Experiences from representing software architecture in a large industrial project using model driven development

Anders Mattsson[1]
Björn Lundell[2]
Brian Lings[2]
Brian Fitzgerald[3]
[1] Combitech AB, P.O. Box 1017, SE-551 11 JÖNKÖPING, Sweden
anders.mattsson@combitech.se
[2] University of Skövde, P.O. Box 408, SE-541 28 SKÖVDE, Sweden
{bjorn.lundell, brian.lings}@his.se
[3] Lero – the Irish Software Engineering Research Centre
University of Limerick, Ireland
brian.fitzgerald@ul.ie

Abstract

A basic idea of Model Driven Development (MDD) is to capture all important design information in a set of formal or semi formal models that are automatically kept consistent by tools. This paper reports on industrial experience from use of MDD and shows that the approach needs improvements regarding the architecture since there are no suggested ways to formalize design rules which are an important part of the architecture. Instead, one has to rely on time consuming and error prone manual interpretations, reviews and reworkings to keep the system consistent with the architecture. To reap the full benefits of MDD it is therefore important to find ways of formalizing design rules to make it possible to allow automatic enforcement of the architecture on the system model.

1. Introduction

Model-Driven development (MDD) [1] is still an emerging discipline [2, 3] where the prevalent software-development practices in the industry are still immature [4]. The success of MDD in practice is currently an open question [4] and there is a lack of reported experience on MDD in large industrial projects. One aspect of such projects is the importance of architecture. This report presents experience from architectural work practices using MDD in a large industrial project.

There exist several approaches to MDD such as OMG's MDA [5], Domain-Oriented Programming [6], and Software Factories [7] from Microsoft. A basic idea of MDD is to capture all important design information in a set of formal or semi formal models that are automatically kept consistent by tools to raise the level of abstraction at which the developers work and to eliminate time consuming and error prone manual work in keeping different design artifacts consistent, or to cite Brent Hailpern and Peri Tarr:

> "The primary goal is to raise the level of abstraction at which developers operate and, in doing so, reduce both the amount of developer effort and the complexity of the software artifacts that the developers use" [4].

An important design artefact in any software development project, with the possible exception of very small projects, is the software architecture:

> "The architecture serves as the blueprint for both the system and the project developing it. It defines the work assignments that must be carried out by design and implementation teams and it is the primary carrier of system qualities such as performance, modifiability, and security – none of which can be achieved without a unifying architectural vision. Architecture is an artifact for early analysis to make sure that the design approach will yield an acceptable system. Moreover, architecture holds the key to post-deployment system understanding, maintenance, and mining

efforts. In short, architecture is the conceptual glue that holds every phase of the project together for all of its many stakeholders."[8]

A primary role of the architecture is to capture the design decisions, the rules that have to be followed in the detailed design of the system, made by the architect to ensure that the system meets its quality requirements. Current architectural methods state that this shall be done by specifying patterns [9] to be followed in the design [8, 10, 11]. Since there is no suggested formal way to describe patterns or pattern usage there is a problem in describing the architecture in an MDD context where the idea is to use formal models for all important design artifacts. This means that with this way of using MDD one cannot achieve the potential benefits of MDD for the architecture. This report shows that this can pose big problems in large industrial software development projects where architecture is very important.

2. Background

In the year 2000 the company Combitech Systems (CS) faced the challenging task of developing a software platform for a new generation of digital TV set top boxes for the DVB standard. The development had to be done in 18 months under strict quality requirements and on a completely new, customized hardware platform developed in parallel by another company.

The challenge consisted of the following elements:

- Short time to market; since CS were in competition with other developers the product had to be ready at a fixed date.
- Since the hardware was to be developed in parallel with the software there would be little time to integrate the two, leading to a significant risk of errors and misunderstandings that would have to be handled by very late changes in the software. Therefore CS needed to test the software on a range of platforms, from host PC to real target hardware with several intermediate hardware platforms.
- The requirements were a moving target where the initial requirement specifications would be overridden by acceptance tests delivered late in the project.
- Maintenance would be long and had to be very cost effective.
- There was a requirement to be able to differentiate the product, releasing different variants for different markets. New variants had to be developed and maintained efficiently.
- Products would be competing on performance and quality; the product with the best performance and quality would win the final contract.

So we found ourselves with the challenging task of building high quality, high performance software supporting efficient maintenance and a lightweight spawning of variants. This had to be done on an imaginary hardware platform with requirements that would change during the development on a fixed, very short, timescale.

