LESSONS LEARNED ON THE DEVELOPMENT OF AN ENTERPRISE SERVICE MANAGEMENT SYSTEM USING MODEL-DRIVEN ENGINEERING

Rodrigo García-Carmona[1], Juan C. Dueñas[1], Félix Cuadrado[1], José Luis Ruiz[2],

[1] Universidad Politécnica de Madrid, ETSI Telecomunicación,
Ciudad Universitaria s/n. 28040, Madrid, Spain
{rodrigo, jcduenas, fcuadrado}@dit.upm.es
[2] Indra, c/José Echegaray 8,28108
Parque Empresarial,Las Rozas, Madrid, Spain
jlrrevuelta@indra.es

Abstract. MDE (Model-Driven Engineering) techniques and tools promise to reduce the complexity and effort of software development. However, although this approach is widely known, there are few reports on its application to real enterprise developments. In this article we present our experience in the creation of an enterprise service management system using MDE. This is a complex system, as it must cope with the heterogeneity and distribution of both software services and their runtime infrastructure. Also, enterprise systems must support multiple non-functional requirements. These requirements are usually fulfilled by enterprise framework solutions, which require a steep learning curve. To overcome these problems we have applied the aforementioned MDE methodologies, starting from a generic information model and partially generating the system from it. We detail the pitfalls found and discuss the strong and weak points of the followed process.

Keywords: MDE, Enterprise Systems, Code Generation, Report on Experience.

1 Introduction

Enterprise service management systems are very complex and costly to develop. Its purpose is the control and automation of the life cycle of software services across a distributed and variable environment. They must adapt to heterogeneous distributed environments, controlling, processing and managing large amounts of data. Also, their internal architecture must support rapid system evolution, in order to keep pace with new business requirements. On top of that, non-functional characteristics such as robustness and security must be maintained.

When we were confronted with the task of developing this kind of system we looked for alternatives in order to simplify its complexity. MDE (Model Driven Engineering) [1] promises to speed the development and reduce complexity by the abstraction of real entities into models, and the application to them of automatic code generation operations. Therefore, we opted to integrate MDE techniques and tools in our development process to try to make use of these capabilities.

In this article we present a report on our experience developing the system. Next section provides an overview over the most important concepts of MDE. Section 3 provides additional information about the target domain, the reasoning behind the adopted approach and the tool selection. The fourth section provides additional details on the case study, detailing the generation processes and system architecture.

Finally, complete discussion on the results and lessons learned after the development is provided, offering some guidelines for similar experiments.

2 Model-Driven Engineering

MDE is a methodology based on the use of abstractions of entities called models. They only contain the information relevant to a particular domain, being oblivious to the remaining details. Their constraints, characteristics and semantics are well defined through metamodels (which are also models), avoiding ambiguities.

The OMG (Object Management Group) is the main standardization organization for MDE languages and processes. Some of its most relevant specifications are MOF [2], a language used for the definition of metamodels, or UML, which is in turn defined using MOF.

MDE processes consist of several kinds of transformations, being model to model and model to text the most prominent. An example model to model transformation allows the enrichment and modification of the definitions of Platform Independent Models (PIM) until they are transformed to Platform Specific Models (PSM). These processes can be automated through the use of transformation languages, such as QVT (Query/View/Transformation).

Code generation activities are the most representative applications of model to text transformations. Under some circumstances, a PSM with enough information can be used to automatically generate the actual source code of the system. In less ideal cases, the generated code base is completed with manual implementation.

Adopting MDE can provide many benefits to the development process. It allows the partial (and in some cases complete) automation of several activities and eases the response to changing requirements or domain specifications. Also, it allows the expression of the problems that need to be solved in a more comprehensible way, providing to architects a clearer view of the system entities.

Applying MDE to the development of enterprise systems has the potential to greatly help in the fulfillment of their particular characteristics [3]. Enterprise management systems present between them many similarities in the software infrastructure and basic requirements, such as communications, or data persistence. These requirements can be captured in model and transformation definitions.

The usage of MDE techniques allows the automation of specific operations and brings “information hiding” principles to the development process, fostering specialization. Work towards solving specific enterprise domain problems using MDE has been performed recently and has shown positive results [4, 5].

However, a considerable effort may be needed for the assimilation of these practices. Thus, the key limiting factor for its enterprise adoption is the availability of

a comprehensive and mature tool chain that seamlessly integrates with the development processes and the specific technologies.

