Aniche and Gerosa *Journal of the Brazilian Computer Society* (2015) 21:15
DOI 10.1186/s13173-015-0034-z

Journal of the Brazilian Computer Society
a SpringerOpen Journal

RESEARCH Open Access

Does test-driven development improve class design? A qualitative study on developers' perceptions

Maurício Aniche* and Marco Aurélio Gerosa

Abstract

Background: Developers commonly affirm that writing unit tests improve internal quality of a software, besides a more obvious effect on external quality. This is particularly common among Test-Driven Development (TDD) pactitioners, who leverage the acting of writing tests to think about and improve class design. However, it is not clear how this effect occurs.

Methods: This study investigates the developers' perceptions on how the practice of TDD influences class design, by means of a qualitative study with 25 participants from 6 different companies in Brazil. In this paper, we share their opinions and experience.

Results: According to them, the practice of test-driven development does not drive directly the design, but gives them a safe space to think, the opportunity to refactor constantly, and subtle feedback given by unit tests, are responsible to improve the class design.

Conclusions: We suggest developers to experiment the practice of TDD, as its effects look positive to software developers. As future work, tools may be developed to automatically warn developers about classes that have testability problems, or even to suggest them to practice TDD in specific parts of the code.

Keywords: Unit testing; Test-driven development; Agile methodologies; Class design

Introduction

Writing automated unit tests is a popular practice in software development. Many developers write automated tests before the actual code is tested, as suggested by the test-driven development (TDD) practice [1]. Scott Ambler shows that 53 % of the respondents of a survey adopted TDD [2]. Similar numbers can be found in the annual surveys from Version One, which, in the 2012 version [3], showed that 40 % of the respondent teams used the practice.

Practitioners commonly argue that TDD helps software design, improving internal code quality. For example, Kent Beck [4], Robert Martin [5], Steve Freeman [6], and Dave Astels [7], state in their books (without scientific evidence) that the writing of unit tests in a TDD fashion promotes significant improvement in class design, helping developers to create simpler, more cohesive, and less coupled classes. They even consider TDD as a class design technique [8, 9].

Nevertheless, the way in which TDD actually guides developers during class design is not yet clear. We observed this phenomenon in a preliminary study with TDD practitioners [10]. We interviewed ten practitioners and none of them could satisfactorily explain how the practice guides them to better design, although they agreed that it indeed improves design. Siniaalto and Abrahamsson [11] also noticed that the effects of TDD are not as direct as expected by most people. Most studies reported in the scientific literature, discussed in Section "Background", evaluated whether the practice of TDD or unit testing makes a difference in the code being produced, but very few of them aimed to understand how the practice makes that difference.

*Correspondence: aniche@ime.usp.br
University of São Paulo, Rua do Matão, 1010, São Paulo, Brazil

This study aims to investigate how the practice of test-driven development influences class design from the point of view of practitioners. As our research question is related to a complex cognitive process (software design), which is heavily influenced by context, we conducted an essentially qualitative study. Professional developers were invited to solve design problems practicing TDD. Based on the gathered data and on semi-structured interviews, we collected their point of view of how the practice influenced their class design decisions.

Background

Most studies investigating the effects of unit testing and TDD on production code focused on external quality (e.g., bug proneness). Munir et al. [12], in a systematic review, show that high rigor studies tend to show an improvement in external quality, with a sense of less productivity. However, these studies only investigate a small set of variables. Studies about internal code quality were inconclusive or not evaluated. In this section, we present studies focusing on the quality of the design and other aspects of internal quality. It is worth to notice that many empirical studies that evaluated the effects of unit testing in class design relied on the TDD technique, as it emphasizes the writing of unit tests and their use to reflect upon the design.

Janzen [13] showed that the algorithms complexity was much smaller, and the code coverage was higher in the code written using TDD. Another study by Janzen and Saiedian [14], with three different groups of students, each one using a different approach: TDD, test last, and no tests, showed that the code produced using TDD made better use of object-oriented concepts, and responsibilities were better distributed into different classes, while other teams produced a more procedural code. In addition, tested classes were 104 % less coupled than non-tested classes, and methods were 43 %, on average, less complex than the non-tested ones.

George and Williams [15] showed that, although TDD can initially reduce the productivity of inexperienced developers, 92 % of the developers in a qualitative analysis thought that TDD helped to improve code quality. Seventy nine percent believed that the practice promotes a simpler class design.

A study by Erdogmus et al. [16] with 24 undergraduate students showed that TDD increased productivity. However, no difference in code quality was found. Langr [17], however, showed that TDD increased code quality, facilitated maintenance, and helped to produce 33 % more tests when compared to traditional approaches.

Dogsa and Batic [18] also found an improvement in class design when using TDD. According to the authors, the improvement was a consequence of the simplicity TDD adds to the process. They also affirmed that the test suites created during the practice favors constant code refactoring.

Li [19] conducted a case study in which she collected the perceptions of TDD practitioners about the benefits of the practice. She interviewed five developers from software companies in New Zealand. The results of the interviews were analyzed and discussed in terms of code quality, software quality, and programmer productivity. Regarding code quality, Li concluded that TDD guides developers to simpler and better-designed classes. In addition, the main factors contributing to these benefits are the confidence to refactor and modify the code, a higher code coverage, a deeper understanding of the requirements, a code easier to understand, and the elevated satisfaction of the developers.

