What Concerns Beginner Test-Driven Development Practitioners: A Qualitative Analysis of Opinions in an Agile Conference

Mauricio Finavaro Aniche[1], **Thiago Miranda Ferreira**[1], **Marco Aurélio Gerosa**[1]

[1] Department of Computer Science
Institute of Mathematics and Statistics
University of São Paulo
PO Box 66.281 - 05.508-090 - São Paulo - SP - Brazil

{aniche, gerosa}@ime.usp.br, thiago.miranda.ferreira@usp.br

Abstract. *Test-Driven Development (TDD) is an important practice among agile practitioners. Many studies in the literature, well-known authors, and developers claim that TDD simplifies code, improves software design, and increases productivity. This paper reports a qualitative analysis on beginners' opinions about TDD, captured in an agile conference. All participants had at most 3 years of experience in TDD, but different levels of experience in software development, which allowed a rich discussion about the effects of the practice in the real world. Based on the participants' answers, beginners consider TDD as primarily a design technique. They also agree that TDD does not solve problems by itself, and programmers should have a deep knowledge about design and OO principles. However, baby steps, productivity, and difficulty in learning are a polemic topic among them.*

1. Introduction

Test-Driven Development (TDD) is one of the agile practices that focus on feedback. In a more formal definition, TDD is the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, programmers first develop a test that specifies and validate what the code will do. Programmers, then, produce exactly as much code as will enable that test to pass. Then they refactor (simplify and clarify) both the production and test code [Alliance 2005].

People usually discuss if TDD is a testing technique, as programmers need to write unit tests all the time, or a design technique, as the generated tests give feedback about design and programmers use it to improve the design. Some well-known authors claim that TDD is in fact a design technique and testing is just a consequence [Martin 2002].

Programmers also discuss about how to do baby steps in real world. Some of them believe that programmers should do baby steps all the time for every situation and, on the other side, some programmers claim that there is no need to go that slow. Productivity is another controversial subject. Although there are some studies showing that TDD does not reduce productivity [George and Williams 2003] [Erdogmus et al. 2005], some people still think that TDD makes the development team go slowly.

This paper reports a qualitative analysis, based on Grounded Theory techniques [Juliet Corbin 2007], of opinions from TDD practitioners inside an agile conference and compares it with the literature. The opinions were gathered in a session conducted by the researchers. The objective of the session was to generate a free discussion about TDD topics with whoever wanted to participate. Most of the participants have less than 3 years of experience in TDD which may classify them as beginners in the practice.

2. Scope

The "Encontro Ágil" conference (a free translation would be Agile Meeting Conference), located in São Paulo, is one of the first brazilian conferences about agile software development. São Paulo is the largest city and the major economic center in Brazil. In the 2010 edition, the conference evolved its format to a different proposal: no lectures at all. The idea was to enhance interactions between all the 200 participants and the exchange of experiences. Because of that, the event program was compound by sessions with open spaces, dojos, workshops, and games.

2.1. The Test-Driven Development Session

The authors conducted an open session about Test-Driven Development. The main idea of the session was to understand the vision that people who actually use TDD have about the practice in their daily work. The objective was to assess what an audience made up of inexperienced practitioners think about TDD, whether they apply it in their routines and what opinions they have about the effects of TDD in software design.

The session adopted an informal chat using a technique called fishbowling [1] to make the discussion more active. In this technique, four people are kept in the center of the discussion with an extra chair available for whoever wishes to participate. In this case, when someone sits on the chair, another participant leaves in order to maintain four people discussing, and one chair available for a new member of the public to join the discussion.

