Refactoring with Unit Testing: A Match Made in Heaven?

Frens Vonken
Fontys University of Applied Sciences
Eindhoven, the Netherlands
Email: f.vonken@fontys.nl

Andy Zaidman
Delft University of Technology
Delft, the Netherlands
Email: a.e.zaidman@tudelft.nl

***Abstract*—Unit testing is a basic principle of agile development. Its benefits include early defect detection, defect cause localization and removal of fear to apply changes to the code. As such, unit tests seem to be ideal companions during refactoring, as they provide a safety net which enables to quickly verify that behaviour is indeed preserved. In this study we investigate whether having unit tests available during refactoring actually leads to quicker refactorings and more high-quality code after refactoring. For this, we set up a two-group controlled experiment involving 42 participants. Results indicate that having unit tests available during refactoring does not lead to quicker refactoring or to higher-quality code after refactoring.**

I. Introduction

The term refactoring was coined by Opdyke and Johnson in 1990 [1]; it is defined as the process of "changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [2]. Fowler adds that "it is a disciplined way to clean up code that minimizes the chances of introducing bugs" [2]. While refactorings are indeed thought-out to be small and relatively simple, thus minimizing the chances of something breaking, unit tests provide a *safety net* against introducing regressions during refactoring [3].

Not only do unit tests provide a safety net, but, because of their small size, they also offer a means to test changes in the code as soon as possible, thereby providing quick feedback to developers [4]. This claim is strengthened by a 2002 report from the NIST that states that catching defects early during (unit) testing lowers the total development cost significantly [5].

Because of these apparent benefits, unit testing is considered a key practice of agile software development methods, like SCRUM and eXtreme Programming (XP). As such, it has gained considerable attention in recent years and more and more practitioners are actively using it during development. XP not only sees refactoring as a key practice, but unit testing as well [6], underlying how intertwined both concepts are [7]. With the relation between unit testing and refactoring seemingly clear, we started questioning what the actual benefits are from having unit tests available during refactoring. In particular, we started thinking whether:

1) unit tests help to understand the code before refactoring, as eXtreme Programming XP advocates unit tests as a form of documentation [8], [3], [9]?
2) unit tests increase the internal quality of the code, by providing a context for the refactoring? (similar to better code quality obtained when applying test-driven development [10])
3) unit tests allow to refactor more quickly, because less debugging is necessary?
4) it actually costs time to also adapt the units tests when refactoring? This final point stemming from the study by van Deursen and Moonen, who established that a number of refactorings invalidate the accompanying unit tests due to changes in the interface [11].

This leads us to our two main research questions, namely:

RQ1 Does having unit tests available during refactoring allow for quicker refactoring operations?

RQ2 Does having unit tests available during refactoring lead to higher-quality refactoring solutions?

In order to investigate these research questions, we carried out a two-group controlled experiment [12]. In this controlled experiment we tested 35 final-year students and 7 experienced software developers and gave them 7 refactoring assignments. The experimental group did have unit tests available, while the control group did not.

The structure of this paper is as follows: we start by covering related work in Section II. Subsequently, we explain our experiment design in Section III, before going over to the presentation of the results in Section IV. We discuss our findings in Section V before concluding in Section VI.

II. Related work

In 2006 Runeson performed a survey on the use of and perceived benefits and weaknesses of unit testing in industry [4]. The positive observations are that developers seem to like unit testing because of its quick feedback, its automatability and its ability to test both the smallest unit and the interaction between units. Developers also indicated that they had issues determining when to stop testing and that the maintenance of unit tests could become costly.

Only a small number of studies have been published on the theoretical relation between the refactoring and unit testing. Studies from Pipka [13] and Basit et al. [14] theoretically explore extending refactoring guidelines to incorporate unit test refactorings. Others like van Deursen et al. [15] concentrated on refactoring test-code itself.

George and Williams performed a study with 24 professional pair programmers. One group developed a small Java program using Test-Driven Development (TDD), while another group used a waterfall-like approach [16]. They observed that while TDD programmers produce higher quality code because 18% more functional black-box test cases passed, the TDD programmers took 16% more time.

Williams et al. studied the effectiveness of unit test automation at Microsoft [17]. Their study makes three notable observations. First, they report that by using an automated unit testing framework like NUnit, the team of 32 developers managed to reduce the number of defects by 20.9% compared to earlier releases where no automated unit tests were in place. Secondly, they witnessed that this increase in quality came at a cost of approximately 30% more development time. And thirdly, interviews with the functional testing team indicated that they found their work to have changed dramatically because the "easy" defects were now found by the developers due to the presence of the unit tests.

Zaidman et al. investigated whether changes to production code are always backed up by changes to the unit tests as well [18]. They observed that this is very project-dependent, with some projects not changing their unit test code for several months, even leading to non-compilable test code.

Moonen et al. noted that, in parallel to test-driven development, test-driven refactoring can improve the design of production code by focusing on the desired way of organizing test code to drive refactoring of production code [6].

Work by Hurdugaci and Zaidman focuses on providing tool support for developers during maintenance of their code [19]. In particular, they have developed TestNForce, a tool which helps software developers to identify test code that should be adapted after having made changes to production code.

Finally, recent work by Athanasiou reveals that there is a relation between the quality of the (developer) test code and the ability of a development team to fix incoming issues (bug reports and feature requests) more quickly [20], thus giving an extra incentive to invest in high-quality testing.

III. Experimental Design

Our goal is to explore the effect of the availability of unit tests on refactoring. For this, we set up a two-group controlled experiment [12] involving seven refactoring assignments. The refactoring assignments were offered with unit tests to the experimental group and without unit tests to the control group. The seven refactoring assignments involved representative refactorings from the refactoring categories as defined by [21]. We specifically looked at the effect of availability of unit tests on the time to perform each assignment as well as on the quality of the work.

