Effects of developer experience on learning and applying Unit Test-Driven Development

Roberto Latorre

Abstract—Unit Test-Driven Development (UTDD) is a software development practice where unit test cases are specified iteratively and incrementally before production code. In the last years, researchers have conducted several studies within academia and industry on the effectiveness of this software development practice. They have investigated its utility as compared to other development techniques, focusing mainly on code quality and productivity. This quasi-experiment analyzes the influence of the developers’ experience level on the ability to learn and apply UTDD. The ability to apply UTDD is measured in terms of process conformance and development time. From the research point of view, our goal is to evaluate how difficult is learning UTDD by professionals without any prior experience in this technique. From the industrial point of view, the goal is to evaluate the possibility of using this software development practice as an effective solution to take into account in real projects. Our results suggest that skilled developers are able to quickly learn the UTDD concepts and, after practicing them for a short while, become as effective in performing small programming tasks as compared to more traditional test-last development techniques. Junior programmers differ only in their ability to discover the best design, and this translates into a performance penalty since they need to revise their design choices more frequently than senior programmers.


1 INTRODUCTION

Test-Driven Development (TDD) is a technique to incrementally develop software where test cases are specified before functional code [1], [2], [3]. Originally, TDD referred to creating unit tests before production code. However, recently, another technique applying a test-driven development strategy at the acceptance test level is gaining attention [4], [5]. In this sense, in the last years, it is usual to distinguish between Unit Test-Driven Development (UTDD), which targets unit tests to ensure the system is performing correctly; and Acceptance Test-Driven Development (ATDD), a technique focused on the business level. Here we aim our attention at UTDD.

Despite its name, UTDD is not a testing technique, but a programming/design practice [6], [7], [8]. In the last years, several studies and experiments have analyzed the influence of this practice on software in terms of software quality and productivity within academia and industry (e.g., see [9], [10], [11], [12]). In literature, UTDD sometimes appears as one of the most efficient software development techniques to write clean and flexible code on time [13], [14], [15], [16]. Nevertheless, these studies report conflicting results (positive, negative and neutral about the use of UTDD) depending on several factors. Thus, no definite conclusions can be drawn, which limits the industrial adoption of this development practice [17].

One of the main problems with UTDD is the difficulty in isolating its effects from other context variables. So, the influence of these context variables must be analyzed in detail to determine the real benefits and cost of adopting this technique. For example, Causevic et al. [17] identified seven possible factors limiting the industrial adoption of this development technique. One of these factors is the programmers’ experience. Nevertheless, as far as we know, just a few empirical studies investigate directly or indirectly the effect of developer experience on applying UTDD. In the context of these studies, experience (or knowledge) usually refers to the degree of practice or theoretical insights in UTDD, and the qualification of the participants ranges between UTDD novices and UTDD experts, where novices are often students. These studies and reports, for example, analyze the correlation between programming experience and code quality [18], highlight some aspects contributing to the adoption of a TDD strategy in an industrial project [19], or compare the characteristics of experts’ and novices’ UTDD development process [20], [21]. Results here are conflicting again.

In this paper, we are interested in the impact of developer experience on learning (with demonstrated ability to apply) UTDD. Therefore, all the participants in our study are UTDD novices. By experience, here we do not refer to the particular level of experience of a programmer in UTDD, but to his/her general experience in the professional development context. We argue that success of adopting UTDD is significantly related to the developer’s experience level. It seems obvious that the success of learning and properly using UTDD in an industrial project depends on the skills and previous experience of the team members in areas such as programming, testing, designing and/or refactoring.

The general perception about UTDD is that it is difficult to learn and apply, requiring much mastery and
a higher level of discipline from the developers than other traditional software development practices [13], [14], [22], [15], [23]. In this paper, we address a controlled quasi-experiment involving 25 industrial programmers with different skills, assessing how difficult it is to actually learn and properly apply UTDD in a first exposure to this software development practice. After an initial training, the research subjects have to individually implement a set of requirements by using UTDD. The efficiency of the learning process is measured as a trade-off between the correct application of the UTDD concepts and the additional time for the development due to the learning curves with UTDD. Then, we study the evolution during the experiment of the development process conformance to UTDD, and the effective development time and the self-training needed to completely implement a set of requirements. In our analysis, we first verify that the functional code produced by the subjects during the experiment satisfies the requirements and has the expected behavior. Then, we estimate the extent to which the development process remains faithful to the principles of UTDD, and calculate and analyze the learning curves for the different subjects and developer experience levels. Finally, we evaluate the effect of these learning curves on performance. Our results point out that the knowledge in UTDD becomes better as participants progress with the experiment and suggest that experienced professional developers are able to quickly learn the UTDD concepts and, after practicing them for a short while, become as effective in performing small programming tasks. Although at the end of the experiment all the subjects but one are able to properly apply UTDD, junior programmers show a performance penalty mainly due to limitations in their ability to design.

The remainder of the paper is organized as follows. In Section 2 we formalize the UTDD process and explain how we measure UTDD process conformance. Section 3 gives details of the experimental design. Results are reported in Section 4. Section 5 addresses threats to validity. And, finally, Section 6 summarizes our findings and draws the conclusions.

2 FORMALIZATION OF THE UTDD PROCESS

Any software development process consists of multiple programming tasks. Evaluating the degree to which a process conforms to UTDD is not easy because it requires the analysis of multiple process measures. A formal definition of the UTDD practice is then needed to facilitate measurement of process conformance.

UTDD is based on a minute-by-minute iteration between writing unit tests and writing functional code. Before writing functional code, the developers write unit test cases (preferably automated unit tests) for the new functionality they are about to implement. Just after that, they implement the production code needed to pass these test cases. The following iterative and incremental cycle defines a typical UTDD implementation:

1. Write a unit test.
2. Run all tests and see the new one fails.
3. Implement just enough code to pass the new test.
4. Run the new test and all other unit test cases written for the code. Repeat the process from step 3 until all tests are passed.
5. Refactor the code. Refactoring includes modifications in the code to reduce its complexity and/or to improve its understandability and maintainability.
6. If refactoring, re-run all tests and correct the possible errors.
7. Go to the next functionality and repeat the process from step 1. New functionalities are not considered as proper implementations unless the new unit test cases and all the prior tests are passed.

According to Beck [1], such as development process contains three different types of programming tasks:

- **Red tasks**, related to write a test that fails (steps 1 and 2).
- **Green tasks**, to make the test work quickly (steps 3 and 4).
- **Refactor tasks** that eliminate all duplications from both test code and application code (steps 5 and 6).

Obviously, the order of these tasks is crucial to the conformance of the development process to UTDD, which has given rise to the so called UTDD mantra: red/green/refactor. Note that refactoring may be omitted if unnecessary. A UTDD implementation process consists of a sequence of UTDD cycles maintaining a continuous red/green/refactor rhythm. Therefore, to determine whether the research subjects actually learned and applied this practice during the experiment, we evaluate the existence of red/green/refactor sequences during the development.

Taking into account the above definitions, we estimate the extent to which the research subjects remained faithful to the principles of UTDD by using a metric proposed by Müller and Höfer [20]:

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

(1)

The larger the value of the metric, the higher the conformance to the UTDD rules.

To classify the changes made by a programmer during the development we consider as:

- **UTDD changes**, the changes satisfying that unit tests were written before related pieces of the application code. We include in UTDD changes those cases where the subject does not validate that the test fails after a test code change (weak UTDD change). To identify UTDD changes we apply the same rules proposed by Müller and Höfer [20].
- **Refactorings**, the changes in the structure of the code that do not alter its observable behavior.

3 STUDY DEFINITION

The research question this study aims to address is: How does the developer’s experience affect the effort required to...
This section describes our study design, planning and execution following well-known guidelines on experimental software engineering [24], [25]. We define the study to be able to compare from different perspectives the learning curves with UTDD of professional developers with different levels of experience, but all of them inexperienced in test-first development practices. The collected data include, between others, all the changes made to the functional and test code during the whole learning process and the time-stamp and the result of each unit test invocation. These data allow us to analyze the effort required for adapting the programmers’ mindset to UTDD.