The session took about 1 hour and a half and the researchers proposed 7 different questions for discussion. Some questions were provocative in order to promote discussion among the participants. The following list presents the questions in the order they were discussed:

1. Is TDD a test practice or a design practice?
2. Discuss the following statement: it is impossible to generate a good design without doing TDD.
3. How do tests help you create a better design, i.e. classes with low coupling, high cohesion, simple code, etc.?
4. Many people apply TDD doing "as simple as possible", even if the code to be written is very simple (e.g. a simple calculator that does sums, in which the programmer creates and returns constant values, adds many conditional clauses until a big refactoring). Do you think it is really necessary?
5. Now that I've used my tests to improve the design of the system, can I throw them away?

[1]http://en.wikipedia.org/wiki/Fishbowl_(conversation). Last access on November 27th 2010

6. Some people say that writing tests before programming decreases productivity (for example, if a programmer writes a hundred lines of code a day, he is going to write 50 lines of tests and 50 lines of production code). What do you think about it?
7. Talk about your first time doing TDD.

The participants had about 7 minutes of discussion for each question. As soon as each slot finished, the researchers asked the public to vote whether the question being discussed should continue. For the poll, the researchers distributed among the participants a card with a green face, which meant that they wanted to keep the current discussion, and a red one, which meant that the topic was not interesting anymore and they could go to the next question.

The session had 10 participants and all of them entered in the fishbowling at least once. None of the researchers participated in the discussions as it would have biased the participants. After the session, the researchers asked them to fill out a survey about their experience in software development and TDD. Section 2.2 explains the participants' profile.

For further analysis and discussion of the ideas that emerged at the session, a voice recorder was placed in the center of the room and all participants were aware that their opinions were being recorded and their statements would be used anonymously and only for the purpose of the research.

2.2. Audience Profile

The researchers asked the participants to answer a short anonymous questionnaire about their experience in software development and TDD. The questionnaire and its possible answers are represented on Table 1.

Question	Options
Time experience in software development (in years)	0-1, 1-2, 2-3, 3-4, 4-5, 5-6, 6+
Time experience in TDD (in years)	0-1, 1-2, 2-3, 3-4, 4-5, 5-6, 6+

Table 1. Questionnaire used during the session

Most respondents had between 0 and 3 years of experience with TDD, and only two were more experienced than that. Also, only 1 participant had never practiced TDD before. Regarding experience in software development, the respondents were fairly homogeneous. Beginner programmers (less than 2 years) represented 30% of the audience and those who were more experienced (more than 3 years) represented 70%.

Based on these numbers, the audience may be classified as an experienced group of software developers that are starting to practice TDD. Such profiles are still very common in industry, as programmers are still getting to know agile practices and TDD specifically.

3. Research Methodology

The research on software engineering practices, since it involves humans, is benefited from the use of qualitative techniques. In particular, when evaluating the effects of TDD, for example, it is hard to separate it from other agile practices that are usually done together [Beck et al. 2001] [Runeson and Host 2009]. Qualitative analysis is well suited for many kinds of software engineering research, as the objects of study are contemporary phenomena, which are hard to study in isolation. Case studies do not generate the same results on e.g. causal relationships as controlled experiments do, but they provide deeper understanding of the phenomena under study [Runeson and Host 2009]. In this proposal the researchers chose some techniques based on Grounded Theory [Juliet Corbin 2007].

The researchers created all the questions and some of them intent to stimulate the discussion. The idea was to generate a discussion and make people comfortable to talk about the questions. During the session the researchers were only observing the discussion without actually participating on it. The researchers took notes about participants' feelings, i.e., when they were agreeing or disagreeing with something. Although the number of participants was quantitatively small (only 10 people), the volume of data gathered during this one hour and a half session is reasonably high and covers different topics about the TDD practice.

As mentioned before, the discussion was entirely recorded. The researchers transcripted the audio and double listened in order to check for eventual errors in the transcription. After that, two researchers individually started the coding process, which is the act of organizing and classifying data into categories or segments of text before trying to give a meaning to that piece of information [Rossman and Rallis 2003]. At the first moment, researchers were free to create any code they want. After that, researchers discussed about each code created and merged them. The intent of this part of the process is to reduce bias. The codes were then grouped into themes by the researchers, which became the subsections of Section 4.

