Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers

Mauricio Finavaro Aniche, Marco Aurélio Gerosa
Department of Computer Science - University of São Paulo (USP) - Brazil
{aniche,gerosa}@ime.usp.br

Abstract

Test-driven development (TDD) is a software development practice that supposedly leads to better quality and fewer defects in code. TDD is a simple practice, but developers sometimes do not apply all the required steps correctly. This article presents some of the most common mistakes that programmers make when practicing TDD, identified by an online survey with 218 volunteer programmers. Some mistakes identified were: to forget the refactoring step, building complex test scenarios, and refactor another piece of code while working on a test. Some mistakes are frequently made by around 25% of programmers.

1. Introduction

Test-driven development (TDD) is an important practice in Extreme Programming (XP) [1]. As agile practices suggest, software design emerges as software grows. In order to respond very quickly to changes, a constant feedback is needed and TDD gives it by making programmers constantly write a small test that fails and then make it pass. TDD is considered an essential strategy in emergent design because when writing a test prior to code, programmers contemplate and decide not only the software interface (e.g. class/method names, parameters, return types, and exceptions thrown), but also on the software behavior (e.g. expected results given certain inputs) [13].

TDD is not only about test. It is about helping the team to understand the features that the users need and to deliver those features reliably and predictably. TDD turns testing into a design activity, as programmers use tests to clarify the expectations of what a piece of code should do [3].

Many other assumptions are made about TDD. Some researches show that it helps the development process by increasing code quality and reducing the number of defects, as presented in Section 2.

Kent Beck sums up TDD as follows: 1) quickly add a test; 2) run all tests and see the new one fail; 3) make a little change; 4) run all tests and see them all succeed; 5) refactor to remove duplication [18]. In order to get all benefits from TDD, programmers should follow each step. As an example, the second step states that programmers should watch the new test fail and the fifth step states to refactor the code to remove duplication. Sometimes programmers just do not perform all steps of Beck's description. Thus, the value TDD aggregates to software development process might be reduced.

This article presents some of the most common mistakes that programmers make during their TDD sessions, based on an online survey conducted during two weeks in January, 2010, with 218 volunteer programmers. The survey and its data can be found at http://www.ime.usp.br/~aniche/tdd-survey/.

This article is structured as follows: Section 2 presents some studies about the effects of TDD on software quality; Section 3 shows the most common mistakes programmers make based on the survey; Section 4 discusses about the mistakes and ideas on how to sort them out; Section 5 presents threats to validity on the results of this article; Section 6 concludes and provides suggestions for future works.

2. The effects of TDD on software quality

Empirical experiments about the effects of TDD have been conducted generally with two different groups: graduate students at universities and professional developers at the industry. Most of them show that TDD increases code quality, reduces the defect density, and provides better maintainability. However, industry studies presented stronger results indicating that TDD is more helpful.

2.1. Studies in an industry context

Janzen [5] 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 in debugging. Janzen also reported that computational complexity is much lower in test-first code while test volume and coverage are higher.

A study from Maximillien and Williams [6] showed a 40-50% reduction in defect density and minimal impact to productivity when programmers in industry were using TDD.

A study from Lui and Chan [7] comparing a group with TDD and another using the traditional test-last approach showed that TDD leads to a significant reduction in defect density. Moreover, defects that were still found were fixed faster with TDD. A study from Damm, Lundberg and Olson [8] showed significant defect reduction as well.

A study from George and Williams [9] found that although TDD might initially reduce productivity among inexperienced programmers, the produced code passed between 18% and 50% more in external test cases than the code produced by groups not using TDD. The code also presented a test coverage between 92% and 98%. A qualitative analysis showed that 87.5% of the programmers believed that TDD approach facilitated requirements understanding and 95.8% believed that it reduced debugging effort. 78% of them thought that TDD improved overall programming productivity; however only 50% of them believed that TDD led to less code development time. Regarding quality, 92% of the developers believed that TDD yielded higher quality code and 79% thought it promoted simpler design.

Nagappan [12] showed a case study in Microsoft and IBM and the results indicated that defect density of four products decreased between 40% and 90% relative to similar projects that did not use TDD. On the other hand, TDD increased in 15% to 35% the initial development time.

Langr [10] showed that TDD improved code quality, provided better maintainability, and produced 33% more tests.

2.2. Studies in an academic context

A study from Erdogmus et al [11] with 24 undergraduate students showed that TDD increased productivity. However no differences between quality effects in TDD code were found.

Another study from Janzen [13] 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.

Müller and Hagner [17] study showed that TDD has no quality and productivity effects. However, students noticed a better reuse in a TDD code.

Pancur [14] showed an experiment with 38 students in which they did not notice any quality effect or productivity improvement. In fact, students thought TDD was not a very effective practice.

