An Agile Driven Architecture Modernization to a Model-Driven Development Solution

An industrial experience report

Mina Boström Nakićenović
SunGard Front Arena
Stockholm, Sweden
email: mina.bostrom@sungard.com

Abstract–**This paper concerns model-driven development (MDD) used in time critical development. We present an agile MDD process developed in consideration of lean and agile development principles and we show its application to the evolutionary development of a real world application supplied to the banking sector. Our approach involves a novel use of concurrent reverse and forward engineering and through our industrial report we are able to provide strong support in favor of the claim that MDD and agile practices can be used together, preserving the benefits of each.**

Keywords-agile; lean; MDD; TDD; reengineering; finance

I. INTRODUCTION

In the world of rapid software development, commercial software companies have to respond quickly to the challenges of volatile business environments in order to achieve a fast time-to-market delivery, necessary for surviving on a tough business market [2]. The incoming requirement changes can concern business functionality or technology or both aspects, demanding adjustments and improvements in the existing systems. Adaptations to the frequent business requirements changes can be fulfilled either through the evolution of existing software systems or through the development of new software systems. The direction and quality of the system evolution is steered by three main drivers: system architecture, organizational structure and development process. System evolution often requires adjustments in all mentioned areas. Therefore, the software systems architecture together with the company's organizational structure and the established development process should constantly be adjusted.

Agile software development techniques have been established in order to help organizations both to evolve and to develop software systems, accelerating delivery time while still maintaining, or even improving, product quality [3]. Many companies have started using the agile techniques to a less or larger extent. Important questions that are constantly rising are: how to combine agile techniques with some other, already existing techniques and methodologies? Agile principles present general ideas and recommendations, but they have not been elaborated enough to be specific on how to work in a particular environment. With the acceptance of agile techniques, the agile principles are also adapting to the different organizations, working environments and methodologies [4]. There are a lot of empirical studies on the agile principles applied on different methodologies, but there is still a need for more empirical results within certain areas. One such area is the application of agile techniques in a Model-Driven Development (MDD) environment. The agile methods and the MDD have appeared separately and evolved on distinct paths, although they address, to a certain extent, the same goals: making systems less sensitive to frequent changes and an accelerated development. Generally speaking, the agile techniques mostly address methodological aspects while the MDD approach is more concerned with architectural issues [5]. Therefore, it became interesting to combine these two approaches in order to get a rapid acceleration of the system development.

This paper, being an extended version of [1], is an industrial experience report that describes an architectural modernization process of an existing system. Despite the fact that the system's architecture is going to be radically improved in the future, there was a need to find an intermediate solution, within a short time-frame, which would both eliminate the existing architectural errors, such as data duplication and system inconsistency, and reshape the system to be less vulnerable to the modifications. Therefore, the existing system was supposed to be transited to MDD, but within a short implementation timeframe as a main requirement. Hence, the main aim of the paper is to answer the following questions:

- How agile and lean principles can help the decision making process when producing a MDD solution within a short time frame?
- How the reengineering process to the MDD solution can be accelerated, fitting the given time frame?

The paper is organized as follows: after the introduction, an overview of the agile and lean techniques is presented in Section II. Section III introduces the Model-Driven Development concept discussing its pros and cons. Section IV describes the problem in details. Section V explains the

architectures of both the present and the long-term solution as well as it introduces reasons for having an intermediate solution, which is separately presented in Section VI. The produced intermediate solution, an Agile MDD approach, is presented in Section VII, while the development process is explained in Section VIII. In Section IX are discussed all benefits of applying agile and lean principles on the MDD. Section X presents the related work. Finally we conclude the paper in Section XI.

II. AGILE AND LEAN TECHNIQUES

Methods of agile software development constitute a set of practices for software development that have been created by experienced practitioners [6]. The main aim of the agile methodologies is to develop qualitative and no cost- effective solutions and deliver them quickly. The core of the agile philosophy is expressed in the agile manifesto, consisting of basic agile principles [3]. The manifesto states that the software development should focus on the following:

- Responding on change over following plan
- Working software over comprehensive documentation
- Individuals and interactions over processes and tools
- Customer collaboration over contract negotiation

If the software development is presupposed on the listed postulates, it can result in fast and inexpensive software that satisfies the customer's needs. Agile practices and recommendations give us answers how to apply the mentioned core values on the development process. The suggested development cycles should be iterative and based on building small parts of the systems, which are tested and integrated constantly. Continuous integration, verification and validation are some of the agile practices that help the organization to check that they are building product in a right way and that the right product is built. The organization should also be arranged to support an efficient, agile development. Agile organizational patterns help in creating a highly effective organization [7]. They concern both the organization of different teams (company management, product management, architects, developers, and test) and the way how people should work within these teams. Some of the most frequently applied organizational agile practices are:

- "Self-selecting teams": The best architectures, requirements and designs emerge from self-selecting teams.
- "Conway's Law": An organization should be compatible with the product architecture and the development should follow the organizational structure.

After one decade of the agile methodologies adoption, empirical studies showed that the best effect is achieved when the agile methodologies are applied on the smaller organizations and projects [6], [8], [9]. Extreme programming (XP), as one of the agile methodologies, is most suitable for single projects, developed and maintained by a single team [10]. For the larger projects and bigger organizations some other methodologies are more suitable.

A lean software development is an adaptation of principles from lean production and, in particular, the Toyota production system to software development. It is based on the seven principles: eliminate waste, amplify learning, decide as late as possible, deliver as fast as possible, empower the team, build integrity and see the whole. The management decisions should be based on a long-term philosophy, even at the expense of short-term financial goals [11]. The decisions should be made slowly, but implemented rapidly. Work load should be limited and systems should be pulled to avoid overproduction. The lean philosophy is more suitable for bigger organizations and larger projects.

Nowadays practice shows that the best effect is achieved when lean and agile practices are combined together. Although it can seem that some of the agile and lean practices are in contradiction, they are not. At the first sight the agile philosophy could be interpreted as a short-term approach, since it says "do not build for tomorrow", while lean is more a long-term approach. But these two approaches are not contradictory; on the contrary, they are complementing each other. One of the main lean postulates is "decide as late as possible", which is another way of prevention for "building for tomorrow". To conclude, both agile and lean principles could be applied together, but to which extent is decided by the type of organization and the type of the project.

III. MODEL-DRIVEN DEVELOPMENT

Model-Driven Development provides an open, vendor-neutral approach to the challenge of business and technology change [12]. This approach makes, on the system architecture level, a flexible system that can respond quickly on frequent changes both in technology and in business requirements. The main goals of the MDD concept are:

- Simplification and formalization of the various activities and tasks that comprise the software system life cycle, through the raised level of abstraction at which the software is developed and evolved.
- Accelerated development, which is achieved by the centralized architecture and automatic generations.
- Separations of concerns both on technical and business aspects, making the system architecture flexible for the changes.

The MDD's intent is to improve software quality, reduce complexity and improve reuse through the work at the higher levels of abstractions cleared from the unnecessary details. Prominent among the MDD initiatives is OMG's Model-Driven Architecture (MDA) in which software development consists of series of model transformation steps, which starts with a high level specification using often a domain-specific language (DSL), specific for the certain domain, and which ends with a platform-specific models describing how the system should be implemented on certain platforms [13]. MDA standard defines different model categories:

- Computation Independent Model (CIM), representing the problem domain.
- Platform Independent Model (PIM), representing the solution domain without platform specific details.
- Platform Specific Model (PSM), representing the solution domain with platform specific details.

