Using TDD to build a game platform: How to maintain the big picture and yet remain agile?

Marko Taipale, Ari Tanninen

marko.taipale@iki.fi, ari.tanninen@iki.fi

Abstract. We built a server-side Java game platform using Test-Driven Development techniques. We learned that even though TDD is a powerful tool it is not a substitute for designing software on the whiteboard. TDD also poses some risks. Adhering too strictly to TDD and refactoring may cause the overall design to erode. This can be mitigated by stepping back, and revisiting the big picture on a whiteboard every now and then. However inexperienced developers are less likely to see problems in the overall design, and probably should not attempt to do TDD without guidance.

Keywords: Test-Driven Development, eXtreme Programming, refactoring, software design

Summary

There is on-going controversy about Test-Driven Development in the industry. On one side there is Ron Jeffries' ten minutes of design followed by coding [1] taken literally and on the other James O. Coplien's pointed statement that "TDD deteriorates the architecture" [2]. We suggest that the controversy is caused by the TDD community advocating and the software industry applying TDD without fully considering what is required to succeed with TDD. According to our experiences of TDD skill and experience are crucial, but they are rarely mentioned in discussions.

We wrote a new version of a legacy game platform from scratch using Test-Driven Development techniques. In our experience TDD increases productivity, makes development fun and increases the quality of code. TDD is not a free lunch, however. We had to put more effort than expected in maintaining our test code or risk turning it into a burden slowing development down instead of speeding it up. We found out that relying only on TDD may produce software that looks good in code, but lacks overall design. Even worse, TDD tends to erode existing design. When refactoring a piece of new code it is possible to inadvertently break the design elsewhere.

To compensate we started supplementing TDD with periodical whiteboard design review sessions – "refactoring-in-the-large" – with good results. Four months later, we had a new game platform with a clean architecture.

Background

European Gaming & Entertainment Technologies Ltd. ("EGET") is a provider of on-line monetary games and services. EGET has over one hundred employees mostly in the cities of Helsinki and Mariehamn in Finland. The challenge for EGET's Games department is to produce more and more games to satisfy the rapidly growing market. EGET's games run on a legacy gaming platform built bit by bit during the last decade. Because the old platform is effectively design dead software, extending it with new features or building new games on top of it is slow and complex.

EGET management decided to build a new game platform that would allow games to be developed within weeks rather than months. To do that a team consisting of mostly senior software developers was formed. The team initially comprised three Java developers with a total of 30 years of experience in developing server-side systems, and a database expert. Each developer had years of experience in TDD, but had not previously built a system from scratch with it. Since the team had little experience of building games on EGET's legacy systems, two game developers joined the team later on.

Building the Platform

In the beginning we knew what systems our platform had to integrate with, but didn't know how. A game has to talk with EGET's payment system, be managed by a back office system, and be integrated with the gaming site. Unfortunately no one could tell us how exactly the old game platform was integrated to these systems. We also knew that each of EGET's games have multiple states, and consist of a Flash client and a Java Servlet backend, are persistent and require access to a random number generator. Also many games share most functionality, but EGET has over 60 games, and no one can tell all of their features in a sufficient level of detail.

From this we envisioned a platform architecture consisting of a server-side MVC framework, a game controller, a random number generator API, a persistence API, a payment API, and some kind of way to integrate with back office and gaming site.

Since the architecture vision lacked details, we moved it aside and decided to implement a few games, and then extract the commonalities to code in the platform. With luck, the extracted and refactored code would fulfil a part of the vision. We could now start implementing the first game and first components of the platform with TDD.

We soon found out that TDD delivered what was promised. It increased our productivity and made development fun. It helped us to develop new features fast and with better quality than ever before. Writing tests first gave us insane focus and freedom to change our software as we wanted and still be convinced that we had not broken anything. We were addicted to TDD and swore there was no going back.

Our approach worked for a while, but on the sixth of our two week iterations we started noticing code smells that were elusive to pinpoint by looking at source code. After hours of frustration we sketched the relevant part of the design on a whiteboard, and immediately pinpointed several flaws: mixed up responsibilities between objects,

objects with too many responsibilities, and complicated communication patterns to name a few. We were astonished that we had created something so messy when we had tried to be so very careful. Now that the problems were clearly visible, fixing them was a matter of few days of straightforward coding at the worst.

After our sixth iteration experience had repeated itself a couple of times, we started to have second thoughts about TDD as a way of designing software. TDD depends on refactoring for design, but our design problems were at a higher level than the refactorings in Martin Fowler's book [3]. Only doing "refactoring-in-the-small" seemed to produce a big soup of highly cohesive, loosely coupled objects, and little higher level design. To create the missing design we started supplementing "refactoring-in-the-small" with half-an-hour whiteboard sessions – "refactoring-in-the-large".

