A Self Funding Agile Transformation

Daniel Poon
Romax Technology Ltd
dpoon@romaxtech.com

Abstract

Funding is often seen an impediment to going agile. If cash is tight, you are not going to be able to pay for a “Big-Up-Front-Agile-Transformation”. However, transitioning to agile can be self-funding. If people take the first bold step themselves, they will find that the funding will sort itself out along the way.

1. Turtles all the way down

An agile training company will send your team on an immersion course. But “Big-Up-Front-Training” doesn't seem to be in the spirit of agile. What is more, having to get authorization and capital investment up front in order to act is counter to the principles of empowerment that.

If you want to be agile, don't you have to eat your own dog food? Designing your agile process up front just doesn't sit right. If you are true to agile principles then, surely you want to take an evolutionary approach to designing your methodology.

An alternative is to bootstrap agile practices yourself and make your transition to agile an “inspect and adapt” cycle. Take the iterative approach to software development techniques preached [6][7] and apply them to implementing the process itself. Observe the tenet of evolutionary architecture and let the architecture of the process emerge.

The real advantage in this is that you will learn how to initiate change yourself. You become self reliant, and that will pay dividends in the future when you have to tackle larger organizational issues.

2. What is in a name?

In this paper, I refer to what we were doing as agile. By this, I am broadly referring to the practices and processes described by the agile methods Scrum [7] and XP [6]. However to understand our mind set, put this in the context of a company that is involved in the automotive industry, where analogous concepts under the umbrella term “lean” [2] are part of the landscape, if not part of everyone's actual practices.

Bear in mind also that some of the companies we do business with were the originators of lean. So the driving force behind a lot of what we did came from an awareness of lean. The managing director still refers to what we do as the “Toyota Process” [1][5]. However I have used the term agile in this paper in preference to lean or anything else simply because that is the term the software industry is most familiar with.

3. Pull yourself up by your own bootstraps

Regardless of whether you can implement agile totally in house, or if you look for outside help, it's hard to deny that getting help from outside will accelerate the implementation. However, this does not necessarily contradict the get-up-off-your-own-backside ethos that is central to agile.

If funds are tight and you can't afford training, then just make the first steps by yourself. Once you start getting benefits, there is nothing stopping you from reinvesting money saved or made from agile into further training.

A self funding transformation is one of the key concepts we stole from lean [2]. Ironically, given Toyota's position today and the size of their war chest, Toyota created lean because they did not have to money to invest in expensive equipment [1].

If you have ever worked for a cash strapped company you will know that necessity is the mother of invention. However we learned that it is not the cost saving aspect that important, more the fact that becoming self funding is a significant milestone in the
The only metric is what you produce at the end of every month and how that translates into sales. And don't forget, whether you have started your journey to agile or not, your software team is still costing you the same amount every month.

It all boils down to your first step. If you can successfully make the first step, then everything else will follow. There are many excuses for not acting, many of them political. But we found there was a lot we could do by ourselves before we started to hit political barriers. And showing that you take the initiative to act where ever you can will stand you in very good stead for when you do eventually hit those political barriers. Political barriers are not an excuse for not acting. They are why you need to act now.

I'm not a high-flying consultant. I look with envy at consultants who can swan into a company and turn teams to agile in a few days. I'm probably like you, a member on a team looking to make a difference. And if you are like me, then you will find that what you lack in consultancy skills you can make up with time and persistence. Plus as a company employee, you also have a greater depth of knowledge about how the company operates.

One last point before we dive in. What follows is a personal account, and it may sound like “I did it all”. Of course, any one who has been involved in agile knows that this could not possibly be the case. But in order to let the prose flow a better, I have indulged my ego and told if from my own perspective. Now would be a good a time to thank my fellow journeymen, especially Gareth Owen, Andrew Poon, Jamie Pears, Andy Smith and Sean Akers, and to Rachel Davies for her input at various stages. For offering their comments and helping me knock this script into a presentable paper, another thank you Rachel Davies and Andy Poon. I'll stop there before this turns into an Oscar acceptance speech.

4. Refactor this company!

Our business is engineering consultancy and engineering software. We started in the late eighties. We produced the first version of our current generation engineering analysis product in 1994. Our value proposition is that our software gives the engineering professional the ability to predict the system wide behavior of a transmission system far beyond what was previously possible.

There are about a dozen people working on code in our company, though the departments they belong to and whether they consider themselves to be coders has varied over the course of history, as you will see.