A. Hypotheses

From our research questions in Section I we define the following null hypotheses:

- $H1_0$: Availability of unit tests does not lead to faster refactoring.
- $H2_0$: Availability of unit tests does not lead to higher-quality production code after refactoring.

The alternative hypotheses are the following:

- H1: Availability of unit tests leads to faster refactoring.
- H2: Availability of unit tests leads to higher-quality production code after refactoring.

The rationale for the first alternative hypothesis is twofold. First, as unit tests provide a specification of the program code to be refactored they support a better understanding of the code [9] and therefore faster refactoring. Second, because program errors are caught and also pinpointed by unit tests, these errors can be detected and repaired faster.

The justification for the second alternative hypothesis is very similar to that of the first. Through better understanding of the code to be refactored and through detection of errors introduced while refactoring, availability of unit tests is expected to lead to qualitatively better production code.

B. Object of research

The software project that we used for our experiment is JPacman, a Java implementation of the well known Pac-Man game. JPacman[1] is built for educational purposes for courses in software testing at the Delft University of Technology [22]. We offered this program to the subjects as an Eclipse project with a Maven build file. This project was chosen because it has the following characteristics:

- JPacman has a clear design with a model and controller package and a clear inheritance hierarchy.
- It is supported with a suite of JUnit unit tests and many JUnit assertions.
- It has substantial complexity with 27 domain classes and about 3000 lines of code.

The JPacman testsuite consists of approximately 1000 lines of code and 60 testcases, of which 15 are high level acceptance tests exercising the top level public interface and 45 are unit tests. Testcases were derived systematically using various strategies mostly obtained from [23], including branch coverage (100%), exercising domain boundaries at *on points* as well as *off points*, testing class diagram multiplicities, deriving tests from decision tables and state and transition coverage.

While small in size, JPacman has substantial complexity, making experimental assignments non-trivial. The availability of a sound and extensive testsuite allows the investigation of the role of these tests in refactoring.

C. Subjects

The subject for this experiment are 35 fourth year professional bachelors in software engineering and 7 participants from industry. The group thus consists of 42 subjects from two populations. The students are Dutch students in their final year of their bachelor program at Fontys Universiy of Applied Sciences, an Eindhoven based Dutch college. This bachelor program is specifically oriented towards software

[1] JPacman can be obtained for research and educational purposes through the 2nd author of this paper.

engineering and teaches refactoring and unit testing integrated in several courses in the curriculum. The participants from industry are software engineers at the Software Improvement Group (SIG)[2], an Amsterdam-based company with focus on software quality and active policies on unit testing.

The students participated in the experiment as part of a course on software quality. As the criterion for passing the course was active participation the subjects are assumed to be properly motivated so that their refactoring and use of unit tests are representative for their behaviour in practical situations. The same holds for the participants from industry as their participation was voluntarily. None of the subjects had any previous experience with JPACMAN.

D. Preparation

For the participating subjects from industry it is presumed that they have sufficient proficiency in refactoring and the use of unit tests as all of them had several years of experience in software engineering in accordance with the quality-oriented policies of their company.

The student subjects were prepared for the experiment by a six hour course on refactoring. In the first two-hour part of this course students were introduced to refactoring by a presentation about the principles of refactoring, demonstration of these principles and some simple refactoring assignments based on the video store example from [2]. Subsequently, the students successfully participated in a 4-hour refactoring lab session, based on the work of Demeyer et al. [24]. This lab session is specifically designed for teaching good refactoring practices in realistic circumstances and had been positively evaluated in an academic as well as industrial context. Therefore it is presumed to be a good preparation for our experiment on refactoring. Based on the results from this lab session students were assigned to the control and experimental to get two balanced groups. Professional subjects were randomly assigned to the experimental and control group as they are expected to be approximately equally proficient.

E. Task design

In order to design refactoring assignments that are both realistic and representative, we looked at work by Counsell et al. [21] that classifies Fowler's catalogue of refactorings [2] based on whether a particular refactoring changes the interface that the unit test expects.

As listed in Table I, Counsell et al. classify the refactorings in 4 categories: compatible, backwards compatible, make backwards compatible and incompatible. Table I also lists the selection of 7 refactorings that we made for our experiment, where there are 2 representatives for the first three categories and only one for the last category. We opted for only 1 refactoring in the last category mainly to keep the total time of the experiment within the range of three hours. The refactorings to be performed in the assignments had an increasing complexity.

Each assignment consisted of the task to perform a specific refactoring on a specific piece of code. In the assignment subjects were pointed to the specific place to perform the refactoring. In particular, the name(s) of the class(es) and method(s) initially involved were given. The instruction with each task was to improve the code in accordance to a specifically named refactoring as described by [2]. When required related unit tests had to be adapted as well to keep them passing, while it was not required to add test code for code that may have been added while refactoring. During the assignment Fowler's book was available and subjects were given a reference to the specific page where the required refactoring was described.

An example of an assignment is as follows[3]:

This assignment relates to:
`jpacman.model.Move.tryMoveToGuest(Guest)`

Task:
In this task the quality of the code has to be improved by applying the refactoring "push down method" (Fowler p. 266)

F. Variables and analysis

The independent variable in our experiment is the availability of unit tests.

The first dependent variable is the *time* needed to perform each of the refactoring assignments. This variable is measured by asking the subjects to write down the current time when each assignment is finished. Because everyone started at the same time and assignments are done consecutively, the time spent on each assignment can easily be computed.

The second dependent variable is the *quality* of the production code after it has been refactored. This is measured by inspection and scoring of the refactored code against a scoring model for each assignment. The scoring model was based on the quality improvements as intended by the refactorings as described by Fowler [2] (e.g. clear localisation of behaviour and properties (cohesion and coupling, sound class hierarchy), naming, encapsulation, etc.). The scoring model was set up after a first general inspection of the code generated by the subjects to be able to differentiate as much as possible between the resulting code handed in. With the scoring model each refactoring assignment for every subject was rated with a number from 0 through 10.