TDD practitioners usually make use of other agile practices, such as pair programming. This makes the evaluation of the practice more difficult. Madeyski [20] observed the results of groups practicing TDD, groups practicing pair programming, and the combination of them, and he was not able to show a significant difference between teams using TDD and teams using pair programming in terms of class package dependency management. However, when combining the results, he found that TDD helps to manage dependencies at class level.

Muller and Hagner [21] showed that TDD does not result in better quality or productivity. Steinberg [22] showed that code produced with TDD is more cohesive and less coupled. Participants also reported that defects were easier to fix.

There is also some work on test code best practices. In his book, Meszaros [23] enumerates a few code smells. As an example, he mentions a pattern called *Obscure Test*, which is a test difficult to understand at a glance. The *Hard-To-Test Code*, which is a code difficult to test, according to him, may happen because of a highly coupled code. He mentions that a possible solution to this problem is to make use of stubs or mock objects. He also proposes the *Test Code Duplication* pattern, which is the same test code repeated many times. Meszaros says it can happen because of clone code reuse but, as we discussed, it can also happen because of a bad abstraction in the production code. We reemphasize that there is a strong connection between the quality of the test code and the quality of the production code being tested.

Discussion

Although some studies revealed that TDD positively influences software design and that developers are aware of this effect, to the best of our knowledge, no study focused on investigating how this influence actually occurs. Josefsson [24], in his discussion about the need

for an architectural phase and the effects of TDD in this matter, came to the same conclusion. According to him, studies about TDD in the current literature are very limited and, as a result, the effects of the practice in class design cannot be properly explained. Besides, most of the studies found were conducted with students in academic settings. Given the complexities of software design, the results may be different in profession settings, with experienced developers. Janzen [25], in his Ph.D., perceived that mature developers obtain more benefits from the practice, by writing simpler classes. Mature developers also tend to choose TDD more often than less experienced ones, who supposedly would receive greater benefits from the guidance provided by the technique.

In addition, studies that analyze TDD from the class design point of view do not come to clear explanations; many of them affirm that the outcomes of teams that practiced TDD were not so different from those teams that do not practice it. Even Janzen's Ph.D. thesis [25] was inconclusive regarding the influence of the practice in coupling and cohesion.

Another limitation of some studies is that many aspects of class design are hard to evaluate quantitatively, such as simplicity and evolvability. Many of them also have no scientific rigor [26]. Developers' point of view and qualitative inspection of the code should also be taken into account.

Methods

Investigating a highly complex cognitive process, like software design, involves many human factors and context-dependent variables. Qualitative research aims to explore and understand a phenomenon considering individual, social, and environmental influences, providing a trade-off between realism and control [27]. Data are usually gathered from the point of view of the participants, and the analysis is carried out inductively, from a very specific to a general theme [28].

As discussed in Section "Background", few studies evaluated the effects of TDD and its feedback on the quality of software design. Some of them found that the practice improves class design, resulting in less coupling, higher cohesion, and more simplicity. However, none of them focused on understanding how the unit tests produced during TDD sessions guide developers through these improvements, which is the goal of this study.

To achieve this goal, we conducted an essentially qualitative study with professional developers. We invited participants to implement a set of pre-prepared exercises. After that, we interviewed them about how the practice influenced their class design decisions. Then, we analyzed their answers using coding techniques. This section details the planning as well as the data analysis procedures.

Research question

The main goal of this study was to understand the relation between the practice of TDD and the class design decisions made by the developer. We raise the following question:

RQ. What are the developers' perceptions on the effects of Test-Driven Development in class design?

Research design

Developers from different software development companies from Brazil were invited to take part in this study. Before taking part in the study, all participants agreed with the research procedures and signed a consent form. Their profiles are discussed in sub-section Participants' profile. They implemented two problems using Java in a defined timeframe. The participants practiced TDD for one problem and were oriented to do not write tests for the other so they could later compare the two approaches. The problems and the order of practicing or not TDD were randomized to possibly reduce bias.

All participants worked on the exercises in their offices, where they were able to use their own computer and development IDE. We were there to explain the procedures of the study and to answer questions. At the end of the exercise, all participants filled out a survey, with questions about the produced code, design decisions, and challenges, and how (and if) TDD helped them. We saved all implementations for further analysis. The survey is available on the Internet[1].

After that, we selected participants for interviewing. The interviews were semi-structured. All questions were open, giving the opportunity to the participants to speak more deeply about the topic. We made specific questions for each participant according to the code they produced and their answers on the survey. Since many different factors may influence the decisions made by a developer during class design, the questions were designed to encourage participants to triangulate their answers and to isolate the practice of TDD from other possible factors of influence.

All interviews were recorded and transcribed. We also took notes about participants' reactions to specific questions. The interviews were conducted according to the availability of the participants.

Participants' profile

We invited developers from the Brazilian software development industry to be part of this study. The only requirement was that the participant should already know how to write unit tests and Java code.

All participants filled out a survey before the start of the study. This survey had objective questions about their

expertise and open questions in which participants could describe their experiences in object-oriented systems, Java, and TDD in a more detailed way.

We had 25 participants from 6 different companies. In Fig. 1, we present the TDD experience among participants. Ten said they had been using TDD for no more than a year, 13 had been practicing TDD for 1 to 3 years, and one had been practicing TDD for 3 to 4 years. Only one participant had never practiced TDD. This diversity was an asset, as it was possible to gather information from people with different levels of experience. Regardless of their level of experience, all of them affirmed they tried to practice their best TDD in the study.

