The Effect of Experience on the Test-Driven Development Process *

Matthias M. Müller
Systeme Infrastruktur Support GmbH
EnBW AG
Durlacher Allee 93
D-76131 Karlsruhe
Germany
matt.mueller@enbw.com

Andreas Höfer
Fakultät für Informatik
Universität Karlsruhe
Am Fasanengarten 5
D-76131 Karlsruhe
Germany
ahoefer@ira.uka.de

Abstract

We conducted a quasi-experiment to compare the characteristics of experts' and novices' test-driven development processes. Our novices were 11 computers science students who participated in an Extreme Programming lab course, the expert group consisted of 7 professionals who had industrial experience in test-driven development. The novices as well as two of the experts worked in a laboratory environment whereas the remaining five experts worked in their office. The experts complied more to the rules of test-driven development and had shorter test-cycles than the novices. The tests written by the experts were of higher quality in terms of statement and block coverage as well. All reported results are statistically significant on the 5 percent level. We conclude that the results of studies which evaluate performance of test-driven development using subjects inexperienced in TDD are not easily generalisable.

Keywords: test-driven development, process, quasi-experiment, experts, novices

1 Introduction

Test-driven development (TDD) alongside pair programming is one of the central programming techniques of Extreme Programming (XP) and has been investigated in several studies. The qualification of the subjects in these studies ranges between two extremes: On the one hand there are (TDD) experts, who have major industrial programming experience and have applied TDD for several years; on the other hand there are (TDD) novices, who have no or little industrial programming experience and have just been trained in TDD techniques. Experts usually are professional programmers, who work or have worked on an XP-project. Novices, in contrast, are mostly students, who have just learned TDD in an XP- or Java-course.

It seems obvious that the difference in expertise between these two types of subjects is reflected in their TDD processes. Yet, it is not obvious what the actual differences are and how the TDD processes of experts and novices are characterised. So far, nobody has examined how the characteristics of an actual TDD process look like, although the knowledge about the characteristics of experts' and novices' TDD processes is helpful for training in TDD as well as for the assessment of studies about TDD. Hence, we have conducted a quasi-experiment which compares the TDD processes of 7 experts to the process data of 11 novices. Our expert group consists of professional programmers with industrial TDD experience, whereas our novice group comprises students who finished an XP lab course at our university. We collected detailed process data transparently

*The study and the second author were sponsored by the German Research Foundation (DFG), project "Leicht" Ti 264/8-1.

for each of the participants. The collected data includes for example the point in time of each unit-test invocation, the development behaviour in the Eclipse programming environment, and all changes made to the application and test code. The data was used to determine the conformance of application-code changes to the rules implied by test-driven development and to identify differences between novices' and experts' TDD processes.

This paper is organised as follows: After a review of the relevant literature in Section 2 we explain the formalised TDD rules and our data collection method in Section 3 and Section 4. One aim of the latter two sections is to explain the reader the data collection mechanism and how we obtained the results from the quasi-experiment. The second aim of these sections is to provide enough information for other scientist to redo our analysis with different subjects in different locations. The quasi-experiment itself is described in Section 5, the results follow in Section 6. Section 7 discusses threats to validity and Section 8 presents our conclusions.

2 Related Work

In recent years, several studies have been published which compared TDD to other development processes. Müller and Hagner [18] compared test-first programming to a traditional test-last approach. All of their 19 subjects – 10 in the test-first group, 9 in the control group – were sixth-semester computer science students. Both groups had to complete the main class of a graph library. The subjects had to work alone in a laboratory environment. The researchers compared problem solving time, reliability of the programs and correct calls of existing methods. The results showed no difference concerning problem solving time and reliability of the programs but the test-first group made less errors when reusing existing methods more than once.

George and Williams [9] compared TDD with a waterfall-like development process in a controlled experiment. The participants were 24 professional developers programming in their usual working environments. They were assigned to work in pairs and asked to develop a bowling game application. The TDD pairs' programs passed 18 percent more test cases of a black-box test than the control group pairs' programs but the TDD pairs also needed 16 percent more time for development. A moderate but statistically significant correlation between the development time and the resulting code quality was found.

Pančur et al. [20] conducted an experiment with 38 undergraduate computer science students in their fourth year. They compared TDD to an highly iterative test-last development process. The results of their work are preliminary and show hardly any difference between the test-first and the test-last approach.

Geras et al. [10] studied the differences between test-first and test-last programming concerning the amount of unplanned failures. Their subjects were professional programmers, which were assigned to write two programs in the domain of business information systems. Due to the small number of participants the results of this study remain inconclusive.

Erdogmus et al. [6] conducted an empirical study, which compared test-first to test-last programming. The 11 participants in the test-first and the 13 participants in the test-last (control) group were third-year computer science students from an eight-week Java course. The programming task was the same bowling game application that was also used in George's and Williams' study mentioned above. Their results indicate that the test-first programmers wrote more tests per unit of effort but there was no difference between the two groups concerning quality of the code and productivity. A second finding of their study is that more tests lead to a higher productivity.

Canfora et al. [3] performed a controlled experiment with 28 professionals. They compared TDD to a test-last approach. In their analysis they report statistical evidence that TDD is more time-consuming but does not produce more assertions in the test code than test-last programming.

Bhat and Nagappan report on two industrial case studies performed at two divisions of Microsoft [2]. In their case studies, they describe two projects using TDD and compare them with similar non-TDD projects at Microsoft. They conclude that development using TDD took more time but the resulting quality in terms of defects per kilo lines of code was at least two times better than the quality of comparable non-TDD projects.

Another relevant paper was written by Erdogmus and Wang [7]. They present a tool that uses data collected by Hackystat [11] sensors to compute metrics about the programmer's TDD process. These metrics include the total time spent on development between two successful test executions as well as the ratio of test code to production code. In opposition to our work, the tool was not used to check process conformance in an empirical study on TDD.

3 TDD Recognition

To be able to decide whether a change of the application code conforms to test-driven development or not the TDD process has to defined. The presented definition is not our own. It is based on definitions found in literature.

3.1 TDD Rules

Beck [1, p. IX] explains TDD as conformance to two rules:

- Write new code only if an automated test has failed.
- Eliminate duplications.

He derives three different *tasks of programming* from these two rules. For our definition of those tasks of programming, we use the terms *application* code and *test* code. *Application* code is later shipped to the customer while *test* code is only used for unit testing purposes. Furthermore, we use the terms *passing* unit test and *failing* unit test. A *passing* unit test is a unit test where all tests pass (green JUnit bar) whereas a *failing* unit test is defined as a unit test where one or more tests fail (red JUnit bar). With these terms, we define the three tasks of programming as follows:

Test task Write a test that fails. If necessary, write few lines of application code such that your test compiles[1]. A test task starts with a passing unit test and ends with a failing one.

Application task Write the application code to make all tests pass. An application task starts with a failing unit test and ends with a passing one.