The division could be done even more granular so that the PIM splits in Architecture Independent Model (AIM) and Architecture Specific Model (ASM). Then the PSM is derived from the ASM [14]. All mentioned divisions provide a good separation of concerns. Working with different types of models, representing different views and aspects of the system, enables an easier understanding of complex systems. MDD is intended for the realm of large, distributed industrial software development and is one approach for solving the software life-cycle development problem. On the other side, the detailed separation of concerns can introduce some other problems in the system. It can cause an additional complexity, requiring the existence of several models describing the same thing, just on the different abstraction levels or from the different point of views. Therefore, it is questionably if the MDD reduces the complexity or it just moves the complexity elsewhere in the development process [15].

Development processes based on the MDA are not widely used today because they are considered as heavy-weight processes, which cannot deliver small pieces of software incrementally [16]. That is why there is a need to rework a MDA to a lighter process, easier to the acceptance. For example, this could be achieved by the introduction of agility in the MDD philosophy.

A. *Agilility in Model-Driven Development.*

A common goal for the MDD and the agile methodologies is to build systems, which can respond quickly on the frequent changes. These two methodologies have different approaches for resolving the mentioned requirement: agile development concentrates on individual software products, while MDD is concerned with product lines, i.e., mass-produced software. Agility mostly addresses methodological aspects while the MDD approach is more concerned with architectural issues [5].

The MDD concept has some drawbacks, which do not suit agile philosophy. Looking from the agile perspective, systems should be built in an incremental way where the small pieces of software are delivered constantly. In contradiction to the MDA modeling's starting curve, which can take a long time before the deliverables are produced. True domain-specific languages are not very agile because they encode commonalities and variations in a narrow, concrete expression of the business form [17]. DSL makes the system being too specific decreasing a possibility to respond to the business changes quickly. If the domain evolves, then the language must evolve with it, otherwise the previously written code becomes obsolete. MDA systems usually become complex while agile claims that "simplicity is essential". People are not an explicit feature in MDD while agile postulates that people and interactions should be over process and tools.

[18] distinguished generative MDD and agile MDD. Generative MDD, epitomized OMG's MDA, is based on the idea that people use very sophisticated modeling tools to create a very sophisticated models that they can automatically transform with those tools to reflect the realities of various deployment platform. [19] proposes the agile MDD, where the agile modeling is used. Agile modeling is practices-based and consists of collection of values, principles and practices. Agile models are models that are barely good enough, where the fundamental challenge with "just barely good enough" is that it is situational and therefore, the most efficient.

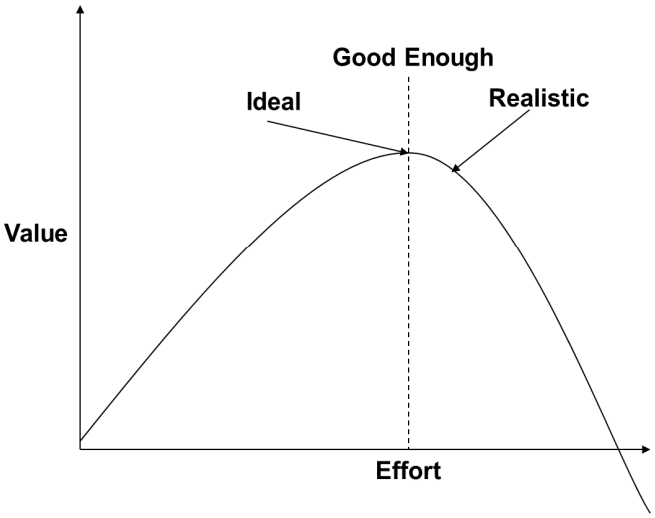

Figure 1 Agile modeling. Adapted from [19]

The main idea with the agile modeling is not to follow strictly the MDA recommendations regarding tools and development environment but to choose ones which fit best the current project and the organizational structure.

IV. PROBLEM DESCRIPTION

A. *Background*

SunGard is a large, global financial services software company. The company provides software and processing solutions for financial services. It serves more than 25000 customers in more than 70 countries. SunGard Financial Systems provides mission-critical software and IT services to institutions in virtually every segment of the financial services industry. We offer solutions for banks, capital markets, corporations, trading, investment banking, etc. [20]. In several areas SunGard is one of the leading providers for the financial solutions and products. Since the finance industry is very though, staying on top of the competitive financial market requires fast delivery, reduction of costs and quick responding to the changes in dynamic market conditions. In order to achieve this, our company has started

adopting agile methods and techniques. The management's decision was to introduce agile software development within each team and on every project. Although many teams have changed its way of working towards the agile development practices, the company is still learning and finding out how to apply the agile techniques on the existing projects and on the existing methodologies.

A software product family, which is developed in the company, is called a Front Arena system and it includes functionality for order management and deal capture for instruments traded on electronic exchanges i.e., markets. Market access is based on a client-server architecture. The clients for market access include the Front Arena applications, while the market servers, called an Arena Market Servers (AMS) provide services such as supplying market trading information, entering or deleting orders and reporting trades for a market.

Clients and AMS components communicate using an internal financial message protocol for transaction handling, called Transaction Network Protocol (TNP) and built on top of TCP/IP. The TNP protocol uses its own messages, which contain TNP message records with fields [21]. TNP messages represent financial transactions like "enter order", "modify trade", etc. The TNP messages have a hierarchical structure. One example of the TNP message, used for modifying order transaction, is presented on the Figure 2.

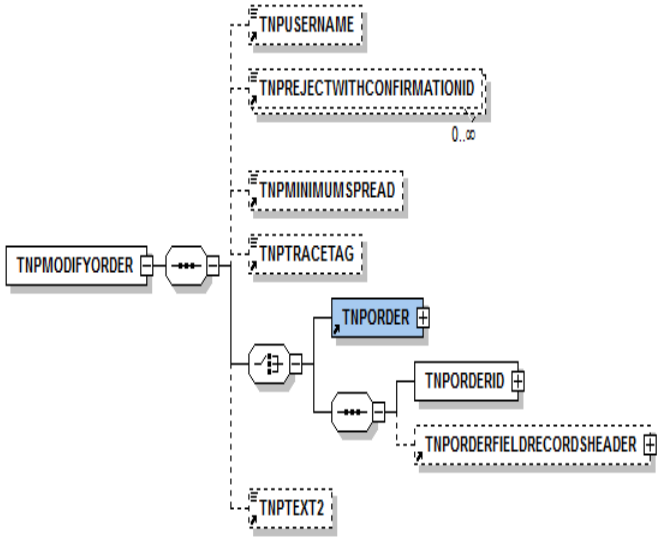

Figure 2 The TNP structure

Each field within one TNPMessage describes some market or business property, such as: order price, trader, order type, etc.

B. *Market Server Capabilities*

Many of the TNP client components query the Market Server Capability (MSC), information about the trading functionality that one electronic exchange (market) offers. Client applications need such information in order to permit/disable the access to the different markets. For example, one market can allow entering orders and modifying orders but does not support entering trades. The other market supports entering trades with the restriction that the shaping trade transaction is not allowed.

The MSC information is embedded and hard-coded into each client application. New client application releases needed to be done before the customers can start using the new AMS. Depending on the current release plans of the client applications this can take a long time. Having to wait for the client application releases may delay the production start of the AMS. Two main problems with the described MSC information are:

- Hard-coded MSC definition. Consequently the client applications have to be recompiled, released to customers and upgraded on the customer's site in order to enable the support for the newly introduced MSC. Such concept conflicts with the agile principles "deliver working software frequently" and "respond to changes quickly" [3].
- Duplication of the MSC definition. It introduced the risk for data inconsistency.

These problems will be resolved in the future by introduction of a Dynamic Market Capabilities (DMC), a new functionality that will be used to retrieve the MSC definition dynamically, in run-time, instead of having them hard-coded. Unfortunately, it will take a long time, probably years, until the DMC solution will be completely implemented and in use (for all AMS and all client components). Until then all components have to support the hard-coded fashion. All new components, which will be developed during this time, have to support the hard-coded MSC way also. That is why there was a need to find an intermediate solution which would remove the duplication and which would be used under the transition phase. Since such architecture would not be long lived company management put some time and resource constraints on the implementation. This paper shows how we created such intermediate solution, taking all conditions and constraints into account.