What are our challenges? The automotive industry is global, fast moving, with demanding customers. So, we had a huge backlog of requests compounded by productivity issues.

95% of our turnover is from export, with each region mandating different engineering standards and specifies in their own language. Our domain may be characteristic as being:

1. Complex (A simulation may be modeled by 1000s of interlinked objects)
2. Numerically intensive (An analysis run may takes between several seconds to several days)
3. Esoteric (our teams are littered with PhD s).

Our code needed refactoring: We had legacy code dating back to the nineties, fragmented into many languages. We had a culture of producing flakey prototypes and then expecting them to ship. Formulating tests in our domain is difficult: Mathematical modeling takes years to comprehend due to the emergent behavior in a complex system: “If I knew what the answer was, I wouldn't need a program to tell me!”, so there were plenty of excuses for not specifying precisely and writing automated tests.

Our process needed refactoring: poor communication between departments was typical and conventional techniques such as UML did not help. In our domain, the mathematical equations are the specification, but equations are esoteric and require domain knowledge. We had lead times of up to 18 months: 6 months to produce Matlab prototype, 6 months to produce Fortran DLL and Smalltalk wrapper.
In the past we found that the key to productivity was cross training, but we had no framework for achieving cross training, and only a handful of people were willing to cross train. It often took 2 years to train a programmer in the problem domain, or 2 years to train an engineer how to program.

5. The land that time forgot

So here I was, working for this company again. A decade ago, I had worked when it was a small company struggling to define itself. It seemed our fates were to be inextricably linked. Back in the late 1980s and myself and another colleague had bet our reputations on a new emerging technology known as object orientation.

Our bet paid off – OO proved to be the right place to be. I left to join an OO consulting gig whilst they, after ditching C++ for mainstream development, went on to use Smalltalk and reap the rewards of having a huge technological edge over their competitors. Remember: this is technical computing, an area of computing that seems to be in “the land that time forgot”. They were using Smalltalk. Their competitors were using Fortran. It was a massacre: this is technical computing; an area of computing that seems to be in “the land that time forgot”. They were using Smalltalk. Their competitors were using Fortran. It was a massacre. I left to join an OO consulting gig whilst they went on to reap the rewards of having a huge technological edge over their competitors.

Fast-forward a decade to the year 2001 when I rejoined the company. Things were not so good. The company was becoming a medium sized. Use of OO, once the secret weapon, was now widespread in the industry and no longer a strategic advantage. Assuming a new technological edge would restore their competitive position, the company had embarked on architecture, platform and framework feast that lead to the paralysis of product development.

6. Will the real analysts please step forward

Analysts have had a bad rap in agile. Particularly, the kind of analysts in business systems. I suspect this is because there is really not that much to analyze. In general, in a business system you are not creating something that requires a new solution. You are probably putting together something that has been done before, in an innovative way, or a way that is highly applicable and appropriate to your situation, or in a way that is going to make you lots of money! To me, that sounds more like a design problem than an analysis problem.

In technical computing, it's not quite the same. You don't have analysts in the business analyst sense of the word. If you ask people in a technical computing project who the analysts are, they will point to mathematical analysts. In technical computing, often you are doing things that nobody has done before, and you need analysts to predict what the outcome should look like and then to tell you why you are not getting the same results as you predicted. The mathematical analysts are domain experts. The program is there as an analysis tool to predict the behavior of the physical (or financial) system under scrutiny. As such, the program is written as a tool for the analysts.

An approach I used back in the day was to teach the domain experts a UML-like notation and get the domain experts to do the analysis themselves[10]. Pretty much what the Domain Driven Design people advocate now [11] learned back in the day [10] was to teach the domain experts a UML-like notation and get the domain experts to do the analysis themselves. Pretty much what the Domain Driven Design people advocate now.

So, off I went and tried to teach mathematical analysts how to draw UML diagrams. The results were less than impressive. Mathematics is an extremely rich way of rigorously expressing complex concepts, and the mathematical analysts in our company gave UML a bloody nose.

What to try next? Now the company was largely a Smalltalk shop. I say largely, because, a lot of work was still done in Fortran, and C++ also cropped up in places. And the mathematical analysts were beginning to look towards Matlab as a possible platform for delivering products in (but we won't go there).