Refactoring task Eliminate all duplication from the test and application code which has been introduced during the two preceding tasks. A refactoring task starts and ends with a passing unit test.

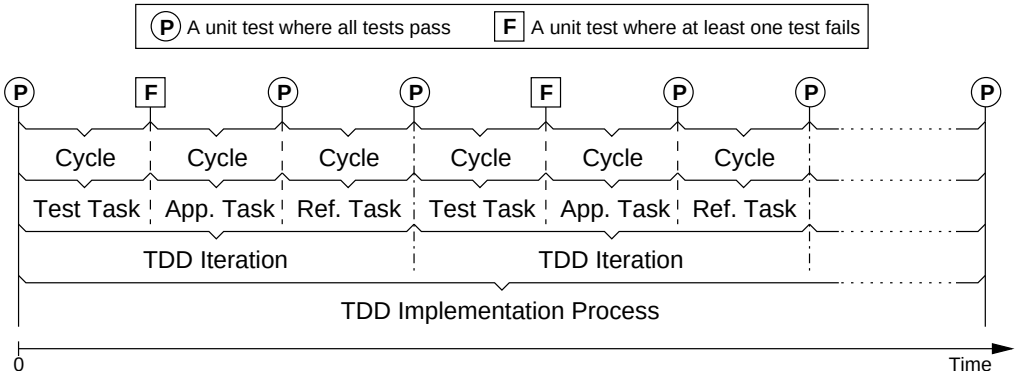


Figure 1: The TDD implementation process.

[1] Beck allows a test that does not even compile. We omit this case because we can not measure this kind of error in our data evaluation framework presented in section 4.

In its ideal form each task maps to exactly one cycle which is the time frame between two successive unit-test executions. The mapping of tasks on cycles in an idealised TDD process leads to an one-to-one relationship between them as depicted in Figure 1. Though, it may happen that a task comprises more than one cycle, e. g. if the developer breaks a test during a refactoring. Test task, application task and refactoring task compose an ideal TDD iteration[2]. Note that the order of the tasks is important for process conformance. Link [15, p. 12] suggests that the length of a TDD iteration should not exceed fifteen minutes. Finally, the TDD implementation process itself consists of a sequence of TDD iterations.

Based on the definition of the TDD implementation process we define three rules. An implementation process is said to be TDD conform if every code change or refactoring complies to one of the following TDD rules[3]:

Rule 1 A change in application code is only allowed in methods which were previously called by a test that failed and thereby caused the failure of the whole unit test.

Rule 2 A new method can only be introduced if it is later called by a test that fails thus causing the whole unit test to fail.

Rule 3 A refactoring changes the structure of the code but not its observable behaviour (see Fowler [8, pp. 53]).

Conformance to the first two rules can be determined automatically if all the changes made by the developer have been recorded. However, the evaluation whether a given set of code changes belongs to a refactoring or not is not automatically decidable, yet. Thus, every framework that tries to evaluate conformance to the TDD process needs some human intervention to classify whether a change belongs to a refactoring or not.

3.2 Formalisation of TDD Rules

In order to calculate whether an application-code change complies to Rule 1 or 2 we have to distinguish between different points in time: the point in time of a modification t_{mod}; the point in time t_{prev} of the unit-test invocation preceding t_{mod}; the point in time t_{next} of the unit-test invocation following t_{mod}. $Methods(t)$ and $Tests(t)$ describe the set of application and test methods available at time t, respectively. $Changes(t_1, t_2)$ is the set of all changes from point in time t_1 to t_2. $calls(tm, m, t)$ evaluates to true if the test method tm calls the application-code method m at time t. The predicate $fail(tm, m, t)$ evaluates to true if test method tm fails and $calls(tm, m, t)$ is true.

Rule 1 requires every application-code change in a method to be preceded by a unit-test invocation which executed a test method that failed and called the changed application method. *failureBefore* models this circumstance:

$$failureBefore(m, t_{mod}) := \exists\, tm \in Tests(t_{prev}) : fail(tm, m, t_{prev}) = True$$

Rule 2 describes another TDD compliant change: the introduction of a new method.

$$newMethod(m, t_{mod}) := \nexists\, t < t_{prev} : m \in Methods(t)$$

However, this new method has to be called by a test method that fails at the next unit test:

$$failureAfter(m, t_{mod}) := \exists\, tm \in Tests(t_{next}) : fail(tm, m, t_{next}) = True$$

To sum up, a change of an application method m at time t_{mod} conforms to TDD if one of the following rules applies:

$$TDDRule_1 := failureBefore(m, t_{mod})$$
$$TDDRule_2 := newMethod(m, t_{mod}) \wedge failureAfter(m, t_{mod})$$

[2] In some cases the refactoring task may be skipped, e. g. if there was no duplication introduced.

[3] Such a strict conformance of all changes to the rules may not be achievable in practice. Therefore, introduction of a threshold may be desirable.

In addition, we also consider the notion of a *weak* TDD process. In a weak TDD process, the developer does not attempt to get a failed unit test after each test-code change. Instead, he skips the unit-test invocation part and starts to modify the method in the application code right away. *testModBefore* accounts for this situation:

$$testModBefore(m, t_{mod}) := \exists\, c \in Changes(t_{prev}, t_{mod}) : c = (tm, t) \wedge calls(tm, m, t_{next})$$

A change on which none of the TDD rules evaluates to true but which complies to *testModBefore* is said to be *weak TDD*.

4 Data Collection

The decision whether a given change of the application code follows the rules implied by test-driven development requires the analysis of multiple process and application measures. We developed a system which collects these measures transparently for a developer and which inserts the desired measures into a database. Figure 2 shows the components of our system. It consists of a plug-ins part and a back-end part. The plug-ins part is responsible for data collection during a programming session. The back-end comprises the postprocessor which prepares the data and fills it into the database and the evaluator. The evaluator checks whether changes are made in conformance to TDD.

Division of the system into plug-ins part and back-end part has the advantage of an easy to install data-collection mechanism with almost no prerequisites. The only requirement is a running Eclipse programming environment. One problem of every field study is the installation of data-collection tools in the context of changing work environments and different operating systems. Thus, installation of Eclipse plug-ins is an easy means to accomplish the data-collection task with almost no effort for the experimenter and the participant.

The remaining part of this section presents the Eclipse plug-ins and the back-end part of our system in more detail.

4.1 Plug-ins

The two Eclipse plug-ins are called *user-action logger* and *JUnit-action logger*. The user-action logger performs two tasks. First, it writes window actions within Eclipse to the *user-actions* log. The logger captures focus changes from one part of the Eclipse GUI to another part of the GUI. For each focus change, the timestamp is recorded and the part which received the focus is described. The second task of the user-action logger is to store backup copies of each modified file into the *file store*. Storage of backup copies is triggered when the user performs a save-file or compile operation. The file store uses the same directory structure as the project that is monitored.