3.1 Study subjects

All the participants in the study were industrial developers with computing degrees and several years of experience using traditional methodologies based on test-last development strategies. However, since our main goal was to compare their learning curves with UTDD, not having a previous experience with test-driven development strategies was mandatory. The subjects belonged to three different companies and voluntarily participated in the study because of a personal invitation.

In an experiment with students, Kollanus and Isomöttönen [26] classify the difficulties the subjects find with the adoption of UTDD in three main categories: UTDD approach difficulties, designing of test problems and technical difficulties. This and other studies point to present examples of UTDD incremental cycles and additional UTDD patterns. There are several patterns that can be used during a UTDD process (see [1], Section III).

1. http://www.junit.org


For the sake of ensuring that the subjects’ skills were the ones required and that no significant differences existed between the groups of subjects, we performed pre-study surveys evaluating their programming, designing and testing expertise; and their knowledge and experience in the particular development/testing environment and programming language. After that, we excluded five participants (one junior, two intermediates and two seniors) because of the following reasons: the junior programmer had not enough knowledge of JUnit; while the intermediates and the seniors did not currently work in a technical job. Therefore, finally, the study involved 25 participants: 9 juniors (J1-J9), 8 intermediates (I1-I8) and 8 seniors (S1-S8).

3.2 Study material

The experiment consisted of individually designing and programming a set of back-end requirements for a Java application, together with their corresponding JUnit automated unit tests. Note that UTDD is not easily applicable to front-end development [6]. The application we selected for the study was a simple project planning tool used during the course of “Project of Software Analysis and Design” imparted by the author in the degree in Computer Science at Universidad Autónoma de Madrid. This allowed us to use the acquired knowledge during the course with the students to select and classify the requirements used as study material into three categories according to their complexity level: easy requirements, moderate requirements and difficult requirements. In the Appendix we include three examples of requirements.

During the experiment, requirements were provided to the subjects in Concordion2 by using a textual description and a set of acceptance tests as described in [19]. The requirements were tested during the experiment by running the corresponding acceptance tests included in the Concordion specification (see below).

3.3 Previous training

As none of the research subjects had a previous exposure to test-first development practices, before starting the development they received a brief course about UTDD. In all cases, the instructor was the same person to avoid differences in the training. The training consisted of two short sessions of three hours each. The time required for these sessions was not included in our analysis. The first session was dedicated to introduce the UTDD main topics. The second was a practical session dedicated to present examples of UTDD incremental cycles and typical UTDD patterns. There are several patterns that can be used during a UTDD process (see [1], Section III).
Some of them are tricks, some are design patterns and some are refactorings. During training we focused on implementation strategies, also called green bar patterns (see [29], Section 3.3.1.1), and refactorings.

Empirical studies suggest an initial resistance to use UTDD due to inexperience [30], [9], [31]. They highlight that when no support for UTDD is available, inexperienced UTDD developers tend to return to more traditional techniques. In our case, we were interested in quantifying this initial resistance within the learning curves with UTDD. Therefore, during the training, we insisted in the importance of properly applying UTDD along the experiment, but the participants did not receive any additional training afterwards. They could search by themselves information about UTDD both in written and online literature, but more support did not take place. The self-training time would be tracked and analyzed during the subsequent analysis.

### 3.4 Study design

Although there was not actually an explicit time limit to finish the implementation, we adopted a very simple study design (see Fig. 1) for the participants to be able to finish the experiment in around one month during their compressed working day of summer 2012. After the initial training, each subject had to individually implement the same 24 requirements. For privacy purposes, results of the experiment were made anonymously.

Three of the seniors and two juniors worked on the development during their official workday in their own office. The rest of the participants worked at home during their free time. All of them were free to work whenever they wanted. The only requisite was to record their working sessions (see Section 3.5 for details).

In order to minimize the effect of task difficulty, requirements were organized in eight groups of three requirements (G1-G8). Each group comprised an easy requirement (R1), a moderate one (R2) and a difficult one (R3), that the subjects had to implement in that order (R1-R2-R3). After the UTDD training, the same sequence of groups of three requirements was presented to every subject. They had to complete the development starting by G1 and ending by G8 (Fig. 1). The participants had to work on each group of requirements until all written unit test cases were properly passed and they thought they had finished the job. At this moment, the Concurrency acceptance tests were executed to validate the new implementations. A new set of requirements was not considered unless the acceptance tests associated to the three requirements of the prior set and all previous acceptance test were properly tested. Hereafter, we call iteration to the process of completely developing a set of three requirements. Each iteration includes several UTDD cycles.

To compare UTDD and non-UTDD metrics and establish baselines for comparisons between the different research subjects, we instructed them to develop the groups of requirements G1 and G2 (a total of six requirements) with a test-last strategy without any constraint. That is, each developer could use the method he/she preferred. This provided an estimation of the design, development and testing abilities of each subject. The other six groups of requirements (G3-G8) should be developed by using UTDD. Note that although UTDD was not used with the groups of requirements G1 and G2, developers had to implement the JUnit tests for these requirements too. The only difference was when these tests were written ("after" vs. "before" the production code under test).

#### 3.5 Study variables and data collection

The dependent variables of our study were:

- The task correctness and completeness, i.e., the application code generated by the subjects during the experiment should work and do what it was supposed to do.
- The process conformance to UTDD, i.e., the extent to which the subjects followed the UTDD rules during the experiment (see Section 2 for details).
- And the programming performance, i.e., the time required to correctly develop a given set of three requirements. During the experiment, the variability in the resting times in between consecutive programming sessions varied from two hours to five days, and, consequently, there were significant differences between the total amount of time required for the different subjects to develop the 24 requirements (from two weeks to a little more than a month). At this regard, it is important to take into account that we were not interested in this overall development time, but in the effective development time spent in each set of requirements. In this time we included both active (described as typing and producing the code) and passive development time (spent on reading the source code, self-training, looking for a bug, etc) [32]. This effective development time was not correlated with the time needed to finish the experiment.

To collect the data needed to measure these variables, we combined the use of questionnaires the subjects had to fill in during the experiment (Section 3.5.1), and automatic and manual tracking tools (Section 3.5.2). Both during the development and the analysis phase, we reviewed and validated coherence between qualitative and quantitative data.

#### 3.5.1 Questionnaires

After the implementation of each set of three requirements (see Section 3.4 for details), the subjects had to fill in with answers a form with six questions about the requirements just implemented. From here on, we refer to these forms as intermediate questionnaires. They consisted of the following two questions for each requirement (R1, R2 and R3):
Fig. 1. Schematic representation of the study design. After a UTDD training, the research subjects implemented eight groups of requirements (G1-G8), with three requirements per group (R1-R2-R3). The groups of requirements were completed always in the same order. G1 and G2 were developed with a test-last development practice (control strategy); and G3-G8 by using UTDD.

1) How difficult has it been for you to understand the requirement? (1-5)
2) How many changes/corrections in the structure of the existing code have you needed to successfully implement the requirement?

The first question was expressed on a five-point Likert scale from 1 (“Extremely easy”) to 5 (“Extremely difficult”). It aimed at better explaining the experiment results, addressing the clarity of the requirements and the ability of subjects to understand them. The second question aimed at gaining insights about the subjects’ behavior during the experiment. The answers to this question helped us to classify the changes in the functional code during the analysis phase.

At the end of the experiment, the subjects had to answer two more questions that tried to measure the acceptance of UTDD:
- Q1. How difficult has it been for you to learn UTDD? (1-5)
  Being 1 = Extremely easy; 3 = Neither easy nor difficult; and 5 = Extremely difficult
- Q2. How useful do you feel UTDD is? (1-5)
  Being 1 = Not at all useful; 2 = Slightly useful; 3 = Moderately useful; 4 = Very useful; and 5 = Extremely useful.