V. THE MARKET SERVER CAPABILITIES ARCHITECTURE

A. *Process flow*

When a new market (AMS) is introduced, the information about functionality that the new market offers (which transactions are supported) should be added to each client, as presented on Figure 3.

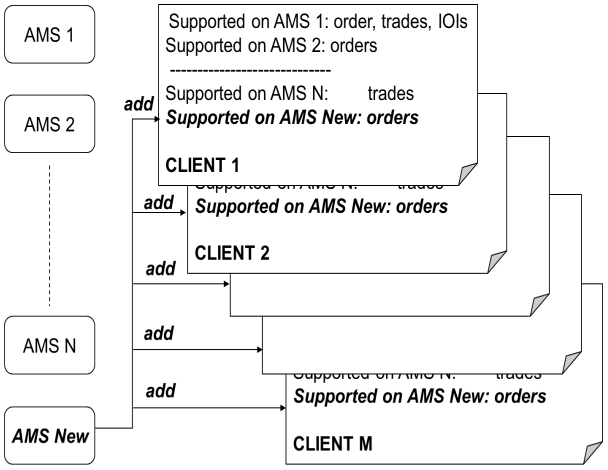

Figure 3. Process flow

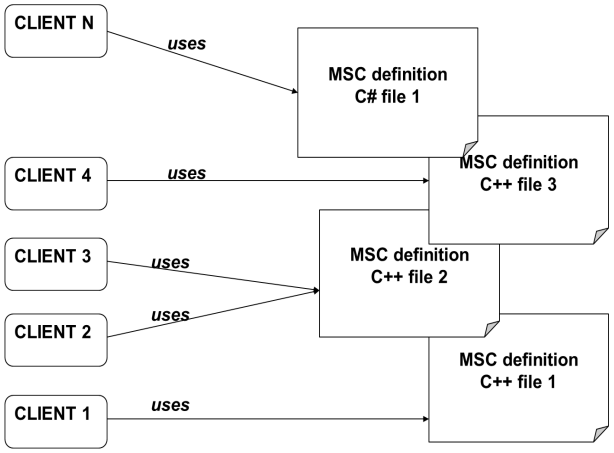

Figure 4. The present architecture with distributed definitions

The MSCs describe market trading transactions (orders, deals, etc.), the commands that are supported for them (entering, modifying, etc.) and the attributes and the fields, which could be accessed on the markets (quantity, broker, etc.). Hence, specifying the MSC for a new AMS requires a detailed description. All components, which use the MSC functionality, must use the same MSC definition. Unfortunately the same MSCs are defined in several different files. Different components are developed in different programming languages so they do not share the same definition file. Because of historical reasons and the fact that some client components were developed within separate teams, even the components developed in the same programming language do not share the same definition file. Each client component has its own MSC definition file. There is a lot of the duplication of information in these files. Even worse they do not present exactly same data since the different clients work within different business domains, so their knowledge about the MSCs is on the different levels. The described situation arouse from bad communication between the teams. Without interacting with each other and without having enough knowledge about the design and the architecture applied on the different projects, it was easy to end up with the described MSC architecture.

B. The present architecture analysis

The client components use the MSC definition from the different sources, developed in different programming languages (C++, C# and Java), where the majority of data is duplicated. This situation, with the usage of the overlapping MSC definitions, is presented on Figure 4.

The present architecture of the MSC definition is not centralized (no single definition of the model) and without control for the consistency. The lack of centralization enormously increases the risk for data inconsistency since the consistency depended on the accuracy of the developers who edits the MSC definition in a source code file. The development of the MSC definition is a continuous process, and new MSCs are defined each time when a new AMS is developed (2-3 times per year) or when a new trading transaction is introduced (once per month). The current process flow is:

- A new AMS is developed or a new transaction is introduced.
- A MSC is added to the MSC definition in each client component. The same information must be added to several different files.
- All client components should be recompiled in order to get the definition of the new MSC.

After the presented files analysis we could state the following facts about them:

- Similar structure: files are structured in the similar way, containing a lot of switch/case statements.
- Data duplication: some data are duplicated
- Different business domains: different levels of the describing aspects are used in the different files.
- Mainly syntax differences: the mainly difference among the files lies in the syntax not in the data structure.

C. Dynamic Market Capabilities architecture

We have already done design plans for the new DMC architecture. In the DMC architecture each AMS will be responsible to provide, to the client components, information about the MSC that the AMS supports. The description of the MSC that the AMS supports will be saved in one XML file.

On the AMS start up, AMS reads the MSC definition from its XML file and sends them, in run time, to all client components, which connect to the AMS. In such a way the client components do not have to be recompiled if something changes in the MSC definition. When a new AMS is developed, a new XML file containing MSC definitions for the AMS is created. On the AMS start up, all client components connect to the AMS and dynamically retrieve the MSC definition for that AMS. In the future, even in this case there will be no need for the recompilation of the client components.

D. Transition phase

The decision is that all AMS components and all client components should be upgraded to the DMC architecture. But this transition is a complicated job. There are over 30 AMS components and more than 5 client components that are using MSC functionality today. There is different prioritizing, from the management side, within the components' backlogs. We know, right now, that some of these components will be upgraded to the DMC in one or two years. This transition project is not marked as a critical since there is already a working architecture, although not the best one. As long as there is at least one component, which has not been upgraded to the new DMC architecture, the hard-coded MSC solution must still be supported. The transition will occur gradually and the transition phase will probably take several years. Under the transition phase some new components are going to be developed; some new components are already under the development. To develop new client components according to the present architecture will introduce even more duplication. Therefore, an intermediate architecture, which will eliminate the duplication, would be introduced. Such a solution should have a short implementation phase, since it must be ready before the new components are completely developed. The solution should be designed so that it eventually leads towards the new DMC architecture. It would be good if the new DMC architecture could benefit from it.

VI. INTERMEDIATE SOLUTION

We work according the lean and agile software development philosophy. One of the key principles of the lean philosophy is to detect and eliminate wastes [22]. The intermediate solution should eliminate, from the present architecture, the three major points of waste.

- Duplication of the MSC information
- Amount of work done during the MSC definition updates
- Amount of time used for communication among groups, informing each other about the MSC definition changes

A. Technical Aspects

The waste elimination adds an important business value, according to the lean philosophy. Even if the transition of the existing MSC architecture to the intermediate architecture does not directly add a business value from the customer's perspective, the existing system's wastes would be eliminated by the reengineering process. In that way the delivery of the new solutions, which are dependent on the MSC architecture, would be accelerated. Hence, we get an implicit business value, which would be produced by the intermediate solution.

In order to eliminate the duplication of data we needed a centralized MSC definition. In order to be able to provide support for the MSC definition in different programming languages we needed to generate code in different programming languages, from the centralized MSC definition. We need a programming language independent architecture. Because of the lack of time, we decided to have an agile approach on brainstorming meetings when we were searching for the architecture of the intermediate solution. We did not want to waste a time on investigating all possible solutions, since the time was more precious for us than perfection. We suggested and analyzed three different approaches and chose one among them, which was the most suitable. Although we did not analyze all possible solutions, we got a methodology that was good enough. To use a good enough solution for the current situation, within a short time frame, suits the agile philosophy.

First we considered a solution, where all client components would be refactored to reference the same central definition file. This would require a lot of work. We did not want to refactor client's components too often, since some of them will be refactored soon regarding the DMC solution.