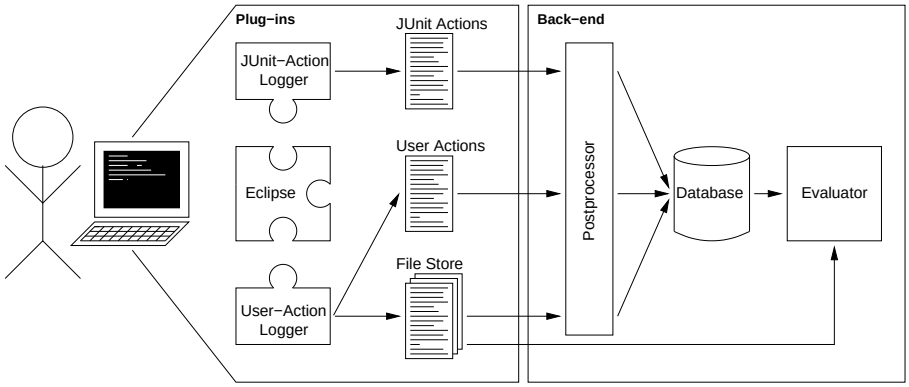


Figure 2: System overview.

The only difference between the actual project sources and the file-store sources is an attached timestamp which indicates the point in time the storage took place.

The JUnit-action logger protocols JUnit invocations in the *JUnit-actions* log. The protocol contains the start and stop time of the whole JUnit run, the number of executed test methods, and for each test method the information whether it passed or failed.

The usage of the plug-ins during development is transparent for the developer. Thus, it is possible to collect data without the knowledge of the developer. However, as the collected data provides details on the personal development behaviour the developer should have agreed on the usage of the two plug-ins. We have signed agreements from all our participants which allowed us to monitor their development process.

4.2 Postprocessor

Before post-processing takes place the sources in the file store are pretty-printed using Jalopy [13]. Jalopy removes all comments and makes the layout of the source files consistent. Then, the postprocessor prepares the data to be suitable for storage in a relational database. It extracts the following information:

- A list of files which were modified during development and a classification of them into *test* or *application* code.
- The differences between two successive versions of a file.
- For each version of a file, the begin- and end-lines of each method.
- For each JUnit invocation: the timestamp, the number of tests, each test method together with the information whether it passed or failed, and the point in time.
- Trace information from the unit tests.

Classification of files in the file list is done according to the name of the file: a 'Test', 'Mock', or 'Dummy' in the file name indicates a test file, all other files are classified as application files. Differences between two successive file versions are calculated using the Unix diff-command. The trace information is necessary in order to decide which application method was executed by which test method during a specific unit test. Trace information is calculated for every user-triggered JUnit invocation. First, the file system that represents the state of the Java classes at the point in time of the JUnit invocation is reconstructed. Reconstruction is based upon the file-version timestamps. The Java sources are then enriched with call backs at each method entry which log the package, class, and name of the method. The system is compiled and the JUnit tests are executed. The call hierachy for each executed unit-test method is constructed from the log. Finally, the timestamp of the JUnit run, the name of the user who triggered it, the name of the test method, and the names of the application methods which are called by this test method are written into the database.

4.3 Evaluator

The TDD evaluator works with the data in the database as well as with the Java files in the file store. It evaluates semiautomatically whether changes made in the application code conform to the process requirements imposed by TDD. For example, TDD rule 1 states that every application-code change should be preceeded by a unit test that failed. Assume the change occurred at line l in file f at time t. The check for conformance to this TDD rule requires 5 steps.

1. Check whether file f belongs to the application code. If yes, proceed.
2. Search for the name of the method m that surrounds line l in file f at time t.
3. Search for the point in time t_{prev} of the latest JUnit invocation triggered before t. If t_{prev} is found, proceed.

4. Check whether a unit-test method tm exists in the JUnit invocation at time t_{prev} that called m. If tm exists, proceed.

5. Check whether tm failed. If tm failed, the change at line l in file f at time t conforms to TDD.

Changes which cannot be classified automatically by the evaluator might belong to a refactoring. In this case, the evaluator shows the difference between the original and the modified file. Now, it is the task of the experimenter to decide whether this change belongs to a refactoring or not. Manual classification of changes took less time than we had expected. The largest number of changes which had to be classified manually for a participant was 34. But for 9 of our 18 participants not more than ten changes had to be inspected manually. The more a developer's implementation process conforms to TDD the less effort for manual inspection is necessary.

For a full automation of the TDD qualification process refactorings have to be recognised. Therefore, one possible improvement for future versions of the evaluator would be to use the renaming detector from Malpohl et al. [17].

5 The Study

The following subsections report on data which has been collected from May 2005 to January 2006.

5.1 Participants

The novice group consisted of 18 students. All of them took part in an XP lab course [19] in which they learned the basic principles of TDD and applied them in a project week. They participated in the experiment in order to get their course credits. They were on average in their fourth year of study and had on average 6 years of programming experience including 2.4 years experience in Java. Only one of them had tried TDD before the programming lab course and four of them had used JUnit before. The experts were seven professional developers. Two of them took part in the experiment because of a personal invitation, two responded to an anonymous poll in a local XP user group mailing list, and the remaining three were selected by a participating company. As the experts were not selected randomly from a defined group of software developers this study is a quasi- and not a randomized experiment [21, p. 14].

All members of the expert group were paid for their participation[4]. The experts received no additional TDD training, because they already had a high level of work experience in TDD: On average, they had 7.3 years industrial programming experience including on average 3.4 years experience with TDD, 4.3 years experience with JUnit, and 6.2 years experience with Java. Table 1 in the appendix lists the programming experience for each participant.

We dropped 7 from the originally 18 novice data-sets because they were not able to develop a solution to the problem within the allocated time frame of eight hours[5]. These drop-outs do not introduce a bias into our data set because ignoring them means to decrease the difference between the experts and the novices. Further more, if the seven novice data-sets had been used in the analysis, another alternative explanation for the difference between experts and novices had been introduced: missing implementation skill. Thus, it is harder for the experts to perform better in this comparison if the seven novice data-sets are omitted from the analysis.

5.2 Implementation Task

The participants had to complete the control of an elevator system. The elevator system resides in a building with multiple floors. The system distinguishes between requests and jobs. A request is

[4] Actually, four of them were paid by us, the other three performed the implementation during their official working hours.