As expected, articulated participants talked more than others. That may influence the discussion to a participant's point of view. However, when a stronger position was placed, all participants argued about it until it converged into something that everyone was comfortable with. Researchers were aware of it and took this into account during the analysis. Afterwards, participants were invited to review this paper in order to find any flaw or bias during the analysis. Their revision can be found in Section 7.

4. Findings

4.1. TDD as a Design Technique

Although some definitions of TDD focus on its testing perspective, most of the participants affirmed that they use TDD mainly as a design technique, just like Robert Martin [Martin 2002] and Kent Beck state [Beck 2001] [Beck 2002]: *"TDD's goal may be to do real testing but in order to make the test you end up by influencing the design."*

Participants often had the opinion that design is a consequence of testing: *"Tests are the means to obtain the design."* This concept was reinforced by a few participants when they talked about the effects of TDD on class coupling. They agreed that if programmers do not decouple their code they would not be able to write a unit test. Also, if programmers spend too much time trying to write a simple test, it may indicate that there is something wrong with the design: *"At the moment you are testing unit by unit and you are writing a unit test and you see that it is too complex, you already know that you need something. You know that you are doing something you should not do. Tests show that, if something is really hard to test, then it is because there's something wrong."*

A participant mentioned the effect of the test feedback. As developers receive constant and rapid feedback about the code design, they are able to find some design smells and fix them when it is still cheap and easy: *"TDD makes you start writing the code from the very beginning. After that, you notice that what you have just written at the beginning is not good enough and then you start to improve the code. This is how TDD ends up influencing."*

Another interesting point of view raised by one of the participants was that when programmers are doing TDD and they write a unit test for a class that sometimes do not even exist, the test is the first client of that class. It encourages developers to write simple code: *"TDD ends up influencing us to simplify the code. As we are the first client of our own system, we end up trying to simplify the design and reduce coupling, and that is why our software design keeps evolving constantly."*

On the other hand, there was one participant who did not believe that TDD is a design technique, but a testing technique. He kept asking questions about testing to the other participants. He believed that the benefits of TDD are the test suite programmers have at the end. He asked: *"But you need to have some automated test suite, even without TDD. It will help me change my design anyway, won't it?"* Others replied that there is a difference in design when writing tests after, as it was mentioned through all this section.

Participants gave most of their opinions about the effects of TDD in design during questions number 1 and 3. A curious fact is that most participants voted for an extra round for question number 1, but question number 3 ended before the 7 minutes limit.

4.2. Refactoring Confidence

The opinion of the majority is that refactoring confidence a great advantage when doing TDD. Programmers can evolve the code and the design without fear. If something goes wrong the test suite warns them: *"TDD influences a lot on design, for sure. However, the most important thing in my opinion is the safety that it gives me at the moment of refactoring the system. Without TDD I don't have a clear criterion whether my refactoring was successful or not"*; *"I like the refactoring part the most. When you do your first refactoring and see a lot of red tests and, then, when you see them turn green, you think: I can change this code with no fear at all!"*. All participants agreed immediately with these opinions.

This was deeply discussed also in question number 5. That was a tricky question: if TDD is a design technique and the programmer used the test solely to improve the design, as soon as the design is done the programmer could delete all tests. This question was promptly replied by most of them with phrases like *"No way!"* and *"You can't do it in any case!"* followed by some laughing. One of them made an interesting comparison: *"I've done the source code, it did compile and meet the needs. Can I delete the source code now?"*

4.3. Initial Skepticism

Writing the test before the code goes against the traditional approach of software programming. All participants that were on the discussion said that, when introduced to TDD, they did not believe in the practice.

