Empir Software Eng
DOI 10.1007/s10664-013-9281-9

EXPERIENCE REPORT

A successful application of a Test-Driven Development strategy in the industrial environment

Roberto Latorre

Abstract Unit Test-Driven Development (UTDD) and Acceptance Test-Driven Development (ATDD) are software development techniques to incrementally develop software where the test cases, unit or acceptance tests respectively, are specified before the functional code. There are little empirical evidences supporting or refuting the utility of these techniques in an industrial context. Just a few case studies can be found in literature within the industrial environment and they show conflicting results (positive, negative and neutral). In this report, we present a successful application of UTDD in combination with ATDD in a commercial project. By successful we mean that the project goals are reached without an extra economic cost. All the UTDD and ATDD implementations are based on the same basic concepts, but they may differ in specific adaptations to each project or team. In the implementation presented here, the business requirements are specified by means of executable acceptance tests, which then are the input of a development process where the functional code is written in response to specific unit tests. Our goal is to share our successful experience in a specific project from an empirical point of view. We highlight the advantages and disadvantages of adopting UTDD and ATDD and identify some conditions that facilitate success. The main conclusions we draw from this project are that ATDD contributes to clearly capture and validate the business requirements, but it requires an extensive cooperation from the customer; and that UTDD has not a significant impact neither on productivity nor on software quality. These results cannot be generalized, but they point out that under some circumstances a test-driven development strategy can be a possible option to take into account by software professionals.

Communicated by: Brian Robinson

R. Latorre (✉)
Dpto. Ingenier'ia Inform'atica, Escuela Polit'ecnica Superior,
Universidad Aut'onoma de Madrid, Madrid 28049, Spain
e-mail: roberto.latorre@uam.es

R. Latorre
NeuroLogic Soluciones Informáticas, 28037 Madrid, Spain

Published online: 27 September 2013

Keywords Test-first design · Acceptance Test-Driven Development (ATDD) · Unit Test-Driven Development (UTDD) · Software engineering process · Software quality/SQA · Software construction · Experience report

1 Introduction

Software professionals in the industrial environment, from IT architects to programmers, often work with time pressure and financial constraints. They cannot elucidate the advantages and disadvantages of changing and/or adapting traditional development processes and usually deploy new strategies. In this scenario, a common mistake is to use a new technique or technology based only in novelty factors or comments and opinions in websites or research papers. To develop projects based on opinions and unfounded sources may eventually lead to failure.

This experience report presents from an empirical point of view a successful case of using a combination of two of these novel software development practices, known as Unit Test-Driven Development (UTDD) (Beck 2003; Astels 2003) and Acceptance Test-Driven Development (ATDD) (Beck 2003; Koskela 2007; Hendricksons 2008), within the industrial environment. They are two different development techniques to iteratively develop software following a test-driven development strategy, i.e. a strategy where the test cases are specified before the functional code. While UTDD focuses on unit tests to ensure the system is performing correctly, ATDD is characterized by acceptance tests "written by the customer", usually with help of the development team, to drive the software development process. Only when the test cases are specified, programmers start writing the functional code to satisfy these tests.

Over the past decade, researchers have conducted several studies within academia and industry on the effectiveness of these software development practices (Müller and Hagner 2002; Andersson et al. 2003; Erdogmus et al. 2005; Bhat and Nagappan 2006; Janzen and Saiedian 2006; Park and Maurer 2009). However, little empirical evidence supports or refutes the utility of test-driven development methodologies in an industrial context and the results are sometimes inconclusive (Jeffries and Melnik 2007; Rafique and Misic 2013). Here, we share our successful experience in a commercial project within an industrial environment. The project had been unsuccessfully started before and then successfully restarted using UTDD and ATDD. The report introduces the project background and describes how we morph ATDD and UTDD to our needs. The development process lasted five and a half months with a development team of 11 people. Most of the team members had no prior experience in test-driven development methodologies. Nevertheless, they reached the goal of finishing the development on schedule without an extra economic cost. We analyze and discuss the effect of applying ATDD and UTDD on requirement engineering, software quality and productivity. Our goal is not to compare results or quality metrics in different development approaches to reach a general result, but to show a specific actual development process where these practices were successfully applied.

One of the problems found in literature about UTDD and ATDD is the difficulty in isolating their effects from other context variables. In our case it happens the same. The adoption of UTDD and ATDD contributed to the project success, although

this success could not only be attributed to them, but also to the combination with other factors such as a continuous integration and verification or the capability of the development team between others. Our results point out that these development techniques can be successfully applied in the industrial environment without an increment in the costs and/or effort. Moreover, in our case, the development team did not require neither a previous expertise on test-first development methodologies nor a high level knowledge about the software tools selected to support the test-driven development implementation. Although each development project is different from the others, with the results reported here, we expect to have software professionals to assess the use of UTDD and ATDD, by themselves or in combination with other techniques, as a possible and valid option to take into account in their developments.

The paper is organized as follows. Section 2 gives an overview of UTDD and ATDD and briefly summarizes related work in industrial contexts. Section 3 presents our project background and explains why we adopt UTDD and ATDD. Section 4 describes how we morph these techniques to our project. Section 5 presents the general project results and the lessons learned about UTDD and ATDD. Section 6 the threats to validity. And, finally, Section 7 draws the conclusions.

2 Test-Driven Software Development Processes

In test-driven software development processes the test cases are specified incrementally before the production code. That means that testing is taken into account from the initial development phases and it drives "the full" development process. Here we need to distinguish between *Unit Test-Driven Development* (UTDD) and *Acceptance Test-Driven Development* (ATDD).

2.1 Unit Test-Driven Development

UTDD is also known simply as Test-Driven Development (TDD). It is based on a minute-by-minute iteration between writing unit tests and writing functional code (Beck 2003; Astels 2003; Erdogmus et al. 2010). It involves working in very small steps, one small unit test at a time. Figure 1 shows a schematic representation of the main steps of the UTDD workflow. Before writing functional code, the developers write automated unit test cases for the new functionality they are about to implement. Just after that, they write the functional code to pass these test cases. In other words, every line of new functional code is written in response to a unit test. New functionalities are not considered as proper implementations unless the new unit test cases and all the prior tests are passed. This technique was used for a long time (Larman and Basili 2003), but only in the last years it has been adopted as a key strategy in agile software development (Martin 2003, 2008).