3.5.2 Tracking tools

The tracking tools were a set of Eclipse plugins that collected data about the software development process for each subject during the programming sessions. This data collection mechanism requires a running Eclipse programming environment where to install the plugins, but it does not require any other prerequisites. Using a similar strategy to the one used in earlier empirical studies about UTDD [20], [33], some of the tracking tools ran in the background of the Eclipse IDE and collected data transparently for the participants. The main goal of these data was to detect deviations from UTDD.

To integrate new code during the experiment, we used a Subversion repository (SVN). We assumed that participants were not going to commit changes in this repository as frequently as we needed for our analysis. Then, we used a first Eclipse plugin to transparently commit in a different SVN repository a new revision of a file (both functional and test code) each time the subject saved it locally. Additionally, a second plugin automatically saved in a database information about the JUnit test invocations. This information included the start and end time-stamp of all the invocations along with their result. All these data allowed us (i) to determine whether the new unit test cases were written after or before the application code and (ii) to calculate deltas of all the files in the project.

In addition to the transparently collected data, the subjects had to manually track their development activities and the required time to develop each requirement. This time included self-training (in the case of using UTDD), requirement comprehension, designing, unit tests and functional code implementation, refactoring and testing. To keep track of the time the subjects spent in each of these tasks, we used BaseCamp.

Automated collected data and BaseCamp data were daily checked during the development for consistency.

4 ANALYZING THE STUDY RESULTS

4.1 Task correctness and completeness

We first evaluate how correct/complete are the programs produced by the research subjects relative to the requirements. If the application code generated by a subject does not do what it is supposed to do, it makes no sense to do further analysis of that subject’s learning curve.

For this purpose, we use the Concordion acceptance tests. As we indicate above, during the development, after the implementation of each set of three requirements, the corresponding acceptance tests and all the previous ones were run to check that the new code had the expected behavior. If some of these tests failed, the subject had to solve the problem in a new UTDD cycle. In this manner, we ensure that at the end of the experiment all the research subjects’ application code passes all the acceptance tests.

4.2 Learning process

Table 1 shows the distribution of the subjects’ answers to the first question in the final questionnaire (“Q1. How
TABLE 1

Distribution of the subjects’ responses to questions belonging to the final questionnaire (Q1 and Q2). See Section 3.5.1 for details. The questions are expressed on a Likert scale as follows: Q1: 1 = Extremely easy; 2 = Easy; 3 = Neither easy nor difficult; 4 = Difficult; and 5 = Extremely difficult. Q2: 1 = Not at all useful; 2 = Slightly useful; 3 = Moderately useful; 4 = Very useful; and 5 = Extremely useful.

<table>
<thead>
<tr>
<th></th>
<th>Juniors</th>
<th>Intermediates</th>
<th>Seniors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 2. Comparison of the average percentage of self-training time during the development of the sets of requirements G3-G8 for juniors, intermediates and seniors.

difficult has it been for you to learn UTDD?”). These answers allow us to analyze the subjects’ personal perception of their own learning processes. Qualitative evidence points out that, independently of their experience level, the subjects found UTDD easy to learn, since 24 out of 25 subjects answered that the learning process was easy (22 subjects, 88%, gave response 2) or extremely easy (2 subjects, 8%, gave response 1). Only one subject (J9) considered it difficult (response 4). This result is a promising starting point, but, nevertheless, it does not reflect whether the subjects actually learned and applied UTDD during the experiment.

During the development, we tried to ensure that UTDD was properly applied by continuously examining deltas and code samples to confirm that JUnit tests were written in step with the functional code. These observations supported that, in general, a test-first development approach was being employed, but they did not allow neither the process conformance with UTDD to be measured, nor the learning curves to be quantified.

To start addressing a more quantitative analysis of the learning process, Fig. 2 compares the average self-training time during the development of the sets of requirements G3-G8 for juniors, intermediates and seniors. Note that the UTDD development started with G3 and finished with G8. Sets G1 and G2 are not included in the figure because they were not developed by using UTDD and, therefore, there was no self-training during their implementation. In terms of effort used, during the first set of requirements developed with UTDD, juniors used ≈11% of effort for self-training, intermediates ≈17% and seniors ≈9%. Independently of the programmer experience level, this effort dropped quickly in the subsequent iterations, until being 0 in just three or four iterations. Note that intermediate programmers spent more time for self-training than junior programmers. These data are in line with the results presented in Table 1 and they suggest that the knowledge in UTDD quickly improved as the development progressed. Although they do not reflect yet whether the technique was or not actually used during the experiment.

To quantitatively assure that the research subjects remained faithful to UTDD during the development, we compare the process conformance (Eq. 1) in our experiment with the values reported by Müller and Höfer for UTDD experts and UTDD novices [20]. The evolution of this metric along the development of the different groups of three requirements (G3-G8) also allows to quantify the learning curves with UTDD of a given subject or group of subjects (see below). As a first step for this analysis, we had to classify all the changes made by the subjects in the functional code into three types according to their nature: UTDD changes, refactorings and other changes. This classification was carried out semi-automatically. On one hand, UTDD changes could be easily identified automatically by using collected data and deltas from the SVN repository. On the other hand, the rest of changes might belong to a refactoring and had to be inspected and classified manually. Manual classification of changes was a hard task because, unlike [20], we performed lots of manual inspections (cf. 34 vs. 332 changes to be classified manually for a subject in the worst case).

Left panels in Fig. 3 show the classification of all the changes made by each individual research subject during the implementation of the distinct sets of requirements developed by using UTDD. Initially, there existed a great variability in the ratios of changes for the three groups of programmers. For example, during the development of G3, there were subjects with a ratio of UTDD changes greater than 80% (or even equal to 100%) in the three groups. But, also in the three groups, there were subjects with a ratio of other changes greater than 50%. As the development progressed, this variability decreased mainly due to the increasing number of UTDD changes. Nevertheless, note that those subjects with the larger ratios of UTDD changes during the initial requirements tended to increase the ratio of other changes as the experiment progressed (e.g., see J6 or S1). In the case of intermediate and senior programmers, each ratio
Fig. 3. Left panels: Classification of the changes made individually by the subjects during the development of each set of requirements. Right panels: Classification of changes grouped by developer’s experience level (junior, intermediate and senior). Sets G1 and G2 are not included because they were not developed by using UTDD.
of changes reached a steady similar value for all the subjects belonging to the same group (cf. G7 and G8). In the case of juniors, this did not happen, although a few more iterations would likely lead to an equivalent result. This general behavior is also observed when we globally analyze the juniors’, intermediates’ and seniors’ development process. Right panels in Fig. 3 show the classification of changes by group of programmers and set of requirements. These panels lead to the following observations:

- All the ratios of changes tended to stabilize in a nearly constant value as the study progressed.
- In general, the ratio of UTDD changes grew; while the ratio of other changes dropped. This result confirms that self-training during the development of the initial sets of requirements (Fig. 2) translated into a better knowledge in UTDD.
- In average, the number of UTDD changes was larger (and similar) for intermediate and senior programmers than for junior developers.
- The ratio of refactorings for seniors was nearly constant during the whole experiment; but, at the end, juniors made more refactorings.

An interesting result observed in our experiment is that UTDD conformance (Eq. 1) is 0 for all the subjects during the development of G1 and G2 (the control groups of requirements developed with a non-TDD strategy). This implies that no refactorings were made during the development of these requirements. In the same way, during the early stages of the UTDD implementation, refactorings were not very usual (Fig. 3), and even some subjects did almost no refactoring of their code during the whole experiment (e.g. I6 and I7). These results suggest that refactorings may not be a commonly performed type of change.