The numbers were a little different regarding their experience in software development. In Fig. 2, we show the distribution. Five of them had been developing software for less than 2 years, 6 for the last 4 or 5 years, and 7 from 6 to 10 years. Sixteen of the participants worked professionally with the Java language and 9 knew Java but did not use it in their daily work. All of them knew JUnit, and 16 used mock objects[2] during their development activities. Only three had never heard about mock objects. When talking about object-orientation, in the open question, most of the participants affirmed they had a good experience.

The elevated number of participants who did not use Java in their daily work is a consequence of a company that uses PHP. However, we ensured that, although they did not use Java frequently, they did not have troubles with the language during the implementation of the exercises.

Proposed problems

We proposed four problems to be implemented by the participants[3]. The goal was to simulate real and recurrent class design problems. They contained complicated design problems that if solved naively would lead to solutions with high coupling, low cohesion, and no encapsulation.

To characterize class design quality, we used the well-known SOLID principles, from Martin [29]. He enumerated a few symptoms of problematic class design, also known as *class design smells*. They are similar to *code smells* but at a higher level.

Each principle is related to a good object-oriented practice. The Single Responsibility Principle (SRP) argues that a class should have only one reason to change; in other words, the class must be cohesive. The Open-Closed Principle (OCP) states that a class should be open to extension, but closed to modification. The Liskov Substitutive Principle (LSP) is related to good use of class inheritance, which