A generative programming concept [23], using a parameterized C++ templates, was discussed as the second solution. Such solution would consist of the generated classes, representing the TNP objects (TNPMessages and TNPRecords). The main intention of the generative programming is to build reusable components. A cost of building the reusable components should be paid off by reusing them in many systems. When a goal is to build just one system and when schedule, to deliver a system, is tight, the introduction of the generative programming idea cannot be the best solution. Additionally, the existence of C# MSC definition file made the usage of the C++ templates impossible.

Finally we analyzed the Model-Driven Architecture (MDA) approach. With the MDA approach we mean the general MDA concept: "A MDA defines an approach to modeling that separates the specification of system functionality from the implementation on a specific technology platform". The common denominator for all MDA approaches is that there is always a model (or models), as the central architectural input point, from which different artifacts are generated and developed. Transformations, mapping rules and code generators are called in common "MDA tools" [24].

We believed that the Model-Driven Architecture (MDA) approach would be the most suitable solution for the intermediate architecture. The main idea was to have just one source, a union of all present MSC definition that is

programming language independent. From such a source, which would be a central MSC definition registry, the present MSC definition source files are generated. All present MSC definition files have a similar structure. The main difference is the programming languages syntax. Because of that the code generation should not be too complicated. The way how the client components work would not be changed, the MSC definition would still be hard coded. Such a solution did not require the refactoring of the client components. But the way how the developers work would be improved. They will work just with the central MSC definition registry and add/edit the MSC definition only there. Then the MSC definition files, for each client component, will be automatically generated from the central registry. The client components will be automatically recompiled. In that way all three mentioned wastes will be eliminated.

Another key lean principle is to focus on long-term results, which is the DMC architecture in our case. That is why we must point out that one important part of the DMC architecture is a MSC XML description file. If the MDA approach is introduced for the MSC definition, the central MSC definition registry would be easily divided into several files (one per AMS), later on. It is clear that the DMC architecture would benefit from having such a central MSC registry. The creation of one central MSC definition registry, with all MSC definitions for all markets, would be a good step towards the future DMC architecture introduction.

B. *Organizational Maturity and Limitations*

Our company management is usually very careful with introducing concepts not already used in the company, since it often requires long implementation and learning time. Additionally, an investment in an intermediate solution is not always a very productive investment. On the other side, the management was aware that the intermediate architecture would increase productivity directly and make some new solutions possible right away. That is why the management listened carefully to our needs and made some general decisions. The intermediate architecture can be introduced, but the time-frame could be only several weeks. No new tools or licenses should be bought. Only tools that are already used within the company or some new, open-source tools, can be used. No investment in change management. Time for teaching/learning cannot be invested for the intermediate solution. The concepts, which our developers are already familiar with, should be used.

Considering these management decisions, we decided to explore if the organization was mature enough to introduce the MDA. Although the MDA approach has been around for a long time, for many companies it is still a new approach. That is why we performed a small survey, with questions presented in Figure 5.

1. *Have you worked in the MDD environment before?*
   a. *If yes, what is your experience regarding the MDD?*
   b. *If yes, have used the UML modeling?*
   b. *If no, have you heared about it?*
2. *Do you think that the MDD approach should be introduced in our project?*

Figure 5. Survey Questions

We asked 60 developers, working in the 6 different teams. 4 teams consisted of C++ developers, 1 team consisted of Java developers and 1 team contained C# developers. The survey showed that the MDA approach hasn't been used within the company and that a majority (80%) of the developers has never used this approach. Consequently the UML modeling is not used in general. Some teams were using MagicDraw, but just as a documentation tool for the state-machines drawing. The architects, who designed the state machines, answered that it was faster to develop own generators, using the state diagrams created in the MagicDraw than to investigate how to use the UML tools and profiles and code generators.

Additionally, there was a previous attempt of introducing the MDD in the enterprise architecture, which unfortunately failed. The former MDD project consisted of a new modeling framework, based on the Eclipse framework, particularly designed for the drawing Front Arena state-machines. The project has never been finished because it took a long time without showing the results. Unfortunately it happened at a bad point of time, when the financial market was extremely poor and when the product delivery to the customer was a matter of the utmost importance. Consequently the company lost time and money by investing in this MDD framework. The main problem was not the MDD concept by itself, but it was difficult to see an explicit business value in it. A time-consuming and cost-effective MDD introduction was in contradiction with a fast and frequent delivery. Because of all mentioned reasons the majority of the developers, as well as the management, did not believe in a new attempt of working with a MDD idea. The introduction of the full scale MDA usually implies: a long starting curve, which we could not afford having a short time-frame and the usage of the MDA tools, which could not be used since developers did not have enough knowledge about them and there was no possibility to invest in learning. In the following section it will be described how we managed to overcome these problems and limitations.

VII. AGILE MDD APPROACH

Our goal was to find an intermediate solution with a MDA philosophy, which satisfied the previously mentioned requirements and fulfills the constraints. In order to achieve this goal, we started from the basics of the MDA concept (models, transformations and code generators), and combined them with the following lean and agile principles [3]:

- "Think big, act small": Think about the DMC as a final architecture but act stepwise, introduce the intermediate solution first.
- "Refactoring": A change made to the structure of software to make it easier to understand and cheaper to modify without changing its existing behavior [25]"
- "Simplicity is essential": We have to find an applicable solution that is simple, keeping in mind that simple does not have to mean simplistic [17].
- "Individuals and interactions over processes and tools": It is important is to find a solution, which fits the developers, as well as to establish such development process, which will be effective in our company.

We used the agile and lean ideas both in the decision making and in the development process. In that way we got our own Agile MDD approach, an applicable intermediate solution.

It is important to emphasize that we had an existing architecture, which should be transformed to the MDD solution. The process of system transformation is called a system reengineering. A system reengineering phenomena has been present in the software development as long as the software systems exist. With the high dynamic of business requirement and technology changes, the systems have to be modernized constantly. System modernization is a way of system adaptation to the changes. Architecture-Driven Modernization (ADM) is an OMG standard for the system modernization [26] and can be briefly described by using the ADM horseshoe model, presented on the Figure 6.

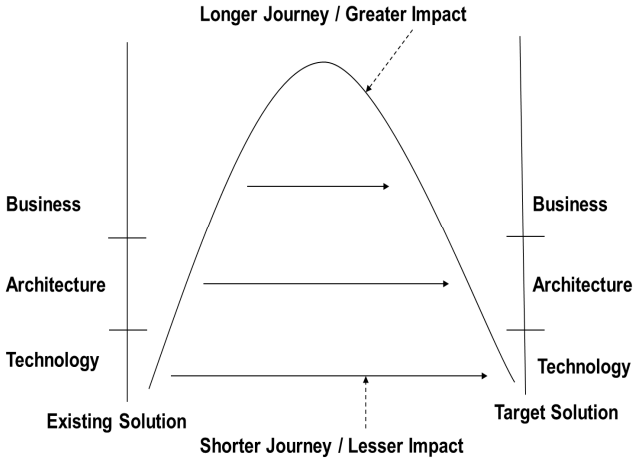

Figure 6. ADM horseshoe model. Adapted from [26].

According to the ADM standard, three areas could be distinguished during the reengineering process; technical, architectural and business area. Depending on the extent of areas that are affected during the reengineering process, the ADM journey can be longer or shorter. Consequently the impact of system changes can be greater or lesser. The duration of the reengineering process directly affects time-to-market. This fact was important to bear in mind when making the architectural decision within the chosen MDD solution, as it is described in the following sections.