Steinberg [15] showed that TDD code was more cohesive and less coupled and students reported that defects were easier to fix.

A study from Edwards [16] with 59 students showed that TDD code has 45% fewer defects and gives a higher programmer confidence.

2.3. Threats to Validity

Several empirical researches investigate the effects of TDD on software quality. However, they do not evaluate if programmers are applying TDD correctly, which might be a factor of influence and might affect the result of the study. As explained in Section 3, mistakes during TDD practice may decrease code and software quality.

Indeed, there is a lack of literature about common mistakes that programmers may incur in while practicing TDD.

3. Common Mistakes in TDD

TDD is theoretically a simple technique since it only has few steps to be followed. However, in practice the steps are not that easy to follow as programmers need to be very disciplined. This might reflect why programmers are induced to make some mistakes, which might lead code to a poor quality and/or unexpected behaviors.

In order to identify common mistakes that programmers are aware to be making, an online survey was conducted during two weeks in January, 2010. The survey was announced in several discussion lists [27] [28] [29] [30] and in the micro-blog Twitter [31].

The survey was conducted with 218 volunteer programmers in order to evaluate their experience, feelings, and which mistakes are more common while practicing TDD. All questions about mistakes were elaborated based on empirical observation of mistakes programmers usually make. Hence, it was focused only on problems related to TDD approach and not on mistakes programmers do when writing unit tests before or after the implementation.

For each question, programmers could choose a number from zero to five, in which zero meant "never" and five meant "always." During the analysis, intermediate numbers received a meaningful term in order to better communicate the results. The terms are respectively, from zero to five: "never," "rarely," "sometimes," "regularly," "frequently," "always." Moreover, in order to make the same comparison, the complement of each answer in sections 3.9, 3.5 and 3.8 were analyzed.

As illustrated in Figure 1, almost 75% of programmers were practicing TDD for at most 3 years and only 22% were doing for more than 4 years.

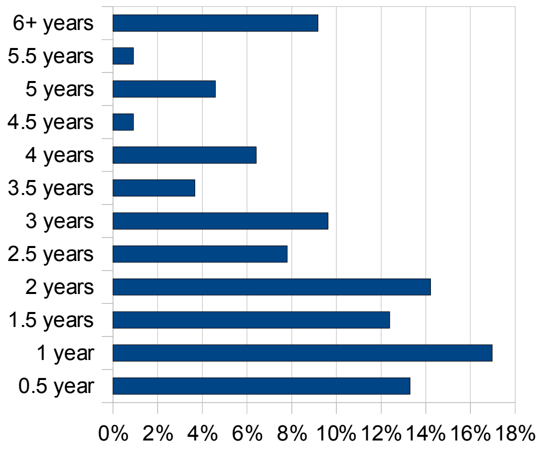

Figure 1. Programmers' experience in TDD

When asked to evaluate their first time using TDD, in a scale from 0 to 5, the average answer was 2.36 and the standard deviation was 1.4. In addition, 50% of programmers chose between 0 and 2 and only 7% thought they did well since the beginning and chose option 5.

Programmers were also inquired about where they practice TDD and available options were: academy, industry, and open source projects. They were able to choose more than one. As illustrated in Figure 2, 90% of programmers use TDD in industry, 50% in open source projects, and only 20% in academy.

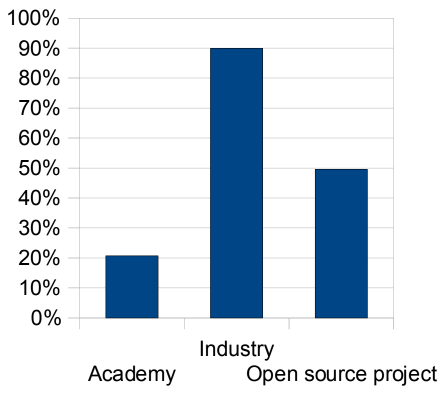

Figure 2. Where programmers practice TDD

Each sub-section below presents a possible mistake programmers might do. All charts show the distribution of the mistakes split in three different ranges, each one representing an interval of years of experience in TDD (at most 2 years, between 2 and 4 years, more than 4 years). The X-axis represents the frequency in which programmers make the mistake and the Y-axis represents the percentage of programmers for each range that makes it.

3.1. Do not watch the test fail

As previously presented, the second TDD step states to watch the new test fail. At first, programmers may think it is an unnecessary step as they just wrote the test, so they know it is supposed to fail. Hence, they skip this step and go directly to the next one, implementing the simplest thing that makes the test pass. This approach may guide the programmer to unexpected errors.