Despite its name, the first thing to take into account if you are new to UTDD is to know that this practice is not a testing technique, but a programming and design practice (Beck 2001; Cockburn 2002; Damm and Lundberg 2006; Janzen and Saiedian 2005, 2008; Crispin 2006). New small design decisions are continuously made in each work cycle (Fig. 1). Note that a high-level design/architecture is performed by the teams prior to coding. Often, this is done during the spiking or exploration phase, which often involves building a throw-away prototype. This high-

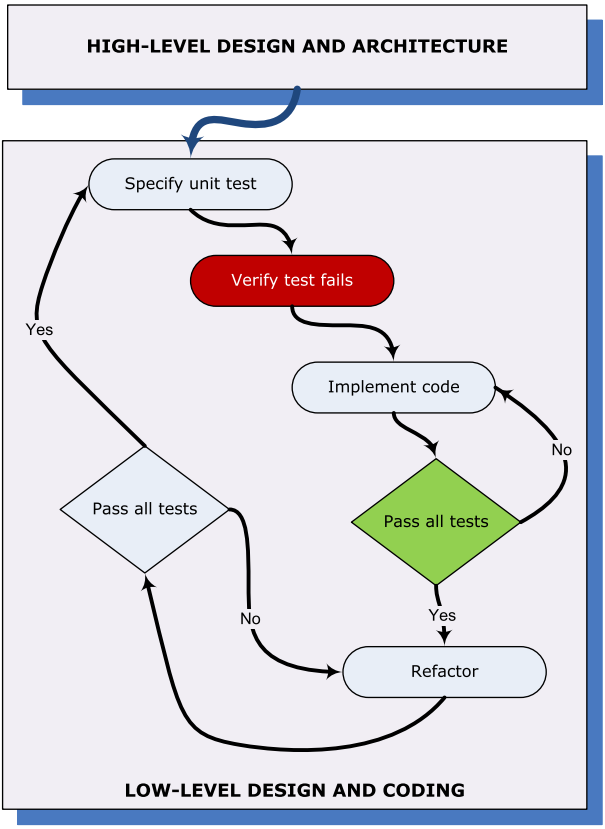

Fig. 1 Unit Test-Driven Development workflow

level design does not proceed to a detailed level. Nevertheless, the following iterative and incremental work cycle allows the low-level design to emerge and evolve during coding:

1. Write the unit test (no functional code yet).
2. Run the newly specified test without new functional code to make sure it fails showing that the capability is not already present. Generally, the code may not even compile because the elements used in the code are not there yet (e.g. in our case using a class that has not been implemented yet).
3. Implement just enough code to pass the test.
4. Run the test and every other unit test cases written for the code. If one test fails, return to step 3.
5. Refactor the code if necessary. Refactoring includes modifications in the code to reduce its complexity and/or improve its understandability and maintainability.
6. Run again all the tests to ensure they pass. If one test fails, return to step 5 to correct the errors.
7. Go to the next functionality and repeat from step 1. Here, the important part is to consider small steps in each increased functionality.

Table 1 Summary of selected previous studies about UTDD in the industrial environment

Study	Type	Quality effect	Productivity effect
Maximilien and Williams (2003)	Case study	UTDD is better	Equal
George and Williams (2003)	Controlled experiment	UTDD is better	UTDD is worse
Bhat and Nagappan (2006)	Case study	UTDD is better	UTDD is worse
Sanchez et al. (2007)	Case study	UTDD is better	UTDD is worse
Nagappan et al. (2008)	Case study	UTDD is better	UTDD is worse

In the last years several studies and experiments have analyzed the effectiveness of UTDD in different terms. Most of them belong to an academic context (Müller and Hagner 2002; Kaufmann and Janzen 2003; Madeyski 2005; Flohr and Schneider 2006; Janzen and Saiedian 2006; Gupta and Jalote 2007). In the research literature UTDD appears as one of the most efficient development techniques to write clean and flexible code on time. Nevertheless, from our experience, the use of UTDD in commercial projects is usually neglected. Just a few case studies and reports are conducted by professionals in an industrial environment (Maximilien and Williams 2003; George and Williams 2003; Bhat and Nagappan 2006; Nagappan et al. 2008). And a detailed review of the literature reveals that results are perhaps controversial (positive, negative and neutral) and depend on several factors. Table 1 shows the main results of a survey of studies belonging to the industrial environment (an interesting and very exhaustive review of UTDD in the academic and industrial environment can be found in Rafique and Misic 2013). As these studies report conflicting results, no definite conclusion can be drawn. The most common assessments in these studies are:

- UTDD increases the quality of software.
- UTDD decreases the productivity.

One of the reasons for the productivity decrease is because of writing the automated tests. However, some studies point out that if the software quality is significantly better, the overall team productivity would be better because the number of defects to fix is reduced.

2.2 Acceptance Test-Driven Development

Like UTDD, ATDD also involves creating tests prior functional code, but this is a technique focused on the business level where the tests represent testable requirements and specifications (Koskela 2007; Hendricksons 2008; Pugh 2010). Thus, it is usually considered as a requirement engineering practice (Cao and Ramesh 2008; Haugset and Stalhane 2012). In ATDD, the software development process is driven by acceptance tests that specify the desired behavior of the system to develop. With this technique, test-driven development is done at a higher level focusing on the end-to-end scenarios. Following the test-driven development philosophy, the ATDD workflow is similar the one shown in Fig. 1 for UTDD. One or more acceptance tests are written for each feature before beginning work on it. Typically, these tests are discussed and captured collaboratively together with the customer. The development team works on implementation with guidance of the acceptance tests, and the implementation is not completed until all the corresponding tests are successfully passed. In this way, the customer specifies the features to be implemented in terms

of externally verifiable behavior ("what"); while the development team defines the specific implementation details ("how").

The same as UTDD, during the last years ATDD has gained a lot of attention as a development method within the agile community (Leffingwell 2010; Pugh 2010; Gärtner 2012). In this scenario, several studies in the academic and industrial environment analyze the advantages and disadvantages of using ATDD (Newkirk and Vorontsov 2004; Jeffries and Melnik 2007; Koskela 2007; Park and Maurer 2009; Ricca et al. 2009). Results in the literature support the utility of ATDD when it is properly implemented. The most common reported benefits are:

- ATDD improves communications between customers and developers.
- ATDD can improve productivity and decrease defects.
- ATDD allows to automatically test software at the business level.

3 Project Background

The product owner is a restaurant chain company. Each restaurant worked with a set of independent text mode applications supporting all its activity (taking and managing orders, cashing, reporting, etc). The migration of these applications to a web-based environment was a key point in the business strategy of the company. Therefore, they began a one year project to migrate and integrate all these tools into an unique web-based application. One year and eight months after the beginning of this project, the development had not finished yet and the product owner had the general feeling that "nothing works".

At that moment, the company's steering committee imposed a time limit of six months for ending the development and beginning the validation tests. After this ultimatum, the customer IT department decided to fire the software company in charge of developing the new web-based application and hire our company as the new "rescue" team to fix the software defects and complete the development.