To get a view on how the unit tests are used when available, we asked the subject whether unit tests are run before, inspected during and run after a refactoring is performed. From these questions we derive the following extra hypotheses:

- $H3_0$: Running unit tests before refactoring does not impact the time needed to perform a refactoring.
- $H4_0$: Running unit tests before refactoring does not impact the quality of the refactoring solution.
- $H5_0$: Inspection of unit tests during refactoring does not impact the time needed to perform a refactoring.
- $H6_0$: Inspection of unit tests during refactoring does not impact the quality of the refactoring solution.

[2]http://www.sig.eu

[3]All assignments can be downloaded from: http://swerl.tudelft.nl/bin/view/Main/RefactoringAndUnitTesting

Category	Description	Representative
A	**Compatible** Refactorings that do not change the original interface and only use refactorings that are compatible as well.	Consolidate duplicate conditional fragments Introduce explaining variable
B	**Backwards Compatible** Refactorings that change the original interface and are inherently backwards compatible since they extend the interface and only use other refactorings that are compatible or backwards compatible.	Push down method Extract method
C	**Make Backwards Compatible** Refactorings that change the original interface and can be made backwards compatible by adapting the old interface and only use refactorings from this or the above two categories. For example, the "Move Method" refactoring that moves a method from one class to another can be made backwards compatible through the addition of a "wrapper" method to retain the old interface.	Move method Introduce parameter object
D	**Incompatible** Refactorings that change the original interface and are not backwards compatible because they may, for example, change the types of classes that are involved, making it difficult to wrap the changes.	Extract subclass

TABLE I
CLASSIFICATION OF REFACTORINGS BY COUNSELL ET AL. [21] AND SELECTION OF REFACTORINGS FOR OUR STUDY.

- $H7_0$: Running unit tests after refactoring does not impact the time needed to perform a refactoring.
- $H8_0$: Running unit tests after refactoring does not impact the quality of the refactoring solution.

To get an overall view on the results we first compute frequencies and averages and print boxplots. To test our hypotheses we first check the normality of the distribution and equality of variances of our samples with a Kolomogorov-Smirnov test and a Levene test. If these tests pass, we use the unpaired Student's T-test to evaluate our hypotheses, otherwise we use the non-parametric Mann-Withney U test.

G. Pilot study

Prior to the actual experiment we did a pilot run of the entire experiment with one subject, an on average performing student. We performed this step in order to make sure that the object of research would run correctly within Eclipse and would work with Maven without issues. We also wanted to verify whether the assignments were clear and if the estimates of the length of the experiment were realistic in the sense that the assignments could be completed in an afternoon session by all subjects. The pilot study pointed out that the assignments were clear and that they could be executed as offered by the subject of the pilot study within two hours. Because some support was needed for the installation of the Maven plug-in, we added a very short description of this instruction for the participants of the actual experiment.

IV. RESULTS

Our experiment is built around gauging the effect of the availability of unit tests on (1) the time to perform a refactoring and (2) on the quality of refactorings. The results of each of those two aspects will be discussed in Section IV-A and IV-B respectively.

A. Effect of unit tests on time to refactor

Overall results. Table II shows the average amount of time that our subjects needed to perform the seven refactoring assignments (and the total time needed), divided over the experimental group and the control group. The same table also shows the significance of the difference between both groups. From these results we see that only the total time needed for performing *all* assignments is significantly different between the two experimental groups. For every single assignment no significant difference between the groups in the distribution of times needed to complete the assignment is observed.

Assignment	1	2	3	4	5	6	7	Total
Average w unit test	7:08	10:43	18:19	5:51	8:27	19:43	46:05	1:56:19
Average w/o unit test	4:23	8:32	13:06	6:15	6:17	14:53	40:35	1:34:05
Mann-Withney U test	.221	.381	.241	.518	.714	.099	.571	**.028**

TABLE II
AVERAGE TIME NEEDED PER ASSIGNMENT AND TOTAL OVER ALL ASSIGNMENTS, WITH OR WITHOUT UNIT TESTS AND THE P-VALUES OF THE MANN-WITHNEY U TEST DISTRIBUTION COMPARISON BETWEEN SUBJECTS WITH AND WITHOUT UNIT TESTS (BOLD MARKS SIGNIFICANCE AT .05 LEVEL).

From the boxplots in Figure 1 we see that the time needed to complete the assignments is similar (assignments 4 and 7) or slightly longer for the unit test group compared to the non unit test group.

The boxplots in Figure 1 also show that for some assignments (e.g., 7) the dispersion of times to complete the assignment is very similar for both groups, while for other assignments (e.g., 2) the dispersion is rather different. In general the dispersion of times to complete the assignments is larger for the unit test group.

The observation that the total amount of time needed to do all the assignments is larger when unit tests are available may be explained by the fact that running the unit test before and after the refactoring and inspecting the unit test during refactoring takes time. We investigated what the effect of the inspection of unit tests on the time needed to do an assignment is, by analysing the subjects who had unit tests available and comparing the time needed to do an assignment between subjects who did and did not inspect the unit tests while refactoring. As displayed in Table III we found no statistically significant difference between these groups. Furthermore, by looking at the average values of time needed, we see a very mixed image where some refactorings take longer with inspection, and others can be performed more quickly with inspection.