Most of the Smalltalkers were feeling like they were getting out of touch with fashionable new technology and longing for the day that they could rewrite the product in Java, the then flavor of the month. You have to remember that around this time, people were trying to follow the exodus out of Smalltalk and into Java, and it didn't help that people like Kent Beck were part of that exodus.

Like I said, the company was at least nominally a Smalltalk shop, and because of this affinity with the
Smalltalk community, there was a copy Kent Beck's book [6] lying around. I picked it up and devoured it. It made so much sense. But how to start applying it? The first step is always the most difficult. Luckily fate has a way of throwing you into situations.

7. What you don't know, you don't know that you don't know

In 2001, I was assigned to work on a simulation project with a mathematical analyst. I'm not sure if it was pure fluke, good judgment or foolish optimism, but with the blessing of our managers, I decided to apply two techniques from the XP [6] – pair programming and test driven development. It worked well, and was seen to work well, and for these reasons:

1. I was seen as the programming expert, so the mathematical analyst did not question my suggestion to use pair programming and test driven development. Besides, he had been struggling by himself with the code for quite some time and was keen to have someone else on the case.

2. It was blatantly obvious that my domain knowledge was zero, and I would never really understand the mathematics behind it, so I really respected the analyst's knowledge.

3. Between us, we had everything we needed to succeed. If we failed it would be our own fault. There was no one we could blame.

4. Because we had everything we needed, we were a self-contained team. There was no way that anybody from the outside could politically sabotage what we were doing. It only needs buy in from two to people to move forward.

The initial work was done in Matlab, which is a mathematical calculation and visualization environment. It is not OO or buzzword compliant in any way. In fact it is a pretty hostile environment for a programmer to be in, at least if you are a programmer who wants to produce well tested, well-factored and robust code.

However, as a symbolic gesture to show you are serious about working with other people, it was just the job. If you show a domain expert that you are willing to work in their environment, they will meet you half way. Anyway, once you have rolled your own testing framework, pretty much any environment is usable, including one like Matlab.

Above I said that “I decide to apply two techniques from XP [6], like I understood all of the 12 XP practices. In reality, I was improvising as I went along. Note also that somehow I managed to create a mini cross-functional team. Again, pure fluke. But what it does show is that you should trust your gut instincts, because they are usually right, even if you don't know why at the time.

And that's the great thing about pair programming. Because it is all about collaboration and communication, it is a microcosm of the agile process. So, if you are bootstrapping your agile implementation, pair programming is a great way to learn about agile and to learn about how to become an agile coach. If you can master an agile team of two, moving up to a larger team becomes much easier.

Another thing it shows is that there is always someone in your organization that is willing to listen to your new ideas. People in other departments who you don't think will be interested often are. In contrast, people in your own department who you think will be interested often aren't.

There is nothing more difficult than tackling an anti-agile software team. Why? Because they are professional programmers – people who are paid to know how best to program. And you are saying to them “I know of a better way of programming”. But if you go to someone who is not a programmer but has a programming problem – and there are a lot of these people – and say “I can help you with that”, then you may get a better reception.

8. That difficult second album

In the music industry, people always talk about “that difficult second album”. And it is the same with implementing agile. After you have taken your first successful step, the next step is often more difficult. It's because the first step is under your control. You will seek out like minded people to work with. In contrast the second step will require you to influence people you cannot control, and often have different and opposing options.

After tasting the success that agile could bring, I wanted more. But what next? Obviously presenting to the managers of the company would be a good idea, wouldn't it? Perhaps it was, but boy was I letting
myself in for it. Some were skeptical, others hostile. Most were indifferent. In that environment, explaining the 12 practices of XP [6] can feel like a marathon.

After the presentation the managing director took me to one side and spoke to me. He was a mechanical engineer, and during the presentation I had made the connection between agile and two practices present in the automotive industry: DFM (Design for Manufacture) and TQM (Total Quality Management). DFM and TQM are part of that general movement in design and manufacture that most of you will have heard of as lean [2]. Even though I hadn't been totally convincing, there was enough there for his mind to make the conceptual jump between DFM/TQM and agile. He told me to go away, work on this idea and not give up. Again, you will always find that there is someone listening, even if it appears that there isn't.

Another thing about management is that, even if they think it is a good idea, they themselves do not know how to act on it. They are not stupid – how do you think they got where they did? But how could they possibly understand agile at a level that they could apply it? They rely on you understand it and implement it. Why do you think they pay you?

