It Has Been a Long Journey, and It Is Not Over Yet

Avraham Poupko(✉)

Cisco, SPVSS, Shlomo Halevi 5, Jerusalem, Israel
apoupko@cisco.com

Abstract. This paper tells the story of my Agile Journey over the last 15 years. It is neither exceptionally rosy nor excessively pessimistic. It is an attempt to give an honest experience report.

Keywords: Experience · Journey · Extreme programming · XP

1 Introduction

> "If you are going through hell, keep going."
> *Winston Churchill*

I have been in the industry for over 20 years, and I have been practicing some or another form of Agile from the very early days. I have seen myself mature, I have seen the industry grow and mature, and in particular, I have seen my organization mature. This paper tells that story. It tells about challenges, hopes, frustrations and accomplishments. I then offer some retrospective.

2 Background

Since 1994 I have been working for NDS that was acquired by Cisco in 2012. I write code and design systems. I take great pride in a job well done. Having fun is a major objective.

My current role is defined as Senior System's Architect. I am expected to be deeply familiar with the core products, to understand the customer's needs, and to lead the task of building something that meets the customer's expectations, while remaining in line with the company's technical and business objectives.

I work directly both with customers and with developers and development leads. As a senior person within the organization, I am also expected to provide technical guidance and inspiration. Currently I do not have any direct reports, but I influence the work of several dozen engineers.

C. Lassenius et al. (Eds.): XP 2015, LNBIP 212, pp. 270–278, 2015.
DOI: 10.1007/978-3-319-18612-2_27

3 The Pre-agile Days

In the early days of my career (1995-1998), the company I worked for (NDS) and myself were heavily invested in Object Oriented Design as a design paradigm and in C++ as an implementation language. We strongly believed that OOD allows the close modeling of the real world. We would often argue: "The world is made of objects, not functions or procedures", and that if done correctly, OOD would make code development much easier. Most importantly we believed that OOD would allow us to "get it right." All you needed to do was understand the domain, map the domain to a class hierarchy, and go implement. Of course you had to make sure that the domain was clear, because if not, you would misunderstand it, and create a wrong hierarchy. That would be painful to fix. But that was an error left to amateurs. Professional programmers and designers would get it right, and once done right, adding or changing requirements would be a piece of cake. Personally, I strongly believed that as I got better at my job, I would be able to design Object Oriented systems that are robust.

4 The Journey Begins

4.1 Getting Started

My personal journey with Agile began in May 1999. This is when I read an article by Kent Beck in C++ report on something called "Extreme Programming." The name caught me. If programming is fun, then extreme programming must be extremely fun. The article started with a discussion of. "How does the cost of changing software change over time?" I knew by then, from painful experience, that the later a mistake is discovered, the more costly it is to fix. That was the reason that upfront design is so important. I knew that the reason we make mistake in upfront design was that we did not design enough. All we needed to do was design more.

The direction the article took surprised me. The article said that we design *too much*. If we designed less, we would make fewer mistakes. The article went on to explain the idea that in extreme programing we do not plan for the future, rather we develop for the known. Less design is better.

I showed the article to my boss, and had a chat with him. He read it and said, "I am not sure if this is naïve, or really smart. Does he mean that we are all doing it wrong?" I did not have the experience or internal conviction to aggressively push this. I needed to learn more.

I started reading up on Extreme Programming, and I tried to convince my boss to let me just try it out. He was not willing to take the risk. He was not willing to forgo code reviews, he was not willing to forgo upfront design, and he was not willing to allow shared owner ship of code. I think I know why he refused to take the risk. My boss was a very moderate, thorough and levelheaded guy. He believed in deep deliberation, and considering all options. Even the name "Extreme Programming" rubbed him the wrong way. At the time "extreme sports" were becoming popular. The participants in these sports came across as thrill seeking, risk takers. Another part of the problem was that I came across as too enthusiastic, and too dismissive of current prac-

tice. In hindsight I realize that this was a tone I had picked up from articles written by the likes of Kent Beck. The XP people are sometimes too enthusiastic, and too optimistic. This can frighten some people.

4.2 The Conference

The first break came when I was invited to the first XP conference that took place in Sardinia in late June 2000. At this conference I met lots of people that were already well known in the field of software development.