Table 2 shows the conformance to the UTDD rules for each subject and set of requirements developed with UTDD (columns G3-G8). To investigate whether the subjects actually learned the development technique, we focus on the last group of requirements (column G8). In the experiments by Müller and Höfer, conformance to UTDD process for a group of UTDD experts varies from ≈64% to ≈96%; while for a group of UTDD novices it varies from 0% to ≈94%. Note that both ranges overlap, so, when an individual developer has a conformance greater than 64%, it is difficult to identify whether he/she behaves as an UTDD expert or as an UTDD novice. In our case, during the development of the last set of requirements, all the research subjects but one (J9) achieved a process conformance in the range 78%–95%. These values are high enough to consider that the corresponding subjects remained faithful to UTDD. Therefore, 24 out of 25 subjects (96%) were able to quickly learn the UTDD concepts during the study and effectively use them when this concluded. These results corroborate the qualitative evidence pointing out that, independently of the experience level, UTDD was easy to learn. The question now is “how easy” was the learning process. To answer this question, we analyze and compare the evolution during the experiment of the UTDD process conformance between different subjects and groups of programmers. We can consider that this evolution characterizes the learning curves, since it shows how the knowledge in UTDD evolved during the study. For example, the subject I4 started out the development with ≈31% process conformance for the set of requirements G3, and ended with ≈85% for G8. The increase conformance points out that this subject adapted his mindset to UTDD along the experiment.

Figure 4 shows the juniors’, intermediates’ and seniors’ learning curves with UTDD. In the case of groups of programmers, these learning curves also allow to characterize the behavior of the group during the experiment by comparing our results with the average UTDD conformance reported by Müller and Höfer [20] for a group of UTDD experts (82%) and for a group of students novices on UTDD (67%). As expected after analyzing the ratios of changes, in general, UTDD conformance grew as the development advanced, i.e. knowledge in UTDD became better. So that, at the end of the experiment, the three groups of programmers achieved a high conformance to UTDD: 79% for juniors, 84% for

---

**TABLE 2**

<table>
<thead>
<tr>
<th>G3</th>
<th>G4</th>
<th>G5</th>
<th>G6</th>
<th>G7</th>
<th>G8</th>
<th>C-Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>58.2</td>
<td>60.4</td>
<td>65.0</td>
<td>75.8</td>
<td>80.8</td>
<td>80.3</td>
</tr>
<tr>
<td>J2</td>
<td>44.2</td>
<td>58.4</td>
<td>71.4</td>
<td>80.5</td>
<td>65.9</td>
<td>81.3</td>
</tr>
<tr>
<td>J3</td>
<td>56.6</td>
<td>59.4</td>
<td>78.7</td>
<td>63.4</td>
<td>84.4</td>
<td>81.9</td>
</tr>
<tr>
<td>J4</td>
<td>59.1</td>
<td>60.8</td>
<td>64.0</td>
<td>74.7</td>
<td>77.9</td>
<td>78.7</td>
</tr>
<tr>
<td>J5</td>
<td>92.3</td>
<td>84.8</td>
<td>86.6</td>
<td>87.1</td>
<td>85.1</td>
<td>82.9</td>
</tr>
<tr>
<td>J6</td>
<td>100</td>
<td>93.1</td>
<td>90.2</td>
<td>90.9</td>
<td>95.4</td>
<td>93.7</td>
</tr>
<tr>
<td>J7</td>
<td>46.7</td>
<td>48.0</td>
<td>70.1</td>
<td>46.0</td>
<td>84.7</td>
<td>80.0</td>
</tr>
<tr>
<td>J8</td>
<td>37.7</td>
<td>39.2</td>
<td>82.5</td>
<td>79.7</td>
<td>53.4</td>
<td>82.3</td>
</tr>
<tr>
<td>J9</td>
<td>27.6</td>
<td>32.9</td>
<td>41.7</td>
<td>37.6</td>
<td>44.7</td>
<td>46.1</td>
</tr>
<tr>
<td>I1</td>
<td>100</td>
<td>98.6</td>
<td>94.5</td>
<td>76.0</td>
<td>74.1</td>
<td>83.1</td>
</tr>
<tr>
<td>I2</td>
<td>74.6</td>
<td>85.1</td>
<td>88.0</td>
<td>93.3</td>
<td>83.1</td>
<td>86.3</td>
</tr>
<tr>
<td>I3</td>
<td>62.7</td>
<td>63.8</td>
<td>78.3</td>
<td>77.1</td>
<td>84.7</td>
<td>82.8</td>
</tr>
<tr>
<td>I4</td>
<td>30.6</td>
<td>37.3</td>
<td>62.3</td>
<td>71.0</td>
<td>78.0</td>
<td>84.5</td>
</tr>
<tr>
<td>I5</td>
<td>79.0</td>
<td>78.1</td>
<td>81.2</td>
<td>86.0</td>
<td>83.8</td>
<td>85.2</td>
</tr>
<tr>
<td>I6</td>
<td>61.4</td>
<td>62.0</td>
<td>70.8</td>
<td>78.1</td>
<td>82.0</td>
<td>84.1</td>
</tr>
<tr>
<td>I7</td>
<td>52.9</td>
<td>52.5</td>
<td>71.7</td>
<td>74.0</td>
<td>79.4</td>
<td>78.7</td>
</tr>
<tr>
<td>I8</td>
<td>45.3</td>
<td>58.7</td>
<td>68.2</td>
<td>76.4</td>
<td>86.8</td>
<td>85.1</td>
</tr>
<tr>
<td>S1</td>
<td>100</td>
<td>100</td>
<td>88.7</td>
<td>91.2</td>
<td>87.2</td>
<td>89.1</td>
</tr>
<tr>
<td>S2</td>
<td>66.6</td>
<td>77.0</td>
<td>86.4</td>
<td>85.7</td>
<td>83.5</td>
<td>85.1</td>
</tr>
<tr>
<td>S3</td>
<td>43.5</td>
<td>54.2</td>
<td>69.2</td>
<td>78.6</td>
<td>94.5</td>
<td>82.2</td>
</tr>
<tr>
<td>S4</td>
<td>83.3</td>
<td>89.4</td>
<td>85.8</td>
<td>87.2</td>
<td>87.3</td>
<td>86.8</td>
</tr>
<tr>
<td>S5</td>
<td>80.3</td>
<td>80.6</td>
<td>87.8</td>
<td>93.1</td>
<td>86.0</td>
<td>86.6</td>
</tr>
<tr>
<td>S6</td>
<td>74.3</td>
<td>83.3</td>
<td>89.1</td>
<td>95.6</td>
<td>94.9</td>
<td>94.5</td>
</tr>
<tr>
<td>S7</td>
<td>94.1</td>
<td>94.0</td>
<td>92.8</td>
<td>87.1</td>
<td>88.3</td>
<td>86.2</td>
</tr>
<tr>
<td>S8</td>
<td>68.4</td>
<td>85.9</td>
<td>93.9</td>
<td>86.9</td>
<td>95.7</td>
<td>83.7</td>
</tr>
</tbody>
</table>

---

0098-5589 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
intermediates and 87% for seniors. If we compare these values with the ones reported by Müller and Höfer, we can consider that the three groups of programmers reached the goal of learning and applying UTDD in a similar manner to an expert group. The difference between the three groups was on how they achieved this final development process conformance to UTDD (learning process). If we consider G3 as the starting point of the learning process, junior and intermediate programmers started out with a similar UTDD conformance to the reported for a UTDD novice group (cf. ≈58% for juniors and ≈63% for intermediates, and 67% reported by Müller and Höfer for a group of students novices on UTDD). Furthermore, in these cases, there was a high variability between the different subjects, a property also observed in an UTDD novice group [20]. Therefore, junior and intermediate programmers needed a few iterations to practice and consolidate the knowledge before effectively applying the TDD concepts. They started the development with a low TDD conformance and became as effective as the experiment progressed. In contrast, senior programmers showed a high and nearly constant average UTDD conformance from the beginning (cf. ≈77% for G3 and ≈87% for G8). Even for the first set of requirements, the seniors’ UTDD conformance was near the value reported by Müller and Höfer for an UTDD expert group (cf. 77% and 82%). Thus, although they did not have a prior UTDD experience, senior programmers remained faithful to the principles of UTDD even during the development of the initial requirements.