The results from Table III show that inspection of unit tests

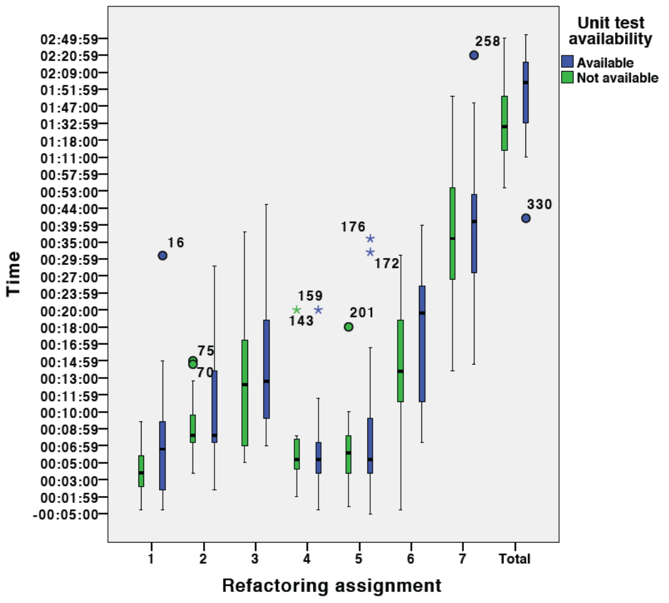

Fig. 1. Boxplots of time needed per assignment and the total over all assignments

Assignment	1	2	3	4	5	6	7
N with inspection	9	15	14	12	14	12	18
Average time w inspection	8:33	11:00	21:25	5:00	6:04	20:19	39:03
N without inspection	13	7	8	10	8	10	3
Average time w/o inspection	6:09	10:08	12:52	6:53	12:37	18:29	34:39
Mann-Withney U test	.713	.199	.374	.370	.945	.895	.167

TABLE III
AVERAGE TIME NEEDED PER ASSIGNMENT FOR SUBJECTS WHO DID AND DID NOT INSPECT UNIT TESTS WHILE REFACTORING AND THE P-VALUES OF THE MANN-WITHNEY U TEST DISTRIBUTION COMPARISON BETWEEN SUBJECTS WITH AND WITHOUT INSPECTION OF UNIT TESTS.

does not lead to significant differences in the time required to perform a refactoring. As such, we have to retain hypothesis $H5_0$: Inspection of unit tests during refactoring does not impact the time needed to perform a refactoring. Likewise the explanation of inspection of unit tests being (partially) responsible for the extra time required for refactoring at least has to be questioned.

The effect of running unit tests before and after refactoring on the time needed to do an assignment could not be analysed from our data because more than 90% of the subjects with unit tests available, ran these tests before and after each refactoring. Because of this, the group of subjects who did not run the unit tests before and after a refactoring was too small (often empty) to make a comparison with. Therefore hypotheses $H3_0$ and $H7_0$ cannot be tested from the data we gathered.

Another explanation for refactoring taking more time with unit tests available is that adapting the tests takes time. Because the solutions created by the subject show that they did adapt the test when necessary this can account for some of the extra time needed.

Refactoring internals. To understand the reasons for the (small) differences in time needed to perform the refactoring assignments, we will now have a closer look at each of the refactorings. The sum of several small differences between both groups may account for the one significant overall difference.

The first two assignments with the *compatible* refactorings *consolidate duplicate conditional fragments* and *introduce explaining variable* show slightly longer refactoring times and bigger dispersion of times for subjects with unit tests compared to subjects without unit tests. As each of these refactorings was relatively easy they only took a short time to complete. The interfaces were not changed in such a way that available unit tests had to be adapted. So for these assignments we can only think of running unit tests before and after refactoring and inspection of unit tests during refactoring as being responsible for the slightly longer time to do the refactoring. The bigger dispersion of time needed to refactor with unit test available may be explained by the fact that the unit tests simply introduce more code the programmer can spend time on trying to understand the job to be done. The normal variance that exists in time required to understand the code to work on will increase with more code involved.

The third assignment is the *backward compatible* refactoring *push down method*. Differently to the first two assignments, this refactoring assignment does not show a bigger dispersion of time to complete the refactoring with unit tests available. In this case the extra time required by inspection of the unit test code may be compensated by the structure that the unit tests provide. This observation has been made before by Moonen et al. who state that well-structured test code can improve the structure of the production code during test-driven refactoring, or first refactoring the test code and subsequently refactoring the production code [6]. Specifically for this assignment, we observed that the resulting code from the non unit test subjects showed much more diversity than that from the subjects with unit tests. Several of these alternative "solutions" would have caused unit tests to fail. So for this refactoring unit tests may have kept the subjects from diverting to complicated and time consuming paths.

The fourth assignment, also with a *backward compatible* refactoring (*extract method*) turned out to be rather easy and showed very similar time scores for both groups. As such, this assignment does not explain the overall difference in time that we found.

The fifth assignment with a *Make Backwards Compatible* refactoring (*move method*) may result in changes to the to be tested interface. As a result, this assignment requires the test to be changed or the refactored code to be made compatible. The extra time required to change the unit tests in this assignment may be compensated by less time needed to identify the required adaptations in the production code because they are pointed out by the feedback from the unit test. The bigger dispersion of time to do the assignment with unit tests available may again be explained by the fact that more code is involved.

In the sixth assignment the code had to be improved with the *Introduce Parameter Object* refactoring, also belonging to the

Make Backwards Compatible category of refactorings. Here the unit test group needed slightly more time as can be seen in Table II. Important to note here is that the assignments did not require to add unit tests for newly added classes or methods. This particular refactoring assignment could be performed without changing the to be tested interface, though most subjects chose a solution that did change the interface. These interface changes always required changes to unit tests. The work on these tests may be an explanation of the extra time required when unit tests were available. It also offers an explanation for the larger dispersion for the time required to refactor this situation. We should also note that the extra time needed to also change the tests may have been reduced by the feedback from unit test, pointing out where code had to be adapted. This reasoning is partly supported by the much bigger diversity in resulting refactoring solutions that we observed within the group of participants without unit tests available as compared to the subjects with unit tests.

The seventh assignment involved the *Incompatible* refactoring *Extract Subclass*. For the group of subjects with unit tests available we see a slightly longer time to refactor and a slightly smaller dispersion of times. This refactoring required adaptation of existing tests. This will have caused some extra time. The instruction that it was not required to write test for newly introduced code will have moderated this effect. The smaller dispersion in times needed to do the refactoring with unit tests available may again be explained by supporting feedback from the unit tests as again the resulting code from the non unit test group showed much more diversity than the code from the unit test group.