[5] Interestingly, one of the drop outs was the previously mentioned participant with a half year of TDD experience.

triggered if an up- or down-button outside the elevator is pressed. A job is assigned to the elevator after a passenger chooses the destination floor inside the elevator. The elevator system is driven by a discrete clock. For each cycle, the elevator control expects a list of requests and jobs and decides according to the elevator state which actions have to be performed next. The elevator control is driven by a finite automaton with four states: driving-up, driving-down, waiting, and open. The task description contained a state transition diagram explaining the conditions for switching from one state to another and the actions that had to be performed during a state switch. To keep the effort predictable, only the open-state of the elevator control had to be implemented. The participants received a program skeleton which contained the implementation of the other three states. This skeleton comprises ten application- and seven test-classes with 388 and 602 non-commented lines of code, respectively. The set of unit tests provided with the program skeleton use *mock objects* [16, 14] to decouple the control of the elevator logic from the logic that administrates the incoming jobs and requests. However, the mock-object implementation in the skeleton does not provide enough functionality to develop the whole elevator control. Other functionality has to be added to the mock-object implementation to test all desired features of the elevator control. Thus, the number of lines of test code may be higher than the number of lines of application code and it also may be higher than for solutions to problems which do not require the usage of mock objects.

5.3 Realisation

Implementation took place during a single programming session. There was no explicit time limit given, however the task description suggested that the task should be solvable in approximately four to five hours. Each participant recorded interrupts such as going to the bathroom or lunch breaks. The experimenter also recorded these interrupts as well as start and stop times of the implementation session. The two logs were later checked for consistency. The novices and two experts performed their work in our programming lab at the university. The novices worked in three rooms of our programming lab at the university and started with the task at the same time. Two of the programmers worked in our lab, too. They sat together in one room and started working simultaneously. The other five experts worked in their usual work environment. Two of them performed the task in one room, the remaining three worked in separate rooms.

Our analysis focuses on the implementation phase which started with a problem description and a running Eclipse working environment. The user-action logger and the JUnit-action logger were installed and the Eclipse working environment already contained the skeleton of the elevator system and all tests of the provided JUnit test-suite passed. The participants worked on the problem until they thought they had finished the job. At this point, participants entered the quality-assurance phase. During the quality-assurance phase, participants had to pass an acceptance test which ensures similar quality of the developed programs[6]. We excluded the quality-assurance phase from our analysis because we wanted to study the core TDD feature-development process and not a process with a focus on bug-fixing.

5.4 Research Hypotheses

This study was motivated by the following research hypotheses:

RH_{Conf} The experts achieve a higher conformance to the rules of the test-driven development process than the novices.

RH_{Len} The average length of the cycles is shorter for the experts than for the novices.

RH_{Var} The variation of the cycle lengths is smaller for the experts than for the novices. The variation of the cycle length is measured for each participant as the standard deviation from the mean.

[6]Expert e5 quit after the first acceptance test because the experiment conflicted with a private appointment.

RH_{CLOC} The number of changed lines of code (CLOC) during the whole implementation process is smaller for the experts than for the novices. We assume that this is true for application and test code.

RH_{Edit} The experts are faster in changing application code and test code. The editing speed is measured in changed lines of code per hour.

RH_{Cov} The tests developed by the experts achieve a higher coverage on the application code than the tests of the novices. Code coverage is measured on statement and block level using Emma [5].

5.5 Power Analysis

As the data samples are small, the non-parametric one-sided two-samples Wilcoxon-Rank-Sum Test [12, pp. 106] is used for evaluation. The power of the respective t-Test at a significance level of 5 percent, an effect size of 0.8 and an harmonic mean of 8.5 is 0.47. The power of the Wilcoxon-Test is in the worst case 13.6 percent smaller than the power of the t-Test [12, pp. 139]. Thus, the probability of detecting an effect is only 40.1 percent. This probability is fairly small opposed to the suggested value of 80 percent [4, p. 531]. To sum up, if a difference on the 5 percent level can be shown, everything is fine. But the probability that we miss the chance to reveal an existing difference with our data set is about 60 percent.

6 Results

Some results of our experiment are visualised using box plots. The boxes within a plot contain 50 percent of the data points. The lower (upper) border of the box marks the 25 percent (75 percent) quantile. The lower (upper) t-bar marks the most extreme data point which is not more than 1.5 times the length of the box away from the lower (upper) side of the box. If the number of data points per group equals the size of the group, each data point is depicted by a circle, otherwise, only the outliers from the above scheme are visualised. The median is marked with a thick line.

In our presentation of the results we distinguish between internal and external process properties. The internal process properties, presented in Section 6.1, could only be detected by analysing the detailed data collected by our loggers, whereas the external process properties in Section 6.2 were obtained with an analysis of the common experiment artifacts such as time sheets and the participants' programs.

6.1 Internal Process Properties

In this part, we examine the conformance of participants to the rules of test-driven development as defined in Section 3.2, present the analysis of the test cycles, followed by an evaluation of the number of changed lines of code and the editing speed.

6.1.1 Conformance to TDD

Research hypothesis RH_{Conf} is evaluated. It states our assumption that the experts achieve a higher conformance to the rules of the test-driven development process than the novices. Figure 3 shows the classification of all changes made by the participants. Table 1 in the appendix shows the raw data set. Data sets of the experts start with an 'e' while the data sets of the novices start with a 'n'. First, we look at the TDD changes. These are changes which conform to the TDD rules 1 and 2. For the expert group, the ratio of TDD changes lies between 45 percent and 85 percent. The corresponding range for the novice group starts at 0 percent and ends at 91 percent. This difference in the ranges leads to the following observations. First, the range of values for the novices' ratio of TDD changes is larger than the range of values for the experts. And second, there are members of the novice group who achieve a higher ratio of TDD changes than the experts.

Now, we examine conformance to the test-driven development process:

$$\text{TDD conformance} = \frac{|\,\text{TDD changes}\,| + |\,\text{refactorings}\,|}{|\,\text{all changes}\,|}$$

Figure 4 shows the conformance to the TDD rules for both groups. The experts seem to achieve higher values than the novices. The medians for both data sets are 82 percent for the experts and 67 percent for the novices. The p-value of 0.028 supports the visible difference between both groups. Thus, the experts achieve a higher conformance to the test-driven development process.

6.1.2 Analysis of Cycles

While the analysis of changes was done semiautomatically, all remaining charts were obtained automatically from the system. The research hypothesis RH_{Len} that the average length of the cycles is shorter for the experts than for the novices is analysed. Figure 5(a) shows the average length of the cycles for each group. The median of the average cycle lengths of the experts is 1.9 minutes whereas the one of the novices is 3.4 minutes. The average cycle length of the experts varies between 1.6 and 3.1 minutes. The values for the novices start at 1.3 and end at 16 minutes. The Wilcoxon-Test supports the visible difference between both data sets with a p-value of 0.022. That is, our research hypothesis RH_{Len} and the assumption behind it could be confirmed.

Figure 5(b) shows the standard deviation of the cycle lengths. We see a picture similar to the average length of the cycles. The data set of the novices is wide spread across the value range while the data set of the experts spans only across a small range. The medians of the standard distributions are 3.4 minutes for the experts and 6.3 minutes for the novices. The Wilcoxon-Tests supports the assumption of RH_{Var} that the experts have a smaller variation in the length of the programming cycles than the novices with a p-value of 0.017.

