A Story About Dinosaur Called Mainframe and a Small Fly Agile

Zuzana Šochová
Prague, Czech Republic
zuzi@sochova.com, http://sochova.com

Eduard Kunce,
CA Technologies
Prague, Czech Republic
eda@kunce.net, http://kunce.net

Abstract—Once, long time ago, there was a very old dinosaur living on the planet Earth. Big yet invisible to most, living in the dark places full of cold, dust and cables; surrounded by several groups of faithful - very specialized software teams, the only ones who truly understood him. But the life on the planet Earth had never been steady and recently spinning faster and faster around, so it happen that even this group had to be changed and follow the fast changing business environment outside, unless they all die. And here our story begins.

The story is based on our experiences from multiple companies and projects operating on high-performing applications environment - mainframes. The companies are mainly from banking services, insurance systems and other very specific software industry areas where mainframes are still very popular.

Keywords - Agile, mainframe, Scrum, enterprise, adoption, development, testing, culture, high-performing applications.

I. INTRODUCTION

This short paper is a real story, an experience report about adopting agile principles on a huge, conservative, and inflexible environment of high-performing mainframe applications. Usually, those applications are designed for critical bank and insurance projects, supporting airport infrastructures, security governmental processes and many others; mainframes are the backbone of many Fortune 500 companies. They have different programming languages than most of the IT world using, different architecture tools and processes. Nothing which could be possibly called as ‘flexible, light or dynamic’.

But the business is not asking if we like the flexibility, it just demands it. And as the demand for such process becomes stronger, we were standing in front of the decision to either keep what we have and tell our customers that the change is not possible, or to go for a change and try to adopt Agile and Lean principles. Seemed like an illusion at first, but at the end, we had succeeded.

A. Mainframes, What?

Mainframes were supposed to be gone many years by now, but we still see them around. Silently running in the background and processing data in the banks, automotive industry and many other fields. Of course there were attempts to remove mainframes from the business, but most of these failed due to the of lack of reliability, stability and scalability in the non-mainframe world.

When dealing with mainframes, we have to realize that we deal with a platform that is on the market since 1950’s and one of the key factors is backwards compatibility. Yes, we talk about full backward compatibility to 1970’s so all applications written in these years will still fully run on the newest and greatest hardware being produced today. As we will see in the following chapters, this and some other platform specifics are great from the customer point of view, but create a very high demand on the knowledge and skills of the application programmers.

B. Starting Agile Transformation

Mainframe development has never been easy, universities do not teach about mainframe as a platform and using languages such as assembler and COBOL aligned with a yearlong release cycles do not support agility out of the box. We would argue though that such conservative environments are ideal places to implement agile practices and achieve an immediate impact on the quality, productivity, and flexibility. In addition, nowadays the mainframe companies are facing a challenge of exchanging generations of developers so new processes and techniques are being introduced to this stringent environment just in time.

Such transformation brings unique challenges, and not only the technical ones. For example, changing the functionality according the customer’s needs, likely interfere with the final product in more significant manner than anywhere else. The architecture is much harder to refactor than we are used to in usual application platforms and so even the small customer request may end up in huge effort of the whole team. There are other challenges too, such as facing tons of legacy code and working with huge amounts of data. Both come with testing challenges as well. Last but not least, you only have a limited set of tools at hand as on mainframes you can’t just google some public solution to your problem. There is no huge community of developers working on the same platform. In most cases, you are just fully dependent on IBM and their willingness to provide some useful tool or extensions of existing instrument. In other case you can write the tool yourself.
II. ISSUES WE WERE FACING

A. Top Five Issues

When you enter the high-performing application environment for the first time, seems like just another huge legacy code environment, operating in mission-critical segment. Nothing really special or new on first view, we did it already many times and it can’t be so difficult to adopt agile here either. However, when you start getting closer, very fast you see there are some unique obstacles. Something you were never facing in any of those conservative companies – huge legacy code, lack of useful tools, extremely hard estimates, very specific customers and huge expectations of team members. So let’s see now in detail how different mainframes are.