If a new test does not fail, programmers receive an indication that the production code was not working as they thought it was and a code revision might be necessary. Another problem that might occur is that programmers cannot be sure about what made the test pass; nothing ensures the new code was actually responsible for it. The test implementation might have been wrong since the beginning.

In all cases presented, if the programmer had run the test before, s/he would have caught the problem just by noticing that the test was not actually failing.

The survey showed that 55% of programmers declared to make this mistake very rarely or never make it. However, as illustrated in Figure 3, 24% of the programmers forget to watch the test fail regularly or frequently and almost 4% always forget to watch it fail, while 15% never forget to do it before starting to code. Moreover, the average frequency of errors was 1.75, which indicates that programmers in general make this mistake rarely or sometimes. The standard deviation was 1.35.

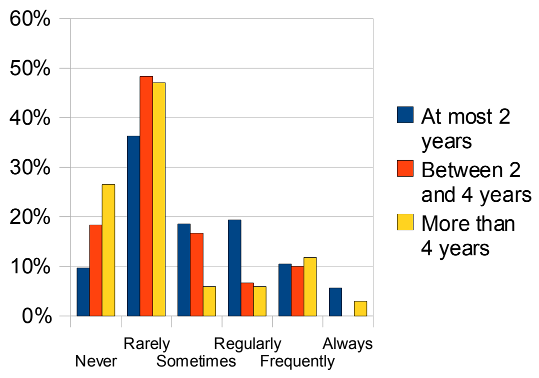

Figure 3. How often programmers forget to watch the test fail

The result of Pearson correlation between how often programmers forget to watch the test fail and years of experience in TDD was approximately -0.22. It indicates that experience is a small factor of influence. Thus, Figure 3 shows that the more experienced the programmer is, the less s/he makes this mistake. However, the chart also shows that around 15% of programmers with more than 4 years of experience forget to watch the test fail frequently or always, which might indicate that experienced programmers are too confident with the process and skip this test.

3.2. Forget the refactoring step

Refactoring is the process of improving the internal structure by editing the existing working code, without changing its external behavior [19]. It is the fifth and a fundamental step in TDD process, because the simplest code that the programmer has written in the previous step is not always the best possible clean code. It also prevents code from being scattered over time.

Ron Jeffries says that TDD is about "clean code that works" and it gets done in two phases: The simplest code step takes care of the "work code" part while the refactoring step takes care of the "clean code" part [18].

When asked about how often they forget to apply a refactoring after the green bar, the average response was 2.37 in a scale of 0 to 5, indicating that programmers in general forget to refactor code almost regularly. The standard deviation was 1.17. About 1% always forget, while only 5% never forget the refactoring step. The chart in Figure 4 shows that experienced programmers tend to forget it a bit less than beginners do. However, 44% of the experienced programmers forget it regularly or frequently, while 52% of beginners forget to do it regularly or frequently. It might indicate that programmers in general think that the newly produced code is good enough and it does not need to be refactored, which might not be always true. Nevertheless, Pearson correlation between how often programmers forget the refactoring step and years of experience in TDD was approximately -0.03, indicating that experience is not a factor of influence.

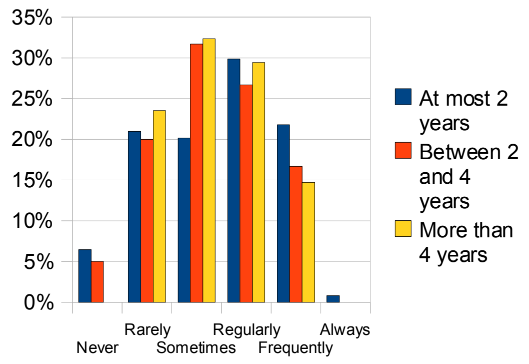

Figure 4. How often programmers forget the refactoring step

Another hypothesis on why programmers forget to refactor may be a psychological factor: when programmers make the test pass they get excited about it and go directly to the next test, forgetting the refactor step. Further investigation is necessary to evaluate this factor.

3.3. Refactor some other piece of code while working on a test

When programmers are trying to achieve a code that makes the test pass, they usually navigate through a code that was written before and as they start reading legacy code they may feel a need to refactor some part of it. TDD states that if there is a failing test, the first thing is to make it pass and do some refactor only after the green bar.

When asked about refactoring some other piece of code while working on a test, the average response was 2.34, indicating that programmers in general do it almost regularly. The standard deviation was 1.36. Almost 40% of programmers do some refactoring while a test is failing regularly or frequently, 5% do it all the time and only 6% of them make the test pass first and then refactor the legacy piece of code. Figure 5 shows that, differently from what would be expected, 38% of the experienced programmers refactor another piece of code regularly or frequently. However, 44% of them never do it or do it very rarely. None of them do it all the time. Although 9% of beginner programmers make this mistake all the time, 28% never do it or do it very rarely. The result of Pearson correlation between how often programmers refactor some other piece of code while working on a test and years of experience in TDD was approximately -0.13, indicating that years of experience in TDD is a small factor of influence.