Both hypotheses RH_{Len} and RH_{Var} could be confirmed. However, not all novices have problems with the TDD process. There are novices which fall into the value range given by the experts. These are the novices n4, n5, n9, and n10. The novices n4 and n5 also achieve a TDD conformance like the experts. Thus, it is possible for novices to achieve similar TDD process characteristics as experts after just one week of intensive training.

We also examined the number of cycles needed to implement a solution. We did not have an expectation whether there is a difference between both groups, nevertheless, it is interesting to look at the distribution of both data sets which are shown in Figure 6. Again, the data set of the novices has a larger variation as opposed to the data set of the experts. But now, in contrast to the length and the variation of the cycles there is no visible difference between the medians of the

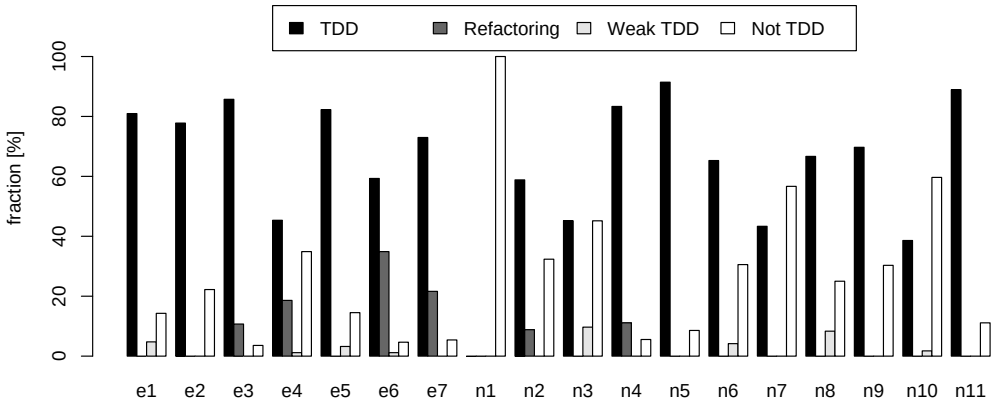


Figure 3: Classification of changes.

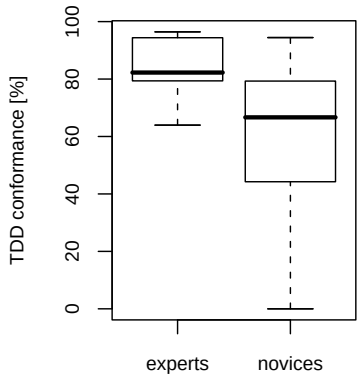
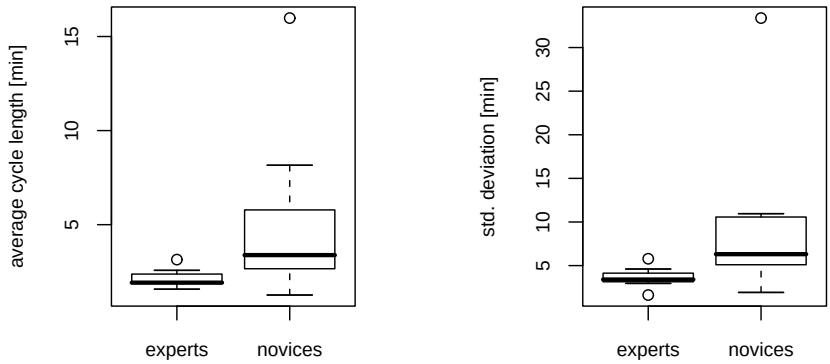


Figure 4: Conformance to TDD process.



(a) Length of cycles

(b) Variation of cycle length

Figure 5: Cycle characteristics.

two groups. The larger spread of the novices' data set is again caused by their differing ability to apply the TDD process. The process of the experts is more stable and thus, there is less variation.

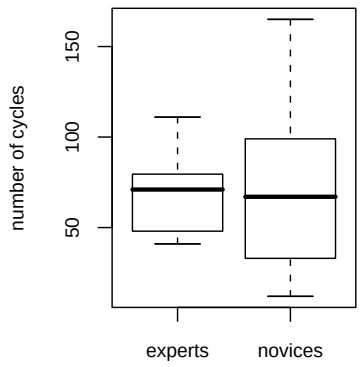
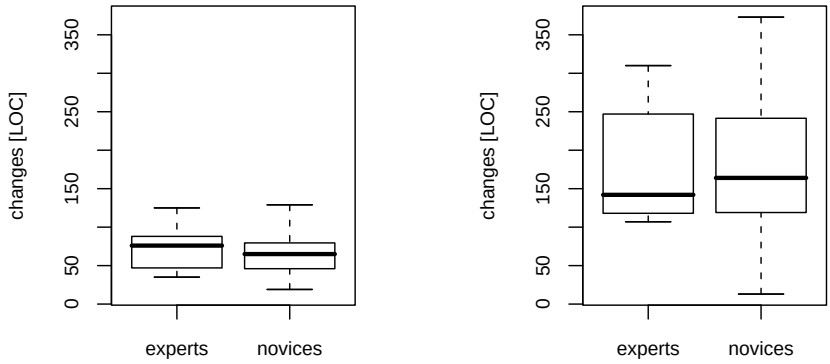


Figure 6: Number of cycles.

6.1.3 Changed Lines of Code and Editing Speed

Figures 7(a) and 7(b) show the number of changed lines of application and test code. For each data



(a) Application-code changes (b) Test-code changes

Figure 7: Code changes.

point in the figure, the change effort in each cycle has been calculated. These values were then summed up. Thus, it is not surprising that the lines of changed application code are larger than the size of 33 lines of code of our own solution. The test-code changes shown in Figure 7(b) were calculated the same way as the application-code changes. If both plots are taken into account, change effort in lines of code differs only slightly, if at all, between the novice and expert group. Both data sets have almost the same median, although, the data set of the novices seems to have a larger variation. The p-values for both data sets is 0.675 for the application-code changes and 0.4649 for the test-code changes. We could not confirm research hypotheses $\mathrm{RH}_{\mathrm{CLOC}}$, but this might be due to our small data set. We examined whether the number of changed lines of application code correlates with the number of changed lines of test code. The Spearman's Rank-Correlation Test was used [12, pp. 394]. The Spearman-Test indicates a correlation on the pooled data set with a p-value of 0.001. This correlation also holds for the data set of the novices ($p = 0.003$) but not for the data set of the experts ($p = 0.236$). The reason for the correlation in the novices' data set might be the ability to follow the TDD process. Actually, the Spearman-Test indicates a correlation ($p = 0.0003$) of the TDD conformance with the changed number of lines of code for the novices. A similar correlation could not be found for the experts ($p = 0.088$). Thus, a novice who is able to follow the TDD process changes the test code more than a novice who is not able to apply test-driven development.