3 Case Study Description

3.1 System Requirements

The system under development is an enterprise service management architecture. Its purpose is the control and automation of the life cycle of software products and services across distributed environments. This system will manage information about the physical structure of the target environment, its runtime state, the available software and services, and the dependencies between them. Moreover, it will interact with the physical elements through a well-defined information model, in order to abstract from the complexity and heterogeneity of enterprise systems.

The development of an enterprise system like the one described in this paper is a complex process. The system must be deployed over a distributed environment, and operate with an adequate quality of service, ensuring its high availability, fault tolerance, and scalability. Some representative non-functional requirements are:

- Information consolidation is a fundamental requirement for any management system. Runtime state, statistics, operation logs and system resources must be persisted, sorted and related between each other.
- System components are designed in a decoupled, distributed way, which in turn imposes a need to expose remote communication mechanisms.

As these requirements are common to most enterprise services, in recent years several frameworks and specifications have been developed to provide pre-packed solutions to these aspects. In fact, they have been so useful that its popularity has turned them into additional requirements for the developed services. However, the result is a framework sprawl where the complexity has shifted from the original requirements to a well-established architecture and technology base.

3.2 Technical Approach

After analyzing the characteristics and requirements of the system, we tried to address these concerns by adopting MDE techniques and tools in our development process. We wanted to achieve two main objectives: First, by using the code generation capabilities of MDE tools, we tried to reduce the development effort of the described system, improving productivity. Second, by selecting which parts of the system will be generated, we wanted to abstract as much as possible from the non-functional concerns and the enterprise frameworks, which were not familiar to the development team.

There was also an additional in factor supporting the adoption of this approach: the existing information model. As this model is the central element of the management system, it must be comprehensively defined in the analysis stage. This will provide us

with an initial input for the selected MDE tool chain. However, it is important to note that it only describes the information and not the system behavior.

In order to apply this approach it is necessary to choose a modeling solution. Although the metamodeling has a huge impact in which solution will be selected (must be powerful, flexible and based upon open and widely adopted standards), the specific requirements of our development process will fundamentally impact the tool support for modeling and code generation. We established the following criteria:

- Comprehensive Java code generation functionality from the available models. The system requirements mandate a Java development, supported by several enterprise frameworks.
- Maturity of the tools. An unfinished or beta solution should be discarded, as tracing errors caused by the code generation are very difficult and costly to detect.
- Out-of-the-box transformations for abstracting from the required frameworks and non-functional concerns (e.g. information persistence through ORM frameworks). Manually defined transformations will not be adopted, as they require the acquisition of a deep understanding in both the transformation language and the underlying framework. Because of that, we will partially adopt an MDE approach.
- Quality of documentation and gentle learning curve. As we will work over the MDE tools, a fundamental factor for its selection is the required effort for applying the technology to our specific problem.

3.3 Tool Selection

After comparing the decision criteria with the available models and tools we chose the following options:

We selected EMF (Eclipse Modeling Framework) [6] ECore as the modeling language for the definition of the information model. EMF is a modeling framework that provides both an implementation of EMOF (Essential MOF) named ECore and a set of supporting tools for defined metamodels, which automatically provide editors for defining model instances, a set of transformations between ECore, XSD and Java, XML-based model persistence and unit test cases. EMF is a very mature and popular project, which has fostered a very active open-source community around the project, providing multiple tools, languages and transformations on top of it.

As our system should support heavy workloads and preserve data integrity, we could not use the base XML serialization provided by EMF, needing relational database support instead. Teneo is an EMF extension that provides a database persistence solution by generating a direct mapping between ECore models and Java ORM (Object Relational Mapping) frameworks, automatically generating the mapping files from the ECore elements. Teneo supports two different types of ORM solutions, Hibernate and JPOX/JDO. We used Hibernate because is the de-facto industry standard (and compatible with the EJB 3.0 specification). It also offers a simplified management interface for the relational operations.

Another system requirement is the ability to distribute the components providing a Web Services remote communication layer on top of the business logic. Web Services

is the leading standard for enterprise distributed communications. It promotes contract-based design and loose coupling, through well-defined XML documents for both the contract definition and the information exchange. The contract is expressed through WSDL (Web Services Description Language) files.

The format of the messages in Web Services is specified inside the WSDL descriptor by XSD (XML Schema Definition). Since EMF allows the usage of XSD for the definition of metamodels, we wanted to use these XSDs to create part of the WSDL. For the implementation of Web Services we chose Spring Web Services, a contract-first Web Services framework which was part of our enterprise middleware layer.

