUnit was modified in such a way that it did not abort a test after a failed assertion. Instead, it continued the test-case so that all assertions were executed. The failed assertions were counted and printed out at the end of the test-run.
Reliability was measured for two programs: the acceptance-test and a random-test with 727, 190 method invocations and about 7.5 million assertions. The reference implementation runs about seven seconds for the acceptance-test and about 150 seconds for the random-test. The random-test calls the methods of the implementation randomly and compares the resulting data-structure with the one built by the reference implementation. Deviations in the structure are caught by subsequent assertions. Each method got a weight to ensure that hot methods, such as insert-edge were called more often than more unusual methods, such as clone-graph.
First, we look at the reliability of the random-test for the final programs, see figure 4.

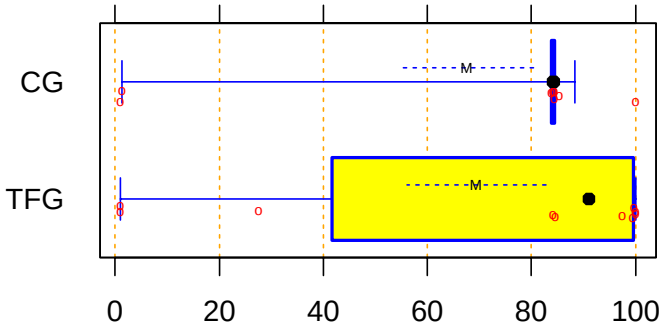

Figure 4: Reliability of final programs for random-test in percent.

Only three programs of the TFG are less reliable than the median of 91% of the CG. Even if the median of the CG is smaller than that of the TFG (84% to 91%, respectively), the observed difference is with $p = 0.2$ due by chance. The variance of the data-distributions differs with 2.38 for TFG to 1.00 for the CG. To interpret the difference in the medians as a trend towards a better reliability of test-first is dangerous because of the large variance of the data-points. Nevertheless, it is remarkable that five programs of the TFG achieve a reliability over 96% compared to only one program of the CG.
Now, the programs right after the IP are examined and the question is asked, what would

have happened if the acceptance-test had been omitted? This question is interesting in as much as these programs represent the output of the pure test-first process without further modification or enhancement by any external quality control. These programs represent the versions the subjects are most confident of concerning accurateness. Reliability of the first run of the acceptance-test is discussed, see figure 5.

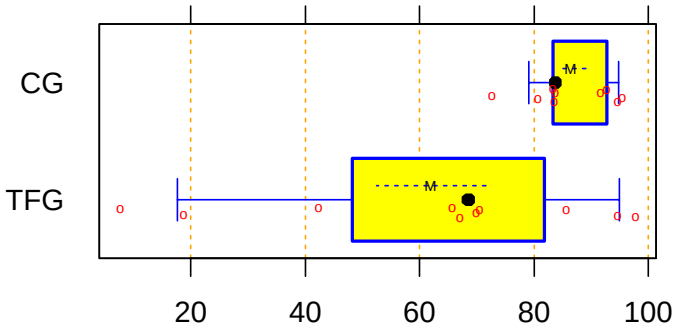

Figure 5: Reliability of programs for acceptance-test after the implementation phase in percent ($p = 0.03$).

The reliability of the TFG is significantly lower $p = 0.03$ than that for the CG. Except for three programs, all programs in the TFG are less reliable than the worst one in the CG and two are even worse than 20%.

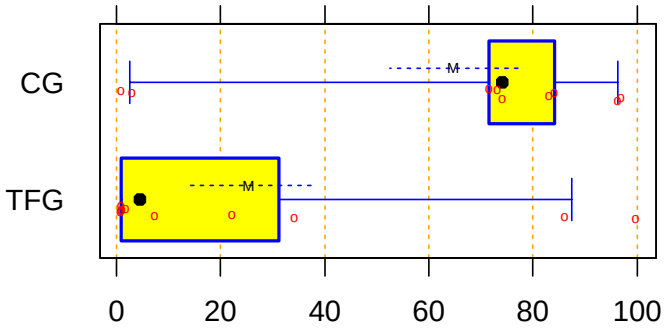

Figure 6: Reliability of programs for random-test after the implementation phase in percent ($p = 0.067$).