Figures 8(a) and 8(b) show the editing speed for application-code and test-code changes. $\mathrm{RH}_{\mathrm{Edit}}$ stated our assumption that the experts change the application and test-code more quickly. While there is a visible difference for the application-code changes, the picture is somehow blurred for the test-code changes. The median of the experts is higher than the median of the novices (147 LOC/h vs. 106 LOC/h) but the data set of the experts has a larger variability than the one of the novices. The p-values of the corresponding Wilcoxon-Test support the impression that the editing speed of the experts for application-code is higher than the novices' one ($p = 0.013$). But the same statement does not hold for the editing speed of the test-code changes ($p = 0.212$). Thus, $\mathrm{RH}_{\mathrm{Edit}}$ holds for application-code changes but not for test-code changes.

The second result is quite interesting because we assumed that a higher skill level in applying test-driven development would increase the editing speed. It seems as if this assumption is false at first glance for test-code changes.

6.2 External Process Properties

This section continues with the analysis of external properties of the TDD process. We study the duration of the implementation until the first acceptance test, the quality of tests, the quality of the program versions at this point in time, and the size of the implemented solutions.

6.2.1 Duration of Implementation

Figure 9 shows the duration of the implementation until the first acceptance test. The experts are significantly faster than the novices (p-value of two-sided Wilcoxon test of 0.018). This result is not surprising because the experts had a higher editing speed for application-code changes and the experts have more programming experience than the novices.

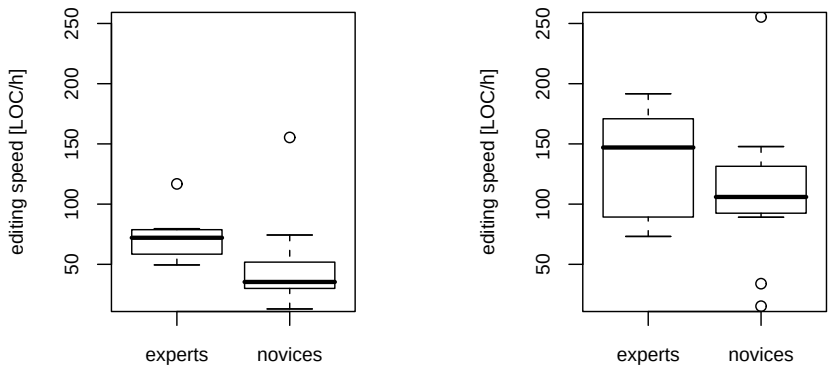
6.2.2 Quality of Tests

Figures 10(a) and 10(b) show the quality of the developed tests in terms of statement and block coverage. The experts achieve higher values for each of the investigated coverage metrics. For statement coverage, the medians are 91 percent for the experts and 90 percent for the novices. The medians for block coverage are 92 and 91 percent, respectively. However, for block coverage the data set of the experts has a smaller variability than for statement coverage. The p-values of the Wilcoxon-Test are 0.057 for statement coverage and 0.018 for block coverage. Thus, the research hypothesis RH_{Cov} that the tests developed by the experts achieve a higher coverage on the application code than the tests of the novices holds for block coverage. The difference for statement coverage is not significant on the 5 percent level but this might be due to the small data set.

We saw in Section 6.1.3 that experts are not quicker in writing test code than novices. This statement still holds but the result of the comparison of the code coverage values sheds new light on the editing speed of the expert group. The experts do not change test code in a faster speed than the novices but their tests are of higher quality, i. e. their tests achieve higher code coverage values. This higher quality is bought at the expense of speed in terms of lines of code per hour.

6.2.3 Number of Failed Tests

Figure 11 shows the number of failed test at the first acceptance test. The proportion of novices whose programs pass the acceptance test is higher than the corresponding proportion of the programs of the experts (7 of 11 vs. 2 of 7). Although this difference is visible to the naked eye, it is not statistically significant. We do not know exactly why the programs of the novices tend to be of better quality than the programs of the experts. We assume that the project experience



(a) Application (b) Test

Figure 8: Editing speed for changes.

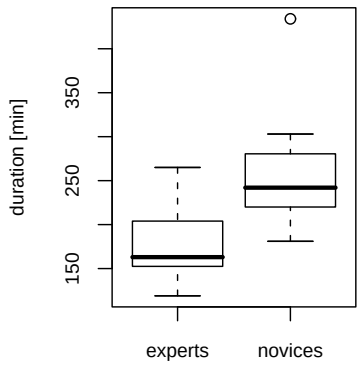
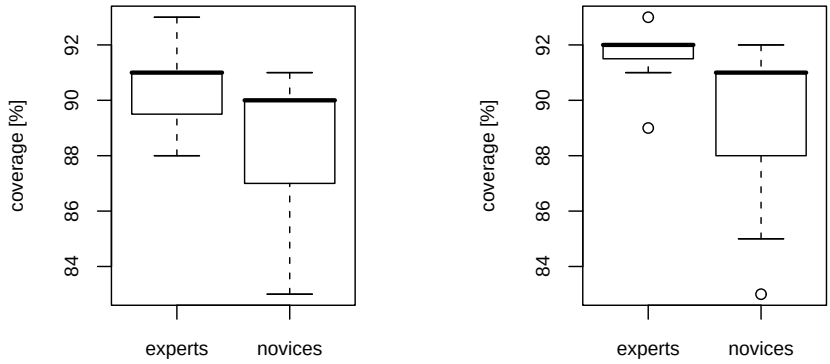


Figure 9: Duration of implementation.



(a) Statement coverage (b) Block coverage

Figure 10: Quality of tests.

of the experts plays an important part. As a result, the experts have a lower threshold to ask for the acceptance test than the novices who see the acceptance test as a formal criteria whereas the experts interpret it as just another means for testing.

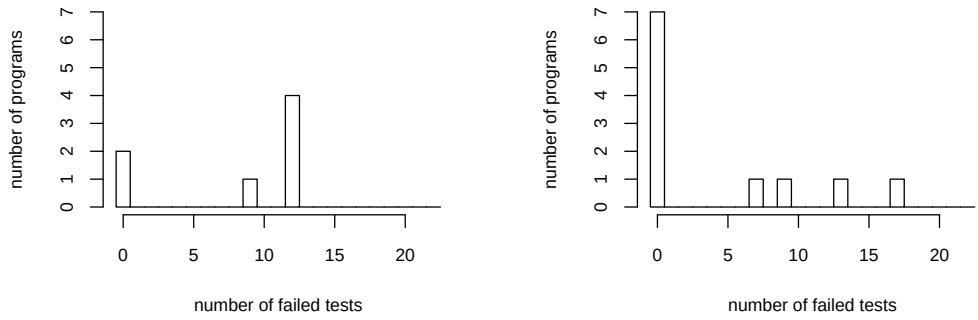
6.2.4 Size of Developed Programs

Figures 12(a) and 12(b) show the size of the application and test code at the time of the first acceptance test. Concerning the size of the application code, the experts seem to develop smaller programs than the novices. The difference is not statistically significant which might be due to our small data set. In contrast to the potential difference in application code size, the size of the test code does not seem to differ between both groups. The median of the experts is smaller but the boxes and the upper whiskers are similar. Remember the analysis of the quality of the tests in Section 6.2.2: The experts wrote tests with a higher block coverage than the novices. Since the amount of test code written by experts and novices is approximately equal, the experts seem to test more effectively than the novices.