HOW WE STARTED

At first, the situation seems to be rather depressing. Only a limited set of our experience seemed to be applicable to mainframe teams, although we had experienced big migration of legacy code to the more modern technologies. Given all the constraints we had to go to the real basics of agile development and look at the most basic best practices in a very flexible way. After a while we realized it’s quite logical, isn’t it? To change the extremely inflexible environment, you have to be ready to implement core practices which allow you such change and help you to achieve a workable state fast.

B. Issue #1: Huge Legacy Code

One of the first obstacle we had met on our way was the extremely huge mainframe legacy codebase. It doesn’t seemed so bad, until you realize there are around 14 million lines in assembler in one product, and apparently no one can say what the code is really up to. No classes, no objects, no procedures or methods. Code which is full of the optimization trick instructions which are not documented in any official manual and are not supported either. There is usually limited knowledge within teams dealing with the product about the legacy code, and as the knowhow was just remembered in people’s heads, there is very often no support from the group who wrote, produced or sold the initial system either. Non-existing coding standards, very limited documentation, it’s all in the people’s memories or hard-coded in the source code itself. Just line after line there are system instructions and macro calls, one after another, thousand after thousand, million after million. Not only the assembler instructions are difficult to grasp, but also the calls to thousands of system macros. If some of the engineers within the team or the company still remember the program logic, it’s good for the team. If not, you can start reading the code from the scratch and hope to understand the memory management, paging and segmentation. Simply put, you have a processor (called your brain) in your head that compiles the code on the fly and tries to guess what the instructions do on the real system - not very effective way to progress with your programs.

In addition to such difficult environmental issues, in many cases it is even worse. To understand what you truly see when you join such a team, is not only difficult and hard to read assembler code, but even very, very old code, which had already survived several attempts to refactor or migrate it to some newer version of addressing, or system macros support. Most of the source code was created more than 25 years ago, and many teams were extending or adapting its functionality or just trying to understand it. In addition, the source code of many modules was lost, so having only compiled versions there was no possibility of changing their behavior. But surprisingly it’s still in use in many organizations, as the internal parts of their systems.

WHAT WE DID ABOUT IT?

First realization was that the knowledge and best practices need to come from the team itself. While in non-mainframe projects you as “outsider” can grasp with some experience the situation quickly, in mainframe world this is close to impossible. We had to rely mostly on the existing team members to share the best practices and become the immediate source of knowledge and improvements [2]. The strongest aspect of our role and involvement in this process was facilitation of the discussion and making sure it finds its way to the successful end. However it was not just about the communication among the team members, sharing experiences and supporting of the learning environment. In order to get team together and cooperate, we had to make sure they have the common goal. Consequently, that they are all linked to the agile adoption process and have a strong feeling that the process relies on them and won’t be successful without their strong support. In the complex environments as the one described above, we were forced to extensively rely on the experienced developers and system administrators to help us to create practices that would be fitting into the environment and would be accepted across the teams.

Lack of generally shared knowledge is one of the obstacles which could be overcome by just placing the team in one room and making a team out of individual developers. Without any special internal training about the specific areas, the cooperation makes the job for us. People start learning from each other and share their knowhow very fast. It was not easy: we had to overcome a number of resistances hidden in the people’s minds, and suppress the conservatism. As you can imagine, our way to the collaborative teams was really

![](image.png)
long. The team members had to leave their desks, where they have been used to sit for more than 20 years (in some cases), and start cooperating, relying on and trusting each other. And that’s even harder in the environment which is highly fragmented and hard to understand. But we have decided to go even further on the cooperation front and implement pair programming [1]. Apparently, as the time went, this was the number one decision we had made during the whole transition project. The pairs were rotating every other sprint, so all team members involved got more used to their other colleagues and step after step started to learn even the so far hidden areas of the software and technology.