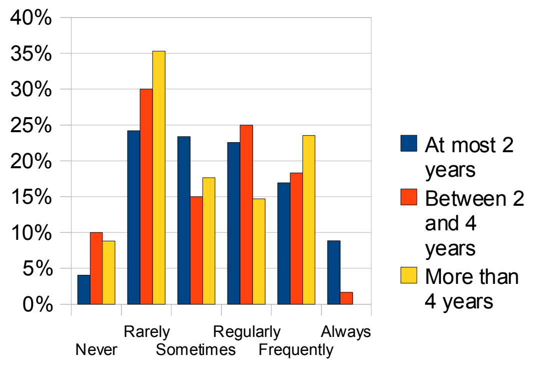

Figure 5. How often programmers refactor some other piece of code while working on a test

Programmers should only refactor code when all tests are passing and never when any test is failing. This way he may detect what part of the refactoring has broken the code and fix it. If programmers find a piece of legacy code that needs to be refactored they need to make the test pass first, make sure all tests in the suite are passing and then refactor the legacy code.

3.4. Use bad test names

Programmers spend more time reading code than producing code [22]. Therefore, all code should be clear enough to be understood very easily. This is valid for test code as well, because TDD programmers spend much time reading test code.

It is a common practice to read all test names before implementing a new feature: programmers get more confident in it. If the test name is not good, then its programmer will need to spend time reading the test code implementation (which might be a bit complex) instead of doing something more valuable. However, if a test has a good and legible name, the programmer will understand what that test does without reading its implementation.

Having understandable test names also makes the use of the test suite as documentation feasible since each test name describes a feature in the system. Putting tests all together, they might become an informal documentation (or even the formal one) and it can be read, as an example, by new programmers in the team or by the product owner in order to know what features are currently implemented and tested.

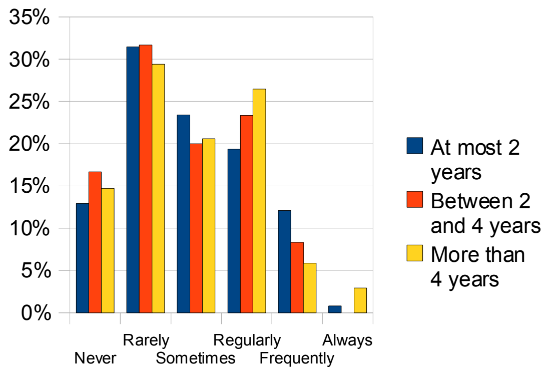

Figure 6. How often programmers write bad test names

When asked about bad test names, the average response was 1.84, indicating that programmers in general recognize to write bad test names rarely or sometimes. The standard deviation was 1.25. On the other hand, as illustrated in Figure 6, 32% of the programmers affirm that they write a bad test name regularly or frequently, and 1% affirm that they write bad test names all the time, when only 14% affirm that they never write a bad test name. Therefore, around 45% of programmers never write a bad test name or do it very rarely. Pearson correlation between writing bad tests name and years of experience in TDD was -0.02 indicating that experience time in TDD is not a factor of influence.

3.5. Do not start from the simplest test

Each new feature in a program is commonly compound by many requirements and to accomplish it in a TDD way, the programmer writes more than one test. As Freeman suggests, the best way is to start testing the simplest success case [3]. Once this test is working, the programmer has a better idea of the real structure of the solution and become more confident in that code.

Programmers were asked if they start from the simplest possible test. In order to compare with other mistakes, the complement of each answer was analyzed. The average frequency was 2.0, indicating that programmers do not start from the simplest test sometimes. The standard deviation was 1.28. As illustrated in Figure 7, 33% of the programmers said that they do not start from the simplest possible test regularly or frequently, 10% always start from the simplest ones and only 2% never do it.

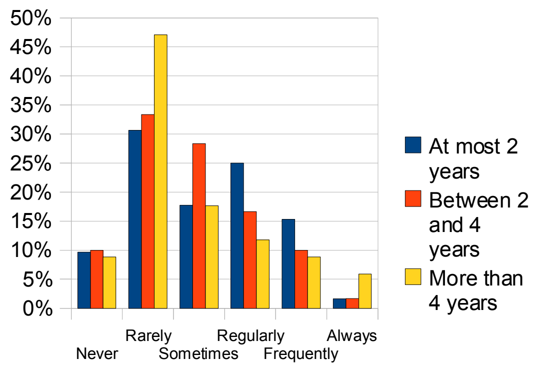

Figure 7. How often programmers start from the simplest possible test

The chart also shows that beginner programmers do it more frequently than experienced (around 20% of experienced and 40% of beginners do it regularly or frequently). It indicates that experienced programmers can evaluate better whether the test is the simplest one or not, while beginners still do not have the required experience for it. However, Pearson correlation between starting from the simplest test and years of experience in TDD was -0.09, indicating that experience is not a factor of influence.