6.3 Summary of Results

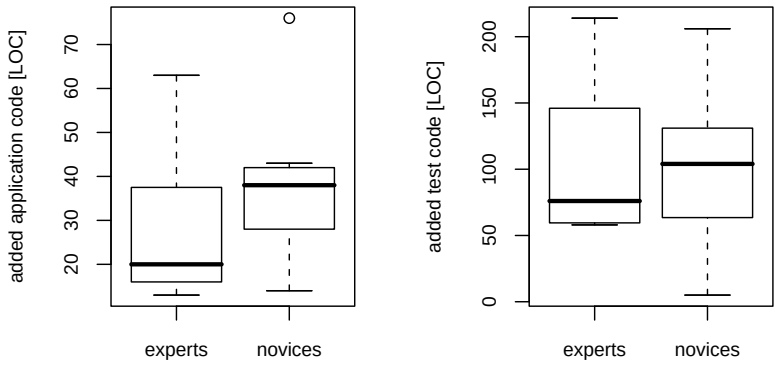
The analysis of the TDD processes of novices and experts lead to the following observations:

- The experts achieve a higher conformance to the rules of test-driven development than the novices. The research hypothesis $\mathrm{RH}_{\mathrm{Conf}}$ holds.



(a) Experts (b) Novices

Figure 11: Histograms of the number of failed test at first acceptance test.



(a) Application code (b) Test code

Figure 12: Number of added lines of code at first acceptance test.

- The analysis of the test cycles yielded that the experts have shorter test cycles than the novices and the test cycles of the experts show a smaller variation. The hypotheses $\mathrm{RH}_{\mathrm{Len}}$ and $\mathrm{RH}_{\mathrm{Var}}$ could be confirmed.

- A novice who is able to follow the TDD process changes the test code more often than a novice who is not able to apply test-driven development.

- The research hypotheses $\mathrm{RH}_{\mathrm{CLOC}}$ could not be confirmed, neither for application nor for test-code changes.

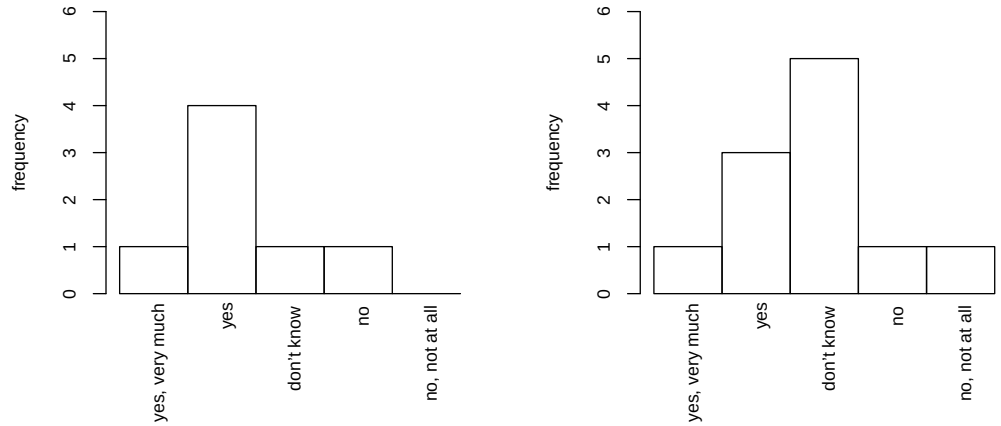
- The experts have a higher editing speed for application code changes than the novices. A similar result could not be found for the test-code changes. Thus, the research hypotheses $\mathrm{RH}_{\mathrm{Edit}}$ holds for application-code changes but not for test-code changes.

- The experts reach the first acceptance test in a shorter period of time than the novices. During that time frame, the experts write smaller programs and better tests than the novices in terms of block coverage.

7 Threats to Validity

Apart from the different TDD experience of the experts and novices other possible explanations for the observed differences in the data set might exist. First of all, the novices do not only have less experience with TDD but also less programming experience than the experts. As mentioned before, we tried to minimise this threat by dropping the data sets of those novices who were not able to finish the implementation task. Secondly, our data might not present the typical TDD process as the participants knew that they took part in an experiment. Thus, their development behavior might be different from their actual development style.

Another cause for differences in the observed data could be the fact that experts were paid for their participation and novices not which might have lead to a bias in motivation. Figures 13(a) and 13(b) show the frequency of answers for the question of the post-test questionnaire "Did you enjoy programming in the experiment?" The experts' distribution of answers seems to be shifted



(a) Experts (b) Novices

Figure 13: Distribution of answers to the question "Did you enjoy programming in the experiment?"

to the left compared to the novices' one which might indicate a higher motivation of the experts. But as our data set is small, this difference is not statistically significant.

The next threat concerns the usage of mock objects which increases the number of lines of test code. Thus the presented numbers of test-code changes and application-code changes may not be representative for the typical TDD process.

And finally, changes which were not classified to be conform to TDD by our system where inspected manually. Manual classification might introduce bias into the data set. However, both authors did the classification independently. Later on, the two change classifications were searched for differences. The few observed differences were caused by misinterpretations of the classification scheme and could be resolved without long discussions.

8 Conclusions and Future Work

This paper presented a comparison of TDD experts' and novices' test-driven development processes. The processes of 7 TDD experts and 11 TDD novices have been protocolled and analysed. This data set lead to the following results:

- The experts achieve a higher conformance to the rules of test-driven development than the novices. However, none of the participants achieved a conformance of 100 percent.

- The average elapsed time between two successive unit-test execution, the cycle time, is smaller for the experts than for the novices. The variation of the cycle time is smaller for the experts as well.
- The experts do not have more application-code and test-code changes than the novices. But, the experts achieve a higher editing speed for the application-code changes than the novices. The latter statement does not hold for the test-code changes.
- The developed tests of the experts have a higher block coverage than the tests of the novices.

Additional results show the correlation between application-code and test-code changes. A participant who had many changes of lines of application code also had many changes in lines of test code. This result holds for the pooled data sets as well as for the data set of the novice group alone. Another results states, that the degree of novices' process conformance correlates with the number of changed lines of test code. Thus, the ability of novices to comply to the rules of test-driven development leads to more changes of test code. The reported differences and correlations are statistically significant on the 5 percent level.