The variability in the ratios of changes observed during the development of the initial sets of requirements, mainly in the case of juniors and intermediates (left panels in Fig. 3), translates into a great variability in the UTDD process conformance for the distinct subjects of a same group (cf. columns G3 and G4 in Table 2). This produces important differences in the individual learning curves regarding the starting conformance and the number of iterations needed to conform to UTDD.

It seems that these differences are independent of the programmer’s experience level, i.e. there was not a clear correlation between how the subjects learned and applied the UTDD rules and our a priori classification of the 25 participants. To study the real effect of developer experience on learning UTDD, we have used the k-means clustering algorithm [34], [35] to find an a posteriori classification of subjects according to their learning curves. The optimal number of clusters6 for a dataset with the 25 learning curves is four. Column “C-Cluster” in Table 2 indicates the result of the k-means classification. Figure 5 shows the characteristic learning curves of each of these clusters:

- Subjects in the cluster labeled as “none” showed a high UTDD conformance from the beginning of the development. These subjects were able to properly use UTDD just after the initial training. They are seniors, intermediates and juniors. Interestingly, some of them achieved 100% UTDD conformance during the implementation of the first set of three requirements (e.g. J6, I1 and S1). Therefore, unlike previous studies suggesting that this technique requires several months of intense use to be learned [22], in our case, there were subjects (5 out of 25, 20%) that just needed a theoretical knowledge and a brief practice (initial training) to effectively apply UTDD (efficiency is analyzed in the next section). Note that, in these cases, UTDD conformance slightly decreased during the development, although we can consider that the development process always conformed to UTDD. Below this behavior is analyzed.

- Subjects in the cluster labeled as “fast” were able to apply UTDD with a good enough process conformance in just one or two iterations.

- Those subjects belonging to the cluster labeled as “slow” started the development with a low or medium UTDD conformance (in the range 30%-63%) and required four or five iterations to conform to UTDD.

- And, finally, the only subject in the cluster labeled as “outlier” is the subject with the worst results and the only one that said that UTDD is difficult to learn (first question of the final questionnaire). In this sense, we can consider him as an outlier.

Note that the final UTDD conformance is similar and greater than 80% for all the clusters except for the “outlier” cluster (Fig. 5). Then, again, quantitative evidence points out that 24 out of 25 subjects easily learned UTDD during the experiment. The only difference between them is that those subjects belonging to the clusters labeled as “none” or “fast”, mostly seniors, followed the UTDD rules almost from the beginning of the experiment (12 out of 25 subjects). The rest of subjects spent a few iterations practicing UTDD until they were

6. The optimal number of clusters is calculated by minimizing the sum-of-squares errors over multiple executions of the algorithm with different coherent values of the parameter k (number of clusters).
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2013.2295827, IEEE Transactions on Software Engineering

able to use it properly (12 out of 25 subjects). The a posteriori classification, together with our personal perception about the subjects, suggests that the learning curves with UTDD depended on a trade-off between experience and discipline (i.e., how businesslike the programmer is). Discipline is a tacit knowledge that may be acquired through experience, but also innate [36]. Those subjects with a high level of discipline, not only the experienced, were the subjects able to follow the UTDD rules from the initial sets of requirements.

Finally, another interesting result is observed during the development of the last sets of requirements, where none of the participants achieved a 100% UTDD conformance, unlike what happened in the initial ones. This produces the conformance decreased in the learning curve of the “none” cluster (Fig. 5). A similar decrease effect was usually observed when a subject achieved a close to 100% UTDD conformance (e.g. see S5), as illustrated in the learning curve of seniors (Fig. 4) and of the “fast” cluster (Fig. 5). After the analysis, we asked the subjects about a possible explanation for this behavior. In the initial stages, they followed all the time the UTDD rules because of our insistence on the importance of properly using UTDD. However, as the experiment advanced and the knowledge improved (and though they were instructed to not do so), they felt that some developments could be more efficiently implemented with a test-last approach. Note that, except in the case of the subject J9, the ratio of other changes in the last iterations ranged between ≈5% and ≈45%. These results are coherent with Müller and Höfer’s results [20] and suggest a trade-off between test-first practices and other development techniques during a typical UTDD development process, i.e., not all the changes are made following the UTDD rules.

4.3 Programming efficiency

Finally, we evaluate the subjects’ efficiency during the learning process in order to investigate if the learning curves generated additional time for the development. Performance is analyzed based on the effective time taken to develop a group of three requirements. The possible performance penalties may be a consequence of self-training and/or of the developer’s learning curve with UTDD. To quantitatively assess the real effort required by the subjects to properly use this development technique, we compare their performance with UTDD and a non-UTDD development strategy. This comparison is uneven in terms of absolute development times, since, obviously, the required time by each subject to implement a set of requirements depended on his/her design, development and/or testing ability. Therefore, we need to define measurements that allow the development efficiency to be compared taking into account this issue.

Here, we do not assume any relationship in terms of efficiency or productivity between UTDD and other software development practices. We use the sets of requirements G1 and G2, implemented using a non-UTDD control strategy, to calculate baselines for normalized comparisons between subjects. The ability of each subject to implement a set of three requirements (an easy one, a moderate one and a difficult one) may be estimated as the average development time for G1 and G2. Formally, for a subject s, this estimation is given by the following equation:

\[
bl_s = \frac{1}{2} \sum_{i=1}^{2} D_{G_i}^s + J_{G_i}^s + JU_{G_i}^s + T_{G_i}^s
\]  

where \( G_i \) identifies the sets of requirements G1 and G2; and \( D_{G_i}, J_{G_i}, JU_{G_i} \) and \( T_{G_i} \) denote, respectively, the time registered for designing, programming/refactoring the Java functional code, programming the JUnit tests and testing the three requirements included in the corresponding set of requirements. We consider \( bl_s \) as a baseline quantifying the development ability of subject s. Thus, for our performance analysis, we normalize the development times of the sets of requirements implemented with UTDD (G3-G8) using the corresponding value of \( bl_s \) (see threats to validity in Section 5). In this vein, for example, when we say that the normalized development time of a given subject is equal to 1.25, it means that the subject’s efficiency during the implementation of the corresponding set of requirements is 25% worse than his/her efficiency when using the control strategy. Table 3 shows the normalized development time for each subject and set of requirements (G3-G8) with and without self-training.

Note that the requirement comprehension time is not included in Eq. 2. Table 4 shows the distribution of the subjects’ answers to the questions in the intermediate questionnaires regarding the difficulty of understanding the requirements. Only in five cases the response differed from 2 (“Easy”) or 1 (“Extremely easy”) and in all these cases the subject gave response 3 (“Neither easy nor difficult”). These answers indicate that the requirement
TABLE 3
Normalized development time to implement the sets of requirements G3-G8. Left value in each column does not include self-training time. Right value includes self-training.