3.1 An Uncompleted Specification Led to a Test-Driven Development Strategy

When we started working on the project, time constraints were our main challenge. The new team had six months to:

- understand the project requirements,
- review the system architecture and design (and maybe modify them),
- fix the software defects,
- and successfully complete the development.

Together with the product owner, we defined a roadmap to reach this deadline. In our initial plan, we did not consider the adoption of a test-driven development methodology. Following the initial planning, the first thing to do was to review the existing documentation and software to understand functional and non-functional requirements and validate the system architecture and design. Surprisingly, we found an uncompleted functional documentation. Furthermore, around 85 % of the identified functionalities did not work as the product owner expected. So, the new development team had not only to fix the software defects and complete the

development, but, in most cases, they had to re-implement the application under the following:

- Time constraints
- Business unknowledgement
- Lack of specifications

As we mention above, some authors claim that one of the main benefits of using a test-driven development technique is aiding software and business people in order to establish unambiguous requirements, helping to deliver exactly what the customer wants (Koskela 2007; Jeffries and Melnik 2007; Park and Maurer 2009; Haugset and Stalhane 2012). So, with the lack of specifications and the time limit in mind, we evaluated the possibility of using a test-driven approach combining UTDD and ATDD, rather than a more traditional waterfall-like practice. The team experience with these techniques was very poor and published results suggested that both UTDD (see Table 1) and ATDD (Haugset and Stalhane 2012) were not easy to learn and use, pointing out that the costs of using them were higher than more conventional practices. Due to the time constraints, this was a great risk for our team. However, to clearly identify and design all the functionalities as soon as possible were critical. Therefore, we decided to assume the risks of using UTDD and ATDD (lack of experience, potential decreased productivity and increased cost, between others) because some of their potential benefits could be an optimal solution to our needs. The main benefits we expected to achieve of adopting these software development practices were:

- A clear understanding and capturing of the business requirements (Koskela 2007; Jeffries and Melnik 2007; Park and Maurer 2009).
- Decreased time for the initial design of the system. Traditional approaches require time for an initial detailed design, while with UTDD this time is lower because the initial design is less formal. The detailed design arises and evolves during the work cycle (Janzen and Saiedian 2008).
- Detection of bugs early in the software development (Williams et al. 2003; Martin 2007; Ficco et al. 2011).

3.2 High-Level Design

The software we found when we started working on the project (hereafter we call this software *original version*) was a *Java* web-based application with the business logic divided between front-end and back-end. It implemented a MVC (Model-View-Controller) architecture (Buschmann et al. 1996) using *Struts*[1] and *Spring*[2] frameworks. We can consider that it was a medium sized application (see Table 3 for details).

The presentation tier was provided by *Java Server Pages* and *Servlets*. The main and critical application requirement was a very strong (and fast) interaction with the user in the front-end (e.g. full control of the application with the keyboard, lots of AJAX requests, etc). In the original version, all the front-end functionalities

[1] http://struts.apache.org

[2] http://www.springsource.org

were implemented with *Prototype*[3] and *jQuery*.[4] The front-end was one of the main sources of error in this version. During the high-level design and architecture revision we decided to fully re-implement the front-end using *Prototype* and a set of *Javascript* tools developed by ourselves.

The back-end consisted of a group a *Java* classes executed in the server. They contained the algorithmic and access a *PostgreSQL* database[5] by using *iBATIS*.[6] This separated the business logic from the data access layer. During the high-level design we decided to use the same architecture in our development.

3.3 The Development Team

When we first started working on the project, we planned to use a waterfall-like methodology. We estimated the need of a development team consisting of 12 people. However, after the adoption of ATDD and UTDD, the final team consisted of 11 members:

- One IT architect that took the role of project manager and technical coordinator. He had some experience in small academic projects with UTDD.
- One senior analyst/designer with a high technical skill and a wide experience in web-based *Java* projects. Initially, when we considered the use of a waterfall-like methodology, we estimated the need of two analysts. But when adopting ATDD and UTDD, the IT architect dedication was increased to play the role of another analyst.
- Four back-end programmers: two seniors with more-than-eight-year and more-than-six-year experience in Java projects; and two juniors with around-two-year experience. They were organized in two groups of a junior and a senior programmer.
- Three web programmers: two seniors (both with more-than-ten-year experience); and a part-time student.
- And, finally, two testers. They were in charge, among others, of the test automation.

In addition to our group, the product owner dedicated a full time person to the project to collaborate in the specification and validation tasks. This customer representative had a technical skill and also knew the business. As we show later, the presence of this person was a key factor for the project success.

When we finally decided to adopt ATDD and UTDD, our main challenge was the lack of previous experience in test-first developments. Reviewing the literature we found some reports claiming that (i) UTDD is difficult to learn (George and Williams 2003; Crispin 2006) and (ii) ATDD is difficult to use and demands a high level of discipline from the developers (Haugset and Stalhane 2012). In our team only the IT architect had some experience with UTDD in small and controlled experiments

[3] http://www.prototypejs.org

[4] http://jquery.com

[5] http://www.postgresql.org

[6] http://www.mybatis.org/

within an academic context. However, we thought that the team members could learn and apply both techniques during the development.

The team members had several-year experience using different traditional methodologies based on test-last development, so the most important thing was to change the programmers' mindset. For that, at the beginning of the project, they received a brief practical course of one week explaining the main test-driven development topics. The most difficult part was to understand the need of writing unit test ahead of time in UTDD and the importance of the fast incremental cycles between writing failing unit tests and writing functional code to pass these tests. Once these concepts were clear, the application of our working methodology (see next section) was not difficult.

Other minor problem at this point was that some of the team members had no experience in the use of the tools that we adopted to implement the automated tests (see bellow). This was easily solved with examples of use and a fluent communication with other team members.

4 Working Methodology

UTDD practice differs from a more traditional test-last practice only in when the unit tests are written. It hinges on identifying and writing single unit tests prior to writing just enough functional code to run that test (see Fig. 1). Following the ATDD basis, in our working methodology, we extend this behavior by requiring that at least one acceptance test for a given feature needs to be developed before the development team starts implementing that feature. There exist many different successful ATDD implementations (Adzic 2011). All of them are based on the principles described in Section 2.2, but they are adapted to specific needs in each development team or project. Figure 2 illustrates how we morph ATDD (and UTDD) to our needs. We define this workflow keeping in mind that UTDD follows ATDD in each work cycle in the sense that acceptance tests define the context in which to derive unit tests.