Interestingly enough it seemed the less experienced members of our team had more difficulty in seeing design problems on the whiteboard, even after prolonged discussions of them. On occasion the discussions reached a point where it was strongly argued that the on-going "refactoring-in-the-large" is a waste of time, and should be terminated. This would seem to support Pekka Abrahamsson's suggestion that TDD may require a certain level of professional expertise, and may not suitable for junior developers [4].

Another thing we discovered is that TDD can erode existing design. When refactoring newly written code it is possible to nibble at the design of a different but related piece of code. After the refactoring is done the new code is clean and all tests pass, but the design of the old code is broken. Perhaps the developer of the new code was not aware of all the aspects of the old code, such as immutability or thread-safety, an intentionally broke the design. If the refactoring introduced a new dependency that just happened to cross an architectural boundary, the architecture was just broken.

While we never managed to break our architecture as described, we constantly had to struggle or at least be keenly aware of "design erosion". It was crucial for us to take a step back and inspect the effect of all refactorings on the whole, and do "refactoring-in-the-large".

Results

After six months of development we have a platform with a clean architecture. Every piece of the platform solves a problem and so far extending and modifying it has been smooth. Whether that flexibility is enough we will see when the platform has gone to production in 2008 and we have added the first new features to it.

About a half of all the code we have written is test code that never sees production. Usually when the platform is changed only a fraction of development effort is spent working on test code, but sometimes test code changes require more effort than platform code changes. It could be argued that maintaining so much test code is a burden, but we are convinced that it is worth it. Spending time writing tests speeds up the overall effort. We feel that this is inherent to TDD, and a necessary evil.

Lessons Learned

While we are happy with the results we got with TDD and could not imagine developing software without it, it is not a substitute for good old fashioned design work on the whiteboard. The TDD community suggests that TDD is about design rather than testing. But what exactly is meant by design? According to our experience TDD is a great tool for low level design – design involving a few classes or perhaps an API. But it offers very little for designing anything larger than that. After all TDD is based on refactoring, and refactoring is about improving the design of code. The design of our game platform is based on hard design work and our experiences of building similar server-side systems rather than TDD magic.

TDD poses certain dangers that should be recognized. First of all adhering strictly to TDD and refactoring risks creating a big soup of classes without an overall design. In addition TDD may cause existing design to erode, and in the worst case cause the architecture to deteriorate [2]. These problems can be mitigated by inspecting the design on a whiteboard every now and then and doing "refactoring-in-the-large".

Another danger to watch for is low quality test code that can kill agility. If test code is not refactored and carefully designed it can end up becoming monolithic just as well as application code. And since the test code is tied to the application code, it does not matter how modular and well designed the application code is, the whole is still a monolith that resists every attempt to change it. A good indicator of this is that it consistently takes longer to change tests than the application. Therefore, simplicity, readability and modularity are even more important for test code than for application code.

Since refactoring - like any software design activity - requires experience and skill it should not be done by junior developers alone. But "refactoring-in-the-large" is even more demanding than refactoring because it operates at a higher abstraction level and involves more moving parts. Refactoring at any level is design, and developers' skills play a big role in the quality of design, as suggested by Lasse Koskela and Bas Vodde in their article about learning TDD [5].

In our opinion a part of the TDD controversy is caused by the software industry's naïve interpretation of TDD where design magically emerges from code as a result of mechanical refactoring. Ironically even eXtreme Programming has mechanisms for supplementing TDD and refactoring. Ron Jeffries explicitly mentions CRC cards and the whiteboard in his book eXtreme Programming Installed [1], and instructs to proceed with implementation test-first, *if possible*.

Finally we propose the arguments for and against TDD are lacking context. The skill level of a team should be a factor when discussing the merits or benefits of TDD.

References

1. Jeffries, Ron: Extreme Programming Installed. pp. 87--88. Addison-Wesley, Boston (2001)
2. Coplien, O., James: Religion's Newfound Restraint on Progress, http://www.artima.com/weblogs/viewpost.jsp?thread=216434
3. Fowler, Martin: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (2002)

4. Siniaalto, M., & Abrahamsson, P. (2007) A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage. in First International Symposium on Empirical Software Engineering and Measurement, ESEM'07, Madrid, Spain, LNCS.
5. Koskela, Lasse, Vodde, Bas: Learning Test-Driven Development by Counting Lines, IEEE Software May 2007, pp. 74--79 (2007)

Acknowledgements

Thanks to James O. Coplien for the provocation, and for pointing out that we have an interesting case. Cheers!