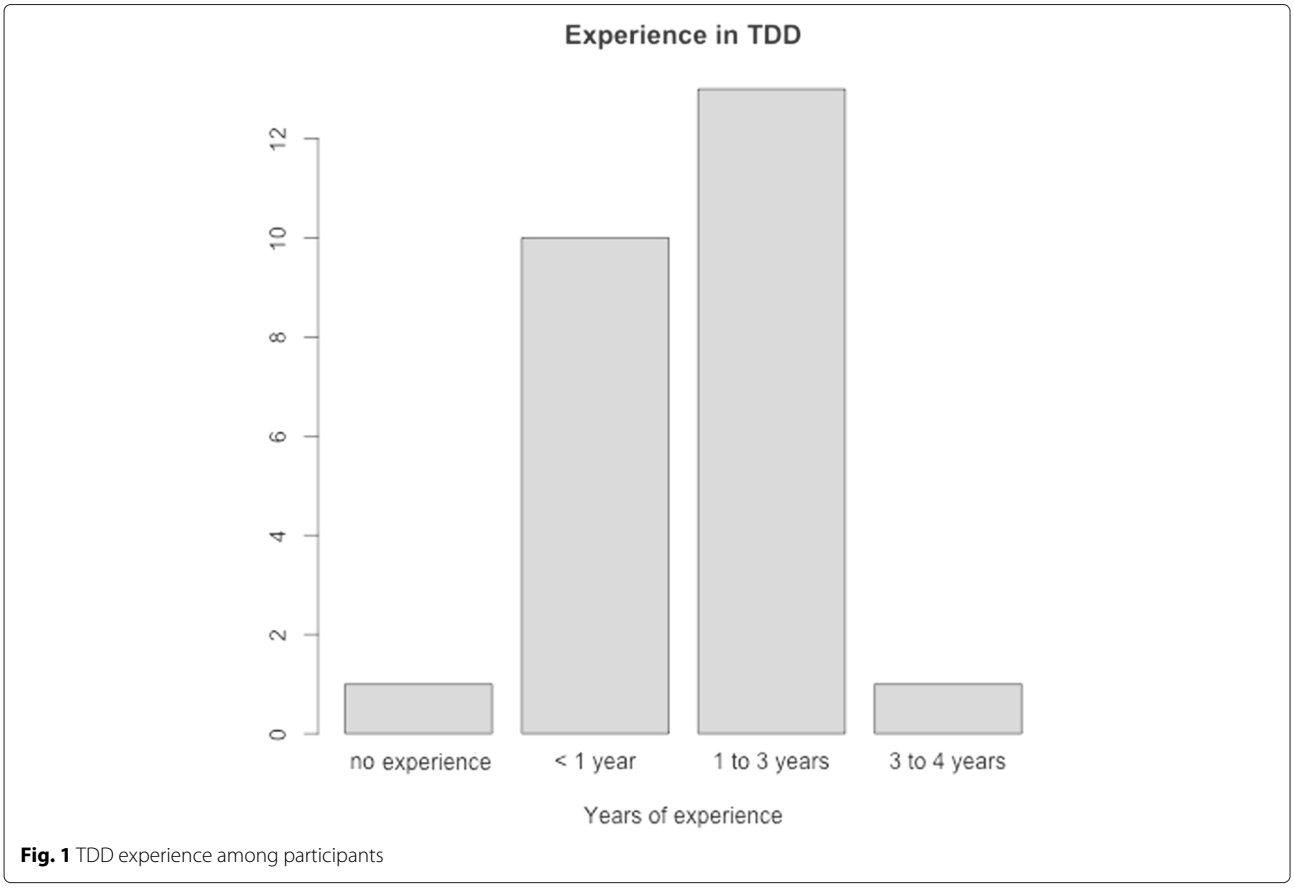

Fig. 1 TDD experience among participants

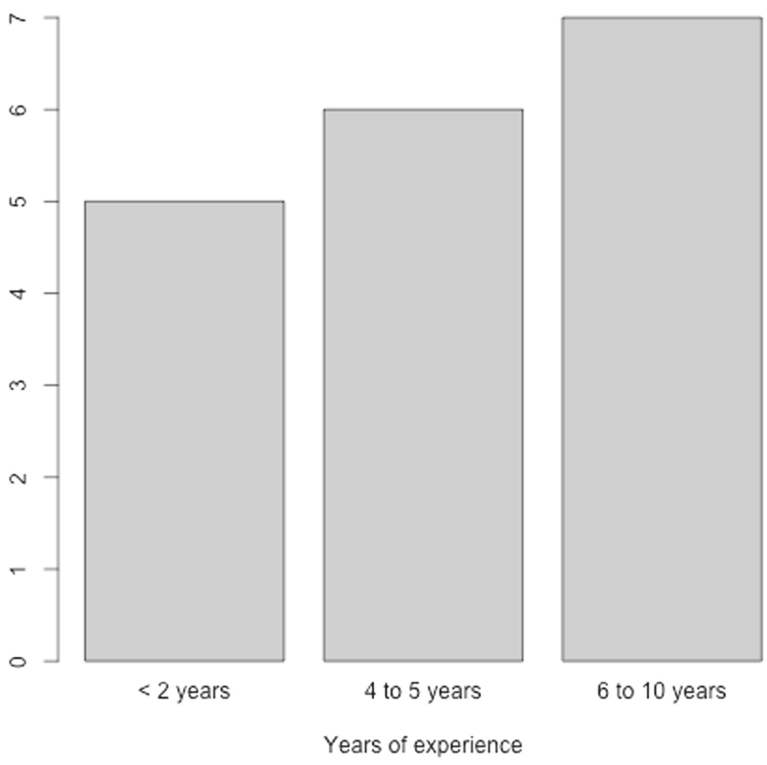

Fig. 2 Development experience among participants

may not be as simple as just having any base class and extend it. The Interface Segregation Principle (ISP) states that an interface should be as thin as possible, in order to increase its reuse. Last, the Dependency Inversion Principle (DIP) argues that classed should always depend upon more stable classes.

Martin also discusses smells a bad design may present. A rigid design is one hard to change, i.e., one change causes a sequence of changes in other modules. Viscosity means that a developer has more than just one way to make a change; some of them, preserving the design, others not. A fragile design is a design that breaks in many places when changed; this one is highly related to a rigid design.

In Table 1, we present the relation of the exercises with class design principles and smells. One can notice that each exercise asks for a different solution in order to be well implemented. In Exercise 1, for example, developers should be concerned about cohesion and extensibility (SRP and OCP). A bad design implementation would be rigid and complex. The table was built based on our own implementation of the solution. On all of them, we followed the five SOLID principles.

Table 1 Proposed exercises and class design smells

Exercise	Design smell	Violated SOLID principles
Exercise 1	Rigidity, needless complexity	SRP, OCP
Exercise 2	Fragility, viscosity, immobility	SRP, DIP, OCP
Exercise 3	Rigidity, fragility	SRP
Exercise 4	Fragility, viscosity, immobility	OCP, SRP, DIP

As said before, each participant received two out of the four exercises to work on. In one of them, s/he made use of TDD, and, in the other, s/he did not.

We explained to the participants that the exercises simulated real world problems and that they were supposed to write code considering that it would be maintained by another team. They were asked to implement the most elegant and flexible solution they could in both exercises. We encouraged them to think carefully about each solution, trying their best, regardless of the use or not of TDD.

Results and discussion

In this section, we present and discuss the analysis and interpretation of the qualitative data. We interviewed 14 developers. The participants were selected for the interview according to their profiles, answers to the survey, and produced code. Participants were selected based on the following criteria: (1) had in-depth answers in

the questionnaire, (2) the participant stated that TDD improved the design, but her code violated SOLID principles (as described in Table 1), writing, for example, a procedural code, or (3) the participant wrote a well designed object-oriented code. We selected participants until reaching data saturation[4].

During the analysis, we repeated the coding and categorization processes [28] until we had only the most important ones to discuss. In the following sub-sections, we present each one of them. The participants, regardless of their experience in TDD or software development, commented on similar points. As a result, we did not separate the discussion according to the levels of experience.

TDD does not drive to a better design by itself

Surprisingly, 13 out of the 14 participants affirmed that the practice of TDD did not change their class design during the execution of the exercises. The main justification was that their experience and previous knowledge regarding object-orientation and design principles guided them during class design. They also affirmed that a developer with no knowledge in object-oriented design would not create a good class design just by practicing TDD or writing unit tests.

The participants gave two good examples reinforcing the point. One of them said that he made use of a design pattern [30] he learned a few days before. Another participant mentioned that his studies on SOLID principles helped him during the exercises. The following transcription was extracted from the interviews:

"It was even funny. I am reading the Design Patterns (book), and it discusses polymorphism. My implementation was based on that because I've never done something a like that before (...), here I rarely create new stuff, I just maintain legacy code."

The only participant who had never practiced TDD before stated that he did not feel any improvement in the class design when practicing the technique. Curiously, this participant said that he considered TDD a design technique. It somehow indicates that the popularity of the effects of TDD in class design is high. That opinion was slightly different from that of experienced participants, who affirmed that TDD was not only about design, but also about testing.