After the initial high-level design (see Section 3.2 for details), as first step of each work cycle, the analysts meet with the product representative to discuss about the business requirements. Following ATDD criteria, each requirement is specified in the form of acceptance tests (high-level functional tests) to represent expectations of behavior the software should have. The use of automated tests improves the quality of the test effort (Dustin et al. 1999). Therefore, these high-level functional tests must be captured and developed in a format supported by a functional test automation framework. But, at the same time, they must be written in a language of the domain so that they become understandable for end users and business people. The team members had no experience with any functional test automation framework. At the beginning of the project, we evaluated the use of three tools for this task: *FitNesse*,[7] *Concordion*[8] and *Cucumber*.[9] We selected *Concordion* because it was the most intuitive for the team members and it did not require too much prior knowledge.

[7] http://fitnesse.org/

[8] http://www.concordion.org

[9] http://cukes.info/

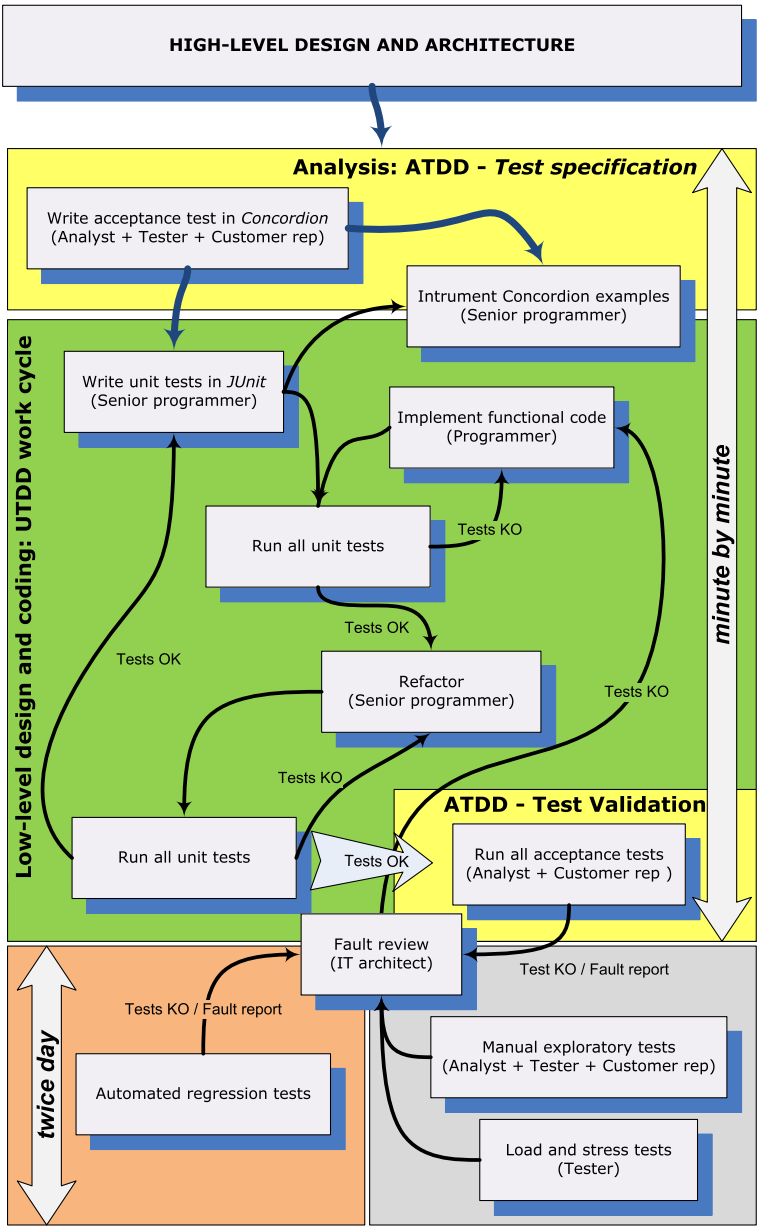

Fig. 2 Workflow used in our project. See the text for details. The label "programmer" in a task means that both junior and senior programmer can perform this task

With *Concordion*, functional test cases are written "in prose" in simple HTML and then linked directly to *JUnit* test cases written by the programmers during the low-level design and coding phase (see below). The use of HTML for specifying the functional tests makes easy to write and read what a feature has to do. The analysts,

the testers and the customer representative are responsible of writing the high-level functional tests and, in the back-end case, the examples[10] (included in the HTML file) that describe each specific test. These tests must be validated by the product owner before beginning work on it. Later, the examples in the *Concordion* functional tests are treated in a similar manner as unit tests during the low-level design. Once an acceptance test is validated, the IT architect assigns it to a group of programmers. As all the groups are equivalent, the assignment follows a workload criterium. Here, the low-level design and coding phase starts. The input for the coding is the *Concordion* functional tests, not a specification or design document.

During low-level design and coding phase (Fig. 2), each group departs from the minute-by-minute iterative cycle of writing unit tests and functional code. The senior programmers take care of the quality and design of their group's code and, therefore, they write the unit tests associated to each acceptance test, including the *Concordion* examples defined during the analysis. In this last case, they instrument the concrete examples in each HTML file with *Concordion* commands (similar to *JUnit* commands, but embedded in the HTML file) that allow the examples to be checked. During the development, new low-level unit tests can be identified. In these cases, senior programmers also specify these tests and update the pool.

Once a *Concordion* acceptance test and the corresponding *JUnit* unit tests are written, each unit test follows a standard UTDD work cycle (Fig. 1). Programmers write the functional code in the same manner they usually do until the unit tests are successfully passed. During the execution of unit tests we use a similar (but simpler) technique for bug detection to the technique presented in Ficco et al. (2011). In order to guarantee each unit test is successfully passed, each time programmers commit their changes in the version control system, an automatic validation of the tests associated with each code fragment is done. If a test is not successfully passed, the commit is aborted. Therefore, programmers have instant feedback about the effect of a design decision or a new code.

Due to the lack of a fully defined functional requirements, when programmers implement a new functionality, they have not a complete view of the system. They do not clearly know how the new code affect the previous one. Hence, to detect software faults as soon as possible in the development process, automated regression tests play a critical role. We have implemented a mechanism that automatically extracts and builds the code twice a day from the version control system, and runs regression tests to validate that the new code does not cause failures in the previous one. After each automated build/test run, an automated report with a list of build failures is sent to the entire team. This is done to ensure a fast bug fixing. Some of the faults discovered during these tests can lead to a change in the general design. In these cases, decisions of changes are made together between the IT architect and the senior programmers. This automated build and test serves as continuous integration and verification.