3.6. Run only the current failing test

Automatic tests should be run after each change of the application code in order to assure that the changes have not introduced errors to the previous version of the code [19]. Test suites can become really big and as they grow, the time needed to run the whole suite increases.

When programmers write a failing test and start making it pass, they should always run the all test suite as the code they are writing might affect another part of the system and break some other test. If programmers only run the actual failing test, they will not notice when an old test breaks.

Running all the test suite only at the end of the process is not a final solution for the problem. If programmers finish the implementation and only after that they see that an old test just broke, it could take more time to find the problem because too many lines of code have been written.

In the survey, when asked how often they forget to run the all test suite, the average frequency of response was 1.4, indicating that programmers in general run the all suite rarely or sometimes. The standard deviation was 1.2. Moreover, 16% of programmers forget to run all tests regularly or frequently, 2% forget all the time while only 25% never forget.

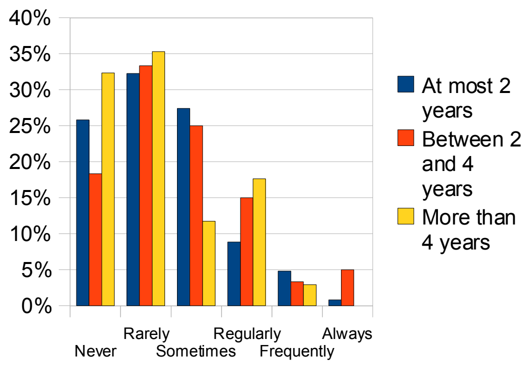

Figure 8. How often programmers forget to run the complete test suite

Figure 8 shows that, differently from what would be expected, experienced programmers tend to forget to run more frequently the complete suite more often than beginners (21% of experienced programmers forget it regularly or frequently, compared to only 14% of the beginners). Hence, the Pearson correlation was -0.009, indicating that years of experience in TDD is not a factor of influence.

3.7. The need for writing a complex test scenario

A test case is usually written for a tiny piece of functionality and the code that makes the test pass should not be too long. When programmers are forced to write a large amount of lines of code just to make one test, it might indicate that the class being tested contains too many responsibilities and should be refactored (maybe dividing it in two or more classes).

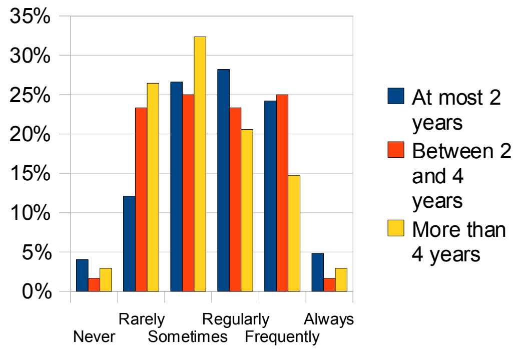

Figure 9. How often programmers need to write a complex test scenario

When asked about how often programmers need to write a complex test scenario, the average response was 2.58, indicating that programmers in general do it almost regularly. The standard deviation was 1.21. In addition, 50% of programmers write complex tests regularly or frequently, 4% write them all the time, and only 3% never write a complex test. The chart in Figure 9 shows that 35% of experienced programmers need to write a complex test scenario regularly or frequently, 27% do it very rarely while almost 53% of beginner programmers need to do it regularly or frequently, and only 12% do it rarely. Thus, Pearson correlation was -0.10, indicating that years of experience in TDD is a small factor of influence.

Programmers should be always aware of it. As soon as the first complex test needs to be written, they should refactor it immediately; otherwise they will be forced to write many complex tests that probably need too much time to be written and probably need too much effort to write code that makes it pass. Later, it might be more difficult to make a big refactoring.

3.8. Do not refactor the test code

As previously mentioned, programmers spend more time reading code than producing it, and TDD programmers spend time reading two different types of code: production code and test code. Because of that, test code should be as clear as possible, and in order to achieve that it needs to be constantly refactored.