I met Ron Jeffries, Kent Beck, Robert Martin, Martin Fowler, Alistair Cockburn and others. I remember feeling that these are all people that are truly passionate about their job. They firmly believe that they can do better, and they want to do better. I attended workshops and lectures, and had great informal conversations with lots of people. I remember that conference as the best I have ever attended. I felt that XP is a real methodology that can solve real problems. I felt inspired to learn more about XP

I came back from the conference deeply inspired. I felt that something very significant was about to take place, and that the software development world is about to undergo a change. I felt challenged, and resolved to do more to get the organization to accept Extreme Programming.

I started rethinking my position that the problems with OOD were due to lack of expertise. I had a particular case where a class hierarchy that I was rather pleased with, contained a structural problem that was only discovered a year too late, and parts of the framework needed to be rewritten. Rewriting the code was long and painful, and in our effort to preserve as much work as possible, and not rewrite everything, we ended up with classes and member functions, with names that no longer represented the actual meaning of those classes. I knew that this code would forever be hard to maintain, and there was little I could do about it. The timing was perfect.

4.3 Trying Harder

I decided that I would try harder to make the company adopt XP. (Spoiler Alert – I failed, but I did succeed in getting some people to adopt some good behaviors). Since then I have learned many times that change in the organization can only happen if there is an individual, or individuals, that are able to drive a change forward. A non-Agile company will only convert to Agile, if there is someone actively pushing.

I decided to start with Test Driven Programming. I knew that Test Driven Programming has value that is independent of any other XP behavior.

First of all, I took aside a senior developer, whom I have a lot of respect for, and I showed him how to do Test Driven Programming. This is a guy that had been programming in many languages for many more years than I had, and who was significantly senior to me in the company hierarchy. I showed him JUnit and CUnit, and I showed him the example that Martin Fowler had shown in his workshop. To my delight, ***he got it***. He understood exactly what test driven programming could do for him, and he has been doing test driven programming ever since.

Lesson Learned: If you are going to promote an idea in an organization, try to convince one person. One person is a great start.

Unfortunately, TDD is still as widely accepted within our company as it should be. We have discovered some major problems with it, but basically it works and its value is recognized.

One bizarre effect of TDD was the following. Sometimes a new behavior is required of a function, and the developer has a choice, whether to add a control variable, or just create a new function. For example there is a function: `f(int x);` that does something, but we sometimes need the function to do something else g, so we have two options:

1. Extend the function `to f(int x, bool do_also_g = false);`
2. Create a new function `f_and_g(int x);`

The decision whether to go for option 1 or option 2 depends on all sorts of considerations. The ease of reading and writing the *program* are certainly legitimate considerations. However, the programmers discovered that option 1 makes writing the *tests* easier. So people were extending the signatures of the function with more and more control parameters, not because that is what made programmatic sense, but because that is what allowed the developer to avoid writing a whole new set of tests.

Where is the fallacy here? By adding more parameters, *proper* test coverage grows combinatorically. That means, in the above example, the entire set of tests needs to be copied, with the control variable set to true.

In general, I noticed that people often treated test writing as sort of an "overhead," something to be avoided. This can be a big problem. I might go as far as saying that if you treat tests as overhead, and not as part of the code to be delivered, you might as well not write them.

Also, people only wrote tests for *new* functions. Writing tests for the thousands of existing functions seemed to be too daunting a task and was never done. So even though all the tests had passed, it was only those tests that we had written. Thus we never really had test coverage.

I tried to introduce pair programming. I took a friend, and we started programming together. That was a miserable failure. It failed because we did not adopt supporting behaviors. For example, the programming was done at my PC, and I left the email client on. Whenever an email popped up, I would take a look to see if it was urgent, and if it was urgent or short, I would take care of it. This is bad behavior even when programming alone, but it is absolutely devastating when working in pairs. While I was checking my email, my partner went to check his. By the time we got back together, 20 minutes had gone by, and we had lost the flow.

4.4 Not Really Agile

Over the next ten years, people within the organization started becoming more and more exposed to Agile methodologies. We adopted some practices, but not others. We never really became a truly Agile organization. The following paragraphs outline that.

Pair programming never really picked up. There might be cultural reasons for that, or issues to do with ego. I tried to do my part in educating people on the value of pair

programming, but I was not very good at it myself, and thus had a hard time convincing others.

Shared ownership never really picked up either.

I know why "shared ownership" did not work. Some people took "ownership" to mean everyone can change any code anywhere, but the *real* "owner" of the code will clean it up if there is a problem. They did not really internalize the idea that with "ownership" anyone that touches the code is responsible to leave it in as good or better condition than it was before he started it. Ownership does not give you a right to break things.