However, different from the idea that TDD and unit tests do not guide developers directly to a good class design, all participants said that TDD has positive effects on class design. Many of them mentioned the difficulty of trying to stop using TDD or thinking about tests, what can be one reason for not having significant difference in terms of design quality in the code produced with and without TDD:

"When you are about to implement something, you end up thinking about the tests that you'll do. It is hard to think "write code without thinking about tests!". As soon as you get used to it, you just don't know another way to write code..."

According to them, TDD can help during class design process, but in order to achieve that, the developer should have certain experience in software development. Most participants affirmed that their class designs were based on their experiences and past learning processes. In their opinion, the best option is to link the practice of TDD and experience:

"The ideal is to put both things together [experience and TDD] (...) I don't believe that TDD by itself could make things get better. There are many other concepts [that a developer should know] to make things good."

In addition, when asked about what TDD is, many participants also reminded of the effects of the practice in the external quality, and the safety it gives to developers during refactoring:

"[TDD] I think it is has a big relation with code quality and regression tests. Two main advantages I have when practicing: the code gets better and the regression tests allow me to safely refactor."

Baby steps and simplicity

TDD suggests developers to work in small (baby) steps; one should define the smallest possible functionality, write the simplest code that makes the test green, and do one refactoring at a time [4]. The rationale is that the bigger the step, the bigger the time a developer stays without feedback. In addition, keeping class design simple is not an easy task. TDD suggests that developers always write the simplest code fulfilling the needs. They should only evolve the class design if the requirement grows. A class design decision can be more complicated than it looks and, without a test to make it visible, a developer would hardly notice the problem [9].

In the interviews, eight participants commented about this. One of them mentioned that, when not writing tests, a developer thinks about the class design at once, creating a more complex structure than needed:

"Because without tests, we don't think in small steps, but in the whole solution, and we end up not noticing the problems that may happen during the way."

One of the participants clearly stated how he makes use of baby steps, and how it helps him think better about his class design:

"Because we start to think of the small and not the whole. When I do TDD, I write a simple rule (...), and then I write

the method. If the test passes, it passes! As you go step by step, the architecture gets nice. (...) I used to think about the whole (...). I think our brain works better when you think small. If you think big, it is clear, at least for me, that you will end up forgetting something. "

This idea is aligned with the principle of avoiding *big design up-front*. Developers keep their code the simplest they can, evolving when needed [31].

Another participant mentioned the lack of focus faced by developers when not practicing TDD. With a short goal (which, in the case of TDD practitioners, is to make the test pass), developers get focused on reaching it:

"Maybe we are not focused by nature. You see one thing and you desire to fix that now (...)"

In fact, some studies also show that the effects of baby steps can go further. In new projects, TDD practitioners affirm that they have a less need of debugging code [13, 32]. The amount of source code written between two tests tends to be small. When a test fails unexpectedly, developers can just revert to the last stable version and start over. Sometimes, this can be more productive than the debug activity.

Refactoring confidence

Eleven participants affirmed that, during the process of class design, changing minds is constant. After all, there is still a small knowledge about the problem, and about how the class should be built. This was the most mentioned point by the participants. According to them, an intrinsic advantage of TDD is the generated test suite. It allows developers to change their minds and refactor all the class design safely. Confidence, according to them, is an important factor when changing class design or even implementation:

"It gives me the opportunity to learn along the way and make things differently. (...). The test gives you confidence."

A participant even mentioned a real experience, in which TDD made the difference. According to him, he changed his mind about the class design many times and trusted the test suite to guarantee the expected behavior:

"I was developing a tool that works with code manipulation, and I've done everything with TDD. I deleted everything many times, I kept the tests and started a new line of thinking. I thought practicing TDD helped me a lot (...). And because of that, I just ran the tool in the end; before that, I just validated it through the tests."

Again, experience is a fundamental factor. When searching for a better code during refactoring, developers again make use of their knowledge:

"(...) if you do not know about single responsibility, cohesion, coupling, I don't think TDD will help. You need to have it in mind to be able to refactor."

A safe space to think

In an analogy done by one of the participants, tests are like *draft paper*, in which they can try different approaches and change their minds about it frequently. According to them, when starting by the test, developers are, for the first time, using their own class. It makes developers look for a better and clearer way to invoke the class' behaviors, and facilitate its use:

"Tests help you on that. They are a draft for you to try to model it the best way you can. If you had to model the class only once, it is like if you have only one chance. Or if you make it wrong, fixing it would give you a lot of work. The idea of you having tests and thinking about them, it is like if you have an empty draft sheet, in which you can put and remove stuff because that stuff doesn't even exist."

We asked their reasons for not thinking on the class design even when they were not practicing TDD or writing tests. According to them, when a developer does not practice TDD, they get too focused on the code they are writing, and thus, they end up not thinking about the class design they were creating. They believe tests make them think of how the class being created will interact with the other classes and of the easiness of using it. The following statements present the same point of view:

"I think the regular behavior of people is not to think before doing something. Looks like the natural thing to do would be to start writing the code (even because of the internal pressure). (...) A few people think before starting. With TDD, you are obligated to think, TDD makes you stop and think, design. It is not natural for me to think before about doing something, but TDD makes me do it."