Students versus professionals. As our subjects came from two populations (35 students and 7 professionals), we also looked at the difference between these two groups. Table IV shows that except for the last two assignments, there was no significant difference in the distribution of the times to complete the individual assignments. Due to the fact that we are actually performing multiple hypothesis testing (students versus professionals and unit tests and no unit tests), we also performed Bonferonni correction, which indicated that also assignments 6 and 7 showed no significant difference. For the last two assignments the professionals needed less time.

The shorter time needed by professionals to do the last two assignments may be explained by the fact that they are more experienced in doing refactorings. That this only shows with the last two refactorings may be explained by the fact that these are the most complicated refactorings that take enough time to demonstrate the difference in proficiency.

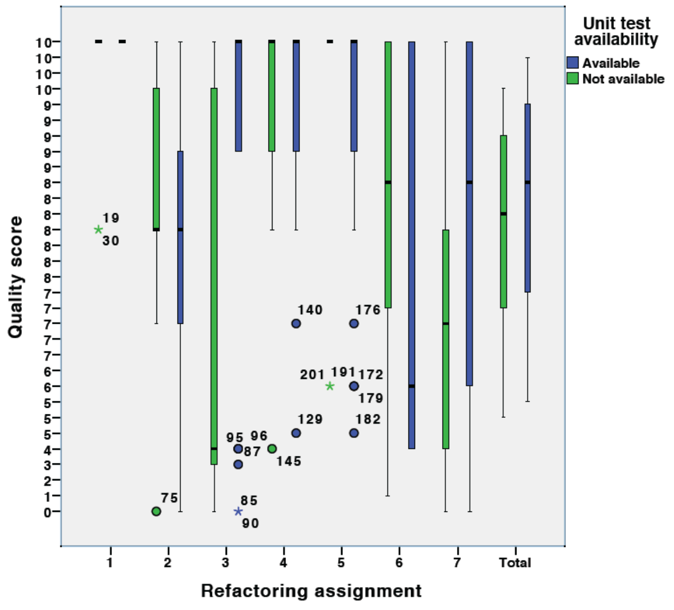

Fig. 2. Boxplots of scores per assignment and average of all scores

B. Effect of unit tests on the quality of refactoring

Overall results. We also investigated each refactoring assignment and attributed a score from 0 to 10 to how well the refactoring was reflected in production code. Figure 2 shows the boxplots of the scoring for each of the seven refactoring assignments, as well as the total score, subdivided per group. From the boxplots we can see that, on average, subjects were able to do the refactorings as intended. The mean score over all seven assignments together, for all 42 subject, was 8.1 (out of 10), with a minimum of 5.1 and a maximum of 9.9.

When testing for significance of the difference in distribution for the quality scores, we see that only for assignment 3 there is a significant difference between the experimental group and the control group (see Table V).

Except for assignment 3, availability of unit tests does not lead to a significantly higher quality refactoring. For assignment 3 the average score for subjects with unit tests available is 8.0 while subjects without unit tests have an

Assignment	1	2	3	4	5	6	7	Total
Average student subject	5:49	9:51	15:18	6:08	7:17	18:54	46:20	1:49:41
Average professional subjects	5:51	8:51	18:25	5:34	8:08	9:59	29:08	1:25:59
Mann-Withney U test	.839	.634	.919	.852	.081	**.007**	**.038**	.085

TABLE IV
AVERAGE TIME NEEDED PER ASSIGNMENT AND TOTAL OVER ALL ASSIGNMENTS FOR STUDENT AND PROFESSIONAL SUBJECTS AND THE P-VALUE OF THE MANN-WITHNEY U TEST DISTRIBUTION COMPARISON BETWEEN STUDENT AND PROFESSIONAL SUBJECTS (BOLD MARKS SIGNIFICANCE AT .05 LEVEL).

Assignment	1	2	3	4	5	6	7	Total
Average w unit test	10.0	7.5	8.0	9.4	9.1	6.7	7.5	8.3
Average w/o unit test	9.8	8.3	5.7	9.4	9.6	7.6	5.8	8.0
Mann-Withney U test	.133	.227	**.027**	.975	.114	.356	.056	.480

TABLE V
AVERAGE QUALITY SCORES PER ASSIGNMENT AND THE MEAN OF SCORES OVER ALL ASSIGNMENTS, WITH OR WITHOUT UNIT TESTS AND THE P-VALUES OF THE MANN-WITHNEY U TEST DISTRIBUTION COMPARISON BETWEEN SUBJECTS WITH AND WITHOUT UNIT TEST (BOLD MARKS SIGNIFICANCE AT .05 LEVEL).

Assignment	1	2	3	4	5	6	7
N with inspection	9	15	14	12	14	12	18
Average score w inspection	10	7.47	8.21	9.87	9.43	6.92	7.78
N without inspection	13	7	8	10	8	10	3
Average score w/o inspection	10	7.57	7.63	9.00	8.50	6.50	5.00
Mann-Withney U test	1.000	.797	.680	.556	.433	.519	.384

TABLE VI
AVERAGE SCORE PER ASSIGNMENT FOR SUBJECT WHO DID AND DID NOT INSPECT UNIT TESTS WHILE REFACTORING AND THE P-VALUES OF THE MANN-WITHNEY U TEST DISTRIBUTION COMPARISON BETWEEN SUBJECTS WITH AND WITHOUT INSPECTION OF UNIT TESTS.

average score of 5.7 (see Table V). So for this assignment having unit tests coincides with better refactoring.

Looking at the role of unit tests we also analysed the effect of inspection of the unit test on the quality of the refactoring. As can be seen from Table VI we found no significant difference in the distribution of the quality scores between the subject who did and who did not inspect the unit tests while refactoring.

With the difference in scores for the seventh assignment (7.78 with unit test inspection versus 5.00 without unit test inspection) we should note that only 3 subjects did not inspect the unit tests while refactoring. Not inspecting the unit test when doing such a complicated refactoring seems uncommon and may be related to less successful refactoring.