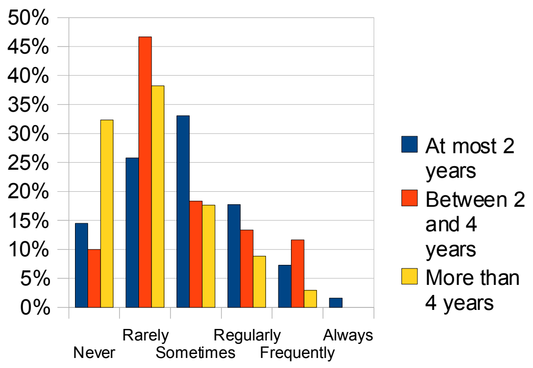

Figure 10. How often programmers apply a refactoring on a test code

In the survey, programmers were asked about how often they refactor test code. In order to compare with other mistakes, the complement of each answer was analyzed. The average response was 1.67, indicating that programmers in general forget to do it almost sometimes. The standard deviation was 1.19. Moreover, as illustrated in Figure 10, only 16% of programmers never forget to refactor test code while 23% of programmers forget to do it regularly or frequently, and 1% always forget to refactor. In addition, Pearson correlation was -0.21, indicating that years of experience in TDD is a small factor of influence and makes difference. Hence, beginner programmers tend to forget to refactor the test code more often than experienced (25% of beginners forget it regularly or frequently while only 12% of experienced programmers do it).

Refactoring is not the only activity programmers should do to keep the tests clear. It should also be updated together with the production code: if a feature is removed from the production code, its test code should be deleted together; if a feature needs to be changed, its tests need to be changed as well. The survey shows that 26% of programmers find test for features that does not exist anymore regularly or frequently, 2% find them all the time, and only 18% never find them.

3.9. Do not implement the simplest thing that makes the test pass

The third step of TDD states that programmers should do the simplest thing that makes the test pass. When programmers do not follow this rule they might be creating unnecessary complex code, and as a consequence, decreasing code quality.

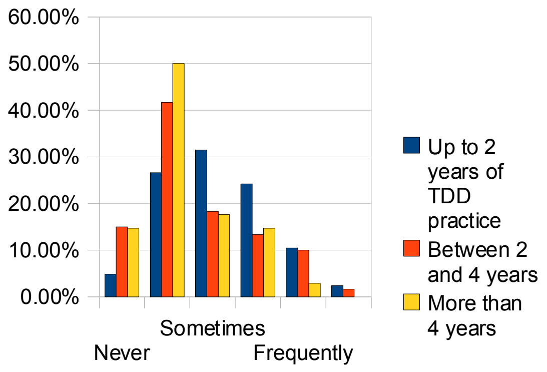

Figure 11. How often programmers do not implement the simplest thing that makes test pass

Programmers were asked about how often they implement the simplest thing that makes the test pass. In order to compare with other mistakes, the complement of each answer was analyzed. The average response was 1.90, indicating that programmers do not implement the simplest thing sometimes. The standard deviation was 1.19. Pearson correlation was -0.24, indicating that experience is a small factor of influence. As illustrated in Figure 11, 35% of beginners tend not to implement the simplest thing regularly or frequently while 20% of experienced programmers do not do it.

4. Discussion

Table 1 summarizes all mistakes programmers make, ordered by the most frequent mistake in average.

Mistake	Avg/Std. Dev.	Pearson correl.	Frequently or always
The need for writing complex test scenario [3.7]	2.58/1.21	-0.1	26.61%
Forget the refactoring step [3.2]	2.37/1.17	-0.03	19.72%
Refactor other piece of code while working on a test [3.3]	2.34/1.36	-0.13	23.85%
Do not start from the simplest test [3.5]	2.00/1.28	-0.09	15.14%
Do not implement the simplest thig that make the test pass [3.9]	1.90/1.19	-0.24	11,01%
Use bad test names [3.4]	1.84/1.25	-0.02	11.01%
Do not watch the test fail [3.1]	1.75/1.35	-0.22	14.22%
Do not refactor the test code [3.8]	1.67/1.19	-0.21	8.72%
Run only the current failing test [3.6]	1.40/1.20	-0.01	5.96%

Table 1. Most Common Mistakes when practicing Test-Driven Development

As previously mentioned, 75% of programmers were practicing TDD for at most 3 years, which might indicate that this technique is still recent to most of programmers. This might explain the frequency of mistakes that are made by programmers.

The average response when programmers were asked to evaluate their first time doing TDD was not high, indicating that TDD is not easy to understand and looks like a non-natural way to develop a software to many programmers.

The mistakes presented in this article may be avoided if programmers follow all TDD steps correctly. Always refactoring code after the green bar, for example, prevents code from a big refactoring need in a long term.

The act of choosing good and understandable test names help programmers to avoid spending much time reading test and production code. Programmers may check if a test has a good name paying attention after finishing its implementation: if the programmer gets to know what the test should do and what the expectations are by just reading the test name, then it is a valid test name.

