Personal Reading: Test Driven Development

Mats Stijlaart - University of Amsterdam, The Netherlands

February 3, 2015

For this week's reading assignment I have chosen Test Driven Development (TDD) as topic. I have some experience with the practice from when I applied it in a 6 month project and wanted to reference my experience to scientific research. The main literature I consulted for this topic are the book *Making Software* [2] book chapter 12 and the paper 'A structured experiment of test-driven development' [1]. The book mainly elaborates on the definition of TDD and presents the results of empirical studies. The paper presents a small-scale empirical study and a survey on the experience on practicing TDD.

What is Test Driven Development?

As presented in *Making Software*, TDD is a software development practice where a repetition of steps should lead to robust code. The defined steps are [2, p. 208]:

1. Choose a small task.
2. Write a test for that task.
3. Run all the tests to verify that the new test fails.
4. Write minimal production code to complete the task.
5. Run all tests (including the new one) to verify that they pass.
6. Refactor the code as necessary.

The literature presents unit testing as a tool to perform TDD. This complies to the first step 'Choose a small task', which corresponds to a testable unit. In the book the writer presents deviations on TDD by mentioning Behaviour Driven Development (BDD) and Acceptance Test Driven Development (ATDD). These are both extensions on TDD that respectively provide a better framework to work with and connect the business expectations more closely to the writing of tests. Neither of the referenced literature mentions the usage of different test types, such as using integration tests and/or system tests. I will present my point of view later on in the section 'Reliability'.

The literature shows that the effectiveness of TDD is uncertain. The results that arise from the performed empirical studies are not conclusive on the fact that it is likely that practicing TDD will improve a project in aspects such as productivity, internal quality, external quality and test quality [1] [2].

Merits and drawbacks

As all the things in life, TDD too has its merits and drawbacks. The main advantages that the literature presents as arguments in favor of this method are: TDD results in better external quality [1, p. 340] [2, p. 212]; TDD utilizes the understanding of code and thus helps to manage complexity [1, p. 338]; creates test assets that utilize (automated) regression testing [1, p. 338] [2, p. 208]; and decrease the chance of defect injection [1, p. 338].

Some of these arguments are presented clearly with scientific foundation, such as the increase of external quality when TDD is applied. This effect is pretty straightforward when you study the above definition that 1) all new code is tested while you are not allowed to write more code than required (step 4 in the definition), 2) new code does not break the existing implementation (step 5 in the definition), and 3) the implementation is always feature-centric. This implies that by definition it is impossible to introduce code that degrades the external quality because all code which is introduced, is introduced intentionally. This statement is verified by the advantage that the defect injection chance is decreased.

What I find a more obscure advantage is the statement that TDD utilizes the understanding and management of complexity. Firstly, this point is not verified with scientific proof in the researched paper. Secondly as the paper [1, p. 338] addresses and the book lacks to do, using TDD does not lead to formal documentation (while the test is your documentation). The alternative to documentation in TDD are the produced tests. However when you open the test code the required information may not be presented in a glance. For example:

> *Suppose there is functionality 'A' with the specification 'When a number is entered between 1 and 10, multiply it by 2 otherwise throw an exception. Using TDD this will result in at least 3 test cases: testing a valid number; testing a number below 1; and testing a number above 10. When a developer has to search for the specification based on the tests he has to look into the details of 3 tests, which possibly will not fit into one developer console due to programming language syntactics. Instead a simple comment above the implementing code or a formal requirement may be more helpful to the developer.*

The above example is rather small and in my opinion will cause more problems when functionality grows and the project has a lack of documentation. The reference implementation I got for this problem is an algorithm for medical images to transform a 16-bit gray-scale image to a 8-bit gray-scale image with additional parameters to shift the domain and the window on the image. The tests written using TDD were created when I knew the domain and the requirements of the algorithm. However when you open the test code at this moment it is unlikely to figure out what the specification was. The point here is that you can not assume that the test code is a replacement for formal documentation.

Reliability

When applying TDD to build a program or software module, the building blocks are tests. The literature states that if you apply TDD the amount of building blocks grows, which leads to certainty as you continue building in the form of regression. I believe that this is true in a certain point of view, the book's point of view. The programmer contuiously increases the test set. But I would like to shed light on the reliability of these building blocks with a simple example.