<table>
<thead>
<tr>
<th></th>
<th>G3</th>
<th>G4</th>
<th>G5</th>
<th>G6</th>
<th>G7</th>
<th>G8</th>
<th>P-Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>1.36</td>
<td>1.34</td>
<td>1.12</td>
<td>1.22</td>
<td>1.16</td>
<td>1.18</td>
<td>no</td>
</tr>
<tr>
<td>J2</td>
<td>1.23</td>
<td>1.28</td>
<td>1.25</td>
<td>1.09</td>
<td>1.02</td>
<td>1.09</td>
<td>yes</td>
</tr>
<tr>
<td>J3</td>
<td>1.18</td>
<td>1.20</td>
<td>1.25</td>
<td>1.14</td>
<td>1.05</td>
<td>1.05</td>
<td>yes</td>
</tr>
<tr>
<td>J4</td>
<td>1.35</td>
<td>1.31</td>
<td>1.28</td>
<td>1.31</td>
<td>1.22</td>
<td>1.22</td>
<td>no</td>
</tr>
<tr>
<td>J5</td>
<td>1.27</td>
<td>1.22</td>
<td>1.22</td>
<td>1.19</td>
<td>1.04</td>
<td>1.04</td>
<td>no</td>
</tr>
<tr>
<td>J6</td>
<td>1.24</td>
<td>1.03</td>
<td>1.00</td>
<td>1.04</td>
<td>1.18</td>
<td>1.18</td>
<td>no</td>
</tr>
<tr>
<td>J7</td>
<td>1.28</td>
<td>1.04</td>
<td>1.21</td>
<td>1.27</td>
<td>1.08</td>
<td>1.08</td>
<td>no</td>
</tr>
<tr>
<td>J8</td>
<td>1.19</td>
<td>1.14</td>
<td>1.16</td>
<td>1.12</td>
<td>1.21</td>
<td>1.21</td>
<td>no</td>
</tr>
<tr>
<td>J9</td>
<td>1.32</td>
<td>1.22</td>
<td>1.20</td>
<td>1.21</td>
<td>1.27</td>
<td>1.27</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td>1.26</td>
<td>1.14</td>
<td>1.11</td>
<td>0.95</td>
<td>1.06</td>
<td>1.02</td>
<td>yes</td>
</tr>
<tr>
<td>I2</td>
<td>1.25</td>
<td>1.04</td>
<td>0.98</td>
<td>0.91</td>
<td>0.98</td>
<td>0.98</td>
<td>yes</td>
</tr>
<tr>
<td>I3</td>
<td>1.29</td>
<td>1.04</td>
<td>1.04</td>
<td>0.98</td>
<td>1.11</td>
<td>1.11</td>
<td>yes</td>
</tr>
<tr>
<td>I4</td>
<td>1.17</td>
<td>1.08</td>
<td>1.04</td>
<td>1.09</td>
<td>1.01</td>
<td>1.01</td>
<td>yes</td>
</tr>
<tr>
<td>I5</td>
<td>1.30</td>
<td>1.22</td>
<td>1.24</td>
<td>1.13</td>
<td>1.20</td>
<td>1.20</td>
<td>no</td>
</tr>
<tr>
<td>I6</td>
<td>1.19</td>
<td>1.15</td>
<td>1.15</td>
<td>1.12</td>
<td>1.03</td>
<td>1.03</td>
<td>yes</td>
</tr>
<tr>
<td>I7</td>
<td>1.29</td>
<td>1.14</td>
<td>1.11</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
<td>yes</td>
</tr>
<tr>
<td>I8</td>
<td>1.20</td>
<td>1.21</td>
<td>1.06</td>
<td>1.11</td>
<td>0.95</td>
<td>0.95</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>1.22</td>
<td>0.91</td>
<td>1.05</td>
<td>0.88</td>
<td>0.92</td>
<td>0.92</td>
<td>yes</td>
</tr>
<tr>
<td>S2</td>
<td>1.17</td>
<td>1.01</td>
<td>1.04</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>yes</td>
</tr>
<tr>
<td>S3</td>
<td>1.26</td>
<td>1.15</td>
<td>1.12</td>
<td>1.06</td>
<td>1.06</td>
<td>1.06</td>
<td>yes</td>
</tr>
<tr>
<td>S4</td>
<td>1.12</td>
<td>0.96</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
<td>yes</td>
</tr>
<tr>
<td>S5</td>
<td>1.04</td>
<td>1.03</td>
<td>1.05</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
<td>yes</td>
</tr>
<tr>
<td>S6</td>
<td>1.17</td>
<td>1.14</td>
<td>1.16</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>yes</td>
</tr>
<tr>
<td>S7</td>
<td>1.26</td>
<td>1.12</td>
<td>0.98</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>yes</td>
</tr>
<tr>
<td>S8</td>
<td>1.18</td>
<td>1.10</td>
<td>0.98</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>yes</td>
</tr>
</tbody>
</table>

TABLE 4
Distribution of the responses to the questions regarding the difficulty of understanding the requirements in the intermediate questionnaires.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Easy requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J1</td>
<td>1.18</td>
<td>68</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Moderate requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J1</td>
<td>1.21</td>
<td>19</td>
<td>53</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Difficult requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J1</td>
<td>1.23</td>
<td>12</td>
<td>57</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Easy requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td>1.18</td>
<td>58</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Moderate requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td>1.22</td>
<td>39</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Difficult requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td>1.25</td>
<td>26</td>
<td>38</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Easy requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>1.18</td>
<td>58</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Moderate requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>1.22</td>
<td>52</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Difficult requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>1.23</td>
<td>41</td>
<td>22</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

For the initial groups of requirements, efficiency with UTDD is very low. For example, during the development of G3, performance decreased ≈41% for juniors, ≈42% for intermediates, and ≈29% for seniors. Initially, most of the subjects needed to look for information about UTDD (self-training), which had a negative impact on performance. As we show above, self-training quickly dropped as the development progressed (see Fig. 2), and its penalty in performance was close to 0 in just a few iterations (no subject spent any time in self-training at the end of the experiment). If now we do not consider self-training, during the development of the first set of requirements developed with UTDD (G3), performance decreased ≈30% for juniors, ≈25% for intermediates, and ≈20% for seniors. In some sense, these percentages can be considered as the effort for applying UTDD. Note that now performance is better for intermediates than for juniors. Independently of the experience level, the UTDD effort dropped as the development progressed. The better the knowledge in UTDD, the better performance (cf. Figures 4 and 6). Therefore, the learning curves impacted on efficiency as this latter improved through practice. Initially, senior programmers had a better performance than intermediates. However, as the intermediates’ development process better conformed to UTDD, both groups displayed a similar understanding was easy for the research subjects, so we can consider the learning curves with UTDD independent of the requirement comprehension (see threats to validity in Section 5 for details).

Figure 6 compares the average normalized development time for each group of programmers and set of requirements. Dark bars on top correspond to the effort used in self-training. These measures estimate the efficiency with UTDD of junior, intermediate and senior programmers as compared to the control strategy.
efficiency. Juniors' performance was always the worst.

These results indicate that, unlike process conformance, efficiency mainly correlates with expertise. If now we classify the subjects with the k-means algorithm according to the performance evolution during the experiment, the optimal number of clusters we find is two. Column “P-Cluster” in Table 3 shows this classification:

- Cluster labeled as “yes” contains the subjects that were able to continuously apply UTDD in an efficient manner when the experiment concluded. All the seniors and all the intermediates but one are in this cluster. During the initial iterations, these subjects used up some effort for UTDD, but this dropped quickly in the subsequent iterations until achieving a similar performance to the one observed with the control strategy (values close to 1). Seniors needed to implement three sets of requirements to achieve a good enough performance, while intermediates needed four iterations (see Fig. 6).
- The subjects belonging to the “no” cluster are mainly junior programmers. Here we have to distinguish between two subgroups of subjects. On one hand, if we individually analyze the development times of these subjects, we observe that some of them alternated good and bad performances (e.g., see J6 or J7). In these cases the penalty in performance at the end of the experiment was not related to training on UTDD. Those cases with a higher development time corresponded with the cases in which the number of structural changes in the existing design was also higher. The correction of these bad design decisions was the main factor to increase the development time for juniors. On the other hand, a different situation occurs for subjects J1 and J5. Although they were able to properly apply UTDD, they could not do it efficiently (=12% of subjects if we include the outlier).

Another interesting result pointing out the difference between juniors and seniors is the response to the last question of the final questionnaire (“Q2. How useful do you feel UTDD is?”). Only three subjects (see Table 1) gave response 1 (“Not at all useful”, J9 and S5) or 2 (“Slightly useful”, S1). That means that they were the only subjects with a poor acceptance of UTDD. However, despite this situation, the subjects S1 and S5 achieved a good performance (even better than 1 in some cases), while the subject J9 always had a poor performance. The rest of the subjects considered UTDD moderately or very useful (the mean value in the Likert scale is always greater than 3.4) and no significant differences in performance are found between them according to UTDD acceptance.