During the development, programmers only have to write the code needed to pass the test. Thus, they quickly write the functional code. This code itself, not mock-ups, prototypes or documents, is used to quickly and continuously validate with the customer representative if the software meets the expectations. For that purpose,

[10]In *Concordion* the term *example* denotes sentences, tables or whatever that demonstrate the specified behavior for a feature. Later these examples will be linked with the functional code through *JUnit* test cases known as fixtures.

we use the *Concordion* HTML files that in a very simple manner (only by opening the file in a browser) show the details of the test and the results of running the examples with a green and red code for indicating successes and failures. Note that the examples in these files are linked with the functional code through *JUnit* test cases. In this way, we solve one of the main problems we have in the project: the early requirement verification and validation.

Passing all the acceptance tests is necessary but insufficient to ensure the system meets the customer needs. For that, together with the automated acceptance testing, manual exploratory testing is done by testers, analysts and the customer representative. During these tests some bugs were found, of course, but the code was clean and worked properly in most cases.

In addition to the functional testing described so far, the testers also prepare and execute performance tests (load and stress tests) of those functionalities identified as critical during analysis or design. For these tests we use a similar platform to the one used in Latorre et al. (2005). If a fault is found during these tests, the IT architect assigns the correction to the programmer with the specific knowledge of how to solve the problem and/or improve the system performance.

5 Results

Here we report our experience in a specific project within the industrial environment from an empirical point of view. Since the goal of this work is not to determine effectiveness of ATDD and UTDD, we cannot show exhaustive comparisons between quality or effectiveness metrics of our development and other approaches (Section 6 addresses threats to validity). The main empirical metrics for us are:

- To have successfully reached the milestone of finishing the development on schedule without an extra economic cost.
- The very positive customer satisfaction at the end of the project.

In the following sections we present the general project results and the lessons learned about ATDD and UTDD.

5.1 Milestone Achieved: Successful Production Delivery

Five and a half months after the development started, we delivered on time a first version of the application. The project was strategic for the customer's business plans. For this reason, and given the problems with the previous development team, before the production delivery three phases of validation testing were performed by the product owner.

- In the first phase, the person of the customer who worked with us during the development process and three more people performed a rigorous testing of the system in order to verify the software against its specifications.
- In the second phase, a testing group of twelve workers of different restaurants tested the application during two weeks in a controlled testing environment.
- And, finally, in the third phase, the application was used for four weeks during production of six restaurants.

Table 2 Results during customer validation testing

	Critical	Back-end	Front-end	L&F
Phase #1	0	2	10	3
Phase #2	0	2	2	1
Phase #3	0	1	0	6

Table 2 shows a summary of the bugs reported during each of these validation phases. Bugs were classified in four levels: critical, back-end, front-end and L&F. One of the requirements imposed by the customer to accept the software was that no "critical" errors should occur during validation. Those bugs belonging to the front-end associated with changes in the look & feel or texts in the screen were graded as L&F. The rest of bugs were graded as front-end or back-end depending on the part of the application they belonged. During the three testing phases some bugs were found, but none of them was critical. If we omit the bugs graded as L&F, the bugs found during validation could be explained by two different reasons. The five bugs belonged to the back-end were minor failures due to insufficient unit tests, i.e., the absence of a *JUnit* test covering a specific behavior. Note the low percentage of uncovered behavior (c.f. more that 1,500 unit test written for the back-end). In the case of the 12 bugs belonged to the front-end, they were mainly associated with the combination of using the keyboard and the mouse for executing the same action. These bugs were not detected during interactive tests. In the final test in the six restaurants, only seven minor failures appeared, mainly related to the look & feel (6 out of 7).

At the end of the validation, the customer was satisfied with the results, not only from the functional point of view, but also from the performance point of view (performance tests were done in parallel to the functional tests, but results of these tests are not shown here). The delivered software met the customer's expectations and the response times were better than those in the old text mode applications. So, after the correction of the seven minor failures found in the last validation phase, the rollout in all the restaurants of the chain started. Four month after the rollout, the product owner had only reported two minor functional bugs, what denotes the quality of the code.

5.2 Effects on Requirement Engineering

The use of ATDD aided to define the project's scope and validate the requirements in a very simple manner during the specification phase. The presence of high level functional test helped the product owner to better understand and validate requirements by solving ambiguities and unclear points. Therefore, these acceptance tests improved the understanding and capturing of the business requirements, using a common language for the needs of the system. The customer did not need to read a document written in a difficult language for him that sometime could be difficult to follow. Using the established conventions in *Concordion*, the customer could read in natural language in a HTML file what the application had to do and validate this functionality by the *Concordion* examples directly linked to the *JUnit* tests. This prevented misinterpretations and misunderstandings during the development process. These benefits also were shown when incorporating a new functionality or a change to the system.

To reach these benefits in our project we needed a close cooperation with the customer. Here, the presence of the customer representative was a key factor. The customer rep participated in the specification and the definition of the acceptance test. This facilitated the communication with the product owner. Without her effort, it had been very difficult to gain all the ATDD benefits.

Finally, from the requirement engineering point of view, it is important to point out that in our project the ATDD specification covered the complete system because all the requirements had determinable results. In other projects and/or domains, this may not be possible. The adoption of ATDD in these projects may not be as good option as in ours, or it must be adopted in combination with other techniques.

5.3 Effects on Quality

Figure 3 shows how the ratio of hidden errors found during the automated regression tests evolved during the first weeks of development. It also allows a comparison with the average number of bugs per week found in a selection of similar project developed without using a test-driven strategy. To calculate these average values we have selected seven similar web projects in terms of architecture and duration developed during the last years by equivalent teams.

The large number of bugs found during weeks #1 and #3 in our project in the front-end was due to the requirement of controlling the application both simultaneously with the keyboard and the mouse. Our methodology allowed to detect and correct this kind of errors early in the development, avoiding to repeat them later. The rest of bugs detected during the automated regression tests were mainly related to collateral effects of new design decisions. Comparing with the metrics of non-TDD projects, we observe that the total number of hidden errors is lower. This is because the bugs were found mainly during the execution of the unit tests. UTDD practice imposed to

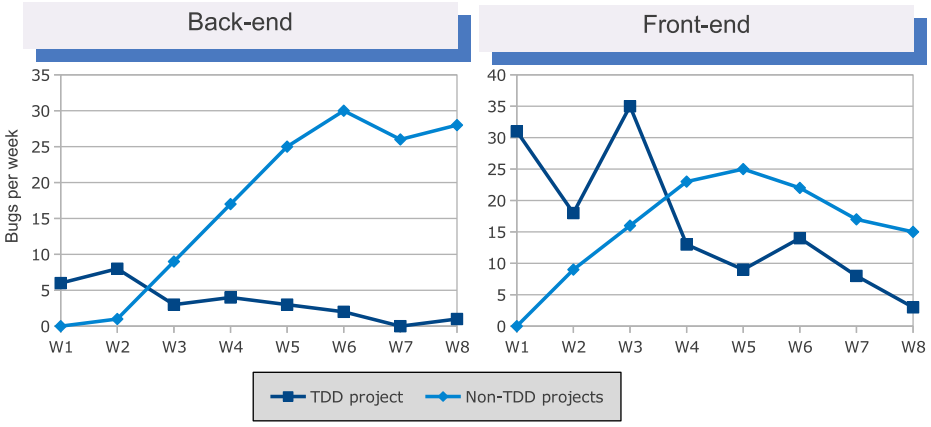

Fig. 3 Evolution of bug detection during the automated regression tests in our project and in other seven similar projects developed without using test-driven techniques. All these projects are Java web projects with a similar architecture, whose duration lies between 6 and 8 months and with development teams between 9 and 14 members. Table 3 allows to compare standard metrics of all these systems. W_n denotes the number of week. Note that these values only provide an approximate comparison since the context of each project is different