C. Issue #2: Lack of Tools

Another difficult aspect of such environment is critical lack of development and system tools the teams and individuals could use. Most common example is source code repository. There is one called Software Configuration and Library Manager - SCLM. It’s very easy to assume that there is at least some easy to understand, effective tool that everybody can understand quickly. Well, not quite. For those who are used to SVN, GIT, TFS or similar advanced tool, this proved to be a really difficult step. The SCLM is a server based solution which provides just very limited (close to none, really) opportunity for team to collaborate at one source code file or module. When you start working on a file or module and lock it, nobody else is able to do that and collaborate on the same piece of software. This implies serialization of development work as well as introduction of many dependencies. No support of merging, no support for branching; so if you need to have a concept of branches, you end up with multiple copies of the sources in completely separate project and manual merges between these. As an outcome, the team has to plan precisely to avoid conflicts not only within the team but among other teams as well.

So to move on, when you finally code some functionality, you are used to test your feature. Well, many times there is almost no test automation and it is again lack of tools, not the laziness of the developers. You can work around the situation with some very limited amount of batch tests; however these usually have no description, nor consistent test design. They just test some random scenarios which were accidentally reported by some important customer years ago, but there is no real test description, only the one based on people’s memories. Eventually you can extend some of those tests, but still, there is huge technology limitation. There are no tools which could be used to generate automated tests, no framework how to write unit tests and run them over tons of assembler code.

Besides these facts, teams are living in the world, which is extremely hard to understand or test by any automated test. When you already identify any problem, the environment provides only very limited debugging options. There are some tools which could read system dumps, however such tools bring along huge lead time before a debugging session can be started. To set up such environment for the first cycle takes several hours, and then every round, you had to prepare the individual run for about 30 minutes. That’s indeed nothing flexible, nothing dynamic.

TESTING IS STILL A PROBLEM

Despite of the enormous teams’ effort and enthusiasm, not all the problems were solved. The lack of consistent test strategy had become bigger problem than we had expected at some projects. The 30+ years old bunch of not documented tests seemed to be bigger problem than we had expected. Even the 2 engineers assigned full time for one year haven’t been able to consolidate such complex test set. However, they had created a reasonable minimum test sets in QC, which teams now can extend step by step as a part of their daily job. Yet the ideal test coverage is not, despite all of these activities, anywhere closer. The second issue of low number automated tests was addressed by the implementation of a Selenium based testing framework. Nonetheless, most of the functionality is technically untestable by such tool anyway, but even a small automated area brought high satisfaction and value to the team.

Regarding the other tools, the team had to get used to it. The limitation of parallel work had been overcome by good communication and careful Sprint planning.

D. Issue #3: Estimates

The third surprise we’ve been facing in most of the teams was rather poor estimates. In the mainframe world, there is an extreme risk coming from on the environmental problems described in the first chapter. Teams including even their most experiences programmers are facing the problem that every small change requires hours of research. This is indeed a very common obstacle identified by most software teams working on large projects, especially in control environment with huge legacy code. However, the mainframes are again very special in this case. While in the non-mainframe world technicians are able to grasp sizing of stories relatively quickly, in mainframe you need years of experience to be able to do so.

Even some simple change is facing a challenge of fitting into memory and instruction paging. That’s something which is almost impossible to predict. Sometimes you are fine and fit in; in other cases you are forced to rewrite many code segments, which is extremely difficult and obviously risky activity.

Another obstacle and a very common situation is that the teams are forced to work in silos and depend on one expert in a given area of code or module. Due to the code complexity and large upfront knowledge required, the analysis and breaking stories into the tasks is usually performed only by experts in specific code area, as there is insufficient knowledge in other parts of the team to really support them. The problem gets worse, as the generation of experienced team members is slowly retiring, and the gap between them and the young men starting their career right is only growing.

COMBINE STORY POINTS AND T-SHIRT SIZES

To address the difficulties with the estimation process, we had asked the teams to put aside a regular time for backlog grooming, where the teams were going through the top of backlog stories in order to identify whether they had enough information to plan a story into a sprint. The initial
sizing of the backlog was done using the T-shirt sizes and later the story points for sprint plans. After some initial period, such practice brought us to a sufficient level of certainty about what is going to be delivered by the next release date. It was in contrary to the other mentioned problems quite easy and straightforward change, which brought us high value very fast.