Thus, keeping test code clean is also a good practice, as programmers might need to read some old tests during their activities. There are some techniques in order to make it clean; Test Data Builders [3], which are simple implementations of Builder Pattern [24], are helpful when there is a need to build instances of complex objects for a specific test scenario. With it, all lines used to create the object are now replaced by the test data builder. Many other ways to improve test code can be found in literature [23].

Starting from the simplest test is a way to make programmers more confident about code and the software needs. As Beck [18] and Freeman [3] suggests, a good way to start from the simplest is to keep a list with the features that should be tested in a piece of paper. The list helps programmers on deciding the next simplest test that should be implemented.

The need for writing complex test scenarios might indicate some design smell in production code such as high coupling, as the programmer needs to code too many test lines just to test one feature. Therefore, in order to avoid that, some known design patterns, as Strategy, State, Observer [24] may come in handy. The utilization of them help programmers to reduce the coupling and class responsibilities, making it possible to test without the need to build a complex scenario.

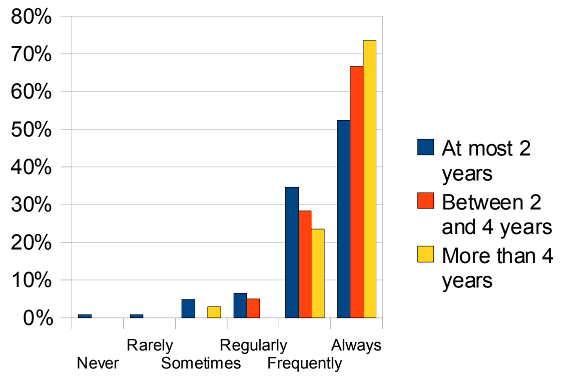

Figure 12. Programmers' opinion about TDD reducing defect density

Although all programmers declared to make mistakes, Figure 12 shows that almost 60% of volunteers are really convinced that TDD helps reducing defect density and only 0.50% of programmers are convinced that TDD does not help in reducing defects at all (average was 4.44 and standard deviation was 0.83). In addition, Pearson correlation was 0.20, indicating that years of experience is a small factor of influence. No programmers with more than 2 years of experience thought that TDD rarely helps or does not help at all.

Their opinions about code quality improvement were almost the same: 65% were convinced that TDD produces better code and 0% of programmers thought that TDD does not help at all (average was 4.46 and standard deviation was 0.88). Hence, Pearson correlation was 0.21, indicating that years of experience is a small influence. In addition, Figure 13 shows that 13% of beginners considered that TDD improves code quality only sometimes or regularly, which might indicate that TDD is not an easy practice.

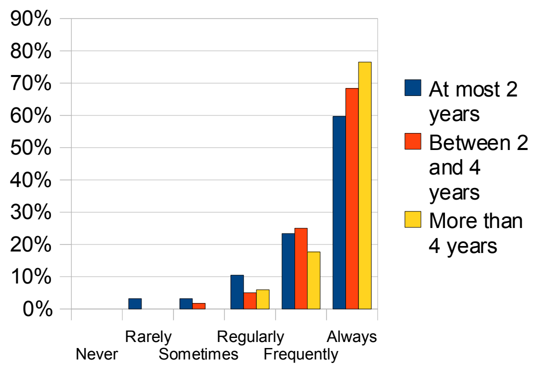

Figure 13. Programmers' opinion about TDD improving code quality

Different from what would be expected, programmers with more than 4 years of experience sometimes make more mistakes than programmers with at most 2 years of experience. It might happen because, as suggested by Dreyfus Model [26], when experienced professionals get more confident with the technique, sometimes they start skipping some steps, and it might lead them to make mistakes. Another possible explanation is that experienced programmers usually tend to be more self-critic about their practices and might choose lower values in the survey scale than reality. On the other hand, beginners in general often oversell their skills as they are still not able to evaluate their technique, so it is possible that the percentage of mistakes this article shows in the beginners' range is underestimated.

5. Threats to Validity

The main threats to validity are:

- The most common mistakes were based on programmers' self-evaluation and as previously discussed, it might not represent the reality.
- All mistakes were raised by empirical observation in industry and there might be many other possible mistakes programmers make while practicing TDD.
- As presented in Figure 1, the number of programmers with more than 4 years of experience is only 16% of total and the sample may not be representative.
- Programmers were only categorized by years of experience in TDD. As studies in Section 2 shows some difference between opinions in academy and in industry, it might be a factor of influence and might affect the results of this study.

6. Conclusions and Future Work

Although TDD is increasingly becoming more popular, the number of mistakes programmers still make is high. In the analysis, “regularly”, “frequently,” and “always” were sometimes grouped, as they might indicate the number of programmers that make that mistake very often. As presented in this article, some mistakes are made frequently or always by around 25% of programmers.