Table 3 Comparison between the code size metrics of the original version, our first release (new version) and a selection of seven similar non-TDD projects

Version	LOC (js)	LOC (java)	#Classes	LOC (tests)
Original	37,644	176,826	763	???
New	18,524	126,962	647	29,264
Non-TDD	12,660 ± 6,059	131,380 ± 27,890	712 ± 115	12,803 ± 4,638

specify these tests ahead of time and to execute them frequently, what gave instant feedback to the programmers about the quality of their code. This feedback aided to catch defects early in the software development process, preventing them from becoming expensive problems and, consequently, reducing the cost of correction.

As Fig. 3 shows, the use of UTDD reduced the percentage of bugs left to be found in later testing. This implied that the number of faults found by the customer was very low (see Table 2), what increased the customer's confidence during the rollout. The reduction in the number of bugs does not necessarily imply that the quality of the code generated with UTDD is higher than the quality of the code generated with other methodologies. Other methodology might have produced a code with a similar quality (or even better). The important thing for us here is that the code generated was a high quality code. The main factors that helped to reach this quality level were the continuous integration and verification of the code.

Although this comparison is not a good metric because the original version was a very poor implementation and we do not know the experience and know-how of the previous development team, if we compare our software with the original version, it looks that ours has a higher quality. Not only in the sense the software did what the customer wanted. If we compare the internal quality in terms of the number of lines of code (LOC), our code size is clearly smaller (see Table 3), even taken into account that we implemented new functionalities not included in the original version. We decreased the LOC around 32 % (50 % the *Javascript* code, and 28 % the *Java* code). It could be a consequence of the generation of a cleaner code because of the refactorings that improved the code structure, and because anything not covered in a test is not implemented. However, if we compare the metrics of the test-driven development project and the averages of non-TDD projects (Table 3), this internal quality improvement is not so clear. Keep in mind that this is just an approximate comparison since the context of each project is different. The *Java* LOC and the number of classes are very similar. The difference is only on the *Javascript* LOC. In this case, the average LOC in the non-TDD project is around 33 % lower due to that in four of the non-TDD projects the client side functionality is simpler (note the percentage of variance). If we calculate the average value for the three projects requiring a similar interaction with the user in the front-end, the *Javascript* LOC are also equivalent (18,524 vs. 17,112). Therefore, the use of ATDD and UTDD does not seem to have a significant impact on software quality. Nevertheless, as expected, if we focus on the LOC of tests, we corroborate that UTDD increases the test cases volume.

Finally, another benefit we found in the adoption of UTDD is that the code is very easy to maintain. During the development the product owner demanded new functional requirements both in the front-end and the back-end. These new requirements were introduced into our work-cycle in the same manner the rest of

requirements. Evaluate and implement the changes was easy and fast. On the one hand, the code was clear and simple. On the other hand, the tests themselves had all the information needed about the requirements and, therefore, the documentation had not to be reviewed. Furthermore, the automation of the regression tests allowed to quickly validate the impact of the changes. This benefit also appeared during the bug correction after release.

5.4 Effects on Productivity

From the productivity point of view, the use of UTDD has positive and negative effects. On the one hand, it allows to produce effective code from the beginning of the project. In this sense, in our project, we did not need a "big design up to front" (BDUF) (Beck and Andres 2004). After the first revision of the system architecture and the high-level design, we started the development without neither a detailed analysis nor design. The detailed design emerged and evolved during coding, so we gained time at the beginning of the project. On the other hand, according to some studies, adapting the programmers' mindset to UTDD is difficult to some extent (George and Williams 2003; Crispin 2006). This adaptation can be a factor with a negative impact over productivity. Furthermore, some experimental results also show that test-driven development takes more time than other methodologies (see Section 2.1 for details). This decreased productivity is usually due to the number of tests written and executed. For example, in our case, including the high-level functional tests, we wrote 1,523 automated tests and 518 interactive tests. These numbers are higher than the ratio in other similar projects without using a test-driven approach (about 1,200 and 250 tests respectively). But it is well known that the use of automated tests improves the quality of the test effort and minimizes the schedule (Dustin et al. 1999). Furthermore, the extra initial development time due to writing the automated tests seemed to be compensated with the early defect detection.

The trade-off between positive and negative factors did not produce a significant decreased productivity (see below). Note that the project finished on schedule when the schedule estimation was performed with metrics for a test-last approach. The team worked overtime during different phases of the development. However, given the schedule pressure, this was something expected even when considering the use of a waterfall-like strategy. After that, when we decided to use a test-driven development strategy, we expected the difference between the real productivity and the estimated one was larger, specially taking into account that we had to re-implement more than 75 % of the code. Nevertheless, actually, the impact in the effort (and consequently in the cost) was minimum. In this sense, a good metric for us was that, at the end of the project, the team members thought that the used methodology was an effective approach. Reviewing the effort by roles:

- Analysts spent more time in the definition than in similar projects because of the use of *Concordion*. However, this effort reduced the testing and validation time in comparison with similar projects. The trade-off between both factors made the analyst productivity approximately the same than in other projects.
- The programming time more or less was the same as the estimated. Programmers wrote a little more automated tests than with other methodologies. However, these tests allowed to catch defects early in the software development process

when they were easily solved. Furthermore, these tests also reduced the testing time.
- Functional testing required much less time.
- And finally, the coordination time was higher than the estimated. In our project, this was the main problem from the effort/cost point of view, but, given the role that the IT architect played in the project, this overtime was something expected. He reviewed all the process and coordinated the workflow. In addition, he also played the role of an analyst. This made the IT architect a bottleneck. To avoid becoming a problem, he needed to spend more time in the project. In other projects this could be solved by increasing the number of analysts.