One of them suggested that the best way to see TDD's benefits is by experimenting the practice with some pet project at home or at work. They all said that after a certain time of practice the benefits become evident despite the initial skepticism caused by the paradigm shift. In their opinion, when developers find that the practice helps them making better software they start to believe in TDD: *"It's hard to believe because the benefit takes some time to show up. However, if he thought that TDD would help him develop better software in a long term, he would believe in it."*

4.4. Experience Matters

All participants agreed that TDD does not solve all design problems by itself. They stated that there were thousands of projects in the past that still work and have a good design.

In spite of that, they also agreed that doing TDD helps developers to create a better design in less time than the traditional approach. This view is represented by the following statement from one of them: *"TDD influences and helps (the design) but it is not mandatory. It is possible to do a good design without TDD but we will face other problems."*.

4.5. Different Opinions About Baby Steps

TDD states that developers should do the simplest thing that makes the test pass. In order to achieve that, programmers do it in baby steps, which are small changes in the code. They help programmers to avoid unnecessary and complicated code, not even at implementation level, but also at design level. Participants were divided about baby steps. Some of them believed that programmers should do baby steps all the time and some of them believed that baby steps all the time are not productive.

The example discussed was the implementation of a simple method *sum(int a, int b)* that sums two integer numbers. The idea was to generalize that example to a much more robust algorithm during the discussion. Baby steps group affirmed that they would only get into *return a+b* after a few tests with integers, for example. The other group affirmed that programmers could go right to the final implementation with just one test for the sum.

Those who believed that baby steps all the time are mandatory argued that when programmers do not do it, they may forget to test some corner cases and a refactoring may change the expected behavior of the system. They would prefer to have 10 unit tests that test the same behavior than forget one of them. They also argued that it is really hard to implement the simplest code if programmers do not do baby steps. If programmers do not do baby steps the chance to write unnecessary code is higher.

On the other hand, the other group said that experience should be taken into account when doing baby steps. They agreed that doing baby steps all the time are not productive. A participant paraphrased Kent Beck: *"I am not sure if you need baby steps* *all the time. Kent Beck says in his XP book that the programmer should use his own experience. In his TDD book he says that, with experience, you feel whether you make a bigger step or not."* Some of them also said that 10 tests for a single case is a waste of time. In addition, they noticed that if there were 10 tests for a feature it would be harder to change that behavior: the programmer would have to alter 10 tests instead of just one.

The interesting part about the last statement is that participants perceived a possible coupling between test and production code. Although they did not mentioned the correct term, they are already aware of it. During another question, a participant commented about testing one case of each equivalence class. It means that, in the calculator example, the programmer would test only once a sum of two positive integers, then only once a sum of two negative integers, and so on.

4.6. No Productivity at the Beginning

Almost all participants mentioned that when they started using TDD, they did not feel too productive. However, in medium terms, the productivity went high as it was much easier to fix bugs. In their opinion, using traditional approaches, programmers may deliver code faster but after some time the productivity decreases as they spend too much time searching for bugs and trying to evolve the software. They also tried to find a definition for productivity. In their opinion, productivity may not be measured in lines of code, but they did not reach a conclusion for that question.

The discussion turned to a more philosophical discussion: *"We should write code with quality. Write a code that does not work may help you to achieve the customer's deadline, but he will call us incompetent!"* They were saying that as developers, they should do the best they can and write tests (not only TDD), and this is essential to the success of a project. One of them also cited the doctor's anecdote: *"If you tell a doctor to do a surgery without cleaning the tools as you want it faster and cheaper he will not do the surgery; if the customer says the same to you, you should not develop the software!"*

4.7. Difficulty in Learning

A topic that was mentioned more than once was that learning TDD is not easy. Some participants said that learning how to write good unit tests or how to mock objects is not simple. Even experienced programmers may feel it too. One of them said: *"Sometimes I look to a code and I think: I don't know how to test this!"* He even said that sometimes he had the feeling of losing productivity.