The analysis shows that years of experience in TDD is only a small factor of influence, at least statistically. A more complete investigation is necessary to evaluate this factor of influence.

Deviations in TDD practice were considered mistakes and were hypothetically connected to defects. A study should be done in order to check whether all these mistakes really reduce TDD benefits.

TDD leads code to better quality and promotes a reduction in defect density. All advantages TDD provides might increase if programmers make fewer mistakes during the process. All studies presented in Section 2 regarding effects of TDD on software quality might be affected by all mistakes programmers affirm to make. A future step of this research is to conduct an experiment isolating this factor of influence and checking whether these mistakes reduce the aggregated value of TDD on software development process or not.

7. References

[1] Beck, K., *Extreme Programming Explained, Second Edition: Embrace Change*. Boston, Massachusetts, USA, Addison-Wesley, 2004.

[2] Beck, K., Beedle, M., et al., *Manifesto for Agile Software Development*, 01.12.2010, http://www.agilemanifesto.org.

[3] Freeman, S., Pryce, N., *Growing Object-Oriented Software, Guided by Tests*. First edition, Addison-Wesley Professional, 2009.

[4] Siniaalto, M. *Test-Driven Development: Empirical Body of Evidence*. Technical report, ITEA, Information Technology for European Advancement, 2006.

[5] Janzen, D., *Software Architecture Improvement through Test-Driven Development*. Conference on Object Oriented Programming Systems Languages and Applications, ACM, 2005.

[6] Maximilien, E. M. and L. Williams. *Assessing test-driven development at IBM*. IEEE 25th International Conference on Software Engineering, Portland, Orlando, USA, IEEE Computer Society, 2003.

[7] Lui, K. M. and K. C. C. Chan. *Test-driven development and software process improvement in China*. 5th International Conference XP 2004, Garmisch-Partenkirchen, Germany, Springer-Verlag, 2004.

[8] Damn, L.-O., Lundberg, L., et al. *Introducing Test Automation and Test-Driven Development: An Experience Report.* Electronic Notes in Theoretical Computer Science 116: 3 – 15, 2005.

[9] George, B., Williams, L., *An Initial Investigation of Test-Driven Development in Industry*. ACM Symposium on Applied Computing. Melbourne, Florida, USA, 2003.

[10] Langr, J., *Evolution of Test and Code Via Test-First Design*, 02.12.2010, http://www.objectmentor.com/resources/articles/tfd.pdf

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

[12] Nagappan, N., Bhat, T. *Evaluating the efficacy of test-driven development: industrial case studies*. Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering.

[13] Janzen, D., Saiedian, H. *On the Influence of Test-Driven Development on Software Design*. Proceedings of the 19th Conference on Software Engineering Education & Training (CSEET'06).

[14] Pancur, M., Ciglaric, M., et al. *Towards Empirical Evaluation of Test-Driven Development in a University Environment*. EUROCON 2003, Ljubljana, Slovenia, IEEE.

[15] Steinberg, D. H. *The Effect of Unit Tests on Entry Points, Coupling and Cohesion in an Introductory Java Programming Course*. XP Universe, Raleigh, North Carolina, USA, 2001.

[16] Edwards, S. H. *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, Orlando, Florida, USA, 2003.

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

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

[19] Astels, D. *Test-Driven Development: A Practical Guide*. Upper Saddle River, New Jersey, USA, Prentice Hall, 2003.

[20] Kerievsky, J. *Refactoring to Patterns*. Addison-Wesley Professional, 2004.

[21] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts D. *Refactoring: Improving the Design of the Existing Code*. Addison-Wesley Professional, 1999.

[22] Begel, A., Simon, B. *Struggles of New College Graduates in Their First Software Development Job*. SIGCSE Bulletin, 40, n° 1, 226-230, ACM, 2008.

[23] Meszaros, G. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional, 2007.

[24] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[25] Beck, K. *Aim, fire*. IEEE Software 18, page 87-89, 2001.

[26] Benner, P. *From novice to expert*. The American Journal of Nursing, 1982.

[27] Test Driven Development Discussion List.Yahoo! Groups, 07.01.2010. http://tech.groups.yahoo.com/group/testdrivendevelopment/.

[28] Agile Testing Discussion List. Yahoo! Groups, 07.01.2010. http://tech.groups.yahoo.com/group/agile-testing/.

[29] Alt.NET Discussion List. Yahoo! Groups, 07.01.2010. http://tech.groups.yahoo.com/group/altdotnet/.

[30] .NET Architects Brazilian Discussion List. Google Groups, 07.01.2010. http://www.dotnetarchitects.net/.

[31] Microblog. Twitter, 07.01.2010. http://twitter.com/mauricioaniche/status/7493800359.