B. Agile modeling

We needed to model the MSC definition registry. This modeling can be done on the different modeling levels and in the different modeling languages. The UML is one the most frequently used modeling language and it became a modeling notations standard, according to the OMG's recommendations. The UML has been developed and evolved to cover many different needs, becoming, at the same time, huge and unwieldy. Although the UML profiles have been introduced in order to help the developers to exclude unneeded UML parts, there are still many cases where the adoption of the MDD has been slowed because of the UML's complexity [15]. Additionally the UML lacks sufficient precision to enable complete code generation [27]. The time frame for our project was short and the developers were without enough UML experience, since the UML is not used in general. According to the limitations, there was no time for learning. Hence, the UML modeling could not be accepted as a modeling solution in our project. Since the XML format is a standard format and the developers are familiar with it, we decided to use a XML description as a "natural language" for the developers. XML was good enough. We had to balance between the familiarity of the XML and abstraction benefits of UML but also a complexity of the related frameworks, keeping the project within the time-frame. Also the XML usage would imply the shorter journey for the reengineering process, compared to the long journey required for the reengineering to the UML model.

The MDA defines different model categories, like a Platform Independent Model (PIM) and a Platform Specific Model (PSM) [24]. As discussed before, although the multi-model concept provides a good separation of concerns, at the same time it could introduce an unnecessary waste in the system, which is in contradiction with a lean architecture. Hence, the multi model concept should be used only if there is a really need for that and when a separation of concerns is required in order to be able to understand and work with a system. The PIM and PSM concept becomes an important issue if there are plenty of different platforms with specifications that differ very much. In our case the different PSMs did not differ too much from each other and, at the same time, did not differ too much from the PIM either. In order to keep it simple we made a pragmatic solution: to have just one model, which contained all info for all programming languages. The code generators had the responsibility for creating the right MSC information to the corresponding programming language.

```
< MarketCapabilities>
  < MarketCapability id="200001" HomeMarketServerId="OMX" version="1.0.0" >
    <Objects>
      < Object name="QuoteRequest" value="RTY_QUOTEREQUEST">
        <Commands>
          <Command name="EnterQuoteRequest" value="RTY_ENTERQUOTEREQUEST"/>
          <Command name="RejectQuoteRequest" value="RTY_REJECTQUOTEREQUEST"/>
        </Commands>
        <Fields>
          <Field name="BidOrAsk" value="RTY_BIDORASK enter="yes"
                 EnumId="MyQuoteRequestBidOrAsk"/>
          <Field name="Quantity" value="RTY_QUANTITY"  enter="yes"/>
        </Fields>
      </ Object>
      < Object name="Order" value="RTY_ORDER">
        <Commands>
          <Command name="EnterOrder" value="RTY_ENTERORDER"/>
          <Command name="ModifyOrder" value="RTY_MODIFYORDER"/>
          <Command name="DeleteOrder" value="RTY_DELETEORDER"/>
        </Commands>
        <Fields>
          <Field name="QuantityCondition" value="RTY_QUANTITYCONDITION"
                 enter="yes" EnumId="MyQuantityCondition"/>
          <Field name="PriceCondition" value="RTY_PRICECONDITION"
                 enter="yes" modify="yes" EnumId="MyPriceCondition"/>
          <Field name="MarketAccount" value="RTY_MARKETACCOUNT"
                 enter="yes" modify="yes" EnumId="MyMarketAccounts"/>
          <Field name="OrderAttributesHidden" value="RTY_ORDERATTRIBUTES"
                 enter="yes"/>
          <Field name="AMASFixOrderState" value="RTY_AMASFIXORDERSTATE"/>
        </Fields>
      </ Object>
      < Object name="PriceDetail" value="RTY_PRICEDETAIL">
        <Commands/>
        <Fields>
          <Field name="LastPrice" value="RTY_LASTPRICE"/>
          <Field name="LastQuantity" value="RTY_LASTQUANTITY"/>
          <Field name="LastDateTime" value="RTY_LASTDATETIME"/>
          <Field name="HighPrice" value="RTY_HIGHPRICE"/>
          <Field name="LowPrice" value="RTY_LOWPRICE"/>
          <Field name="ClosingPrice" value="RTY_CLOSINGPRICE"/>
        </Fields>
      </ Object>
      ...
    < Objects>
  </ MarketCapability>
      ...
</ MarketCapabilitys>
```

Figure 7. XML Model

We have created two models. One was a logical model that describes the entities in the MSC definition registry. Another was the MSC definition registry by itself, expressed in a XML dialect, which is presented on Figure 7. Consequently the logical model was expressed as a XSD schema and was used to validate the entries in the registry.

C. *Code generators*

We needed code generators for generating the different types of files: C++, C#, Java. We decided to use XSL transformations as the code generators. They satisfied our needs and could be widely used, since the XSL is a common standard for all developers, who program in the different programming languages. In that way a "collective code ownership" [3] is achieved for the code generators. The maintainability is also better if all developers can maintain/develop the transformations.

VIII. THE DEVELOPMENT PROCESS

Our company has introduced the agile software development several years ago. Scrum is used as a process tool. Each team runs its own sprints, typically lasting for 4-5 weeks. The sprints are synchronized, meaning that the start and the end sprint date is the same for all sprints within the company. Such synchronization makes the releases and the delivery process of the dependent components easier. Although we use Scrum, all teams do not strictly follow all Scrum recommendations, it is more up to the team how the Scrum is performed, dependent on the currently running project.

The intermediate solution, as an internal project, was supposed to be done in parallel with other running projects. In order to fit in the company's culture, we decided to run our project according to Scrum, based on the sprints. In general, working in Scrum sprints suited our project well. When we worked on the common data, which were present in several MSC definition files, it was easy to plan the coming sprints, since we knew the next required steps. For example, we knew that we should extract all capabilities per market. Scrum suited well for the major parts of the project as we were planning one sprint at time.

On the end of project, when only odd data, specific for a certain market or a certain component, was left it was difficult to plan the sprints. When we had many small tasks, which were not very related to each other and which were not easy to separate and divide into the sprint tasks, Kanban [11] was more suitable. Therefore, when we were approaching the end of the project, we switched our development process to a Kanban. In contrast to Scrum, tasks in Kanban are performed one after the other, without collecting them into sprints. One of the main Kanban principles is to limit "work in progress" by defining the maximum of tasks, which can be performed in parallel. If this number is exceeded, no new tasks are taken from the backlog until there is an available capacity for a new task. Changes to the product backlog take effect as soon as capacity becomes available. A typical Kanban board is presented on the Figure 8 where both short and long running tasks can be executed in parallel.

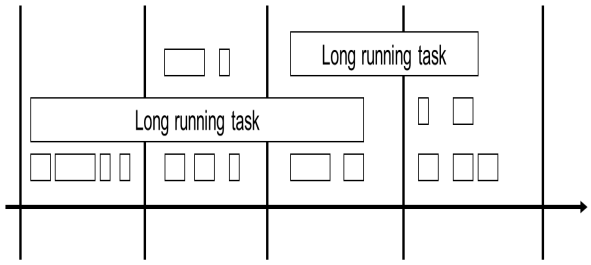

Figure 8. Our Knaban process. Adapted from [11].

Since we could not appreciate time for the tasks that were left, we just put them on the board and took them as soon as the previous task was finished.

A. *Team Selection*

A good team communication is one of the necessary prerequisite for a successful development [2]. The absence, irregularity and incompleteness in communication among the company's teams caused the duplication and inconsistency in the present MSC architecture. According to the agile

manifesto people and interactions should be over tool and processes [3]. As the reengineering to the MDD solution was a new challenge for the developers in our company, it was important to have a "self-selected" team [3], with the developers that were interested and willing to work on it. Since the main part of our project consisted of working with the legacy code, we wanted to have experts in the team, with deep domain knowledge about all existent MSC files. Applying organizational patterns "architects also implements" [7], three architects from different teams where chosen, one for each MSC product owner team. In that way we got a good expertise for different business domains, for all types of the MSC files. On the other side, the chosen experts worked together in a pair-development sessions, supporting the concept of "generalizing specialists". Generalizing specialists are often referred to as craftspeople, multi-disciplinary developers, cross-functional developers and deep generalist [28]. It was important that the experts working on the project could see the whole impact of the changes, not only within their expertise domain.