"As I first think about what I am going to need for the tests, I mean, I need this and that, the test makes me think before starting to write the code. With the tests, I stop to think about. Then I believe we can think better, we can develop a better solution."

One of the participants was even more precise in his statement. According to him, developers that do not practice TDD, as they do not think about the class design they are building, they end up not doing good use of OOP. TDD forces developers to speed down, allowing them to think better about what they are doing:

"In the heat of the moment, you start coupling, aggregating, inheriting, and you don't think that it can cause you a problem in the future. With TDD, you are forced to slow down. It gives you time to think better about stuff."

Janzen and Saiedian [14] agrees with it. According to him, the test is the first client of the class being developed. Because of that, developers think more and make a better decision about the class interface (decisions such as class' names and methods, return types, exceptions thrown, and so on).

Rapid feedback

More than half of the participants also commented that one difference they perceived when they practiced TDD was the constant feedback. In traditional testing, the time between the production code writing and test code writing is too long. When practicing TDD, developers are forced to write the test first, and thus receive the feedback a test can provide sooner.

"You would look at a test and say: "Is it ok? Is it not?", and do it again."

One participant commented that, from the test, developers observe and criticize the code they are designing. As the tests are done constantly, developers end up continuously thinking about the code and its quality:

"When you write the test, you soon perceive what you don't like in it (...). You don't perceive that until you start using tests."

Reducing the time between the code writing and test writing also helps developers to create code that effectively solves the problem. According to the participants, in traditional testing, developers write too much code before actually knowing if it works:

"[The test] is not only a specification; it should actually work. So, as you really reduce the time between writing software that works and testing it, you end up perceiving whether that part works or not more quickly (...)"

The search for testability

Maybe the main reason for the practice of TDD helping developers in their class design is the constant search for testability. It is possible to infer that, when starting to produce code by its test, the production code should be, necessarily, testable.

When it is not easy to test a specific piece of code, developers understand it as a class design smell. When this happens, developers usually try to refactor the code to make it easier to test. A participant also affirmed that he takes it as a rule; if it is hard to test, then the code may be improved.

"I take it as a rule: every time [the test] is very complex, I think we should stop and refactor because it certainly can be simpler."

Feathers [33] raised this point: the harder it is to write the test, the higher the chance of a class design problem. According to him, there is a strong synergy between a highly testable class and a good class design; if developers are looking for testability, they end up creating good class design; if they are looking for good class design, they end up writing testable code.

In the search for testability, developers are encouraged to write an easily testable code. As Freeman [6] states, testable code contains a few interesting characteristics, such as the ease of invoking the expected behavior, the non-need of complicated pre-conditions, and the explicit declaration of all dependencies.

The participants went even further. During the interviews, many of them mentioned patterns that made (and make) them think about possible design problems in the class they build. As an example, they told us that when they feel the need to write many different unit tests to a single method, this may be a sign of a non-cohesive method. They also said that when a developer feels the need to write a big scenario for a single class or method, it is possible to infer that this need emerges in classes dealing with too many objects or containing too many responsibilities, and thus, it is not cohesive. They also mentioned how they detect coupling issues. According to them, the abusive use of mocking objects indicates that the class under testing has coupling issues. These small patterns match with the cited Feathers' opinion [33]; a class that is hard to test probably has design issues.

Discussion

From the analysis, we were able to understand the point of view of many developers about the effects of test-driven development. We believe the findings are interesting to the community, as it discusses many of the questions and myths around the practice.

The first interesting myth contested by the participants was the idea that the practice of TDD would drive developers towards a better design by itself. As they explained, the previous experience and knowledge in good design is what makes the difference; however, TDD helps developers by giving feedback by means of the unit tests that they are writing constantly. As they also mentioned, the search for testability also makes them rethink about the class design many times during the day—if a class is not easy to be tested, then they refactor it.

We agree with the rationale. In fact, when comparing to test-last approaches, developers do not have the constant feedback or the need to write testable code. They will have the same feedback only at the end, when all the production code is already written. That may be too late (or expensive) to make a big refactoring in the class design. This is what we show in Fig. 3.

We also agree with the confidence when refactoring. As TDD forces developers to write unit tests frequently, those tests end up working as a safety net. Any broken change in

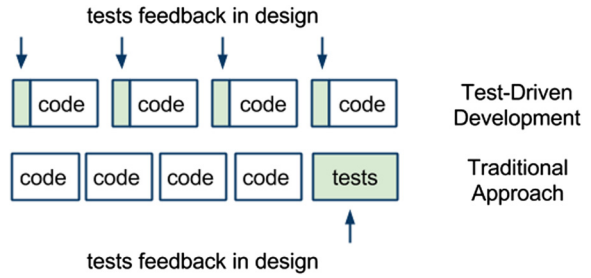

Fig. 3 Difference between TDD approach and test-last approach

the code is quickly identified. This safety net makes developers more confident to try and experiment new design changes—after all, if the changes break anything, tests will warn developers about it. That is why they also believe the tests are a safe space to think.

Therefore, we believe that is is not the practice by itself that helps developers to improve their class design; but it is the consequences of the constant act of writing a unit test, make that class testable, and refactor the code, that drives developers through a better design.

Threats to validity

In the following sub-sections, we discuss some possible threats to the validity of this study and how we dealt with them.

