An Agile Model-Driven Development Approach

A case study in a finance organization

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

***Abstract*–In the Sungard Front Arena, current software portfolio a business functionality called Market Server Capability (MSC) is embedded and duplicated in many components. By the application of Agile and Lean principles on model-driven development, we will get an Agile approach for constructing the architecture of a new MSC definition which will eliminate the duplication and inconsistency, while still maintaining a short implementation phase. The resulting architecture has a single modeling level, with merged PIM and PSMs. The model is designed by reverse engineering of the legacy code in a Test Driven Development fashion.**

Keywords-agile; lean; MDD; TDD; finance

I. INTRODUCTION

SunGard is a large, world-wide 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. [1].

The Front Arena system includes functionality for order management and deal capture for instruments traded on electronic exchanges. 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 (AMAS) provide services such as supplying market trading information, entering or deleting orders and reporting trades for a market.

Clients and AMAS 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 [2]. 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.

A. Problem description

When a new market (AMAS) is introduced, the information about functionality that the new market offers (which transaction i.e., TNP messages are supported) should be added to each client. MSCs describe market trading transactions (Orders, Deals, etc.), which command are supported for them (entering, modifying, etc.) and which attributes and fields could be accessed on the markets (Quantity, Broker, etc.). This information is presently hard-coded into each client application. New client application releases need to be done before the customers can start using the new AMAS. 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 AMAS.

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. Two main problems with this architecture are:

- Hard-coded MSC definition, requiring the recompilation of components when a new MSC is introduced
- Duplication of the MSC definition, introducing the risk for data inconsistency.

These problems will be resolved in the future by introducing 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 AMAS and all client components). Until then all components have to support the hard-coded fashion.

 ISBN: 978-1-61208-165-6

All new components, which will be developed during this time, have to support the hard-coded MSC way also. That is why there is a need to find an intermediate solution which will remove the duplication and which will be used under the transition phase. Since such an architecture will not be long lived company management put some time and resource constraints on the implementation. The question we address in this paper is how to create such intermediate solution, taking all conditions and constraints into account.

Introduction and problem description are presented in Section I. Section II explains, in more details, the architectures of both the present and the DMC solution as well as it introduces reasons for having an intermediate solution. In Section III, requirements and constraints are explained. The produced intermediate solution, an Agile MDD approach, is presented in Section IV. In Section V, the benefits are discussed of applying Agile and Lean principles on the MDD. Finally, Section VI presents our conclusion.

II. ARCHITECTURE OF THE MSC DEFINITION

A. The present architecture

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. 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 AMAS 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 AMAS 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.

B. Dynamic Market Capabilities architecture

We have already done design plans for the new DMC architecture. In the DMC architecture each AMAS will be responsible to provide, to the client components, information about the MSC that the AMAS supports. The description of the MSC that the AMAS supports will be saved in one XML file. An example of an extract from a XML file, containing the MSC definition for the AMAS called OMX, is presented in the Figure 1. In this example, a MSC defines that the market OMX supports trading transaction order with the following commands: enter, modify and delete, combined with the following fields: price and quantity.

```
<MarketCapability id="200" MarketServer="OMX">
 <Object name="Order">
  <Commands>
   <Command name="Enter"/>
   <Command name="Modify"/>
   <Command name="Delete"/>
   ----------------------------
  </Commands>
  <Fields>
   <Field name ="Price"/>
   <Field name ="Quantity"/>
   ----------------------------
  </Fields>
 </Object>
</MarketCapability>
```

Figure 1. Market Server Capabilities for market OMX

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

C. Transition phase

The decision is that all AMAS components and all client components should be upgraded to the DMC architecture. But this transition is a complicated job. There are over 30 AMAS 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, will 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 can benefit from it.

III. INTERMEDIATE SOLUTION

We work according Scrum in the company, trying to apply Lean and Agile software development

 ISBN: 978-1-61208-165-6

philosophy. One of the key principles of the Lean philosophy is to detect and eliminate wastes [3]. 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

In order to eliminate the duplication of data we need a centralized MSC definition. In order to be able to provide support for the MSC definition in different programming languages we need to generate code in different programming languages, from the centralized MSC definition. We need a programming language independent architecture. First we considered a solution, where all client components would be refactored to reference the same central definition file, but 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. That is why we believed that the Model-Driven Architecture (MDA) [4] approach can be the most suitable solution for the intermediate architecture. 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" [5].

The main idea is to have just one source, a union of all present MSC definition that is programming language independent. From such a source, which will 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 will not be changed, the MSC definition will still be hard coded. Such a solution does not require the refactoring of the client components. But the way how the developers work will 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 AMAS), 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.

A. 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. A small survey which we performed showed that the MDA approach hasn't been used within the company and that a majority of the developers has never used this approach and that the UML modeling is not used in general. Also, the introduction of the full scale MDA usually implies: a long starting curve, which we cannot afford having a short time-frame and the usage of the MDA tools, which cannot be used since developers don't have enough knowledge about them and there is no possibility to invest in learning. In the following section it will be described how we managed to overcome these problems and limitations.

IV. AGILE MDD APPROACH

Our goal is to find an intermediate solution with a MDA philosophy, which satisfies 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 [6]:

- "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 [7]"

 ISBN: 978-1-61208-165-6

- "Simplicity is essential": We have to find an applicable solution that is simple, keeping in mind that simple does not have to mean simplistic [8].