At the time of the project, CS was a Swedish consultancy company specialising in services for developing embedded real-time systems. CS had approximately 250 consultants of which about 75 where involved in the project. The total effort of the project was 100+ person years expended over an 18 months period. The project was distributed across five sites located in Jönköping, Linköping, Växjö, Helsingborg and Trollhättan.

CS developed the platform as a phased fixed price (price negotiated for each phase) assignment for a customer company. CS had full responsibility for the software but the customer wanted control of the architecture of the software so the architecture team was actually managed by the customer and not by CS, although they were CS employees. This meant that there was a need for a counterpart on the CS side, making sure that the architecture also was feasible within the budget. This was the project role of one of the authors of this paper, technically responsible within the CS project management team, with prime responsibility to negotiate the architecture with the architecture team.

3. Rationale for choosing MDD

Importantly, CS had relevant previous experience from maintenance on the previous generation of the product. There was to be more functionality and completely different hardware, but the base functionality would remain the same. So there was pretty good knowledge about the capabilities of the product and how to realize them. The older product was developed by another company. When CS took over the maintenance of the product the company had difficulties both in maintaining it and adapting it for different markets. This was caused by poorly structured code in C with documentation that was obsolete through lack of maintenance.

Within CS there was also extensive experience of working with models in UML and earlier modeling languages, both for analysis and design models. CS also had experience of using rule-based transformation from design models directly into code which executed on a platform. However, in real projects CS had only done the transformations manually so far, although experimentation with automatic code generation had been done to a degree where the company felt ready to apply it in a real project. Given this experience there was conviction that model driven development would help in taking on the challenges of the project.

3.1. Agility

The approach made it possible to work in an agile way where one could quickly go from requirements to tested implementation without having to skip documentation, something very important for the maintainability of the product.

3.2. Test on several HW platforms

MDD would make it possible to test most of the code without access to the actual hardware by simply generating code for different platforms, as the project gained access to hardware that became increasingly similar to the final target.

3.3. Performance

For performance reasons it was important to pay special attention to generated code, which had to be efficient. But if this requirement was satisfied, the ability to keep the work on a higher abstraction level would enable refactoring with less effort, which in turn made it easier to fine tune the system to obtain better performance.

3.4. Product variability

Of course product variability is primarily an issue for the architecture, but there was also a thought that MDD would make it easier to both communicate and enforce the architecture since a big part of it would actually be represented directly in the system model.

3.5. Tool selection

Since there was very little calendar time an out-of-the-box tool solution was required that would give the following:

– modeling in standard UML, to minimize the need for training;
– generation of code with good performance on our target platform, the host platform and the G1 platform since this would be used as an intermediate test platform;
– 100% of the developed code generated from the model (to avoid synchronization problems with code and model) having at the same time the ability to use pure C++ code where there was a need (to remove any risk that one could not do some things that it was possible to do in the traditional way);
– the ability to debug at the model level;
– support for distributed team working;
– a high probability that the vendor would continuously improve the tool towards the requirements of embedded real-time system development.

After a short evaluation Rhapsody from Ilogix was selected as the only tool that seemed to satisfy all these requirements.

The selection of Rhapsody meant that a system is designed in a UML model with action code in C++. This model (the system model) is then automatically converted to full production code in C++ by the tool. The generated code uses an execution framework (OXF), provided by the tool, to abstract out the target execution platform. In terms of MDA [5], the model built in Rhapsody corresponds closest to the PIM, where OXF and the generated code corresponds to the PSM.

4. Capturing the architecture

To be able to meet the deadline quite a lot of developers had to work in parallel with different parts of the system. This required a stable architecture before the project scaled up. A product line architecture approach [10] was selected to address the requirements for efficient development and maintenance of product variants. In addition to this there were other important quality requirements such as portability (the software was anticipated to outlive the hardware) and overall performance, which had to be handled by the architecture. So, architecture was very important.

One of the first problems to face was how to represent the architecture when working in a MDD context. A basic idea in MDD is to use models instead of documents to represent the requirements and design of a system and to generate the implementation code from these models. The traditional way of representing the architecture is in a document that guides and constrains the detailed design. In model oriented methods, like Rational Unified Process, where models have replaced requirement specifications and design descriptions you still represent the architecture in a document. Our aim was higher than this; the aim was, as much as possible, to automatically connect the architecture to the design to minimize both the maintenance problem and the effort to enforce the architecture on the design. In the end the project settled for the following approach:

- High level structure was to be captured in the system model.
- Architectural design rules were to be captured in a combination of natural language in a text document and a framework in the system model.
- Example components would be designed in the system model illustrating how to follow some of the architectural rules.