The selected tools (EMF, Teneo, Spring-WS) partially address our requirements. They support the definition of both models and metamodels and their transformation to database mappings, WSDL files and Java source code. We chose these solutions discarding more generic transformation model tools because of the previously mentioned requirements (out-of-the-box functionality, abstraction from middleware layers, simplicity and ease of learning).

Figure 1 depicts the relations between these tools and how they work to generate the base artifacts for different aspects of the system (logic, persistence, and communications). In the middle box, EMF automates the generation of both Java classes and XSD files which represent the metamodels obtained from the ECore information model. On the data persistence layer, Teneo automates the generation of database mappings and schemas from the same ECore model that was used in EMF.

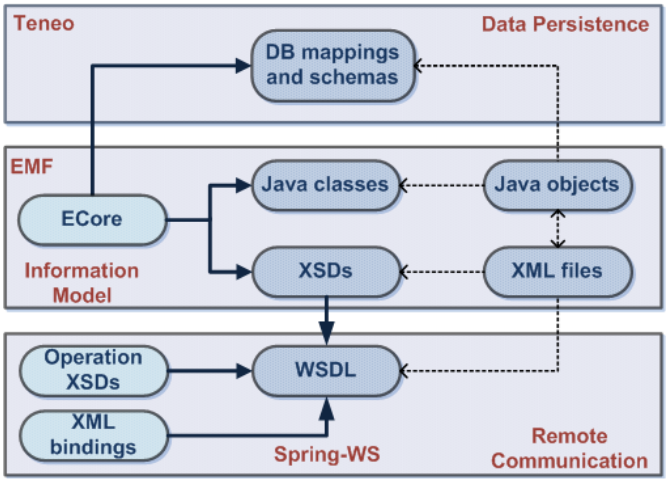

Fig. 1. Transformation flows.

Lastly, on the remote communication domain, Spring-WS generates a WSDL descriptor from the XSDs created in the information model layer, XSDs specifying the operations of the interface and XML bindings of the remote interfaces to Java code.

4 Report on Experience

4.1 System Description

As we have described previously, the developed system is a distributed enterprise application, with multiple entities collaborating to provide the required functionality. For its design we have followed a layered architecture, adopting the middleware open-source stack (Spring, OSGi, Hibernate, Web Services) for modular, enterprise applications. The adoption of middleware and framework components greatly reduces the coding effort, and promotes best practices for solving common concerns of every development project.

Figure 2 shows a model-focused structural view of one component of our distributed system. It shows three different areas. The system runs over a runtime environment, formed by hardware, operating system, a Java virtual machine and a set of provided libraries, On top of this substrate reside the models layer. These components are the result of our generation process. Finally, the third group is composed by the actual functionality of the application, the service layer. Developers should focus only on these elements, which are the business logic units, user interfaces, remote services, and inventory services. As the model layer provides automatic transformations it abstracts from the middleware infrastructure in charge of the remote serialization and persistence.

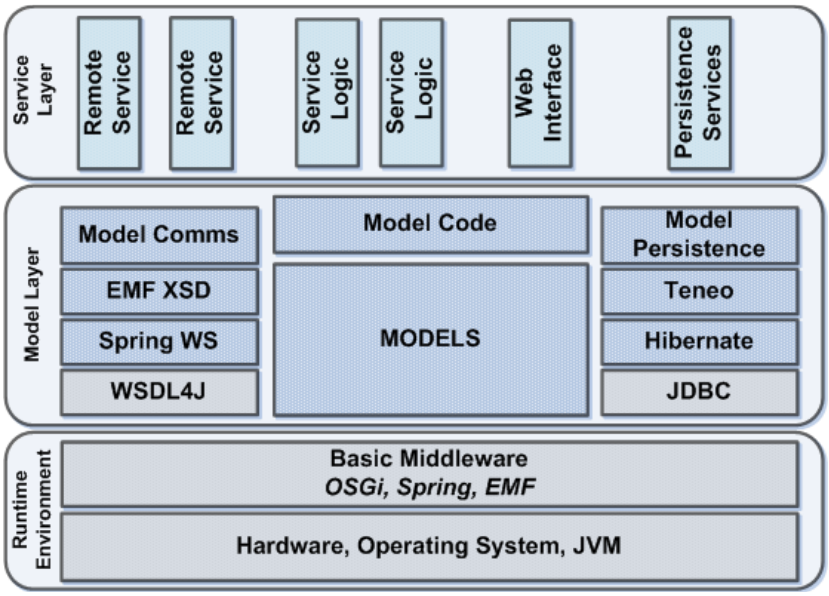

Fig. 2. System structural view.