Daily build and Test Driven Programming did pick up a bit. However, we cheated a little, and as a consequence we never really got the full advantage of those two great ideas.

For example, if there was some code that did not pass all the tests, we just commented it out. We did not want to break the daily build. We would start in the morning, uncomment our code, and start writing. At the end of the day, if we did not pass all tests, we would comment out the code again, and do our daily build. So over a few days we ended up having quite a bit of code that did not fit into the daily build.

4.5 Retrospective

Looking back at the early days, I realize several things. First and foremost, in order to do XP, you must do more than just follow the rules. You must follow the spirit, and understand how to apply the rules. Also, in order to push XP within the organization, you need a champion that gets what it is about and believes in it. He needs to be technically excellent and able to show results rather quickly. The champion needs to be passionate, charismatic and convincing. This is true about any change, but XP involves so many changes at once that this is much more significant. Not enough thought was given by the initial pioneers of XP as to how to migrate code that was written in a non-XP environment to become "XP friendly".

5 Where We Are Today

5.1 Growth

Over the following ten years, the company was growing, fast. Our branch of the company had grown from 200 people to over 1000. Some parts of the company adopted Agile practices, mainly Scrum, and others did not.

Personally, I was becoming more involved in design and less in the day-to-day coding.[1] As I got into more and more of a design and architecture role, I became more proficient at "lightweight design." I resisted making the design flexible in support of requirements that might one day come along. But design was never refactored, so

[1] At the time, the company encouraged that separation. Architects design, and Coders implement.

when we made bad design based on wrong assumptions, the bad design often stuck even after our assumptions were corrected.

In 2012 our company was acquired by Cisco. Cisco was a whole new ball game. Cicso is a large company, by any standard, that puts a huge emphasis on optimal performance. Cisco is very aware of the cost of developing and maintaining software, and will go to lengths in order to reduce costs and keep them down.

One of the things that our new mother company was concerned about was that even though our customers were getting most of their requirements met, the cost of developing software was on the rise. Our systems were large and complex and brittle.[2] One day, management of the newly created division prescribed the following: "From now on, we will be an Agile organization. We will all do Agile 'by the book'."

Everyone wanted to be part of the Agile transformation. The particular form of Agile chosen was Scrum. So people started training as Scrum Masters, and Product Owners. We were supposed to transform from a company that exhibits some Agile practices in some places, to a company that was fully Agile. My personal reaction was one of "reserved optimism". On the one hand, I was excited that we were now going "fully Agile" and I was hoping to play a significant role in that transformation. On the other, I was not really sure what "fully Agile" means, and if being "fully" anything was a good idea.

Over time, I noticed the following phases. Not every part of our unit went through the phases in the exact same way, but all parts of the organization certainly went through some variant of them.

5.2 Phase 1 - Optimism

Agile is great. If it is Agile, it is good. In this phase, people were going around touting Agile as the solution to all problems. There were people from other parts of the company going around telling these great Agile stories of how if we would only adopt Agile, many of our problems would be solved. Everyone wanted to become a scrum master or product owner, and everyone was attending workshops. I was delighted. More than 10 years after having attending the first XP conference, it seemed that the company I worked for was willing to fully commit to being Agile. I volunteered to mentor, to teach, to coach, to preach and guide on anything and everything that had to do with Agile.

---

[2] While the two often go together, it is worth explaining the difference between "complex" and "brittle".

"Complex" means "Many parts with many interactions". This goes together with high coupling, where the knowledge to implement requirements requires knowledge of lots of "neighboring" components. This makes the cost of change high.

"Brittle" or fragile is when the design is precarious, in the sense that every small change, force changes to the design.

5.3 Phase 2 - The Problems

Not everyone understood Agile, but almost everyone liked it. Once in a while something really troubling would come up. For instance Agile was occasionally used as an excuse for laziness, but it was hard to tell when. I myself, when asked for a detailed design of some particular aspect would often say: "Oh, we are Agile now, we don't need that." But deep inside, I was not sure if I was being Agile, or being lazy. The YAGNI (You Ain't Gonna Need It) principle can give easy rise to procrastination. I do not like doing very tedious detailed work, and the YAGNI principle was too convenient. After seeing this behavior in others, and myself I started to try to articulate "Patterns" of Agile design. I tried to define heuristics and rules to help us identify which elements need to be done up front, and what can be deferred. One activity I found particularly productive was to review the relationships between the real world entities and to have a discussion as to which relationship is intrinsic and which is not ("incidental"). More important than the actual architecture was this common understanding. I tried to make sure that everyone involved in the coding, understood these underlying assumptions so that the design decisions that were taken every day accounted for them.