Construct validity

Exercises

The proposed exercises were small compared to real systems design tasks. However, all exercises contained localized class design challenges and the participants were oriented to come up with the most elegant and flexible solution, considering that the produced code would be maintained in the long term by other team. We refined the exercises during the pilot studies and during its use in other contexts. At the end of the implementation, the participants answered a question about the similarity between the exercises and real world problems. Most of them affirmed that the problems found in the exercises were very similar to the ones they deal with in the professional development. A participant said "we frequently face problems like that, in which we have trouble defining the responsibilities of each class."

Number of exercises per participant

In order to reduce bias, each developer should do four exercises: two with TDD and two without TDD. However, as we noticed in our pilot that four exercises would take around 3 h for each developer, which would be too much for the companies, we decided to reduce the experiment. On the other hand, as this study is fundamentally qualitative, we do not think that doing more exercises would change their mind about TDD.

TDD x unit testing

TDD emphasizes the progressive writing of unit tests and the reflection upon the class design, leveraging the feedback the tests offer. Tests act as first clients to the classes' API. Thus, it was not our objective to separate the effects of unit testing from the ones of practicing TDD. Some other aspects of the mechanics of TDD may also influence class design, even though they did not appear in our interviews and analysis.

Internal validity

Recent effects of TDD in participants' mind

Many participants use TDD in their daily work. This can bias participants so they would not be able to analyze the disadvantages of the practice. To reduce this bias, participants implemented one exercise without TDD, so both development practices were fresh in their minds. However, participants familiar to TDD potentially may think about the tests mentally, even when not using them. To reduce this threat, we removed from the analysis any statement mentioned by a participant with no clear explanation. Also, as we were interested in the effects of TDD, not only during the exercises, the participants were incentivized to mention anything from their professional experience. This way, we aimed to filter biased personal opinions, not supported by concrete situations and examples.

Researcher's influence

Researchers are central in a qualitative research, which rely on interpretative data. To reduce this bias, we reviewed all the analyses in pairs, looking for unclear or incorrect conclusions. We also conducted pilot studies and refined the exercises iteratively to avoid bias.

Desirability bias

Desirability bias is related to the tendency of some participants to reply questions in a way that would make them accepted by the other members of the community [34]. Agile methods and TDD have a strong discourse. The Brazilian agile community is still young, and it is common to notice some developers repeating what the community says without great experience or knowledge about the subject. To reduce this bias, we would eliminate participants that replied questions superficially. In this study, only a few answers were superficial (and were eliminated).

Participant selection for the interviews

We selected participants who gave in-depth answers in the questionnaire, mentioned explicitly TDD, or produced either a good or a bad solution for the exercises until

reaching data saturation. We may have not considered potentially interesting answers in the interviews from people who produced a bad code (using and not using TDD) and did not see value on the practice. However, we included in the interviews developers who produced good solutions regardless of the use of the practice—even those who did not see value on the practice.

External validity

Selection of participants

In this qualitative study, generalization for the whole population was not our main goal, which was more related to deeply understand how TDD help developers during the class design process. Nevertheless, to avoid bias, we selected professional developers from different companies, backgrounds, and level of expertise in software development and TDD. We interviewed participants until reaching data saturation.

Localization

All developers and companies are located in Brazil. There might be some specific characteristics we did not noticed.

Conclusions

Test-Driven Development is a well-known technique in which developers write the test before the code. It is said that the practice drives developers through a better class design. In this study, we presented developers' perceptions of how the practice of TDD may influence classes design.

"Therefore, to answer the research question: *What are the developers' perceptions on the effects of Test-Driven Development in class design?"*

Developers believe that the practice of test-driven development helps them to improve their class design, as the constant need of writing a unit test for each piece of the software forces them to create testable classes. These small feedbacks—is your test easy to be tested or not?—makes them think and rethink about the class and improve it. Also, as the number of tests grow, they act as a safety net, allowing developers to refactor freely, and also try and experiment different approaches to that code.

Based on that, we suggest developers to experiment the practice of test-driven development, as its effects look positive to software developers. As future work, tools may be developed to automatically warn developers about classes that have testability problems, or even to suggest them to practice TDD in specific parts of the code.

Endnotes

[1]The survey can be found at https://gist.github.com/mauricioaniche/5895261. Last access on April 10, 2014.

[2]Mock objects are objects created during a unit test. They mock the behavior of another object. Usually, they are used to isolate the unit test from other classes. More information about it can be found on [35].

[3]The exercises can be found at https://gist.github.com/mauricioaniche/5694865. Last access on April 10, 2014.

[4]Data saturation is a common term in grounded theory and qualitative studies. It represents the phase of the research in which no new data appears anymore, and conclusions may be drawn.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
MA met the developers, executed the experiment, participated in the data analysis, and worked on the manuscript. MG participated in the data analysis, and worked on the manuscript. Both authors read and approved the final manuscript.

Acknowledgements
We thank the companies that participated in this study: Caelum, Bluesoft, Amil e WebGoal (São Paulo and Poços de Caldas). We also thank the developers who took part in this research independently. Marco Gerosa received funding from CNPq and FAPESP. We also thank NAWEB-PRP-USP for supporting this research.