4.1. High level structure

The high level partitioning of the system, down to a level at which individual components were to be developed by individual developers, was captured in a package hierarchy populated with classes acting as facades [12] for the actual components.

4.2. Architectural design rules

A number of patterns and rules were to be followed when the components in the model were designed. To support these patterns and rules an architecture package with an executable framework was developed as a part of the design model (Figure 1).

In principle, the framework contained abstract base classes representing the core abstractions of the system, relations between them and operations that were to be overridden in specializations of the base classes. The framework also contained full implementations of some basic mechanisms that operated solely on the level of the abstract base classes, such as inter-process resource locking and component registry handling (registering, allocation and de-allocation).

Unfortunately the project didn't reach all the way to capturing the architecture in a formal model. There was a need to use natural language to express the rules on how to use the architectural framework to design the components in the architecture. There were more than sixty rules that had to be followed. Below is a small subset of these rules.

- "Any *arcComponent* with behavior similar to an *arcPort* should be a specialization of *arcPort*."
- "All specializations of *arcPort* may have one overloaded method for each of the methods Open(), Close() and Write()".
- "All specializations of *arcPort* may have several methods for the method Ctrl(). These methods shall be named as ctrl_<specific_name> and may not change the parameter list of the base class, except for specialization of the parameter classes given for the base class. However, a method may omit the second (parameters) parameter."
- *"arcPort*::Write() shall be used to stream data to a port's data output stream."
- *"arcPort*::Ctrl() shall be used to control and manipulate a port's properties."
- "All specializations of *arcPort* must use its ***parent's*** implementation of the base class method for their respective purposes. "
- "All specializations of *arcPort* require a specialization of the *arcPortUser* interface."
- "All specializations of the *arcPortUser* interface base class may have one overloaded method for each of the methods RxReady(), TxDone() and GetMem()."
- "All specializations of the *arcPortUser* interface base class shall have one overloaded CtrlAck() method for each of the asynchronous ctrl_<name> methods."

4.3. Providing Example components

To guide the developers in how to develop the components using the architectural framework, a number of example components were developed by the architects as a part of the model. In addition to showing the design the examples also showed how to use different diagrams to capture the design.

5. Lessons learned

5.1. Manual enforcement and guidance is time consuming and error prone

Using natural language to describe architectural design rules meant that we had to rely heavily on manual reviews to enforce conformance with the architecture. The rules also proved to be hard to