Refactoring internals. To get a clearer view on the data we gathered, we will next have a closer look at each of the refactoring assignments.

The first assignment concerned the *consolidate duplicate conditional fragments* refactoring, classified as being a *compatible* refactoring. The average score for both groups was high (10.0 with unit tests and 9.8 without unit tests) and the code handed in was very similar, making it very difficult to differentiate between the groups based on this refactoring.

With the second assignment the *compatible* refactoring *introduce explaining variable* led to an average score of 7.5 for subjects with unit tests. Subjects without unit tests scored 8.3 on average. Inspection of the code that was handed in revealed that participants with unit tests lost some marks on wrong protection levels of variables. We could not identify a reason in the unit tests to explain this observation.

The third assignment with the *backward compatible* refactoring *push down method* showed a significant difference in scores between the groups with and without unit tests: 8.0 versus 5.7. The assignment intended to push one complex method down to two subclasses, in which the method would become simpler, due to the elimination of the type check. What we see, however, is that several subjects from the group without unit tests moved the method to a wrong class. We found a possible explanation for this when we inspected the test classes that intend to test the two subclasses of the class in which the method that should be pushed down is residing before the refactoring. In particular, we found that the method that should be pushed down is indirectly called from these test classes, thereby possibly influencing the participants who did have access to the unit tests to move the method to these two classes.

The fourth and fifth assignments concerned respectively the *Backwards Compatible* refactoring *extract method* and the *Make Backwards Compatible* refactoring *move method*. Subjects from both groups succeeded very well in performing the refactorings. With average scores of 9.4 for both groups for the fourth assignment and 9.1 and 9.6 for the fifth assignment, we found no differences by which we could differentiate the solutions that were handed in by both groups.

From the sixth *Make Backwards Compatible* refactoring *Introduce Parameter Object* we see a slightly lower score of 6.7 for subjects with unit tests compared to 7.6 for subjects without unit tests. From the handed in code we see that whenever the tested interface is adapted the tests are also adapted. The adaptations of the tests look more like satisfying the unit test system than critically testing the new specifications. This may have been caused by the instructions that existing (test) code should keep working, but that it is not necessary to add new test code for code that may be added with the refactoring. This instruction may have given support for a quick and dirty way of testing, but, to our opinion, it cannot explain the slightly lower scores for subjects with tests available.

The scores for the seventh assignment with the *Incompatible* refactoring *Extract Subclass* were just not significantly different between the two groups at a 0.05 significance level, in particular, we obtained a p-value for the Mann-Withney U test of .056 and average scores of 7.5 for subjects with unit tests available and 5.8 for subjects without unit tests available. When looking at the code handed in, we see that the participants did adapt several unit tests (when available to the participants). For these subjects, some aspects of the the code for the two newly created subclasses and the resulting superclass got significantly better scores than for the subjects without unit tests. Table VII shows the average scores for subjects according to our scoring model.

At least in several cases, for this assignment the feedback from the unit tests is likely to have pointed to how changes should be made to the code. When for instance the resulting super class was turned into an abstract class every instantiation

Criteria for subscores	
1	Does resulting code compile?
2	New classes defined in correct class hierarchy with the correct constructor.
3	Content of newly created subclasses.
4	Content or resulting Superclass.

Subscore	1	2	3	4
Maximum subscore	3	3	2	2
Average w unit test	2.45	2.36	1.36	1.27
Average w/o unit test	2.4	1.95	.85	.95
Mann-Withney U test	.882	.439	**.034**	.12

TABLE VII
MAXIMUM AND AVERAGE QUALITY SUBSCORE FOR THE SEVENTH ASSIGNMENT AND THE P-VALUES OF THE MANN-WITHNEY U TEST DISTRIBUTION COMPARISON BETWEEN SUBJECTS WITH AND WITHOUT UNIT TESTS (BOLD MARKS SIGNIFICANCE AT .05 LEVEL).

Assignment	1	2	3	4	5	6	7	Total
Average student subject	9.89	7.63	6.34	9.34	9.26	6.71	6.26	7.92
Average professional subjects	10.00	9.00	9.71	9.57	9.71	9.29	8.57	9.41
Mann-Withney U test	.522	**.041**	**.026**	.611	.869	**.023**	.068	**.001**

TABLE VIII
AVERAGE SCORE PER ASSIGNMENT AND TOTAL OVER ALL ASSIGNMENTS FOR STUDENT AND PROFESSIONAL SUBJECTS AND THE P-VALUES OF THE MANN-WITHNEY U TEST DISTRIBUTION COMPARISON BETWEEN STUDENT AND PROFESSIONAL SUBJECTS (BOLD MARKS SIGNIFICANCE AT .05 LEVEL).

of an object of this class in the tests had to be changed, which may have pointed to the use and functionality to be moved to the newly created classes. In this way the tests may have guided the subjects in their refactoring and led to better code in the super class and newly created subclasses.

Students versus professionals. Looking at the difference between professionals and students in the quality of the refactorings they performed, the Mann-Withney U test indicates significantly different and higher scores on assignments 2, 3, 6 and 7 as can be seen in Table VIII. However, after Bonferroni correction, which we applied because of multiple hypothesis testing, the results are no longer significant.

V. DISCUSSION

Our results have shed some light on the role of unit tests during refactoring. We now come back to our research questions and also discuss threats to validity.

A. Revisiting the research questions

RQ1: Does having unit tests available during refactoring allow for quicker refactoring operations? Based on the results from our controlled experiment with 42 student and professional participants, we have to answer this question negatively. The overall time needed to perform the refactorings in our assignments was longer when unit tests were available. This effect may be explained by more code to be handled when refactoring, both for inspection and adaptation. The fact that it was not required to add unit tests for added production code will have reduced this effect.