Finally, if we try to find a correlation between UTDD conformance and efficiency, we observe that, as the development advanced and the knowledge in UTDD became better, some subjects reached a trade-off between the use of UTDD and efficiency. As we describe above, Fig. 5 points out that when the subjects acquired a good knowledge in UTDD, the corresponding process conformance usually decreased to a nearly stable value that depended on the specific programmer. These small UTDD deviations usually imply a performance improvement (e.g., compare the conformance and the performance evolution of subjects I2 or S3).

In brief, our results show that during the study, not only effectiveness on UTDD was improved as the subject progressed with the experiment (Fig. 5), but also efficiency was improved through practice on UTDD (Fig. 6). However, for junior programmers, although the UTDD concepts were clear, the efficiency was lower due to what looks like a poor ability to design.

4.4 Retention ability

When we talk about the adoption of a new software development technique, an important feature to take into account is the retention ability, not only the learning process. As far as we know, some of the intermediate and senior programmers that participated in our study have included UTDD (and also ATDD) in the arsenal of development techniques of their teams and, nowadays, they use them in projects within the industrial environment. Nevertheless, we cannot have conducted a controlled experiment to study in detail the retention and later applying of UTDD within the professional environment. In this sense, we can only provide a simple estimation of retention in the case of three of the original research subjects (I2, I5 and S5). Six months after finishing the experiment explained so far, we conducted a very simple empirical study with these three subjects trying to evaluate their ability to continue using UTDD. None of them had used UTDD again after our initial experiment. In the new quasi-experiment, they had to develop by using UTDD two new groups of requirements (G1' and G2'), each one comprising an easy requirement, a moderate one and a difficult one. Figure 7 shows
Fig. 7. Estimation of the retention ability of subjects I2, I5 and S5. G3-G8 are the groups of requirements included in initial experiment. G1' and G2' are two new groups of requirements with three requirements per group to be developed six months after finishing the experiment.

their UTDD conformance (top panel) and performance (bottom panel) during the initial development (G3-G8) and the later one (G1'-G2'). The three subjects reached a similar UTDD conformance to the one reached during the development of the groups of requirements G7 and G8. From the performance perspective, they showed a little decline, although in all cases performance was near the non-UTDD control value. These results point out that after six months the three subjects were able to continue using UTDD effectively and efficiently.

5 Threats to Validity

Like most empirical studies, the validity of our results is subject to several threats. In order to limit the scope of our claims as well as to explain their utility, the first important fact to take into account is that this is an empirical study that presents results based on a small number of developers. In some sense, they knew that we were measuring their development capabilities, leading to a possible Hawthorne effect. Furthermore, the lack of time pressure during the development could help the participants to focus on using UTDD in the right way instead of on performance, which facilitates learning and increases UTDD conformance.

A crucial internal validity threat is related to how we calculate the baseline for the time required by each research subject to implement a set of three requirements (Eq. 2). Note that we use these baselines to quantify the designing, programming and testing abilities of the subjects, not to compare UTDD and non-UTDD performances. UTDD forces developers to write unit tests, because the tests are an essential part of the development. With more traditional approaches unit testing is sometimes less detailed or even disregarded [37], [13], [14], [38]. To allow the performance comparison in an equivalent scenario, we instructed the subjects to implement JUnit tests with the control strategy. Intermediates and seniors created approximately the same mean number of unit tests with the UTDD and the non-UTDD approach. In the case of juniors, the number of unit tests slightly increased with UTDD. In order to analyze whether the method depended on the sets of requirements used in the estimation, we tested that the level of complexity and the development time of all the sets of three requirements were similar. Before the study, three professional programmers (one for each of the developer categories considered in the study) developed the eight sets of three requirements validating that the performance in all cases was equivalent between sets and programmers. After that, we chose the order of the requirements and the two groups to calculate the baselines.

In our analysis we have omitted the requirement comprehension time because of the subjects’ responses to the intermediate questionnaires. If we include this time within the effective development time, small differences appear in the performance results. However, this is an expected effect since different requirements of the same category require different time to be read and understood.

Finally, concerning the generalization of the findings, as with all empirical studies, our analysis should be repeated in different environments and in different contexts before generalizing the results. In particular, all the participants in our study are skilled programmers. They have experience in unit testing and know the development language and the automated testing tool. Our results are only applicable to similar scenarios. The lack of experience/knowledge will produce different learning curves with UTDD. Threats to external validity are also related to the application we used as study material. Although it is a real world application, requirements were adapted to the time available for the study and, therefore, all the development tasks were simple. Shallower learning curves (mainly from the efficiency point of view) are possible when considering more complex tasks. In this vein, for example, our results are the same considering the developers’ general software development experience, the Java and JUnit knowledge or the design experience. However, in more complex real world developments, the testing or design experience, or a specific programming language knowledge, may be key factors with significant impact on UTDD adoption.
6 CONCLUSIONS

The principal conclusion drawn from our study is that skilled developers with the appropriate knowledge and without any prior experience in test-first development can quickly learn the UTDD rules and, after practicing them for a short while, properly apply them in small programming tasks. Interestingly, there even existed some subjects whose development process conformed to UTDD from the beginning of the experiment. In other words, they did not need a training or learning period to apply UTDD in the right way. Furthermore, it also seems that the research subjects were able to retain the UTDD knowledge and even use it in their companies within the industrial environment.

All the participants in our study except one (96% of the research subjects) believed that UTDD was easy to learn. The quantitative results corroborate this belief. The research subjects wrote a correct and complete functional code and, at the end of the experiment, only one of them (outlier) was not able to properly apply this software development practice. If we do not take into account the outlier, the participants followed three possible learning processes (Fig. 5), although all of them adapted their mindset to UTDD in just a few iterations.

Analyzing the learning curves by experience level (Fig. 4), the development process of seniors looks like an UTDD expert process from the beginning of the experiment; while the processes of juniors and intermediates initially look like an UTDD novice process. Knowledge in UTDD became better as the participants advanced with the development. Not only self-training dropped (Fig. 2), but also UTDD conformance grew until reaching a steady level (Fig. 4). When the study concluded, the three groups of programmers behaved similarly to an UTDD expert team. If we consider 0% conformance the starting point of the learning process, the learning curves are steeper for intermediate programmers than for juniors (although it is important to keep in mind that in any case they are always steep), but juniors needed more self-training along the learning process. Nevertheless, the learning curves with UTDD do not seem to only depend on the programmer’s experience level. Some juniors and intermediates were also able to be faithful with UTDD with just a little practice. This fact points out the dependence of the learning process on how businesslike the programmer can quickly learn the UTDD rules and, after practicing them for a short while, properly apply them in small programming tasks. Interestingly, there even existed some subjects whose development process conformed to UTDD from the beginning of the experiment. In other words, they did not need a training or learning period to apply UTDD in the right way. Furthermore, it also seems that the research subjects were able to retain the UTDD knowledge and even use it in their companies within the industrial environment.

The general perception about UTDD is that it is difficult to learn and apply [13], [14], [15], [26]. It is commonly thought that although it seems easy at first glance, it requires a high level of discipline to be carried out effectively and efficiently [22], [23]. In this sense, previous studies show that, at a first exposure to this software development practice, there often exists an initial resistance to use it due to inexperience and a potential growth in the amount of work [9], [30]. Then, it is usually highlighted that support is required for UTDD inexperienced developers at least in the early stages of the learning process [31]. In a context like the industrial environment, where software professionals often work with time pressure and financial constraints, all these factors may make UTDD novices revert to well-known traditional techniques. In contrast to the general perception, recent empirical results indicate that, in an UTDD novice development team, problems related to lack of experience can be limited with an UTDD mentor or following a pair programming approach [14], [39], [21], [19]. Our results are aligned with these last findings. The difference with previous studies likely lies in UTDD (i.e., he/she achieved a close to 100% UTDD conformance). These subjects usually started combining the use of UTDD with other development practices because they believed that some developments could be more efficiently completed with other approaches. Therefore, they showed small UTDD deviations. When this change in the behavior happened, performance improved (cf. Table 2 and Table 3). So, the developers’ observation seems to be truth. A possible explanation is that UTDD may not be a suitable or effective solution for all kind of requirements [6]. These results suggest that experienced programmers are able to dynamically adapt their development process in order to reach an optimal balance between conformance against performance and, therefore, a strict conformance to the UTDD rules may not be observable in practice within expert groups.