4.2 Process Practices

With the characteristics of the selected tools and the requirements of the system in mind we defined a flow for the detailed design and implementation activities. Figure 3 shows its steps and the transitions between them.

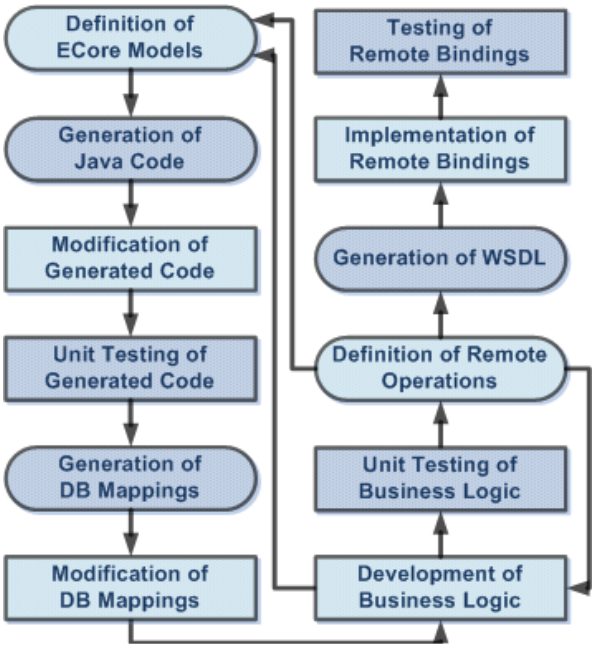

Fig. 3. Development process.

The application of MDE translates into the following tasks:

- Definition of models using MDE models and metamodels.
- Modification of already created models, in order to adapt them; either by using transformations (model-to-model) or by hand (model tuning).
- Generation of code from the models, mainly using model to code transformations.
- Modification of generated code (code tuning).
- Implementation of code not covered by MDE, which in our case pertains to the system logic.
- Testing of both the generated and manually created elements.

Concerning testing it is important to note that EMF generates unit tests that validate the generated source code. Therefore, the testing tasks can be performed with automatically created or hand-written tests.

5 Discussion

This section provides additional discussion on the followed approach after its completion. We will present both a small quantitative analysis of the finished system and a summary of the lessons we have learned. We think this information can be useful to not only evaluate the success of the approach but also improve similar processes.

5.1 Quantitative Analysis

To evaluate the generated code some metrics have been performed. The results of this analysis are depicted in Table 1.

The first two rows contain the most basic information that can be obtained: the raw number of lines of code and Java classes. It is important to note that the size of the modeled part weights roughly half of the system (60.000 lines of code excluding libraries). Most of this code contains the information model and the XML serialization engine.

The remaining rows comprise some software metrics that try to measure the quality of the code. Efferent couplings indicates how focused are the classes. The remaining metrics (cyclomatic complexity, number of lines per method and number of locals) indicate the complexity and comprehensibility of the code. All the values are averages for all the classes or methods.

Since generated and manually written code cannot be compared side by side, we compared the amounts of code of the model definitions and the generated elements. The model definitions span 1609 lines, the ratio is of 20.6 Java lines generated per line of model definition written.

Table 1. Code metrics.

Metric	Value
Lines of Code	33125
Classes	290
Average Efferent Couplings	6.91
Average Cyclomatic Complexity	2.06
Average Number of Lines per Method	13.63
Average Number of Locals	1.44

5.2 Lessons Learned

During the process we identified some critical risks for the success of the development with this approach. Most of these pitfalls could be avoided taking some factors into consideration. Further on, we expose the most remarkable issues:

Application of mature transformations. Our intent with the described generation process was to take models as a foundation, trying to abstract whenever possible of

the specific middleware for the previously described concerns, such as persistence or remote communications.

Although our experience was positive (used these capabilities seamlessly over the model layer), we found some problems using one of the transformation frameworks (Teneo 0.8): its data persistence service did not work as expected in common situations (updating operations). Detection of such failures was difficult because the source of problems could be in any of the layers, and we had to look into their source code, losing the theoretical advantages of abstraction. Therefore, tool and framework maturity are a fundamental risk to be assessed for adopting this type of approach.