The results for the random-test are quite similar as shown in figure 6. The lower reliability of the programs of the TFG is significant with $p = 0.067$.
But what is the reason for this difference? Is it because of the ongoing testing that lulls the developer in a false sense of security? False sense, because all of the tests run at 100% but do not cover the main faults. Or is it, because the subjects of the TFG are so used to testing that the acceptance-test degenerates to just another test and looses its importance as a control instance? And is it for that reason, that the subjects of the CG are more motivated to have less faults entering the acceptance-test phase which leads consequently to more reliable programs? But these reasons are all speculations. So far, we do not know it.

At this point, it is of no further importance and not surprising that the TFG has a significant larger improvement in reliability during the acceptance-test phase than the CG because it follows quite naturally from the above presented results.

3.3 Code reuse

Examining code reuse might lead to some conclusions about program understanding. Three measures were used to get a perception of it. These are (1) the number of reused methods, (2) the number of failed method calls, and (3) the number of method calls that failed at least twice. The data-sets of the last two measures were obtained with silent assertions inserted into the existing graph library. Their output was written to a log file without notice of the subjects.
Figures 7 and 8 show the results for the number of reused methods and the number of failed method calls, respectively.

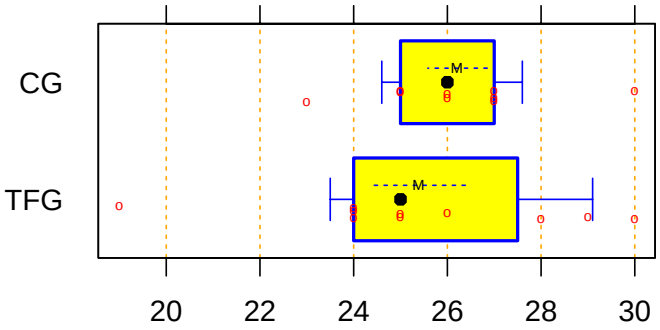

Figure 7: Number of reused methods.

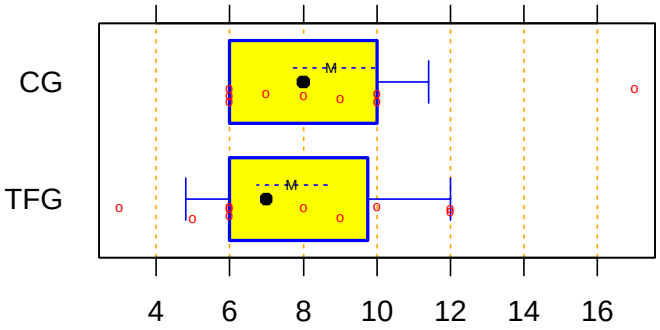

Figure 8: Number of assertions that failed at least once.

Neither figure shows a remarkable difference between the two measures. But there is a difference between the two groups when studying the number of method calls that failed at least twice, see figure 9.
The test-first group had significantly less errors ($p = 0.086$) by reusing a method more than once. By combining these two observations into a single result the authors have to say: test-

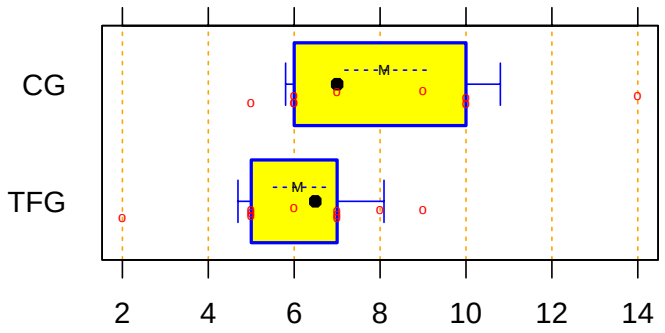

Figure 9: Number of methods that failed more than once ($p = 0.086$).

first does not aid the developer in a proper usage of existing code but it guides him through ongoing testing in the process of fixing the fault.

3.4 Tester quality

The quality of the subjects' tester-classes is the last studied measure. Branch coverage was used to measure subject's tester quality. To get an unbiased comparison, each tester executed the reference implementation and not the subject's implementation. In order to collect this measure, the reference implementation was modified. Each ground block of the reference implementation got a distinct number. On entry of the ground block, this number was written out. The different numbers were collected and summed up for each tester. The reference implementation had a total of 45 branches. This measure was collected at the end of the experiment, after the AP. Figure 10 shows the fraction of the actually executed branches related to all possible branches of the reference implementation.