The principal conclusion drawn from our study is that only two subjects achieved a UTDD conformance greater than 90% during the development of the final requirements, unlike what happened during the initial requirements, where even some of the participants showed 100% process conformance (Table 2). This also happens in UTDD expert groups [18]. In the same manner, a similar effect can be observed when a subject reached a high knowledge level in UTDD (i.e., he/she achieved a close to 100% UTDD conformance). These subjects usually started combining the use of UTDD with other development practices because they believed that some developments could be more efficiently completed with other approaches. Therefore, they showed small UTDD deviations. When this change in the behavior happened, performance improved (cf. Table 2 and Table 3). So, the developers’ observation seems to be truth. A possible explanation is that UTDD may not be a suitable or effective solution for all kind of requirements [6]. These results suggest that experienced programmers are able to dynamically adapt their development process in order to reach an optimal balance between conformance against performance and, therefore, a strict conformance to the UTDD rules may not be observable in practice within expert groups.

Another interesting result of our study is that only two subjects achieved a UTDD conformance greater than 90% during the development of the final requirements, unlike what happened during the initial requirements, where even some of the participants showed 100% process conformance (Table 2). This also happens in UTDD expert groups [18]. In the same manner, a similar effect can be observed when a subject reached a high knowledge level in UTDD (i.e., he/she achieved a close to 100% UTDD conformance). These subjects usually started combining the use of UTDD with other development practices because they believed that some developments could be more efficiently completed with other approaches. Therefore, they showed small UTDD deviations. When this change in the behavior happened, performance improved (cf. Table 2 and Table 3). So, the developers’ observation seems to be truth. A possible explanation is that UTDD may not be a suitable or effective solution for all kind of requirements [6]. These results suggest that experienced programmers are able to dynamically adapt their development process in order to reach an optimal balance between conformance against performance and, therefore, a strict conformance to the UTDD rules may not be observable in practice within expert groups.

The general perception about UTDD is that it is difficult to learn and apply [13], [14], [15], [26]. It is commonly thought that although it seems easy at first glance, it requires a high level of discipline to be carried out effectively and efficiently [22], [23]. In this sense, previous studies show that, at a first exposure to this software development practice, there often exists an initial resistance to use it due to inexperience and a potential growth in the amount of work [9], [30]. Then, it is usually highlighted that support is required for UTDD inexperienced developers at least in the early stages of the learning process [31]. In a context like the industrial environment, where software professionals often work with time pressure and financial constraints, all these factors may make UTDD novices revert to well-known traditional techniques. In contrast to the general perception, recent empirical results indicate that, in an UTDD novice development team, problems related to lack of experience can be limited with an UTDD mentor or following a pair programming approach [14], [39], [21], [19]. Our results are aligned with these last findings. The difference with previous studies likely lies in UTDD (i.e., he/she achieved a close to 100% UTDD conformance). These subjects usually started combining the use of UTDD with other development practices because they believed that some developments could be more efficiently completed with other approaches. Therefore, they showed small UTDD deviations. When this change in the behavior happened, performance improved (cf. Table 2 and Table 3). So, the developers’ observation seems to be truth. A possible explanation is that UTDD may not be a suitable or effective solution for all kind of requirements [6]. These results suggest that experienced programmers are able to dynamically adapt their development process in order to reach an optimal balance between conformance against performance and, therefore, a strict conformance to the UTDD rules may not be observable in practice within expert groups.

The general perception about UTDD is that it is difficult to learn and apply [13], [14], [15], [26]. It is commonly thought that although it seems easy at first glance, it requires a high level of discipline to be carried out effectively and efficiently [22], [23]. In this sense, previous studies show that, at a first exposure to this software development practice, there often exists an initial resistance to use it due to inexperience and a potential growth in the amount of work [9], [30]. Then, it is usually highlighted that support is required for UTDD inexperienced developers at least in the early stages of the learning process [31]. In a context like the industrial environment, where software professionals often work with time pressure and financial constraints, all these factors may make UTDD novices revert to well-known traditional techniques. In contrast to the general perception, recent empirical results indicate that, in an UTDD novice development team, problems related to lack of experience can be limited with an UTDD mentor or following a pair programming approach [14], [39], [21], [19]. Our results are aligned with these last findings. The difference with previous studies likely lies in UTDD (i.e., he/she achieved a close to 100% UTDD conformance). These subjects usually started combining the use of UTDD with other development practices because they believed that some developments could be more efficiently completed with other approaches. Therefore, they showed small UTDD deviations. When this change in the behavior happened, performance improved (cf. Table 2 and Table 3). So, the developers’ observation seems to be truth. A possible explanation is that UTDD may not be a suitable or effective solution for all kind of requirements [6]. These results suggest that experienced programmers are able to dynamically adapt their development process in order to reach an optimal balance between conformance against performance and, therefore, a strict conformance to the UTDD rules may not be observable in practice within expert groups.
in the subjects’ experience. While in most of the previous UTDD studies the subjects are students, in our case they are industrial developers with testing skill and several years of experience. This provides them the ability to write efficient and effective unit test cases [40], [41], which for us is one of the key factors for the UTDD success.

From the industrial point of view, our results point out that UTDD can be taken into account as an effective software development practice even if the development team has no experience in such technique. Intermediate and senior programmers can quickly take advantage of the benefits of using UTDD in those cases where the adoption of this practice benefits the project (see [42] for a detailed analysis).

Finally, our results suggest that junior programmers do not have good design abilities in many occasions. At this regard, an interesting question already raised by Janzen and Saiedian [7] is the possibility of incorporating UTDD into the undergraduate curriculum to improve the students ability to design. In this line, promising results have been obtained recently in introductory programming courses with WebIDE [43], [44].

**APPENDIX**

The application to be implemented during the study was a simple project planning tool. The requirements used as study material were classified in three different categories as a function of their complexity level. In this Appendix we present an example of requirement belonging to each category: easy requirements, moderate requirements and difficult requirements.

Before providing these examples, we need to introduce some concepts. A project has different properties (not important here) and contains zero or more tasks. A task belongs to one and only one project. It consumes time and requires resources (persons or materials) to be carried out. A task has the following attributes: Name, Description, Minimum Start Date (the start date if the task has not dependencies) and Time (the time required to accomplish the task). Each association between a resource and a task has an availability attribute indicating the percentage of the resource availability allocated for the associated task. Each resource can have different costs and percentages of availability and, in the case of resources of type “person”, different percentages of over-allocation.

- **Easy requirement:** Create a new task.
- **Moderate requirement:** Calculate the project cost.
- **Difficult requirement:** Evaluate whether a resource is over-allocated.

The authorized users must be able to calculate the cost of a given project. The cost of a project is defined as the sum of the cost of all its tasks. The cost of each task is calculated as the sum of the cost of the resources associated with the task and all its subtasks.

A resource is over-allocated in a given time period if the sum of the availability attributes of all the corresponding resource-task associations are greater that the resource’s availability in that period. In the case of resources of type “person”, this sum cannot be higher than the corresponding percentage of over-allocation.

**ACKNOWLEDGMENT**

The author would like to thank Mónica Cabanillas for her comments and suggestions that helped to improve the final version of the manuscript.

**REFERENCES**