Limits in the abstractions. We were also affected by the law of leaky abstractions [7], as the transformations hid useful concepts in the lower levels that could only be obtained by respecting these low-level constraints in the business logic (lazy loading from the database improves efficiency but imposes session management in the upper layer).

Model definition accuracy. The success of the complete development is heavily dependent on this. During our development, an error in the business logic was finally traced to a mistake in the definition of the information model. We expressed a relationship between elements as a composition instead of an aggregation, and the generated code did behave as we defined (but not intended).

Application of corrective changes. Probably the most important model transformation that a solution can offer is the generation of code. In our experience almost all the chosen solutions behaved perfectly on this matter. However, the generation process can in some cases be far from perfect and the generated code could not be used directly.

We experienced this drawback with Teneo. The generated mapping files had to be manually edited to solve various problems. The greatest time sink here was to trace the failure to the generated model and figure what tweaks were needed.

On the other hand, the automatic generation of unit test cases that EMF provided helped greatly to discard those models as the source of any failure.

Application of perfective changes. Sometimes the generated elements do not accomplish all the goals that have been set. In these situations the missing features have to be implemented into the generated code by hand. If the generated artifacts are well documented and easily readable, applying these improvements is a good way to build over the base functionality.

In our case, the code produced by EMF lacked proper methods for asserting the equality between two elements, managing collections or generating a unique identifier. But fortunately the generated code did not require a deep knowledge of EMF. With the help of annotations to preserve these non-generated methods in future transformations and thanks to the cleanliness and organization of the code, the application of these perfective changes was straightforward.

Cost of starting a new iteration. It is very common during the development process to go back to a previous step, introducing some changes and continue from there. This usually forces the redefinition of models and regeneration of code. In these cases it is very important to keep track of all the manual changes and procedures that have to be applied after finishing the automated tasks. For instance: performing the correct code modifications after its regeneration.

Therefore, is vital to have a detailed and documented process for the development with MDE. We addressed this point by adopting the detailed flow shown in previous sections.

Coverage of transformations. MDE is based upon transformations. However, special attention needs to be devoted to the system components where the necessary transformations are not automatically performed. These sections must be reviewed after each code regeneration operation and some parts need to be manually implemented.

During the development of the system we found that Spring Web Services, although generated the WSDL, lacked the tools to do the same with the bindings between the logic and the interfaces. In the end we implemented those bindings manually. However, in retrospective we think that defining and implementing these transformations could have been a better solution. The workload would have been similar but in further iterations the benefits of extending MDE coverage would have been considerable.

6 Conclusions

In this case study we have developed a real-world enterprise management system in a model-centric view through MDE processes. This approach has allowed us to implement some non-functional requirements such as remote communications or information persistence with model transformation techniques and tools, using available open source tools and libraries.

The results obtained during this development have been satisfactory. The reduced effort obtained by the code generation capabilities greatly helped to speed the process. The general perception of both the developers and project managers are that the use of these methodologies, albeit the problems faced, has eased the development process and improved the quality of the produced system. It seems clear that the characteristics of the enterprise domain make it perfectly-suited for automating the generation of parts of the system.

However, regarding the level of achieved abstraction from the middleware layers we identified several key factors that greatly impact the results in this area. We believe that our lessons learned in this case study can help with the execution of similar processes and greatly reduce the risks involved and shorten the development cycles.

Acknowledgments. The work presented here has been performed in the context of the CENIT-ITECBAN project, under a grant from the Ministerio de Industria, Comercio y Turismo de España.

References

1. Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. Computer Vol. 39, 2, 25–31 (2006).
2. Object Management Group, January 2006. Meta Object Facility Specification 2.0. http://www.omg.org/spec/MOF/2.0/
3. Frankel, D.S.: Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley (2003).
4. Quartel, D.; Pokraev, S.; Pessoa, R.M.; van Sinderen; M.: Model-Driven Development of a Mediation Service. In: 12th International IEEE Enterprise Distributed Object Computing Conference, Munchen (2008).
5. White, J.; Schmidt, D.C.; Czarnecki, K.; Wienands, C.; Lenz, G.: Automated Model-Based Configuration of Enterprise Java Applications. In 11th International IEEE Enterprise Distributed Object Computing Conference, Annapolis (2007).
6. Steinberg, D.; Budinsky, F.; Paternostro, M; Merks, E.: EMF: Eclipse Modeling Framework, Second Edition, Addison-Wesley Professional. (2008).
7. Spolsky, J.: Joel on Software. Apress. (2007)