> *With unit testing a tester focuses on testing the external quality of a single unit. To correctly test a single unit the tester should 1) remove all dependencies on IO such as file system interaction and network activity, 2) take control over the instances the unit depends on performed by mocking dependencies [3], 3) improve testability by introducing patterns such as dependency injection. Suppose the tester fully tests*

functionality 'A' and mocks a functionality 'B' on which 'A' relies. When the tester mocks the behaviour of object 'B' he should know what the domain and range of the behaviour is to create correct tests for 'A'. So far, so good. The problems arise when there is a request for change on functionality 'B' whereas the domain and/or range changes. A developer is asked to modify 'B' and he updates the functionality and its related tests to the new specification. There will still be 100% test coverage if both implementations are truly developed with TDD and all tests will pass because 'B' is mocked in the tests of 'A', but with this example we can conclude that functionality 'A' is not tested properly i.e. it does not integrate properly with functionality 'B'.

The above example shows that pure TDD development by using the notion of unit tests may lead to the assumption that a system is tested properly since most components are tested. However, bugs may show up in both tested and untested code. The point is that the foundation on which TDD is built might give a wrong sense of confidence. Although the outlined problem can be simply solved by using integration tests or system tests instead of unit tests, this contradicts with the definition of TDD 'Choose a small task'. With integration and system tests the developer will write considerably more code before all tests pass.

The mentioned confidence is not completely inappropriate. The following statement will hold 'When the tests do not pass, then the code is not correct.'. However, this does not imply that the inverse will hold: 'When the tests pass then the code is correct'. Developers applying TDD must be aware that the inverse of the original statement is a dangerous assumption.

Approach

A developer can build functionality with or without TDD in two directions: bottom-up or top-down. For example when requesting a user from a webservice, the developer may start at the endpoint in the server and work his way down, or start at the database and work his way up. From my personal experience I encountered that you keep rewriting your upper layered implementations because you have encountered that you are missing information in one of the lower layers or vice versa. By using TDD the developer needs to rewrite both implementation and tests. Both the paper and the book both state TDD lacks upfront design [1, p. 338] [2, p. 212], resulting in a greater chance that a developer can't foresee all the aspects of the detailed implementation. I believe that this happens in both top-down and bottom-up workflows.

Questionable Aspects

The research shows different aspects of the effects of TDD. Some interesting aspects I want to outline are summarized below.

Productivity

The research clearly points out that the productivity decreases (a developer requires more time to develop new functionality) when applying TDD [1, p. 341] [2, p. 213]. This is understandable when you realize developers may spend more time on testing wehn applying TDD, than when developing in a code-first style. This is also shown with the number of tests and the code coverage related to the test quality when applying TDD

[1, p. 341]. A question which remains unanswered is wheter this productivity remain lower in the overall picture. A project where TDD is applied may have less bugs than a system developed without TDD. The absence of creating bug fixes might make up for this decrease in productivity. I did not find related literature which proves or disproves this.

Level of difficulty

Both the paper and the book qualify TDD as hard or difficult. Personally I believe this is not true and the difficulty statement is context-dependent. When a developer starts applying TDD he has to cope with the side-effects in the code such as when a developer starts applying TDD. Mocking these side-effects, such as input/output, web-requests and database behaviour, is hard. A developer will face these challenges directly when applying TDD. This does not make TDD hard, but only the concept of unit testing. But that was also hard when the developer applied the code-first style. However, with the code-first style the developer was free to skip testing functionality in full extend. This can be addressed as lack of discipline, which brings me to my point to make: Discipline is hard, not TDD. TDD requires discipline because you have to strictly adhere to the guidelines, while code-first allows developers to cheat. This might be why developers find TDD hard: they lacked the discipline in the first place to write properly tested code in their code-first style.

Conclusion

As the literature shows, Test Driven Development improves external quality of code. The increase of test quality contributes to this aspect and allows a developer to be more confident, but when carried out too much may result in an unfair amount. As I have presented with different examples: TDD does not replacing documentation, it is built on top of an incomplete practice (unit testing) and requires a certain amount of discipline. I conclude that the concept of TDD is strong, but still too weak. For example I think the concept might be vastly improved by writing integration tests up front.

References

[1] Boby George and Laurie A. Williams. "A structured experiment of test-driven development". In: *Information and Software Technology* 46.5 (2004), pp. 337–342. DOI: 10.1016/j.infsof.2003.09.011.

[2] Andy Oram and Greg Wilson. "How Effective Is Test-Driven Development". In: *Making Software: What Really Works, and Why We Believe It.* 1st ed. O'Reilly, 2011. Chap. 12, pp. 207–219.

[3] Steve Freeman Tim Mackinnon and Philip Craig. "Endo-testing: unit testing with mock objects". In: *Extreme programming examined.* Addison-Wesley Longman Publishing Co, 2001. Chap. 17, pp. 287–301.