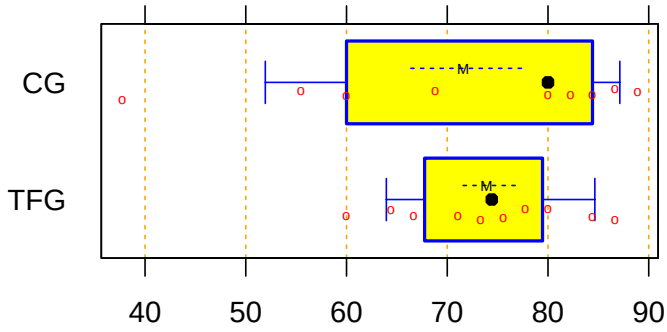

Figure 10: Percentages of branches the subjects' testers covered in the reference implementation.

Originally, we assumed that the testers of the TFG have a better branch-coverage than those of the CG. This assumption based on the hypotheses that the TFG delivered more reliable programs. But as our hypotheses does not hold to some extent, we do not expect our assumption to hold either. The medians of both groups differ with 80% for CG to 74% for TFG. This difference is with $p = 0.451$ most likely due by chance. The following observation

is remarkable. Despite the fact, that 80% of the TFG's data-points are smaller than the median of the CG, the TFG had a slightly better reliability in their final implementations, see figure 4.

4 Conclusions

This paper presented an experiment about test-first programming conducted at the University of Karlsruhe at the end of the summer lectures 2001. Subjects were CS graduate students who participated in an Extreme Programming practical training course. The study compared test-first programming with the traditional development process. In particular, it investigated the influence of test-first programming on (1) programming speed, (2) program reliability, and (3) program understanding measured as proper reuse of existing methods.
The experiment data led to the following observations.

1. If a developer switches from traditional development to test-first programming, he does not program necessarily faster. That is, he does not arrive at a solution more quickly.

2. Test-first pays off only slightly in terms of increased reliability. In fact, there were five programs developed with test-first with a reliability over 96% compared to one program in the control group. But this result is blurred by the large variance of the data-points. Concentrating on the program versions after the implementation-phase, the result just turns around. The test-first group has significantly less reliable programs than the control group. So far, we do not know, if this effect is caused by a false sense of security, less importance of the acceptance-test for the test-first group, or if it is quite simply a result of too little testing.

3. Test-first programmers reuse existing methods faster correctly. This is caused by the ongoing testing strategy of test-first. Once a failure is found, it is indicated by a test-case and, while fixing the fault, the developer learns how to use the method or interface correctly.

Despite the observed results, this study is far from being a complete evaluation of test-first programming. The authors encourage other researchers to repeat the experiment or to conduct a similar one in order to extend the knowledge about test-first.

References

[1] Code unit test first. http://www.c2.com/cgi/wiki?CodeUnitTestFirst.

[2] D. Bisant and J. Lyle. A two-person inspection method to improve programming productivity. *IEEE Transactions on Software Engineering*, 15(10):1294–1304, October 1989.

[3] L. B. Christensen. *Experimental Methodology.* Allyn and Bacon, 1994.

[4] A. Cockburn and L. Williams. The costs and benefits of pair programming. In *eXtreme Programming and Flexible Processes in Software Engineering, XP2000*, Cagliari, Italy, June 2000.

[5] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Academic Press, 1977.

[6] R. Gittins, S. Hope, and I. Williams. Qualitative studies of xp in a medium sized business. In *Proceedings of the 2nd Conference on eXtreme Programming and Flexible Processes in Software Engineering*, Cagliari, Italy, May 2001.

[7] David Goldschmidt. Design and implementation of a generic graph container in java. Master's thesis, Rensselaer Polytechnic Institute in Tray, New York, April 1998.

[8] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[9] junit.org. http://www.junit.org/.

[10] M. Müller and W. Tichy. Case study: Extreme programming in a university environment. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 537–544, Toronto, Canada, May 2001.

[11] J. Nosek. The case for collaborative programming. *Communications of the ACM*, 41(3):105–108, March 1998.

[12] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair-programming. *IEEE Software*, pages 19–25, July/August 2000.