5.5 Required Expertise

Some studies indicate that with UTDD the productivity is lower for expert groups (Höfer and Philipp 2009). However, from our point of view, the experience and know-how of the team members were two of the main factors for the project success. In our case, most of the team members had around 6 and 10 years experience in waterfall-like development approaches, without any experience in test-first development techniques. In some cases, they had a complete unknowledge of the software tools we use to support our development process (e.g. nobody has any experience with *Concordion*). The team members only received a brief course on ATDD and UTDD. And, nevertheless, the productivity was good since the first day. A more detailed analysis of the productivity metrics shows that the transition to UTDD mindset was quiet easy for the seniors and the student. In the case of the juniors, they needed a little more time.

6 Threats to Validity

In this section we list threats that could potentially invalidate our results. In order to limit the scope our claims as well as to explain their utility, the first important fact is that this is an empirical report. Like most empirical studies, the validity of our results is subject to several threats. Here we analyze external, internal and statistical conclusion validity (Shadish et al. 2001).

Concerning the generalization of the findings (*external validity*), as with all empirical studies (Basili et al. 1999), the results showed here should not be taken as valid in the general sense. In our case a Hawthorne effect is not possible, but we present results pertaining to one project and one team. Therefore, we cannot obtain statistical results (*statistical conclusion validity*). In a real industrial project like ours, the environment cannot be controlled to the same extent as in isolated research experiments and it is nearly impossible to control all variables. Thus, there exist several threats concern external factors that may affect the results (*internal validity*). In particular, we do not know whether the success of the project can be attributed to ATDD and UTDD adoption or to many other possible factors like the presence of the customer rep, the capability of the new development team, etc.

Thus, the utility of our result is just to show an example of project in the industrial context where ATDD and UTDD are successfully applied.

7 Discussion and Conclusions

In a context like industrial environment where it is of crucial importance to reduce development costs, it is difficult to change a well-known practice. In the case of test-driven development practices, from my experience in different software development companies, in many occasions they are not usually applied in commercial projects because the prevailing idea is that they are expensive and/or risky. As an example, Causevic et al. (2011) shows seven possible factors limiting the industrial adoption of UTDD. Before this project, each time I had to evaluate the adoption of a test-driven development approach in a new project I always had doubts about what I could expect from it in terms of required effort and cost, resulting quality and required expertise in a "real world" project within the industrial environment. I never found any conclusive response to these questions. So I had never used it before in an industrial environment. After this project, I believe that ATDD and UTDD have a place in the arsenal of development techniques of a software professional in the industrial environment.

Looking back, it seems very difficult to successfully complete our project with a different methodology. When we reviewed the original version of the software, we thought that it was not possible to successfully finish the development in just six months. The workflow defined in this paper imposed a continuous quality assurance of the code that, under the project circumstances, was key for success. The expected benefits of adopting ATDD and UTDD have been shown to be true. Note that in our project there were some factors facilitating the adoption of ATDD and UTDD: the presence of the customer representative, the fact that all the requirements had determinable results and the capability of the development team.

The results of the project presented in this report point out that the use of ATDD and UTDD presents both advantages and disadvantages. Therefore, they are probably not the best solution for all the projects within the industrial environment. However, as we show here, under some circumstances (time constraints, lack of specifications, expert development team...), they can be successfully adopted by themselves or in combination with other techniques as a possible option. Although the reader must take into account that each project in the industrial environment is different and the fact that a practice works for a specific context (project, team, company...), does not mean that the same practice may not fail in another; we expect to provide a piece of evidence to software professionals to become more confident in the adoption of ATDD and UTDD as effective solutions to be considered. Our main conclusions after these projects are:

- ATDD requires an extensive cooperation from the customer.
- ATDD helps to clearly capture and validate the business requirement with the customer during the specification phase.
- ATDD helps developers to better understand the requirements.
- Using UTDD does not change significantly neither the required effort nor the software quality.
- UTDD allows to catch defects early in the software development.
- UTDD generates code that is easy to maintain.
- Executable acceptance test facilitates the system verification.
- A high level of expertise in test-first development is not required for success. Obviously, when a team has experience with test-first development techniques,

a part of the way is walked. In any other case, training is needed, but a high level of expertise is not mandatory. Moreover, if the team is experienced, this fact is not critical for the project. If the team members have enough experience, they could easily adapt their mindset to test-driven development.

Acknowledgements The author would like to thank Ramón Huerta for his comments and suggestions on the initial version of the manuscript. I am also very grateful to the insightful and helpful feedback of the anonymous reviewers which helped substantially to improve the paper.

References

Adzic G (2011) Specification by example: how successful teams deliver the right software. Manning Publications Co

Andersson J, Bache G, Sutton P (2003) Xp with acceptance-test driven development: a rewrite project for a resource optimization system. In: Proceedings of the 4th international conference on extreme programming and agile processes in software engineering, XP'03. Springer, Berlin, Heidelberg. ISBN 3-540-40215-2, pp 180–188. http://dl.acm.org/citation.cfm?id=1763875.1763904

Astels D (2003) Test driven development: a practical guide. Prentice Hall Professional Technical Reference. ISBN 0131016490

Basili VR, Shull F, Lanubile F (1999) Building knowledge through families of experiments. IEEE Trans Softw Eng 25(4):456–473. ISSN 0098-5589

Beck K (2001) Aim, fire. IEEE Softw 18(5):87–89. ISSN 0740-7459

Beck K (2003) Test driven development: by example. Addison-Wesley Professional (2003)

Beck K, Andres C (2004) Extreme programming explained: embrace change, 2nd edn. Addison-Wesley Professional. ISBN 0321278658

Bhat T, Nagappan N (2006) Evaluating the efficacy of test-driven development: industrial case studies. In: Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering, ISESE '06. ACM, New York. ISBN 1-59593-218-6, pp 356–363

Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-oriented software architecture: a system of patterns. Wiley. ISBN 0471958697

Cao L, Ramesh B (2008) Agile requirements engineering practices: an empirical study. IEEE Softw 25(1):60–67

Causevic A, Sundmark D, Punnekkat S (2011) Factors limiting industrial adoption of test driven development: a systematic review. In: 2011 IEEE fourth international conference on software testing, verification and validation (ICST). IEEE, pp 337–346

Cockburn A (2002) Agile software development. Addison-Wesley Longman Publishing Co., Inc., Boston. ISBN 0-201-69969-9

Crispin L (2006) Driving software quality: how test-driven development impacts software quality. IEEE Softw 23(6):70–71

Damm L-O, Lundberg L (2006) Results from introducing component-level test automation and test-driven development. J Syst Softw 79(7):1001–1014. ISSN 0164-1212

Dustin E, Rashka J, Paul J (1999) Automated software testing: introduction, management, and performance. Addison-Wesley Longman Publishing Co., Inc., Boston. ISBN 0-201-43287-0

Erdogmus H, Morisio M, Torchiano M (2005) On the effectiveness of the test-first approach to programming. IEEE Trans Softw Eng 31(3):226–237. ISSN 0098-5589