Given the frosty reception, you might have expected that an agile planning meeting with the management would have been out of the question. But that didn't stop us from trying. Thinking back, I don't know which was worse – the fact that management didn't feel it was worth spending a couple of hours with programmers to discuss priorities, or that most of the programmers didn't think it was worth doing either. Nevertheless, we persevered with planning meetings until management eventually killed them dead by asking for power point presentations instead.

Why were we failing? The answer was simple. We didn't have the experience. The paradox is that this failure gave us understand that we were able to leverage a few years down the line. The other thing is that timing, and choosing your battles is very important in moving agile forward. And you need to learn that timing by failing as soon as you can.

9. Turning defeat into victory

Failing to get the management on board, we turned our attention inside, and found that there was plenty to be getting on with whist we were waiting for the management to change. In the body, most cells get replaced in a 2-year cycle. The same sort of thing happens to people in companies if you wait long enough.

We are in the business of simulation. Our product was a simulation tool, and as such could be seen as one great big mathematical model. The amount of hard core mathematical code actually present in simulation is actually not that much. Most of the programming effort is spent creating an intermediate representation that a non-mathematical user can understand and manipulate easily. Therefore the programming team was staffed by programmers rather than mathematicians.

Although the quantity of hard-core mathematical code is not that much, it is the life and soul of the program. It was also a part that was programmed by another department. This division of labor was severely hampering our agility. The next challenge was to find some way of integrating people from these disparate departments into a single team focused on product development. We had two things that enabled us to move on this. Firstly, the two departments were in the same room, only separated by a partition. Secondly, it was now 2002 and we had already shown enough results to justify going to learn Scrum with Ken Schwaber[9].

We instigated Daily Scrum meetings [7]. Although we didn't have the political clout to move the mathematicians and programmers into a single department, it was relatively easy to get everyone to agree to meet up every day for a daily stand-up meeting.

Brilliant though the mathematicians were, the code they produced had stability issues. They did not have the luxury of working in a highly productive environment or one that was conducive to testing. These stability issues had a knock on effect on the stability of our entire code base, and that's not something you can live with.

Luckily, we had the unit-testing framework SUnit, another gem that Kent Beck left the Smalltalk community. We produced integration tests for the mathematical code, and published the test results everyday during the daily scrums. This feedback gave the mathematicians the information they needed to help them produce more stable code.

We found it to be cost effective to attend events run by the agile community rather than commercial consultancy and training companies. One of the nice
things was to connect up with other practitioners to sanity check what we were doing.

Another “cheap day out for the family” we did was to take a group of programmers round a tour of the local Toyota plant. Toyota are quite open about what they do, and they organize free tours for the public, and their competitors alike.

10. The code is the documentation

Technical computing has a lot of interesting characteristics. But the thing that really characterizes it is that the complexity of the domain. Given the complexity, it is common practice for domain experts to program the algorithms. Given the complexity, excel doesn't cut it, hence the ubiquity of numerical and visualization packages such as Matlab.

So, it is common to see that the authority on a particular calculation method is actually on your programming team. And when a customer asks how precisely a value is calculated, often what happens is that the numerical analyst will open up the development environment and go and look at the code.

Technical computing is an approximation of reality. Most of the methods we have for approximating reality rely of iteratively improving an approximation. And these methods do not lend themselves well to the kind of acceptance testing found elsewhere in the agile community.

In the past we have had acceptance tests written by another expert not in the programming team. But often the calculations in the acceptance test itself become so complicated that you need a full-blown programming language with iterative solvers in order to express the acceptance test.

This gives you two problems. Firstly differences in approaches that different solvers take lead to the results that the acceptance test gives differing from the results that the implementation gives. Secondly, you now have two code bases to maintain – the acceptance test and the implementation.

A far better approach is to coach mathematical analysts how to write good tests, and have them written in the production environment of choice. Enabling this to happen was another win for pair programming.

11. Feeling good about the code base again

Our Smalltalk code base was over a decade old, and in dire need of refactoring. Bringing mathematicians in had a few effects that we did not anticipate.

Firstly, they really took to Smalltalk. If you add some visualization tools to Smalltalk, then you can perform mathematical experiment in Smalltalk in much the same way that you can in languages like Matlab.

Secondly, they understood the code base very quickly. That's the benefit of having an instinctive understanding of the problem domain.

Thirdly, programming with them was a joy: layers of badly written code that we based on wrong assumptions could be swept away when you have a domain expert pairing with you.