Our results do suggest that when refactorings become more complicated (e.g., changing the type hierarchy by introduction of subtypes with for instance the refactoring "Extract subclass"), unit tests also start to have a positive effect on the time to perform a refactoring. This observation may be explained by the fact that the unit tests reduce the solution space for the refactoring through the feedback of the unit test. It has to be mentioned that the time required to write unit tests for newly added code has not been taken into account because this was not required from the subjects. When this would have been a requirement, the time required to refactor would probably have increased for some refactorings. It would be interesting to investigate the trade-off between writing these extra unit tests, versus subsequent gains from these new unit tests. This being said, based on our findings we can neither reject or accept our first null hypothesis *$H1_0$: Availability of unit tests does not lead to faster refactoring* due to a lack of information.

RQ2: Does having unit tests available during refactoring allow for higher quality refactorings? Again, the results do not show a statistically significant quality effect of the availability of unit tests while refactoring, that we could explain by the role of unit tests. Overall the null hypothesis *$H2_0$: Availability of unit tests does not lead to more correct refactoring* can neither be accepted or rejected.

Detailed inspection of the code handed in after refactoring gave suggestions for a positive role of unit tests on refactoring quality with more complicated and extensive refactorings. Here we saw some indications of feedback from unit tests pointing out the adaptations to be made. With this, we again have to remark that it was not requested to write new unit tests, an activity which could have had a positive influence on the resulting production code. Writing the tests in a test first approach may have the effect of specifying the code to write, by which the writing of the production code may become easier [25].

Because of the absence of the instruction to write new unit tests for newly created code during refactoring, we have to mention that in several cases where unit tests had to be adapted, this was done with the easiest solution that satisfied the unit test, without adding any critical test functionality. This may have influenced the role of unit tests while refactoring, but we think that does not deteriorate the validity of our results as in practical situations the same habit may exist.

B. General observations

Adapting unit tests. The behaviour of the subjects in our experiment may reflect the actual way of working of software developers rather well. In particular, we observed that a majority of subjects ran unit tests before and after each refactoring during the experiment. While these operations take extra time, they offer direct feedback to the developers. Adapting the unit tests on the other hand, requires more additional effort from the developers, while the possible gain comes at a later time. As such, the developers are less inclined to invest time to do this. This is also in line with observations by Zaidman et al. that indicate that unit tests do not always co-evolve nicely with changes to production code [26]. This may explain why our subjects, in several cases, choose the fastest path while adapting unit tests.

Running unit tests. Because unit tests were run before and after refactoring by almost all subjects our result do not allow to draw conclusions about the following hypotheses where we try to relate running unit tests to time required for refactoring and the quality of refactoring.

- $H3_0$: Running unit tests before refactoring does not impact the time needed to perform a refactoring.
- $H4_0$: Running unit tests before refactoring does not impact the quality of a refactoring.

- $H7_0$: Running unit tests after refactoring does not impact the time needed to perform a refactoring.
- $H8_0$: Running unit tests after refactoring does not impact the quality of a refactoring.

Inspection of unit tests. In relation to the following hypotheses, related to the inspection of unit tests, our results do not allow us to reject either of these null hypotheses.

- $H5_0$: Inspection of unit tests during refactoring does not impact the time needed to perform a refactoring.
- $H6_0$: Inspection of unit tests during refactoring does not impact the quality of a refactoring.

In this case, the complex situation with its many parameters may have prevented results that allow straightforward rejection of these hypotheses.

General. In general we come to the conclusion that the effect of unit tests on refactoring is definitely not as straightforward as we would expect. From the resulting code handed in by our subjects we conclude that in the case of unit testing and refactoring, practice is something very different from theory. While theory explains how unit testing makes refactoring easier, e.g., Feathers [25], we could not observe this in our experiment, neither in a quicker refactoring process, nor in higher quality of code.

C. Threats to validity

This section discusses the validity threats in our experiment and the manners in which we have addressed them.

Internal validity. In relation to the quality of refactorings with and without unit tests, we should mention that we did not take into account the quality of the (adapted) unit tests and only looked at the quality of the production code. As the control group in our experiment did not have unit tests available we think that the only straight comparison of the quality of refactoring between this group and the experimental group with unit tests available is to only compare the production code that both groups wrote.

Another threat to the internal validity may be the subjectivity in quality ratings of the code handed in. In order to mitigate this threat we set up a solution model and grading scheme to ensure consistent grading. Furthermore, to verify the soundness of the grading process, the authors of this paper independently graded seven refactoring solutions. Five out of those seven were graded exactly the same, one assignment was rated with a 0.2 difference and one assignment was rated with a difference of 1 one a scale of 0 to 10. With a total difference of 1.2 on a total of 70, this suggests a high inter-rater reliability.

External validity. Threats to the external validity of our results are the possibility of non-representative subjects, tasks and task conditions. Against each of these threats we took some precautions as described below.

Our subjects were mainly final year professional bachelor students who have had three years of practical training in object-oriented programming and unit testing, including a one semester internship. To support their refactoring knowledge and experience they followed a six hour training dedicated to good refactoring practices and the use of unit tests therein. As such, they were thoroughly prepared for the task at hand and judging from the assignments they handed in, they were able to perform all the refactoring assignments. In order to make our set of subjects more realistic, we extended our group of 35 students with 7 professional subjects. Results show that they too were able to complete all assignments, albeit slightly faster and qualitatively better, which can be explained by the higher level of experience.

As always the (small) number of participants is a risk to the external validity of our results. Our statistical results thus have to be regarded from this perspective. To support our statistical analysis we added qualitative data by describing the refactoring internals.

The object of research, JPacman, is smaller than an industrial software system. However, we still think that working on it in the context of refactorings approximates real life situations, because the system has substantial complexity and the refactorings to be performed only had local impact, which is typical for most refactorings.

Furthermore, concerning the refactoring assignments, we chose 7 different refactorings from four different categories varying in impact and complexity (see Section I).