When the participants were talking about their first time in TDD they started by mentioning how they have learned it. At least three of them said that they learned it in Dojo sessions [Sato et al. 2008]. Although they mentioned that they do not usually like the way programmers do baby steps at Dojo sessions, they agreed that it is a good way to spread knowledge about the practice.

There were other ways of learning cited during the session. A participant commented about his participation in discussion lists. Another one talked about some practical videos he has found. The same participant made lots of references to books from famous authors like Kent Beck and Robert Martin during the discussions and, when that happened, other participants made a gesture indicating that they already read the book. Therefore, it is possible to infer that books are another way of learning TDD.

A participant commented about an interesting situation. He was forced to do TDD by a colleague at work. He said he did not believe at the beginning, but as soon as he himself started practicing and saw improvements in his design and all automated tests advantages, he started believing in TDD. Thus, peer learning was the way a participant learned TDD and in his opinion it was very effective.

5. Discussion

The main topic in the session was that TDD is actually a design technique, which matches with what is found in literature. Participants talked about many effects of the practice, such as the need to manage dependencies, and the design simplicity achieved by the urge to test a class without much effort. It is interesting to notice that practitioners that are experienced in software development but are beginners in TDD, notice these effects. The confidence when refactoring was also highly discussed by the participants. The test suite enables developers to change code with safety. It shows that the testing part of the technique is also useful.

As participants noticed, experience is fundamental to the process. When trying to easily write a unit test to a class, a developer needs to use good object orientation principles. TDD is a technique that gives feedback from the design, and the practitioner should use it to drive design to better solutions.

Baby steps were also a popular topic. It continued for one more round as most part of the audience voted for it. It can be justified by the fact that baby steps are one of the most misunderstood part of the practice. Baby steps try to prevent developers from creating a complex solution when a simpler one solves the problem. A programmer should be driven by his experience: if s/he is comfortable with that part of the implementation, the step may be bigger; if not, a smaller one should be taken. In researchers' interpretation, the goal of baby steps is to simplify both implementation and design, and not to remember programmers about corner case, which is a software testing activity.

However, developers face many problems when starting to practice TDD. As presented before, most of developers tend not to believe in the practice at first. The relation between testability and good design is not clear enough at the beginning. It is hard to notice TDD's effects on design when writing small projects that does not require a flexible design. In order to believe in the practice, programmers should try to use it so that they can make their own conclusions, noticing the substantial improvement in the quality of their code and design.

As participants said, the practice may reduce developers' productivity at the beginning. However, it is hard to perceive if productivity is reduced when programmers do not know how to write a unit test or because they were not familiar with OO principles. But, as noticed by participants, although they do not feel productive at the beginning, in medium terms productivity grows. This also may explain the the learning difficulty presented by the participants. Besides learning the fundamental tools to write a unit test, developers should write decoupled code; both activities have a learning curve.

Interestingly, the effects of TDD in external quality was not mentioned. That can be explained by the fact that most questions were focused on the design effects. However, the first question was clearly unbiased, and participants only discussed about the design part. It may indicate that TDD effects are only in internal quality and external quality is just a side-effect.

6. Related Work

Many empirical experiments have been done in order to evaluate TDD's effects on both internal and external quality. Also, these experiments can be divided into two categories: industry and academia. Most of experiments in industry show results similar to what is found in this study.

When talking about external quality, Janzen [Janzen 2005] demonstrated that programmers using TDD in industry produced code that passed in up to 50% more external tests than code produced by control groups not using TDD and spent less time to fix defects. A study from George and Williams [George and Williams 2003] that produced code passed between 18% and 50% more in external test cases than the code produced by groups not using TDD. The study from Edwards [Edwards 2003], with 59 students showed that TDD code has 45% less defects.

Although many other studies relate TDD and external quality, no participants mentioned effects on external quality, which may indicate that TDD effects are only in internal quality and external quality is just a side-effect.