E. Issue #4: Customer Side

Due to used technology, a very easy change in the Java world can cause a massive rewrite and changes in the mainframe subsystems. One of the most common problems are linked to the memory management and the large amounts of legacy code which is hard to learn, even harder to understand and almost impossible to refactor. The traditional User Stories based on functionality and business value are very often intangible for the mainframe team. Again, nothing new in large legacy projects in the corporate world; however at mainframes we are facing even more significant problem than anywhere else. For the team it is hard to match the solution to business case. They still need an extensive analysis of the existing solution and new functionality to be able to accept it and start working on it. Needless to say this upfront effort is significantly higher than in any other, more traditional, complex environment.

Customers on the other side do not understand any of mainframe specifics, so they are staying far away, without any connection to the team. They know the mainframes are supposed to be reliable. Thus not always understand there is a downside of such solution in lower flexibility, and sometimes higher cost. And consequently can’t imagine the complexity of the solution, environmental obstacles and low know-how burden. Furthermore, they are generally very reluctant to adopt new releases quickly and therefore run late on the upgrades, waiting for others to solve all the issues and bugs with migration.

**CAN WE USE AGILE FOR SUCH CUSTOMERS?**

First, we started with the realization that having a release every sprint is only a dream. Well, honestly, many companies are not able to release every sprint, but in the non-mainframe environment you are usually able to prepare the release quickly, if there is a customer demand. The mainframe world is a bit different again. Not only the teams would not be able to produce a full and stable release every sprint, but the very conservative customers would never ever accept the releases that frequent. Even small, relatively isolated modifications are seen as extreme risk that no one is willing to accept. This is obviously not very surprising in banking industry, automotive or airport operations, where every minute of system downtime cost millions of dollars / euros. The most accepted timeframe for a product release (based on our experience with multiple teams) is about one year. But that is only to the testing environments, getting the products to the production environments usually takes between 1-3 years in the better case. However, using agile fast iterations we were able to get the PSI available to the customers about 3 times per year. Through their testing environments we gathered better quality feedback on how the final product should be.

In addition we had to overcome the huge gap between the customer and the team. Even when we put in place Product Owner role, he was still quite far away from the team. Often team has no direct access to the necessary business background. They still didn’t understand the high-level User Stories and the discussion between the team and Product Owners was frustrating. Nonetheless, after some time of struggling and trying to improve the communication between team and Product Owner, we found out, we just need a role of Product Owner Proxy who would be closer to the team and overcome the growing gap.

F. Issue #5: Team

And last but not least, there is another obstacle – the team. The learning curve in the mainframe world is not very steep and only a limited number of young engineers stay on the path long enough to make a real impact. It is very common that it takes between 3-5 years to learn the basics of mainframe development and administration. So while in the more traditional world you would be senior in 5 years, in the mainframe world you are still considered very junior. The original generation that grew on mainframes is slowly retiring and younger ones are not that interested to learn the technology. The university environment typically doesn’t help here either. There are no mainframe curricula in the majority of schools.

Development on mainframe is unlike anything else the average developer ever experienced. Applications are so complex, that to become familiar with even a small part takes long time. On mainframes, the experience is what matters the most. Even the young bright engineer could never been better than engineer with 20 years of experience on mainframe - a situation that is unlikely to be seen anywhere else, especially in Java world.

However, nothing is only black and there are some benefits as well. After few years of working on mainframe, the engineers usually come to the realization that mainframes are extremely stable, very reliable, and engaging. There are no grey zones, no unpredictable behavior of the system, no surprises and you always know what’s going on. You can always tell why there was an infrequent delay in data processing or slowdown in the system response. The mainframe, besides all its complexities, becomes their platform of choice, fast, reliable and dependable.