As for rewriting in another language, with Smalltalk you control the whole software stack, so when you are in vertical application like us and you have to roll your own solutions. Smalltalk is still a very good fit.

All this is great, except that legacy code means no unit tests. Ideally, you want your unit tests to detect all problems. In theory, if your unit tests are perfect then you should never get a failing acceptance test. However, we frequently had to rely on acceptance tests to catch error. If we only ran unit tests before check in, then the code base would not be clean enough to ship at short notice. And we wanted to be able to ship the currently checked in code at any time with the certainty that the quality of the code would be at least alpha level. However with poor unit test coverage that would mean running all the acceptance tests before code check in.

The other reason why the traditional divide between unit and acceptance tests was not that useful to us is that, with end users on the programming team programming unit tests, aren't those unit tests then really acceptance tests? And with those end users programming the acceptance tests, sometimes they optimized them so that they ran faster than the unit tests.

With tests that took 3 hours to run, how could we do continuous integration? We were never going to get multiple check-ins per day, but we might be able to keep a clean code base if all the tests were run before check-in. Our solution was to automate the commit process, but not in the way Cruise Control does it. We did it using an automated two phase commit process,
made possible because of the tight integration between Smalltalk and its repository:

1. Code is checked into a branch
2. Repository runs unit & acceptance tests
3. If the tests pass, the repository then moves the branch onto the main line

With acceptance tests always at 100% we never have regressions. This has a great effect on the team dynamics in that a major reason for playing the blame game is removed. Now we have to think a lot harder to find an excuse to blame other team members!

For a while, I struggled with coming to terms with having non-classical XP integration process. But eventually I realized that putting in the quality checks early in the cycle to intercept issues was more important than following the classic XP unit test/acceptance test separation, at least in our field, something that some of my other more practical team mates realized a lot sooner than I did.

Earlier I mentioned that I had pair programmed with the mathematical analysts. A lot more of this followed, and eventually management took the bold step of combining the two teams in 2004. Part of what triggered this change was the relocation of our office at that time, and the unrelated fact that we had a lot of staff turnover during that period. Also, that in 2004, that four of our staff also became Certified Scrum Masters [8] probably contributed to the acceleration.

12. May you live in interesting times

If you are a programmer, then you might feel that commercial people are experts in creating drama from a crisis. If you are a programmer, you may have already built up a knee-jerk defense mechanism for dealing with the crisis that you feel commercial people create. Carefully documented requirements and implementation plans are excellent walls to throw up when a commercial person starts acting out a drama.

In our company, most commercial dramas start when a sales person sells a non-existent feature. I suspect that this kind of thing happens everywhere. As a programmer I often postulated that all a sales person is only concerned about is his sales quota, and selling things, regardless of whether they exist, was a way for them to make their quota. However, one sales person let me into a secret.

He asked: “You know why we sell things that we know we don’t have?”. “No” I replied.

He continued: “It’s because its the only way we have to getting the programming team to do what we want them to do! We want new features, so if we sell it then you have to build it!”

Errr.... That’s good isn’t it? We actually want sales to steer our development, don’t we? That’s part of agile, isn’t it? After all the sales person represents the customer, and it is our job to produce things that the customer wants. And in reality, there is so much software that gets written which does not satisfy this most basic brief.

So, the sales people actually do want to steer the development process, just like they are supposed to in agile [6], they just are not very good at it, and nor are we at very good at being steered.

Forget that the sales person's mode of operation is dysfunctional and remember that our behavior is every bit as dysfunctional. Forget that as a programmer how would up it makes you feel when a sales person forces a deadline on you despite your best efforts to explain to him that you are already overworked.

Inside the sales person's dysfunctional behavior is a bit of communication struggling to get out and make itself heard. And given how far we have come so far, surely it would it not be too difficult to change our own dysfunctional response so that we can start a positive rapport with the sales people.

13. If you build it, they will come

And so, it was that programming team continued to improve their XP engineering practices. The failed attempts at planning meetings were a distant memory. We heard nothing from the sales team, and then suddenly, out of the blue they did it again. They sold something that did not yet exist and to make matters worse, they had promised delivery in one months time.