Received: 10 February 2015 Accepted: 17 August 2015
Published online: 03 September 2015

References

1. Beck K (2004) Extreme programming explained. 2° edn. Addison-Wesley Professional, Boston, USA
2. Wambler S (2010) How agile are you? 2010 Survey Results. http://www.ambysoft.com/surveys/howAgileAreYou2010.html. Último acesso em 28/10/2010
3. One V (2012) State of agile development survey results. http://www.versionone.com/state_of_agile_development_survey/11/. Último acesso em 29/02/2012
4. Beck K (2002) Test-driven development by example. 1° edn. Addison-Wesley Professional, Boston, USA
5. Martin R (2006) Agile principles, patterns, and practices in C#. 1st edition. Prentice Hall, Upper Saddle River
6. e Nat Pryce SF (2009) Growing object-oriented software, Guided by Tests. 1° edn. Addison-Wesley Professional, Boston, USA
7. Astels D (2003) Test-driven development: a practical guide. 2nd edition. Prentice Hall
8. Janzen D, Saiedian H (2005) Test-driven development concepts, taxonomy, and future direction. Computer 38(9):43–50. doi:10.1109/MC.2005.314
9. Beck K (2001) Aim, fire. IEEE Softw 18:87–89. doi:10.1109/52.951502
10. Aniche MF, Ferreira TM, Gerosa MA (2011) What concerns beginner test-driven development practitioners: a qualitative analysis of opinions in an agile conference. 2nd Brazilian Workshop on Agile Methods
11. Siniaalto M, Abrahamsson P (2008) Does test-driven development improve the program code? Alarming results from a comparative case study. Balancing Agility Formalism Softw Eng 5082:143–156
12. Munir H, Moayyed M, Petersen K (2014) Considering rigor and relevance when evaluating test driven development: a systematic review. Inf Softw Technol 56(4):375–394
13. Janzen DS (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. ACM, New York, NY, USA. pp 240–241. doi:10.1145/1094855.1094954. http://doi.acm.org/10.1145/1094855.1094954
14. Janzen D, Saiedian H (2006) On the influence of test-driven development on software design. In: Proceedings of the 19th Conference on Software Engineering Education and Training (CSEET'06):141–148. doi:10.1109/CSEET.2006.25

15. 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, NY, USA. pp 1135–1139. doi:10.1145/952532.952753. http://doi.acm.org/10.1145/952532.952753
16. Erdogmus H, Morisio M, Torchiano M (2005) On the effectiveness of the test-first approach to programming. IEEE Trans Softw Eng 31:226–237. doi:10.1109/TSE.2005.37
17. 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. Last access on March, the 1st, 2011
18. Dogsa T, Batic D (2011) The effectiveness of test-driven development: an industrial case study. Softw Qual J:1–19. doi:10.1007/s11219-011-9130-2
19. Li AL Understanding the efficacy of test driven development. Master's thesis, Auckland University of Technology
20. Madeyski L (2006) The impact of pair programming and test-driven development on package dependencies in object-oriented design—an experiment. In: Munch J, Vierimaa M (eds). Product-Focused Software Process Improvement. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Heidelberg, Berlin, Germany. Vol. 4034. pp 278–289. doi:10.1007/11767718_24
21. Muller MM, Hagner O (2002) Experiment about test-first programming. Softw IEE Proc 149(5):131–136. doi:10.1049/ip-sen:20020540
22. Steinberg DH (2001) The effect of unit tests on entry points, coupling and cohesion in an introductory java programming course. XP Universe. Citeseer. 8:2001
23. Meszaros G (2007) xUnit test patterns: refactoring test code. Pearson Education, United States
24. Josefsson M (2004) Making architectural design phase obsolete—TDD as a Design Method. T-76.650 Seminar course on SQA in Agile Software Development Helsinki University of Technology. Last access on March, the 1st, 2011. http://www.soberit.hut.fi/T-76.5650/Spring_2004/Papers/M.Josefsson_76650_final.pdf
25. Janzen D (2006) An empirical evaluation of the impact of test-driven development on software quality. PhD thesis, University of Kansas
26. Shull F, Melnik G, Turhan B, Layman L, Diep M, Erdogmus H (2010) What do we know about test-driven development? Softw IEEE 27(6):16–19
27. Runeson P, Host M (2009) Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng 14(2):131–164. doi:10.1007/s10664-008-9102-8
28. Creswell JW (2008) Research design: qualitative, quantitative, and mixed methods approaches. Third edition. Sage Publications, New York
29. Martin RC (2002) Agile software development, principles, patterns, and practices. Primeira edn. Prentice Hall, Upper Saddle River
30. Freeman ET, Robson E, Bates B, Sierra K (2004) Head first design patterns. Primeira edn. O'Reilly Media, Sebastopol, CA, USA
31. Fowler M (2004) Is Design Dead? http://martinfowler.com/articles/designDead.html. Last access on October, the 28th, 2010
32. George B, Williams L (2004) A structured experiment of test-driven development. Inf Softw Technol 46(5):337–342. doi:10.1016/j.infsof.2003.09.011
33. Feathers M (2007) The deep synergy between testability and good design. https://web.archive.org/web/20071012000838/http://michaelfeathers.typepad.com/michael_feathers_blog/2007/09/the-deep-synerg.html. Último acesso em 27/10/2010
34. Crowne DP, Marlowe D (1960) A new scale of social desirability independent of psychopathology. J Consult Psychol 24:349–354
35. Mackinnon T, Freeman S, Craig P (2001) Endotesting: unit testing with mock objects. In: Succi G, Marchesi M (eds). Extreme Programming Examined. Addison-Wesley Longman Publishing Co. pp 287–301