A qualitative analysis in George and Williams showed that 87.5% of the programmers believed that TDD approach facilitated requirements understanding and 95.8% believed that it reduced debugging effort. Regarding quality, 92% of the developers believed that TDD yielded higher quality code and 79% thought it promoted simpler design.

Another study from Janzen [Janzen and Saiedian 2006] with three different academic groups (each one using a different approach: test-first, test-last, no test) found that the code produced by the test-first team better used object-oriented concepts, and responsibilities were separated in thirteen different classes while the other teams produced a more procedural code. The test-first team also produced more code and delivered more features. Moreover, tests produced by the test-first team had twice more assertions than the others and covered 86% more branches than the test-last team. Furthermore, tested classes had 104% lower coupling measures than untested classes and tested methods were 43% on average less complex than the untested ones. Langr also [Langr 2001] showed that TDD improved code quality, provided better maintainability, and produced 33% more tests. Steinberg [Steinberg 2001] showed that TDD code was more cohesive and less coupled and students reported that defects were easier to fix.

Participants agree with most of this results; in their opinion, TDD improves code quality both in implementation and design, also making it as simple as possible. The written code is also easier to maintain as the test suite ensures the behavior, giving freedom to programmers to refactor the code.

Higher productivity was also an effect claimed by the participants of this study. In their opinion, although programmers do not feel productive at the beginning, in medium terms productivity grows. This result is also confirmed by George and Williams who also found that although TDD might initially reduce productivity among inexperienced programmers, in a qualitative analysis, 78% of them thought that TDD improved overall programming productivity. However only 50% of them believed that TDD led to less code development time. A study from Erdogmus et al [Erdogmus et al. 2005] with 24 undergraduate students showed that TDD increased productivity.

The learning difficulty presented by the participants was also evaluated by researchers. Mugridge [Mugridge 2003] identified two main challenges in teaching TDD over the last two years: to get students to rethink about design, and to really engage with this new approach. Also, it is hard to explicitly develop students' skills in testing, design and refactoring.

Participants sometimes mentioned steps that are not suggested by TDD cycle. This is not anomalous. Aniche and Gerosa [Aniche and Gerosa 2010] reported that programmers sometimes do not follow TDD steps as described by Kent Beck [Beck 2002]. Some mistakes identified were: to forget the refactoring step, to build complex test scenarios, and refactor another piece of code while working on a test. Some mistakes are frequently made by around 25% of programmers.

7. Participants' review

Some participants accepted to review the paper. They all agreed with all the research findings. Some of them also showed their own interpretations of the discussion. For instance, one participant said that his conclusion about the baby steps discussion was that there is no need to do small steps for simple code; however, for a more complex one, baby steps would help him earn experience with the code he is developing. He also said that, besides the design effect, TDD helps programmers to understand the business they are dealing with.

8. Threats to Validity

Although many participants showed great theoretical and practical knowledge about TDD, it is hard to know if they do exactly the way they reported. In addition, a considerable amount of them talked about practices that differ from what TDD suggests. As some of them do not follow TDD steps the way they theoretically should, it may influence their opinion.

A few participants also affirmed that do not practice TDD regularly during their daily jobs. They often practice TDD in pet projects. It may influence the opinion as some of them lack TDD experience in real world projects.

The 10 participants represented over 7% of the event's audience and there were no pre-requirements to join the session. The sample may not be representative in order to generalize the findings on this paper.

9. Conclusions and Future Work

There are still many experienced programmers adhering to Test-Driven Development. This paper showed that most of TDD beginners' opinions and concerns about the practice match with what is reported in literature. TDD is a technique that makes design problems more visible, regardless of the level of experience with the practice; it is up to the developers to see them and improve the design. Moreover, what makes developers fix design flaws is their experience in software design, and not their experience in TDD itself. This is reinforced by the participants' opinions on the influence of experience in the process.