Erdogmus H, Melnik G, Jeffries R (2010) Test-driven development. In: Encyclopedia of software engineering, pp 1211–1229

Ficco M, Pietrantuono R, Russo S (2011) Bug localization in test-driven development. Adv Soft Eng 2011:2:1–2:18. ISSN 1687-8655. doi:10.1155/2011/492757

Flohr T, Schneider T (2006) Lessons learned from an xp experiment with students: test-first needs more teachings. In: Proceedings of the 7th international conference on product-focused software process improvement, PROFES'06. Springer, Berlin, Heidelberg. ISBN 3-540-34682-1, 978-3-540-34682-1, pp 305–318

Gärtner M (2012) ATDD by example: a practical guide to acceptance test-driven development. Addison-Wesley Professional

George B, Williams L (2003) An initial investigation of test driven development in industry. In: Proceedings of the 2003 ACM symposium on applied computing, SAC '03. ACM, New York. ISBN 1-58113-624-2, pp 1135–1139

Gupta A, Jalote P (2007) An experimental evaluation of the effectiveness and efficiency of the test driven development. In: Proceedings of the first international symposium on empirical software engineering and measurement, ESEM '07. IEEE Computer Society, Washington. ISBN 0-7695-2886-4, pp 285–294

Haugset B, Stalhane T (2012) Automated acceptance testing as an agile requirements engineering practice. In: 2012 45th Hawaii international conference on system science (HICSS). IEEE, pp 5289–5298

Hendricksons E (2008) Acceptance test driven development (atdd): an overview. In: Seventh software testing Australia/New Zealand (STANZ). Wellington

Höfer A, Philipp M (2009) An empirical study on the tdd conformance of novice and expert pair programmers. In: XP, pp 33–42

Janzen D, Saiedian H (2005) Test-driven development: concepts, taxonomy, and future direction. Computer 38(9):43–50. ISSN 0018-9162

Janzen DS, Saiedian H (2006) On the influence of test-driven development on software design. In: Proceedings of the 19th conference on software engineering education & training, CSEET '06. IEEE Computer Society, Washington. ISBN 0-7695-2557-1, pp 141–148

Janzen D, Saiedian H (2008) Does test-driven development really improve software design quality? IEEE Softw 25(2):77–84. ISSN 0740-7459

Jeffries R, Melnik G (2007) Guest editors' introduction: Tdd–the art of fearless programming. IEEE Softw 24(3):24–30. ISSN 0740-7459

Kaufmann R, Janzen D (2003) Implications of test-driven development: a pilot study. In: Companion of the 18th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA '03. ACM, New York. ISBN 1-58113-751-6, pp 298–299

Koskela L (2007) Test driven: practical tdd and acceptance tdd for java developers. Manning Publications Co., Greenwich. ISBN 9781932394856

Larman C, Basili VR (2003) Iterative and incremental development: a brief history. Computer 36(6):47–56. ISSN 0018-9162

Latorre R, López F, Martínez, AE (2005) Sharing of precompiled database statements in j2ee applications. Softw Pract Exper 35(3):301–311. ISSN 0038-0644

Leffingwell D (2010) Agile software requirements: lean requirements practices for teams, programs, and the enterprise. Addison-Wesley Professional

Madeyski L (2005) Preliminary analysis of the effects of pair programming and test-driven development on the external code quality. In: Proceedings of the 2005 conference on software engineering: evolution and emerging technologies. IOS Press, Amsterdam. ISBN 1-58603-559-2, pp 113–123

Martin RC (2003) Agile software development: principles, patterns, and practices. Prentice Hall PTR, Upper Saddle River. ISBN 0135974445

Martin RC (2007) Professionalism and test-driven development. IEEE Softw 24(3):32–36. ISSN 0740-7459

Martin RC (2008) Clean code: a handbook of Agile software craftsmanship. Prentice Hall, Upper Saddle River. ISBN 978-0-13235-088-4

Maximilien EM, Williams LA (2003) Assessing test-driven development at IBM. In: Clarke LA, Dillon L, Tichy WF (eds) ICSE. IEEE Computer Society, pp 564–569

Müller MM, Hagner O (2002) Experiment about test-first programming. IEE Proc-Softw 149(5):131–136

Nagappan N, Maximilien EM, Bhat T, Williams L (2008) Realizing quality improvement through test driven development: results and experiences of four industrial teams. Empir Software Eng 13(3):289–302. ISSN 1382-3256

Newkirk JW, Vorontsov AA (2004) Test-driven development in microsoft .net. Microsoft Press, Redmond. ISBN 0735619484

Park S, Maurer F (2009) Communicating domain knowledge in executable acceptance test driven development. In: Abrahamsson P, Marchesi M, Maurer F (eds) Agile processes in software engineering and extreme programming. Lecture notes in business information processing, vol 31. Springer Berlin Heidelberg. ISBN 978-3-642-01852-7, pp 23–32

Pugh K (2010) Lean-agile acceptance test-driven development: better software through collaboration. Addison-Wesley Professional

Rafique Y, Misic VB (2013) The effects of test-driven development on external quality and productivity: a meta-analysis. Softw Eng IEEE Trans 39(6):835–856. ISSN 0098-5589. doi:10.1109/TSE.2012.28

Ricca F, Torchiano M, Di Penta M, Ceccato M, Tonella P (2009) Using acceptance tests as a support for clarifying requirements: a series of experiments. Inf Softw Technol 51(2):270–283. ISSN 0950-5849. doi:10.1016/j.infsof.2008.01.007

Sanchez JC, Williams L, Maximilien EM (2007) On the sustained use of a test-driven development practice at ibm. In: AGILE Conference (AGILE), pp 5–14

Shadish WR, Cook TD, Campbell DT (2001) Experimental and quasi-experimental designs for generalized causal inference, 2nd edn. Houghton Mifflin. ISBN 0395615569

Williams L, Maximilien EM, Vouk M (2003) Test-driven development as a defect-reduction practice. In: Proceedings of the 14th international symposium on software reliability engineering, ISSRE '03. IEEE Computer Society, Washington. ISBN 0-7695-2007-3, pp 34–45

Roberto Latorre is a software engineer and researcher in NeuroLogic Soluciones Informáticas. He is also a Profesor Ayudante Doctor of computer science at Dpto. Ingeniería Informática, Escuela Politécnica Superior, Universidad Autónoma de Madrid, Madrid, Spain. His research interests include different topics in software engineering (e-business software engineering, quality software development) and neurocomputing (from the generation of motor patterns and information coding to pattern recognition and artificial neural networks). He has Ph.D. in computer science and telecommunications from Universidad Autónoma de Madrid.