**INVOLVE PEOPLE IN THE PROCESS**

Once the cooperating in the team was increased and estimation process fixed, we started to make sure the teams are gaining not only the technical know-how, but they are also connected to the Product Owners and customers as well. As the Product Owner was not always sitting together with the team, we have started to promote the idea of a Product Owner proxy. Simply someone, who would be in a daily contact with the team and is able to answer the product related questions without a direct involvement of the ‘real’ Product Owner. The Product Owners than gained more time to focus on the customers and their involvement in the whole
process, while Product Owner proxy was instantly available to the team. These changes were introduced quite recently and results of these changes are yet to be seen in the coming months.

Last but not least our progress and implemented changes were more than usually based on people’s willingness to support the change and to find unique solutions to their specific issues. The effective retrospective really helped in this area. Teams got frequent regular opportunity to identify the biggest issues and were directly involved in the solution brainstorming process. Many of the solutions were rare and fully unique as there are not many similar teams around and no published theories were really matching their problems. For example, if you need to generate a PDF in a Java world, you can search the internet for a component or a piece of code you can call or reuse. On mainframes you are just on your own. There is no prepared functionality like that. You have to go back to the initial PDF format description and write if from the scratch, byte by byte. Another example would be the access to the real machine physical layer. On mainframes, you are faced with double layered virtualization. So even a simple ask for number of processors, you can’t be really sure what you get back. Multiple factors come to play and if you ask a second later, you can get a completely different number, because, for example, some parts of the system could be hidden from security reasons, some are shared among all. The question of free memory doesn’t make any sense either. All processes can get all memory they ask for, virtually there is no limit.

As the quality had never been big issue in our products, it’s quite hard to measure any improvement. The number of problems reported is the same; however most of the bugs are solved right away, which is more convenient for the team then the previous process. We believe we improved customer satisfaction, and keep the functionality. We definitely improved the team health. But that’s intangible and hard to measure. However, there is one area where the improvement was outstanding. That’s the learning curve of new team members. Using agile we’ve been able to speed up new teams 30% faster than on traditional projects. After some time we have learnt from the retrospectives that we succeeded in our efforts to make the team motivated, focused on improvements and support of each other even between teams. Nowadays, the teams are actively spending their time in discovering of better usage of the existing tools and even creating new extensions to make their work more efficient. “Nobody gets a gold star for being agile; the goal is to get better, not to become agile.”[3]

III. SUMMARY

To summarize our experiences with high-performing mainframe applications, we concluded in the outcome that agile environment is flexible enough to embrace even such conservative and complex environment as mainframes. For all our projects, using agile principles was exceptionally beneficial for both teams and customers. The way to such conclusion was not easy, and included a number of tailoring and customization of agile processes. The first building block was, as in any other environment, to build a good team. But the knowledge sharing rate and team learning curve was much flatter here. The huge amount of the legacy code base did not make it any easy. We had to rely more on the expert programmers and senior engineers’ expertise. The involvement of the whole team into the adoption process was crucial. The lack of useful tools forced them to adapt the individual daily development practices to fit the agile culture [4] to mainframe environment and even adjust or develop some tools by themselves. Using pair programming steepened the learning curve and secured the expected quality. In the second step we had implemented a new role of Product Owner proxy, who finally arched the growing gap between the Product Group and the development team. Finally the functional retrospective boosted high team involvement and made the last step to our success.

We have learnt our lesson here. While we thought, that after years spent in the software industry and agile environments, we have seen and experienced many situations to be able to generalize a solution for “every” problem, we couldn’t have been more wrong here... we were challenged a lot to learn, adopt and think about the issues from completely different perspectives. But we have also met with many dedicated and highly skilled engineers who work on a platform that runs reliably in the background of many of our daily activities.

The success of the changes would not be possible without the strong will of the engineering teams to change and make their lives easier and better. The will was exceptionally strong compared to the other teams we have met. Possibly the difficult environment and hard learning requirements help to shape the will, commitment and persistence needed for every adoption of every new change in the process.

IV. ACKNOWLEDGMENT

Our best thanks to all teams we had been part of or involved in, without their enthusiasm, help and active support we won’t be able the achieve any deep understanding and learning in this complex mainframe and high-performing applications environment.

V. REFERENCES