Another important task, related to the team communication, was informing all teams, which were the product owners of the MSC files, about this project. We wanted to avoid making the same mistake as it was done before. Therefore, it became very important with sprint demos, combined with the result presentation, for all affected teams. The two important purposes of demos were:

- Spread the knowledge about the done and to-be done project tasks
- Show that the MDD project can be rapidly developed.

Additionally, the knowledge spread was even more effective with the chosen experts, belonging to the different teams, since each expert talked to its colleagues about the ongoing project.

B. Reverse engineering of the Legacy code

We needed to do a one-time reverse engineering in order to convert a large amount of the existing MSC data, legacy code, to the new MSC XML format. We developed our own tool for this purposes since no open-source tool was completely suitable. The main question was: when to start with the reverse engineering? At the end or at the beginning of the project? Very soon we realized that we could not design our model in detail without the data from the existing MSC definitions. It was data stored in the MSC definition, which lead the reengineering process. This data became a kind of business requirement in our project. Consequently the requirements were not developed; they were discovered during the reversing process.

We decided to adopt a spike principle. The spike is a full cross-section of the modeling and architecture aspects of the project for a specific scenario. The aim of the spike approach is to develop the whole chain for only one, chosen user scenario. The first chosen scenario is a simple one, and during the incremental development process every next scenario is a more complex one [29]. We started with the round-tripping (the whole chain: model – code generation – reversing back to the model) for simple scenarios, which we expanded, in each sprint, to the more complex scenarios. In that way we could develop the reverse engineering tool, the code generators and to design the model in parallel. The results of the reverse engineering helped us with the specification of the model objects for both the logical model and for the central MCS registry. Working in that way, we allowed "the business requirements coming late in the project". In our case, the business requirements were mainly the results (predictable and unpredictable) from the reversing process, which steered the reengineering project. Since we could do the round-tripping very early in the project, it was a way in which we could start testing our MDD approach early, under the development.

C. Round tripping with the TDD approach

According to the lean principles, we wanted to specify our model just according to the existing data, without unnecessary objects or unnecessary properties, which risk never to be used. In order to be able to do that, we wanted to do the reversing first and specify the logical model and fill the data in the MSC registry upon these results. We used a TDD approach and started with writing unit tests first. For this purpose we used test framework developed and already used in the company. This framework simulates the execution of the TNP messages sent among server and client components. Because of that the test scenarios that we wrote can be reused later on, for testing AMS components, when the DMC is introduced.

According to the TDD principles we wrote the tests first, run them on "empty" code and developed the code, until the tests passed. Since we had to test several parts of our MDD approach (the logical model, the central MSC registry, the code generators and the reverse engineering tool), we established our own TDD process for the MDD testing. The main idea was to use the same tests, which reflects the parts of one spike, both to develop the reverse engineering tool and the code generators. Our TDD process is presented on the Figure 9 and will be described now through one real spike. The chosen spike is called "Get all markets" and the goal is to get all existing markets, described in the present MSC files. We started with writing a test, which consisted of sending a TNP message "TNPGetAllMarkets". The next step was to develop the reverse engineering tool for this scenario. The legacy code was used as input data. We developed the corresponding methods in the reversing tool, which extracted markets from the existing data, producing the results in the XML format, and inserted them in our MSC registry. It was a list of all markets. Then we redesigned the model and registry entities and refactored the reversing tool according to the model changes. This process flow is presented with semi-dashed arrows on the Figure 9.

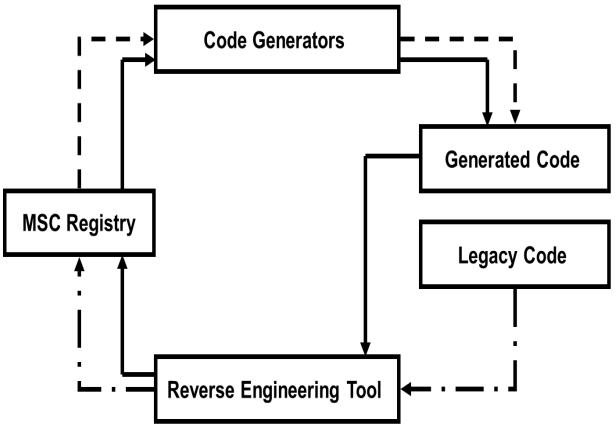

Figure 9. Our TDD process

The TDD logic for the code generators was the following. What we had, so far, was the reversing tool working for the chosen scenario, and some data in the central MSC registry. We used the same test against the code generators, trying to get all markets from the MSC registry. We developed the code generators using the mentioned test. This process is marked with dashed arrows on the Figure 9. The final goal was to get the same code lines in the generated code files, as we had in the corresponding legacy code files, concerning the data affected by the chosen spike. In order to verify this, we run the whole round-trip but this time we used the generated code files as the input for the reverse engineering tool. Eventually we compared the newly generated files with the previous ones and the legacy code files and if there were some differences we adjusted the reversing tool and the code generators accordingly. This process is marked with full arrows on the Figure 9. After this sprint we had a list of all markets in the MSC registry, the code generators methods, which generate files containing such a list, and the reversing tool methods for extracting such a list from the generated files. In the following sprints we used more advanced scenarios, such as, for example, "Get all markets where is Order supported with commands: Enter, Modify".

At the end of each sprint we run the whole round tripping, starting from the legacy code. In that way we could confirm that both the newly implemented code worked, as well as that the previously implemented code was not broken. As the final verification process we confirmed that all client components could be compiled without errors. We did the usual integration tests also, in order to confirm that the communication among the client components and the AMS components has not been changed. When we completely finished with the reversing, we disabled this functionality. We needed the reversing only for extracting the existing data. It has not been possible do the reversing nor the round tripping since the project was released.

D. Test-first Tests

Two reasons were crucial for choosing the TDD approach in the reengineering project. As first, we believed that the TDD approach could accelerate the development. The second reason was the fact that we did not have enough knowledge about the MSC files content so we did not have a clear idea how to start the implementation. Therefore, writing the tests first was a good start. We usually started with file investigation and wrote the test-first tests as soon as we understood the existing code. It was an excellent start to begin the implementation of one spike. We applied often the "learn it" TDD pattern [30] in order to examine the files and write the test-first tests accordingly.

The introduction of the TDD approach was important because of the following reasons:

- By developing and testing in parallel we shortened the implementation phase.
- We did not produce any wastes in the logical model (unnecessary info). We designed the model just according to the data that we got from the reverse engineering. We achieved to avoid the usual modeling mistake when a large amount of metadata is put in the model.
- The reengineering process was accelerated since the reverse and forward engineering were performed simultaneously.
- We showed how the TDD can be an efficient way to work with, since this development method has not been yet widely spread within the company. When it has been introduced once, it would be easier to introduce the TDD thinking in other projects too.
- We learned a lot about the different TDD patterns.
- We can reuse some of these tests later on, for the DMC architecture testing.

It is important to say that we had to reverse the legacy code from the code, which was written in the different programming languages. We had to develop separate methods for the reversing from C++, Java and C#. Fortunately, the respective legacy code files had a similar structure; the syntax was the main difference. So we could develop the corresponding reversing methods based on the common objects.

E. Continuous Integration with Automation

The continuous integration is a software development practice where the software is integrated frequently, having the integrations verified by automatic builds to detect integration errors [31]. TDD and Continuous Integration (CI) are agile practices, which complete each other. TDD produces code that is well designed and relatively easy to integrate with other code. The incremental addition of small parts to the system, together with the automatic builds, provides the continual system development without extensive integration work [32]. The general continuous integration concept was already introduced in the company. All client and server components, which use the MSC

definition, have the automatic builds enabled. When the code changes in the code repository, which is Clear Case in our case, the automatic builds are started by a trigger scripts. The trigger scripts, integrated in the Clear Case and specially developed for this purposes, are responsible to start the automatic builds on the build servers. If the builds fail, the responsible product owners are immediately notified by email.