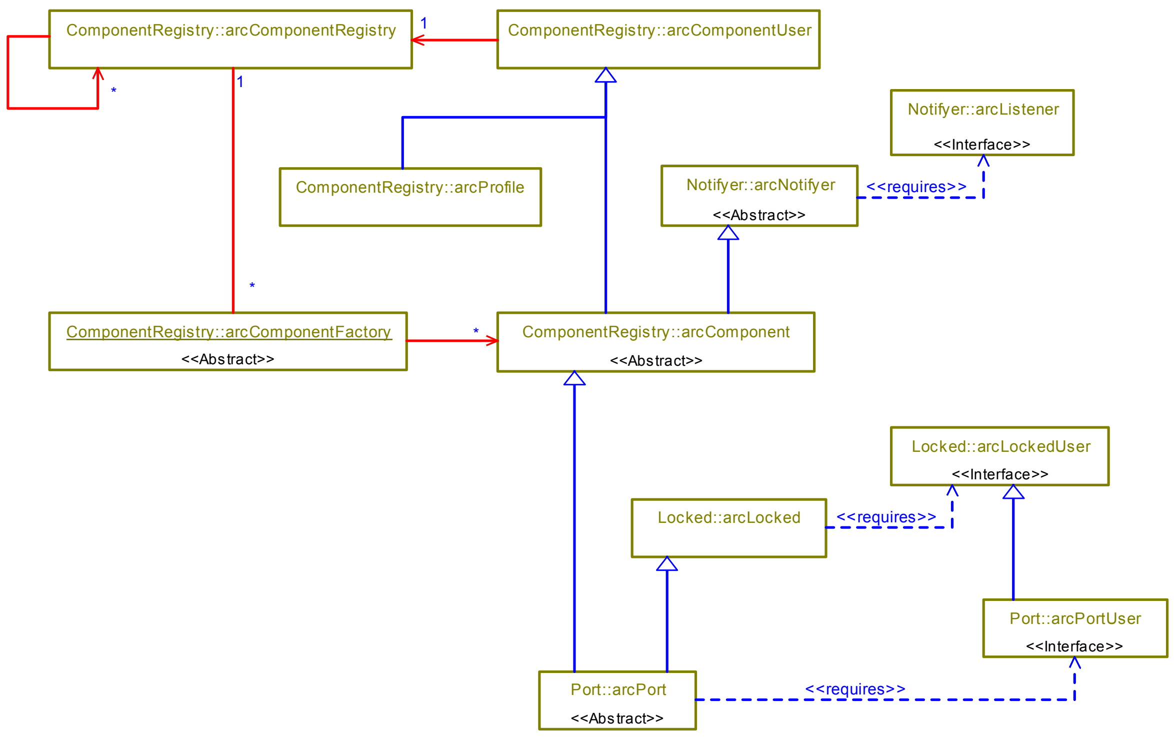

Figure 1 Architectural framework

enforce and ambiguous, and thus prone to different interpretations.

Several developers reported having a hard time trying to understand and follow all of the detailed rules. This was manifested by the fact that a lot of time was spent by the architects on reviews and by the fact that corrective actions were generally needed after reviews. Sometimes this meant that a lot of reworking had to be done since reviews were often held when design had continued too long. This was caused by work overload on the architects, which in turn was caused by a lot of effort having to be spent on communicating and reviewing the designs generated by the different teams.

5.2. Late changes to design rules are time consuming and error prone.

Since part of the architecture had to be manually transformed into model constructs throughout the model, it required a lot of effort to change the rules or the architectural framework. This made it almost impossible to make late changes to the architecture. So the initial idea that MDD would allow us to make late changes to the architecture in order to fine tune it against its quality requirements did not hold.

6. Summary

As we have seen in this example MDD allows some of the architecture to be captured in formal models but current methods and tools for MDD do not support formal representation of architectural rules that go beyond structure. These rules are instead typically represented as text in natural language to be followed by developers in the detailed design. This means that one has to rely on manual interpretations and reviews of rules that often are hard to understand and ambiguous. This requires a lot of effort from developers and places a heavy burden on architects, who represent a sparse resource and so quickly become a bottleneck in the development process. It is also bad for quality, since manual routines are error prone, especially when time is scarce, which it is if you are a bottleneck.

Another problem is that since the architectural rules are manually introduced throughout the models it is very hard to make late changes to them: it often involves massive reworking. These rules are typically an attempt to find a compromise between conflicting solutions to different quality requirements. Early in the project you generally don't have enough information to make an optimum solution but you still have to have a stable architecture before you can ramp up the project. So, you have to guess some things. Some of these guesses are probably wrong. This means that you generally have a need to change some of your architectural rules late in the project, so the ability to have a tool to automatically update your model when you change the rules could save a lot of time, or enable optimization of the system.

Therefore, if one could capture architectural rules in a way that would make it possible to automatically enforce them on, and propagate changes to the system model, it would represent an opportunity to dramatically improve both efficiency and quality.

From this experience we believe that to obtain the full benefits from MDD there is a need for support of formal modeling of architectural rules and automatic enforcement of these rules on the generated models of the system.

7. References

[1] B. Selic, "The pragmatics of model-driven development," *IEEE Software,* vol. 20, pp. 19-25, Sep-Oct 2003.

[2] D. C. Schmidt, "Model-driven engineering," *IEEE Computer,* vol. 39, pp. 25-31, Feb 2006.

[3] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, "Developing applications using model-driven design environments," *IEEE Computer,* vol. 39, pp. 33-40, 2006.

[4] B. Hailpern and P. Tarr, "Model-driven development: The good, the bad, and the ugly," *IBM Systems Journal,* vol. 45, pp. 451-461, 2006.

[5] MDA, "MDA Guide version 1.0.1," OMG, 2003.

[6] T. Dave and M. B. Brian, "Model driven development: the case for domain oriented programming," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* Anaheim, CA, USA: ACM Press, 2003.

[7] J. Greenfield and K. Short, *Software factories : assembling applications with patterns, models, frameworks, and tools*. Indianapolis, IN, USA: Wiley Pub., 2004.

[8] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 2nd ed. Boston: Addison-Wesley, 2003.

[9] F. Buschmann, *Pattern-oriented software architecture : a system of patterns*. Chichester ; New York: Wiley, 1996.

[10] J. Bosch, *Design and use of software architectures : adopting and evolving a product-line approach*. Reading, MA: Addison-Wesley, 2000.

[11] L. Bass, M. Klein, and F. Bachmann, "Quality attribute design primitives and the Attribute Driven Design Method," *Software Product Family Engineering 4th International Workshop, PFE 2001 Revised Papers Lecture Notes in Computer Science Vol 2290,* vol. 2290, pp. 169-86, 2002.

[12] E. Gamma, *Design patterns : elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, 1995.