Another aspect that can be seen as a threat to the external validity is the experimental setting where time and quality might play a different role than in real live situations. This is a complicated issue as in real live situations the role and influence of time and quality constraints can vary a lot. In the experiment there were no time constraints, which we think at least did not reduce the role of unit tests in refactoring. In relation to quality of refactorings we asked to refactor as good as possible. As participation in the experiment for the students was part of a course they were graded for their participation. The binary criterion for passing this course, was thus a serious attempt to refactor as good as possible. We chose not to give them an explicit grade, because we feel this is less in accordance with real life situations.

Construct validity. The scattered results we found might suggest a follow-up experiment in which we control how and when unit tests are used during the experiment. Yet, we question if this would increase the validity of the results. Adding constraints to the behavior of the subjects to our opinion would make their behaviour less realistic and therefore would say less about the real role of unit tests and refactoring in practice. By taking both the concept of unit testing and refactoring "by the book" and putting them in realistic situations, as described above, we think our results relate to the concepts as theoretically defined.

VI. Conclusion

In this paper we have reported on an experiment in which we investigated the role of unit tests during refactoring. We designed seven refactoring assignments where refactorings were taken from different categories related to the impact they have on the related unit tests. In a two-group controlled experiment the experimental group had unit tests available while the control group had to do the refactorings without unit tests. As results we looked at the time needed to perform the refactorings and the quality of the resulting code.

From the results we found, we can not conclude that having unit tests available during refactoring leads to quicker refactoring operations *(RQ1)*. With simple refactorings, having unit tests is related to longer refactoring. With more complex refactorings this effect is less strong. The feedback from the unit tests may support faster refactoring but next to that inspecting, adapting, adding and running the unit tests add to the time needed to do a refactoring, in total leading to longer times to do the refactorings as related to the situation without unit tests.

In relation to the quality of refactorings we did not witness that having unit tests available during refactoring increases the quality of the resulting production code *(RQ2)*. We see a slight effect that suggests that for more complex refactorings feedback from unit tests may scaffold the refactoring operations.

To conclude, our investigation suggests that the relation between unit testing and refactoring in practice is less obvious than theory may predict. In particular, while there might be benefits from having unit tests available in terms of preserving correctness of the software system during development and maintenance, we did not observe a link between a decrease in time and an increase in the quality of the resulting production code from having unit tests available during refactoring.

Future work

In the future, we aim to further investigate the role, possible benefits and costs of unit tests during refactoring. As discussed in Section V, a next step in this line of research is to get a clear view on the practical role of unit tests in practical situations, e.g., through a longitudinal study.

Acknowledgements

This work was funded by the RAAK-PRO project EQuA (Early Quality Assurance in Software Production) of the Foundation Innovation Alliance, the Netherlands. Our gratitude goes out to the volunteers from the Software Improvement Group that participated in our experiment.

References

[1] W. F. Opdyke and R. E. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proc. of Symp. on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, 1990.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the design of existing programs*. Addison-Wesley, 1999.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[4] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 25, no. 4, pp. 22–29, 2006.

[5] G. Tassey, "Economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology (NIST), Planning Report 02-3, May 2002.

[6] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, "The interplay between software testing and software evolution," in *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 173–202.

[7] K. Beck, "Embracing change with extreme programming," *Computer*, vol. 32, no. 10, pp. 70–77, 1999.

[8] A. van Deursen, "Program comprehension risks and benefits in extreme programming," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2001, pp. 176–185.

[9] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman, "Visualizing testsuites to aid in software understanding," in *Proc. Conf. on Softw. Maintenance and Reengineering (CSMR)*. IEEE, 2007, pp. 213–222.

[10] D. Janzen and H. Saiedian, "Does test-driven development really improve software design quality?" *IEEE Software*, vol. 25, no. 2, pp. 77–84, 2008.

[11] A. van Deursen and L. Moonen, "The video store revisited–thoughts on refactoring and testing," in *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, 2002, pp. 71–76.

[12] E. Babbie, *The practice of social research*. Wadsworth Belmont, 2007, 11th edition.

[13] J. Pipka, "Refactoring in a test first-world," in *Proceedings of the International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, 2002.

[14] W. Basit, F. Lodhi, and U. Bhatti, "Extending refactoring guidelines to perform client and test code adaptation," *Agile Processes in Software Engineering and Extreme Programming*, pp. 1–13, 2010.

[15] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Extreme Programming Perspectives*. Addison-Wesley, 2002, pp. 141–152.

[16] B. George and L. Williams, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, pp. 337–342, 2004.

[17] L. Williams, G. Kudrjavets, and N. Nagappan, "On the effectiveness of unit test automation at microsoft," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 2009, pp. 81–89.

[18] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[19] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2012, pp. 11–20.

[20] D. Athanasiou, "Constructing a test code quality model and empirically assessing its relation to issue handling performance," 2011, http://resolver.tudelft.nl/uuid:cff6cd3b-a587-42f2-a3ce-e735aebf87ce.

[21] S. Counsell, R. Hierons, R. Najjar, G. Loizou, and Y. Hassoun, "The effectiveness of refactoring, based on a compatibility testing taxonomy and a dependency graph," in *Testing: Academic & Industrial Conf. on Practice And Research Techniques (TAIC-PART)*. IEEE, 2006, pp. 181–192.

[22] M. Greiler, A. van Deursen, and A. Zaidman, "Measuring test case similarity to support test suite understanding," in *Proceedings of the International Conference on Objects, Models, Components, Patterns (TOOLS)*, ser. LNCS, vol. 7304. Springer, 2012, pp. 91–107.

[23] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 1999.

[24] S. Demeyer, B. Du Bois, M. Rieger, and B. Van Rompaey, "The LAN-simulation: A refactoring lab session," in *Proc. Workshop on Refactoring Tools (WRT)*. University of Berlin, 2007, p. 53.

[25] M. C. Feathers, *Working Effectively with Legacy Code*. Prentice Hall, 2004.

[26] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production and test code," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2008, pp. 220–229.