On the end of the project, we have automated some of the processes reducing the amount of work and time spent on working with the MSC definition architecture. We use ClearCase (CC) as a configuration management tool and we have a build server for automatic build processes. Since all client MSC definition files were in CC, we decided to keep even the generated files in the CC repository, at least under some period. This decision was made by the management.

When the MSC definition registry file is updated and checked into CC, the following steps are executed automatically:

- The MSC definition files with hard-coded data, belonging to the client components, are checked out from CC.
- The code generators are invoked by a CC trigger script. All MSC definition files are generated.
- All generated files are checked into CC, if the generation did not fail. Otherwise the "undo checkout" operation is done.
- All client components, affected by the mentioned code generation, are recompiled. If some compilation fails, the error report is immediately sent to the component owners.

Continuous integration verified that all parts of the MDD solution are synchronized. To clarify, previously existing tests are run against the generated files, as they were run against the hard-coded files before. In that way we had an automated check that the legacy code was not broken. New test suit, containing tests used for the code generators development, were also added to the automated test execution. The corresponding tests were not run against the reverse engineering tool, since this functionality was disabled on the end of the project.

F. Light-weight Documentation

Although the documentation generation was not among the project requirements, the MDD solution enabled a possibility to generate documentation about the MSC for the different markets in a light way. The MSC registry, expressed in the XML format, supported a possibility for writing comment lines. In that way we could easily develop a generator that generates a HTML files presenting different MSC aspects. For example, beside basic tables describing the supported capabilities on the separate markets, it was interesting to create lists for each capability, presenting all markets where the capability is supported. The latter information was very useful for the product management team, to get quick information about the market capabilities. The user-friendly presentation was highly appreciated, since it shortened time when searching for information. In the described way we achieved to get light-weight documentation, which is easy to update and does not cost much to maintain. Thanks to the XML format of the MSC registry, the documentation generators development was a trivial job, lasting for just one developer day. Such way of documenting MSC definitions fitted well the agile philosophy.

G. Results

The project was completed within the 4 sprints, lasting for 4 weeks each, and one month of Kanban process. At the early stage of the project, without enough experience, we could not plan the first sprint in the most efficient way. It was during the first sprint, which took more than the one month, when we made the decision to do the reverse and the forward engineering in parallel. After that, the development was accelerated as well as the sprint's velocities. Velocity of the first sprint was only 10 story points. Every next sprint was executed with a velocity of 15 to 20 story points. We planned the coming sprints according to the results and experience from the previous sprints. During the Kanban process, the development speed decreased again since we were stacked with a lot of small problems which were supposed to be solved separately. On the other side, the fact that we were approaching the end of the project encouraged us with completing the tasks. Although we could have completed the project several weeks earlier, if we planned the first sprint better, the management was satisfied with the performed results. The extenuating circumstance was the fact that the intermediate solution was an internal project, without fixed released date to the customer.

IX. AGILE AND LEAN PRACTICES IN MDD

The agile and lean methods are light in contrast to the MDD that can become complex. Through the application of the agile and lean principles, the MDD becomes more pragmatic and more useful. Some of the agile and lean principles, used in our Agile MDD approach, are explained below.

A. Architectural aspects

"Eliminating waste" Eliminating the duplication of information was also according to the XP's principle "Never duplicate your code" [33]. This principle is the heart of the MDD – to have one central input point, model (models) from which everything else is generated.

"Think big, act small" We were thinking on the DMC as a final architecture but acted in a stepwise way, via an intermediate solution.

"Simplicity is essential." We have simplified the full scale MDA. Instead of the UML modeling language we used the XML. The PIM and PSMs were merged, avoiding the maintenance of several models and transformations among them. On the other side, by merging PIM and PSMs in one model we lost a good separation of concerns but it was a price worth paying.

B. Organizational aspects

"Self-organizing teams" contributed to the successful MDD introduction, since the evolved people were indeed interested to complete the project.

"Empower the team" Roles are turned – the managers are taught how to listen to the developers [22]. Despite the fact that the management puts non-technical constraints on our project, they allowed the developers to make decisions, regarding the intermediate solution, on their own. It contributed to faster development, since the developers did not have to wait for feedback from the management, for each decision.

"We became a constantly learning organization, through relentless reflection and continuous improvement." Since the organization was without enough previous knowledge within the MDD area, we learned a lot about applying this concept in practice.

C. Development process aspects

"Deliver as fast as possible". The implementation phase of our Agile MDD approach was short.

"Spike principle" applied on the round-tripping, which includes both the reverse and the forward engineering, made the introduction of the TDD philosophy spontaneous and natural.

"Forward and reverse engineering attain the same importance." Since the model was designed upon the results of the legacy code reversing, this process, although being only a one time process, was equally important as the forward engineering.

"Constant feedback" practice was particularly important in the reengineering process since we did not have a clear idea, from the beginning, how the data reversing should be performed.

"Welcome changing requirements, even late in development." The industrial experience report presented an iterative development, which allowed late model changes. We worked in sprints, according to the Spike principle, which implied the frequent model changes, in each sprint.

"Combine Scrum and Kanban process tools" as it is suitable. When the projects tasks can be strictly divided and planed, than the Scrum is more appropriate. But for some long running task, such was a reengineering process in our case, the Kanban was more appropriate since it was difficult to plan sprints in advance.

D. Benefits of the Agile MDD approach

We got a lot of benefits by introducing the Agile MDD approach.

1. Agile principles can make the starting curve for the MDD shorter. Through the application of the agile principles the long learning curve and introduction gap of MDD methods and tools could be avoided. Instead of spending a long time building a big thing we had a small team spending a little time building a small thing but we integrated regularly to see the whole system [11].
2. We introduced the TDD approach, showing the effectiveness of such an approach. TDD approach contributed to the accelerated reengineering to the MDD solution since the reverse and forward engineering were performed in parallel.
3. "Base your management decisions on a long-term philosophy, even at the expense of short-term financial goals." We have prepared, in advance, for the introduction of the DMC architecture: the model specification and the reverse engineering job are already done. As well as the test cases, some of them are going to be reused.
4. The Agile MDD approach could be used instead of the full scale MDA. When all MDA recommendations could not be applied, we adjusted them to our system and organization, with a help of Agile and Lean principles.
5. Agile modeling helped against building gargantuan models [15] and specifying potentially unused data.

X. Related Work

The idea of combining the agile ideas with the MDD concept has been present in both research and industry world for some time. But as a relatively new idea it is still without enough empirical results, which should lead to the right direction where and how this idea should be evolved even more. Many authors agree with the conclusion that the agile principles can be combined with the MDD, making, usually long and time-consuming process of modeling, being iterative and incremental. [34] explored this idea even more, by applying a set of agile principles on UML modeling, such as pair-modeling, test-driven development and regression testing. In [35] a comprehensive framework, showing the various ways to take advantage of the complementarity between the agile methods and MDD, is proposed.

Although it could be assumed that the agile methods should be used for the development of new software systems, they could be used for the legacy code evolution as well as discussed in [36]. By applying the agile methodology on the reverse engineering process, some authors have already made proposals for an incremental agile reverse engineering process. [37] and [38] describe a framework support for an agile reverse engineering process. [39] proposes an iterative reengineering approach that uses reverse engineering patterns for the reverse engineering and test-driven development for the forward engineering, where the reverse engineering and forward engineering activities are done independently, one after the other.