Almost all participants agreed that design is a consequence of testing. Programmers use the feedback from the tests to improve the design. However, the specific question about how programmers get feedback from tests did not reach the time limit, which may indicate that programmers do not know exactly how they get this feedback. They also talked a lot about design, and researchers were expecting more citations from good design techniques, but there were only few mentions to good object-oriented principles. It may indicate that a study relating unit tests and object-oriented principles need to be done.

When start practicing TDD, beginners are mainly concerned about their productivity; writing unit tests, constant refactoring, and doing baby steps all the time suggest that developers will spend too much time to write code. However, as participants suggested, the best way to see the benefits is by trying to practice TDD.

This work also contributed with a different way to gather data about any agile practice. Agile conferences with open spaces are becoming popular and it is a good place for researchers to interview people from industry and also enable participants to learn with each other. Also, researchers noticed that proposing a fishbowling was a good way to make people with diverse experience to discuss about the same subject and get different answers.

10. Acknowledgements

We would like to thank Encontro Agil 2010 organization for the acceptance on our TDD open space session. In addition, participants were an essential part of the research, and researchers acknowledge their participation and the allowance for the data analysis. Also, Caelum Learning and Innovation sponsored Mauricio Aniche and Thiago Ferreira during this research. Marco Gerosa received an individual grant awarded by CNPq.

References

[Alliance 2005] Alliance, A. (2005). Tdd. http://www.agilealliance.org/programs/roadmaps/Roadmap/tdd/tdd_index.htm.

[Aniche and Gerosa 2010] Aniche, M. F. and Gerosa, M. A. (2010). Most common mistakes in test-driven development practice: Results from an online survey with developers. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:469–478.

[Beck 2001] Beck, K. (2001). Aim, fire. *IEEE Software*, 18:87–89.

[Beck 2002] Beck, K. (2002). *Test-Driven Development By Example*. Addison-Wesley Professional, 1^o edition.

[Beck et al. 2001] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., and Thomas, J. S. D. (2001). Manifesto for agile software development. http://agilemanifesto.org/. Último acesso em 01/10/2010.

[Edwards 2003] Edwards, S. H. (2003). Using test-driven development in a classroom: Providing students with automatic, concrete feedback on performance. *International Conference on Education and Information Systems: Technologies and Applications*.

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

[George and Williams 2003] George, B. and Williams, L. (2003). An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 1135–1139, New York, NY, USA. ACM.

[Janzen and Saiedian 2006] Janzen, D. and Saiedian, H. (2006). On the influence of test-driven development on software design. *Proceedings of the 19th Conference on Software Engineering Education and Training (CSEET'06)*, pages 141–148.

[Janzen 2005] Janzen, D. S. (2005). Software architecture improvement through test-driven development. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 240–241, New York, NY, USA. ACM.

[Juliet Corbin 2007] Juliet Corbin, A. S. (2007). *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 3rd edition edition.

[Langr 2001] Langr, J. (2001). Evolution of test and code via test-first design. http://eisc.univalle.edu.co/materias/TPS/archivos/articulosPruebas/test_first_design.pdf. Último acesso em 01/03/2011.

[Martin 2002] Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, first edition edition.

[Mugridge 2003] Mugridge, R. (2003). Challenges in teaching test driven development extreme programming and agile processes in software engineering. *Lecture Notes in Computer Science*, 2675.

[Rossman and Rallis 2003] Rossman, G. B. and Rallis, S. F. (2003). *Learning in the Field: An Introduction to Qualitative Research*. Sage Publications, second edition edition.

[Runeson and Host 2009] Runeson, P. and Host, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164.

[Sato et al. 2008] Sato, D. T., Corbucci, H., and Bravo, M. V. (2008). Coding dojo: An environment for learning and sharing agile practices. *AGILE Conference*, 0:459–464.

[Steinberg 2001] Steinberg, D. H. (2001). The effect of unit tests on entry points, coupling and cohesion in an introductory java programming course. *XP Universe*.