Another example of the misunderstanding of Agile is in the way people understood user stories. I often heard people say, "Stories?? They are just requirements. Take all the requirements and translate them into user stories (As an X I need Y so that Z)."

They simply missed the point that stories are meant to first be *told,* and only then captured as user stories. The *telling* of a user story shifts the emphasis from the formal and context free description of what the system does, to the highly contextual. I later came to learn that this is an extremely common misunderstanding in the world of Agile. A lot of people really don't get user stories. As a consequence, people often would just read the user story, but never had the story told to them, and sometimes missed important information, that would have been conveyed had they actually been told the story. This is certainly a battle worth fighting and whenever we use Rally or some other tool, I also make sure that before the story is written, it is *told.*

While we were quickly adopting some Agile behaviors, the core values of "Agile" were took longer to sink in. I realized that a value such as "prefer individuals over process" must come from deep within the individual and within the organization. And if you do not believe in the values, then it is very difficult to really be Agile.

A typical example might help illustrate. I once spent quite a bit of time at a customer site to learn about the customer's needs and business. I came back from that customer, gathered the team, and told them the story. I told in dramatic detail, how the operator (Fredrik) gets his instructions, and how he configures the system on a daily basis. I told about how frustrated Fredrik gets with the current system, and how some modifications would make his life much easier. I then wrote some "place holder" stories such as "When adding a channel, Fredrik needs to be able to easily copy a configuration from an already configured channel. This allows Fredrik to save time, and to make fewer configuration errors." I thought that the story was clear, and so did the developers. We entered the story into Rally, and expected that whenever people read the story, they would remember what I told them in detail, and would know how

to implement. Yet, one of the PMs objected, "How can you put a first name in a requirement? And you must follow form. This is what the story should say – 'As an operator, I need to copy configuration from an already configured service, to a newly defined service, so that the correct specification of the service parameters is ensured'." He went on to explain that since they are the basis of acceptance, the user stories must be properly formal, and have full detail.

I chose not to fight this particular battle, and just complied. But since then, I have been campaigning to craft user stories that explain what the system must do to bring value to the users, and acceptance criteria that explain what the system must do so that we get paid.

5.4 Maturity

It seems like at the moment, we are reaching some sort of equilibrium. We are figuring out what Agile practices work for which groups and which parts of the organization. We are not yet a fully "Agile organization" (does such a thing exist?), and we might never become one, but we are doing much better.

A lot of the effort is focused on process training. There is also technical training, in the use of tools and languages. What I find lacking most is training in proper design. I am currently working on creating a community of analysts and architects that train and mentor each other in good design. Because design is in the grey area between skill and art, it is not an easy thing to teach, but we are trying.

Over the last year I have noticed that some of the very good developers are not properly trained or skilled in analysis and design. This is a shortcoming that I think can and must be fixed. If we claim that everyone is responsible for the overall quality of the final product, then everyone is responsible for the design.

6 Retrospective

It has been a very long journey, and I learned a lot along the way. Here are some of the main takes that I would like to share.

Legacy is an extremely powerful force. If we have a large organization that has been around for a long time and thus has lots of legacy software and behaviors, it will require a great deal of force to change the organizational behavior. Whenever we discuss adopting Agile behavior, we need to discuss how we get there.

Adopting behaviors is not enough. In order for Agile to be successful, you need to believe in the values.

The evolution from extreme programming to Agile shows a great deal of maturity. Extreme programing preached that you must follow all the rules in order to be an extreme programmer. Agile preaches only core values, and lets the individual or organization, blaze their own trial.

The process of going from waterfall or partially Agile to Agile, is an important process. We are not yet done, and it will not be easy. It will probably continue to be a very bumpy ride.

Acknowledgements. Special thanks to all my friends and most of my managers in NDS and then Cisco, who patiently listened to all my ideas. Your feedback was not always kind, generous, or even welcome. But it was always honest, and after a while – appreciated.

Thanks to Rebecca Wirfs-Brock for great comments and insights.