To our best knowledge there is no author who explores a simultaneous application of TDD both on the reverse engineering and forward engineering process when the legacy system is reengineered to the MDD. Additionally, there is no author who discusses the whole agile development process for the system's evolution to the MDD solution.

XI. Conclusion and Future Work

This industrial report presented an evolution process of an existing system - the architecture of the MSC definition, to a Model-Driven Development solution. The main point of this paper was to show how agile and lean principles helped us in a decision making process during the intermediate solution production, within a short time frame. In that way we coped successfully both with the management constraints as well as with the complexity and time-consuming introduction of the MDD concept.

This industrial report showed that the agile philosophy and the MDD concept can be successfully combined, resulting in an accelerated development process. Agile principles relax the OMG's recommendations reducing the complexity from the MDD concept, making the MDD easier to adopt in organizations. As this paper showed, an agile MDD could be a key success factor for organizations, which are not ready for the introduction of the full-scale MDA. Consequently we could expand Ambler's "agile modeling" philosophy on the whole MDD, including the reengineering process, meaning that it should be situational, adjusted to the running project, organizational structure and development process. Additionally, a development process based on the TDD logic can contribute to improved development efficiency and decrease the total time spent on the development and testing.

By being aware of the "Think big act small"-principle, we got a simple and applicable solution, which could easily grow to a more complex one. With a help of agile and lean ideas we modernized the MSC architecture. Such evolution made this architecture more flexible and more responsive to the future changes, regarding both the technical and the business aspects. "It is not the strongest of the species that survive, nor the most intelligent, but the ones most responsive to change [40]."

References

[1] Mina Boström Nakicenovic: An Agile Model-Driven Development Approach – A case study in a finance organization. Proceedings of ICSEA 2011.

[2] M. Pikkarainen, J. Haikara, o. Salo, P. Abrahamsson, J. Still: The Impact of Agile Practices on Communication in Software Development. Journal Empirical Software Engineering, Vol. 13, Issue 3, pp 303-337, June 2008.

[3] AgileManifesto, www.agilemanifesto.org. Accessed in May 2012.

[4] Dave West, Tom Grant: Agile Development – Mainstream Adoption Has Changed Agility, Forrester Research, 2010.

[5] Hans Wegener: Agility in Model Driven Software Development? Implication for Organization, Process and Architecture, 2002.

[6] T. Dybå, T. Dingsoyr, Empirical Studies of Agile Software Development: A systematic review, Inform. Softw. Technol. (2008), doi:10.1016/j.infsof.2008.01.006

[7] James O. Coplien, Neil B. Harrison: Organizational Patterns of Agile Software Development, Prentice Hall, 2005.

[8] Paloma Caceras, Francisco Diaz, Esperanza Marcos: Integrating an Agile Process in a Model Driven Architecture. http://www.sciweavers.org/publications/integrating-agile-process-model-driven-architecture Accessed in May 2012.

[9] Marko Boger, Toby Baier, Frank Wienberg, Winfried Lamersdorf: Extreme Modeling, 2000. http://vsis-www.informatik.uni-hamburg.de/getDoc.php/publications/70/XM.pdf Accessed in May 2012.

[10] Pritha Guha, Kinjal Shah, Shiv Shankar Prasad Shukla, Shweta Singh: Incorporating Agile With MDA Case Study: Online Polling System. International Journal of Software Engineering & Applications (IJSEA), Vol.2, No.4, October 2011.

[11] Henrik Kniberg: Kanban Vs Scrum. http://www.infoq.com/minibooks/kanban-scrum-minibook Accessed in May 2012.

[12] Oliver Sims: Enterprise MDA or How Enterprise Systems Will Be Built. MDA Journal, September 2004.

[13] Arie van Derusen, Eelo Visser and Jos Warmer: Model Driven Software Evolution: A Research Agenda. CMSR 2007 Workshop on Model-Driven Software Evolution (MoDSE) Amsterdam 2007.

[14] Nourchene Elleuch, Adel Khalfallah and Samir Ben Ahmed: Software Architecture in Model Driven Architecture, IEEE, pp 219-223, 2007.

[15] Hailpern B, Tarr P: Model-driven Development: The good, the bad and the ugly. IBM Systems Journal, Vol.45, No.3, 2006.

[16] I. Lazar, B. Parv, S. Motogna, I-G Czibula, C-L Lazar: An Agile MDA Approach For Executable UML Structured Activities

[17] James O. Coplien, Gertrud Bjornvig: Lean Architecture for Agile Software Development, Wiley 2010.

[18] Scott W. Ambler: Agile Model Driven Development Is Good Enough. IEEE Software 2003.

[19] Scott W. Ambler: Agile Model Driven Development, http://www.xootic.nl/magazine/feb-2007/ambler.pdf Accessed in May 2012.

[20] SunGard, www.sungard.com. Accessed in May 2012.

[21] TNP SDK documentation: SunGard Front Arena

[22] Mary Poppendieck, Tom Poppendieck: Lean Software Development, An Agile toolkit. Addison Wesley, 2005.

[23] Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming, Addison Wesley 2000.

[24] MDA, www.omg.org/mda. Accessed in May 2012.

[25] Martin Fowler: Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.

[26] Vitaly Khusidman, William Ulrich: Architecture-Driven Modernization: Transforming the Enterprise. www.omg.org Accessed in May 2012.

[27] Thomas O. Meservy, Kurt D. Fenstermacher: Transforming Software Development: An MDA Road Map. IEEE Software, September 2005.

[28] S. W. Ambler, Generalizing Specialist: Improving Your IT Carrer Skills (2012), http://www.agilemodeling.com/essays/generalizingSpecialists.htm. Accessed in May 2012.

[29] Ray Carroll, Claire Fahy, Elyes Lehtihet, Sven van der Meer, Nektarios Georgalas, David Cleary: Applying the P2P paradigm to management of large-scale distributed networks using Model Driven Approach, Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP Volume, Issue , 3-7 April 2006 Page(s):1 – 14.

[30] Kent Beck: Test-Driven Development By Example, Addison Wesley, 2003.

[31] Martin Fowler: Continuous integration, http://martinfowler.com/articles/continuousIntegration.html Accessed April 2012.

[32] Michael Karlesky, Greg Williams, William Bereza, Matt Fletcher: Mocking the Embedded World: Test-Driven Development, Continuous Integration and Design Patterns. Embeded System Conference Silicon Valley, April 2007.

[33] Ron Jeffries, Ann Anderson, Chet Hendrickson: ExtremeProgramming. Addison Wesley, 2001.

[34] Yuefeng Zhang, Shailesh Patel: Agile Model Driven Development In Practice. IEEE Software 2010.

[35] Vincent Mahe, Benoit Combemale, Juan Cadavid: Crossing Model Driven Engineering and Agility – Preliminary Thoughts on Benefits and Challenges, 2010. ECMFA 2010.

[36] Dave Thomas: Agile Evolution – Towards The Continuous Improvement of Legacy Software. Journal of Object Technology, vol. 5, no.7, September-October 2006, pp.19-26

[37] Maria Istela Cagnin, Jose Carlos Maldonado, Fernao Stella Germano, Paulo Cesar Masiero, Alessandra Chan, Rosangela DelossoPenteado: An Agile Reverse Engineering Process based on a Framework

[38] Maria Istela Cagnin, Jose Carlos Maldonado, Fernao Stella Germano, Rosangela DelossoPenteado: PARFAIT: Towards a Framework-based Agile Reengineering Process

[39] Vinicius Durelli, Rosangela Penteado, Simone de Sousa Borges, Matheus Viana: An iterative reengineering process applying Test-Driven Development and Reverse Engineering Patterns, INFOCOMP – Special Edition, p. 01–08, fev. 2010

[40] Charles Darwin: The Origin of the Species, 1859.