In that way we got our own Agile MDD approach, an applicable intermediate solution, which will be described in detail in the following section.

A. Agile modeling and code generators

We need to model the MSC definition registry. This modeling can be done on the different modeling levels and in the different modeling languages. Considering the limitations, the UML modeling cannot be accepted as a modeling solution in our project: it is not used in general and there is no time for learning. 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.

We have created two models. One is a logical model which describes the entities in the MSC definition registry. Another is the MSC definition registry by itself, expressed in a XML dialect. As a consequence of that, the logical model is expressed as a XSD schema and is used to validate the entries in the registry.

The MDA defines different model categories, like a Platform Independent Model (PIM) and a Platform Specific Model (PSM) [5]. This is an important issue if there are plenty of different platforms with specifications that differ very much. In our case the different PSMs didn't differ too much from each other and, at the same time, didn't differ too much from the PIM either. In order to keep it simple we made a pragmatic solution: to have just one model, which contains all info for all programming languages. The code generators have the responsibility for creating the right MSC information to the corresponding programming language.

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" [9] is achieved for the code generators. The maintainability is also better if all developers can maintain/develop the transformations.

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. 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 [10]. 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. 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 development. Round tripping in combination with the Test Driven Development (TDD) [11] will be explained in more detail in the following section.

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 AMAS 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 test, which reflects one Spike scenario, both to develop the reverse engineering tool and the code generators, but with the input from the different sources: the legacy code was used as input when the reversing tool was developed and the generated code files was used as input when the code generators were developed. Our TDD process is presented on the “Fig. 2”. Modules presented on “Fig. 2” are parts of our MDD approach where the following abbreviations are used: RE for the reversing engineering tool, CG for the code generators, LC for the legacy code and GC for the generated code.

 ISBN: 978-1-61208-165-6

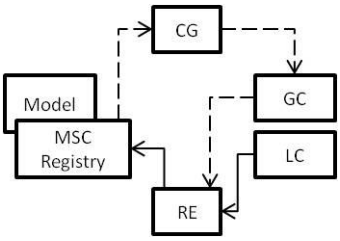

Figure 2. Our TDD process

Our TDD process will be described now through one real Spike scenario. The chosen Spike scenario 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 extract 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 full arrows on the Figure 2. The TDD logic for the code generators were more complicated. 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, trying to get all markets, but this time from the generated code instead (which was empty when we started), via the reversing tool (where we have some code implemented). We developed the code generators using the mentioned test. The final goal was to get the same entries in the MSC registry by the reversing of the generated code as we got by the reversing of the legacy code. 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. This process is marked with dashed arrows on the Figure 2. 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 AMAS 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.

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.

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.
- 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 can reuse some of these tests later on, for the DMC architecture testing.

D. Automation

We have automated some of the processes, supporting a kind of continues integration also. We reduced the amount of work and time spent for 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.

V. AGILE AND LEAN PRACTICES IN MDD

The Agile and Lean methods are light in contrast to the MDA that can become complex, because of all standards and OMG recommendations. Through the

 ISBN: 978-1-61208-165-6

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.

"Eliminating waste": Eliminating the duplication of information was also according to the XP's principle "Never duplicate your code" [9]. 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.

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

"Empower the team": Roles are turned – the managers are taught how to listen to the developers [3]. Despite the fact that managements put 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.

"Spike principle" applied on the reverse and round-trip engineering made the introduction of the TDD philosophy spontaneous and natural.

"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.

"Welcome changing requirements, even late in development." The case-study 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.

A. *Benefits of the Agile MDD approach*

We got a lot of benefits by introducing the Agile MDD approach. Now we will list them:

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.
2. We introduced the TDD approach, showing the effectiveness of such an approach.
3. 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.

VI. Conclusion and Future Work

The main point of this paper was to show how Lean and Agile principles helped us with producing an intermediate solution, with a short implementation phase, for the architecture of the MSC definition. In that way we coped successfully with the management constraints, achieving the implementation within the short time-frame and without investment in change management.

Our Agile MDD approach is based on the general MDA idea but is shaped then with the Lean and Agile principles. "Eliminating waste" helped us to detect main wastes. The most important was the duplication, which we eliminated by applying the MDA philosophy. "Simplicity" Agile principle reduced the MDA concept to the single modeling level, expressed in the XML dialect. By being aware of "Think big act small", we could produce such an intermediate solution, which can be easily improved in the long-term solution. The TDD logic improved the development efficiency and decreased the total time spent on the development and testing. We got a simple and applicable solution which will easily grow to a more complex one.

"A complex system that works has usually been evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a simple system. [12]"

References

[1] SunGard, www.sungard.com. Accessed in May 2011.

[2] TNP SDK documentation: SunGard Front Arena

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

[4] James McGovern, Scott Ambler, Michael Stevens: A practical guide to Enterprise Architecture. Prentice Hall PTR, 2003.

[5] MDA, www.omg.org/mda. Accessed in May 2011.

[6] AgileManifesto, www.agilemanifesto.org. Accessed in May 2011.

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

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

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

[10] 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.

[11] Michael C. Feathers: Working Effectively with Legacy code. Prentice Hall PTR, 2005.

[12] John Gall: Systemantics: How Systems Really Work and How They Fail. Quadrangle, 1975.

 ISBN: 978-1-61208-165-6