These results implicate that studies investigating TDD can not be easily generalised if they use novices (e. g. students without prior TDD experience) as subjects. Studies that want to avoid this problem need to provide evidence that the subjects are able to apply the TDD process to the same extent as programmers experienced in TDD. One way to show process conformance is a protocol of subjects' programming activities. Such a protocol might also ease comparison to other TDD studies and theory building. For a better understanding of the TDD process further questions need to be answered. Among others, these questions include the following:

- How long does it take for the majority of our novices to acquire comparable skills as the experts?
- How do the TDD processes of professional programmers without any prior experience in TDD look like?
- How does a typical TDD process look like? Since none of our experts achieved 100 percent process conformance, strict conformance to the rules may not be observable in practice.
- Does pair programming lead to a higher conformance to the TDD process?

To answer the above questions further experiments and field studies are sought.

9 Acknowledgements

The authors would like to thank David Burkhart for developing the Eclipse plug-ins and Urs Reupke for providing the tools to fill the database.

References

[1] K. Beck. *Test Driven Development: By Example.* Addison-Wesley, 2002.

[2] T. Bhat and N. Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *Procedings of the 2006 International Symposium on Empirical Software Engineering (ISESE'06)*, pages 356–363, Rio de Janeiro, Brazil, September 2006. ACM Press.

[3] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. A. Visaggio. Evaluating advantages of test driven development: a controlled experiment with professionals. In *Procedings of the 2006 International Symposium on Empirical Software Engineering (ISESE'06)*, pages 364–371, Rio de Janeiro, Brazil, September 2006. ACM Press.

[4] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Academic Press, 1988.

[5] http://emma.sourceforge.net.

[6] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, March 2005.

[7] H. Erdogmus and Y. Wang. The role of process measurement in test-driven development. In *Proceeding of XP Agile Universe 2004: 4th Conference on Extreme Programming and Agile Methods*, Calgary, Canada, August 2004.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[9] B. George and L. Williams. An initial investigation of test driven development in industry. In *ACM symposium on Applied computing*, pages 1135–1139, Melbourne, Florida, USA, 2003.

[10] A. Geras, M. Smith, and J. Miller. A prototype empirical evaluation of test driven development. In *International Symposium on Software Metrics (Metrics)*, pages 405–416, Chicago, Illinois, USA, September 2004.

[11] Hackystat. csdl.ics.hawaii.edu/Tools/Hackystat.

[12] M. Hollander and D. Wolfe. *Nonparametric Statistical Methods.* John Wiley & Sons, 2nd edition, 1999.

[13] http://jalopy.sourceforge.net.

[14] J. Link. *Unit Testing in Java: How Tests Drive the Code.* Morgan Kaufmann, 2003.

[15] J. Link. *Softwaretest mit JUnit: Techniken der testgetriebenen Entwicklung.* dpunkt.verlag, 2005.

[16] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In *Extreme Programming and Flexible Processes in Software Engineering - XP2000*, Cagliari, Sardinia, Italy, June 2000.

[17] G. Malpohl, J. Hunt, and W. Tichy. Renaming detection. In *Automated Software Engineering*, pages 73–80, Grenoble, France, September 2000.

[18] M. Müller and O. Hagner. Experiment about test-first programming. *IEE Proceedings Software*, 149(5):131–136, October 2002.

[19] M. Müller, J. Link, R. Sand, and G. Malpohl. Extreme programming in curriculum: Experiences from academia and industry. In *Conference on Extreme Programming and Agile Processes in Software Engineering (XP2004)*, pages 294–302, Garmisch-Partenkirchen, Germany, June 2004.

[20] M. Pančur, M. Ciglarič, M. Trampuš, and T. Vidmar. Towards empirical evaluation of test-driven development in a university environment. In *EUROCON 2003. Computer as a Tool. The IEEE Region 8.*, volume 2, pages 83–86, September 2003.

[21] W. Shadish, T. Cook, and D. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference.* Houghton Mifflin, 2002.

Table 1: Original data set.

User	Duration [min]	No. of failed tests at first acceptance test	Did you enjoy programming in the experiment?	No. of changes	TDD	Refactoring	Weak TDD	Not TDD	Over-all exp. [yr]	Java exp. [yr]	JUnit exp. [yr]	TDD exp. [yr]	Aver. cycle length [min]	Dev. cycle length [min]	No. of cycles	App. changes [LOC]	Test changes [LOC]	Added app. code [LOC]	Added test code [LOC]	App. edit. speed [LOC/h]	Test edit. speed [LOC/h]	Line cov. [%]	Block cov. [%]
e1	160	12	yes	42	34	0	2	6	6	6	6	6	3.1	5.8	45	76	223	55	214	77	178	91	92
e2	119	0	don't know	18	14	0	0	4	6	5.5	5.5	6	1.6	1.6	51	35	142	20	105	79	147	91	92
e3	163	9	yes, very much	28	24	3	0	1	4	4	3	1	1.9	3.7	73	47	108	17	58	49	73	91	92
e4	202	12	yes	86	39	16	1	30	10	9	3	1	1.9	3.0	71	82	310	20	60	72	191	89	91
e5[a]	265	12	no	62	51	0	2	9	8	4	0.75	0.75	2.2	3.4	86	94	128	15	59	64	74	88	89
e6	206	0	yes	86	51	30	1	4	10	8	6	5	1.9	3.3	111	125	271	63	187	116	163	93	93
e7	145	12	yes	37	27	8	0	2	7	7	6	4	2.6	4.6	41	47	107	13	76	52	103	90	92
n1	222	17	no, not at all	16	0	0	0	16	9	5	0	0	16.0	33.4	12	19	13	14	11	12	15	86	87
n2	300	7	no	34	20	3	0	11	7	7	0	0	4.1	6.0	70	86	164	39	104	50	95	88	89
n3	255	0	don't know	31	14	0	3	14	2	0.5	0	0	3.4	6.3	67	63	162	36	88	35	105	90	91
n4	181	0	yes	90	75	10	0	5	8	3	0.5	0	1.3	1.9	119	129	373	42	135	155	255	90	91
n5	303	0	yes	35	32	0	0	3	6	3	0	0	1.6	2.9	165	65	321	42	206	74	126	89	90
n6	434	0	don't know	72	47	0	3	22	10	4	0	0	3.2	6.8	128	124	225	76	88	34	102	86	85
n7	218	0	don't know	30	13	0	0	17	12	5	0.5	0	6.4	11.0	23	45	20	22	5	25	33	83	83
n8	242	13	yes	24	16	0	2	6	13	0	0	0	5.2	10.7	43	73	191	43	115	49	136	91	92
n9	261	0	don't know	33	23	0	0	10	10	2	0	0	3.0	5.8	79	65	258	38	149	53	147	90	91
n10	186	9	yes, very much	57	22	0	1	34	3	2	0	0	2.4	4.4	59	47	82	16	39	34	89	90	91
n11	222	0	don't know	9	8	0	0	1	2	0	0	0	8.2	10.4	23	41	156	34	127	25	122	91	92

[a]Quit after first acceptance test.