To address the crisis, and create enough drama, the business director called a panic meeting. Given that in the past this particular program had been a poison chalice and a black hole for resources, there was not surprisingly an air of trepidation about the meeting. However there was also a feeling on our part that this could be a historic opportunity to move things forward and finally lay to rest the ghosts of the past.
The sales people did not realize it, but we were about to turn their panic meeting into an iteration planning meeting. We took the program and scoped the functionality down to a month. We then promised to demonstrate something in exactly one month's time, implicitly setting an iteration length of one month. Off we went for one month, and exactly one month latter we meet up with the sales people again and demonstrated the functionality. They were happy with the progress, but were looking forward to seeing more functionality. So, we agreed to do what they requested and meet again next month. And so it came to pass that henceforth every month there was an iteration planning meeting with the commercial people.

Our operations manager was a bit concerned that our iteration planning meetings were poorly organized. Of course he was right – after all we had never done it successfully before. He was worried that it may come across as unprofessional and that the commercial people would react badly to the lack of professionalism. But things improved gradually, and now we are almost professional about it, and so far the commercial people are yet to walk out in disgust!

This happened in 2005, and latter on that year the number of Certified Scrum Masters on our team went up to 9, and we held the first retrospective run by an outside facilitator.

Once the monthly cycle time was chosen, we didn't change it because it seemed to fit in with the rest of the organization so well. Our sales force travels all over the world and we can only round them up once a month. We have toyed with the idea of introducing an internal one or two week cycle within the monthly cycle, but we feel that may result in a local-optimization of our process.

Along a similar vein, our story writing is not very refined. Well-phrased highly granular stories all of the same size would make the job of the programmers a lot easier. However, this approach is not optimal for our whole organization as it is at the moment. Writing good stories requires a lot of skill, and I would rather have poorly written stores written by a sales person than beautifully written stories written by a programmer. One day this may change, but this is where we are at right now.

14. Explaining the offside rule

Implementing planning has been the most difficult thing we have done. And we have found through bitter experience that almost every practice we tried failed first time around. So, we learned not to implement too much too soon. It is best to first try the simplest thing that could possibly work. And when it is seen to break, then use that opportunity to introduce something new to fix it.

An example of this is the use of ideal-days. When you are just getting used to planning, it is best not to introduce too many concepts at once. Working out your capacity is easiest if you work in real-days. After a few iterations, you will feel more comfortable with planning and you will start to see that working in real days has its disadvantages. At this point, you can introduce ideal-days as a concept that helps estimation and capacity planning. This way you can avoid much of the philosophical arguments that tend to surface when you try and explain the advanced rules of the planning game to an inexperienced team.

Similarly, we let the stakeholders in the planning meeting discover for themselves the best way of organizing their stories. When planning an iteration with the stakeholders, the simplest solution was to work on a white-board, so that's what we did for a year. After a year, the sales people realized that white boards get erased (the simplest things are often the most difficult to realize!). So to make the stories more permanent they re-invented story cards. Sometimes it doesn't matter how you get there, so long as you get there in the end. Our story cards are magnetic, like great big fridge magnets. They are very tactile, and you can hurl them against the magnetic planning board and watch them stick. They are built to survive the most hectic planning session, sessions that would lay to waste mere index cards or post-it notes.

15. This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

There are many pros and cons in implementing agile yourself. You don't need capital, but you do need time. You will fail a lot, but you will learn how to initiate change yourself. You will get a sense of achievement a lot of the time, but you will feel frustrated the rest of the time. You are not dependent on consultants, but you do need staff to work for long-term goals.

It’s now 2006. In the last 5 years, we have gone a long way. I haven't got any metrics for you other than this: in the last two years revenue from software sales
has grown 100% per annum. Agile is now seen by our company as the only sensible way to organize programming teams. We wouldn't know where to start if we wanted to do it any other way. We are starting to have problems that stem from having to scale the team, but these are problems that it is good to have: After all, we are only scaling the team because we want spread the success out to more areas.

The managing director recently asked what we were going to do next. I said “If I knew what to do next, we would be doing it already!”.

I don't think he realized how literally we had been bootstrapping ourselves up using the “inspect and adapt” cycle. He still thought there was a master plan hidden somewhere that we were working to!

There is plenty to still do both within software development and elsewhere in the company. Toyota, the founder of lean and the inspiration for much of what we have done, spent decades refining their production system, and we in our own humble way will continue to refine ours. There are plenty of software engineering and process issues that need attention. What excites us though is to see how far agile can be pushed beyond the bounds of programming into the realm of a general purpose management practice in the service orientated areas of our company.

16. References


