

An Efficient Migration to Model-driven Development (MDD)

Jens Knodel¹, Michalis Anastasopolous, Thomas Forster,
Dirk Muthig,

*Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6, D-67661 Kaiserslautern, Germany*

Abstract

Model-driven development envisions raising the abstraction level of software development. To fully realize this vision, technology-specific aspects must be completely hidden from developers. They produce only platform-independent models (PIM), which are automatically transformed into executable systems. To enable an efficient migration to MDD, we recommend taking advantage of concepts from software architectures, product line engineering and reverse engineering.

Keywords: architecture-driven migration, model-driven development, product lines, reverse engineering, PuLSE

1 Introduction

When a new technology like MDD is introduced into industrial practice, it always must be adapted to and integrated with the current practices in the target organizations in order to minimize the degree of novelty introduced. That is, the transition has to consider the existing working environments, and MDD concepts have to be tailored to the current processes in order to have benefits like increased productivity and reduced complexity right from the start and to be successful, whereby success means that the people in the organization are satisfied with the new technology, adopt it and really use it in their daily practices. Hence, MDD requires in the beginning additional

¹ Email: {knodel, anastaso, forster, muthig}@iese.fraunhofer.de

effort to adjust the existing development environments and to make the organization fit towards the new technology. To keep the migration cost low, to not lose knowledge hidden in already existing applications, and to get the full benefits of MDD and appendant tool(s), the migration should be automated as much as possible. By exploiting technologies from areas such as software architectures, product line engineering, and reverse engineering the migration bypasses common technological change problems from the beginning. The concepts from these areas have in common the focus on patterns and how patterns are applied in model-driven development.

The remainder of the paper is structured as follows: Section 2 discusses the four areas we consider as enabling an efficient migration to MDD in more detail. Then section 3 presents a summary and gives an outlook on the full paper.

2 Enabling an Efficient Migration

Driven by concepts out of technologies like software product lines, software architectures, and reverse engineering, the migration to MDD can bypass common technological change problems from the beginning and minimize typical introduction problems like technology scepticism.

2.1 Software Architectures

Software architectures as defined in [7] describe relevant aspects of software systems including functional requirements, quality characteristics, or business goals of different stakeholders. Architectures are usually described in a view-based manner (see or [6], [8], or [2]) separating the concerns of the system. Well-understood software architectures are a foundation for successful software systems, since they enable a clear communication about all kinds of system aspects among diverse roles and stakeholders. Product line architectures are software architectures that provide the skeleton for all systems within a software family. Hence, product line architectures consider besides existing instances also anticipated future variations among these systems. Architectures are realized with sets of architectural patterns that systematize all implementation activities and ensure the fulfillment of quality requirements. For instance, the definition of persistence patterns ensures that data is made persistent in a consistent, standardized way throughout the whole application using a database management system. Different patterns for the realization of the persistence are possible, on the one hand optimizing the performance, and on the other hand emphasizing on data integrity with backup mechanisms. Decisions at this architectural level have deep impact on the implementation.

In order to fulfill the given requirements by the stakeholders, it is necessary to know about the consequences of architectural means and patterns already at this early phase in the development process.

2.2 *Product Line Engineering*

A software product line, as defined in [10] is a family of products that are designed to take advantage of their common aspects and that have predicted variabilities. Product line engineering, in general, is an approach for systematically developing software families and aims at significantly improving development cost, time-to-market, or software quality. PuLSE (Product Line Software Engineering, [1], [2]) is a well-known and concrete product line method developed by Fraunhofer IESE, which is applied in industry contexts since 1997 and that emphasises the definition and evolution of product line architectures including their implementation. Both, the definition and the evolution of architectures rely in most cases on reverse engineering activities to efficiently reuse all existing artifacts (ranging from user documentation over design documentation if available down to source code). When migrating from a manually implemented application to an MDD-based approach, product line concepts enable a systematic identification of commonalities and differences between the existing application and the equivalent application to be generated later by the new MDD-driven software development. The knowledge of commonalities and differences is the key to a systematic transformation of the existing application and thus to a planned, predictable and successful migration to an MDD-approach. Product line architectures allow to build different products sharing a common core, but with variations that address certain requirements. Hence, they provide the skeleton for all systems within a software family considering besides existing instances also anticipated, future variations.

2.3 *PuLSE-MDD*

Model driven development approaches have demonstrated that they may improve current practices of software development. Design and concepts of an application can be reused in the context of another platform through divorcing business logic from technical implementation details. Current MDD approaches support the abstraction from concrete implementation technologies through meta-models, which practically define domain-specific languages. Interest groups like the OMG are often responsible for creating these meta-models. This leads to more complete and standardized meta-models but often results in epistemic meta-models capturing concepts that are not of any interests to products of a particular organization. Therefore successful MDD

approaches are often architecture-centric and bound to a given implementation technology such as J2EE. In other words, some MDD approaches are driven by the target platform and not by the envisioned system architecture. This constitutes a gap between theory of model-driven development, which proposes a waterfall approach, and practice, which in this case proceeds in a bottom-up manner.

PuLSE-MDD (Model-Driven Development, see [11]) is a systematic approach that uses the ideas of the MDD approaches above but avoids the mentioned deficits through fully concentrating on optimizing code generation for systems an organization is going to build. Such a product-centric scope consequently does not require assumptions on potential patterns in complete technical domains but focuses on product line members only. PuLSE-MDD is part of the architecture design phase as depicted in Figure 1. If PuLSE-MDD has been started synchronously with the architecture definition (see left side of Figure 1), its initial input is the (empty or partially filled) set of architectural views selected and customized during a stakeholder analysis. Otherwise (i.e. in a reverse engineering-driven process), it starts with documenting the architecture(s) of the past. These documents (i.e. mainly architectural views) are analyzed with the focus on capturing how these architectures address reoccurring problems. Moreover, non-functional requirements are analyzed one by one to identify further pattern candidates. The architecture(s) implementation view is then a good starting point to explore how identified pattern candidates are realized at implementation level. To find economically optimal pattern sets, PuLSE-MDD incrementally derives and implements patterns following a process defined with respect to the product line architecture that typically follows a component-oriented style. At a certain point, an exact identification or correct implementation of patterns becomes too complex. Then, the process stops by providing an initial pattern set. The percentage of the implementation generated is measured: if it is low, pattern identification and implementation is restarted to achieve further improvements; otherwise, the initial pattern set is used to develop first components or products.

2.4 Reverse Engineering

Reverse engineering as defined in [4] analyzes the various artifacts of existing software systems and extracts information about them (with techniques such as pattern matching, component identification, etc.). Assuming that both, the manually implemented application and the MDD-implementation, are based on the same set of technologies (e.g., J2EE), then there are technical similarities but also alternative solutions of identical technical problems, such as GUI, transactions or persistence. The goal is to transform an existing applica-

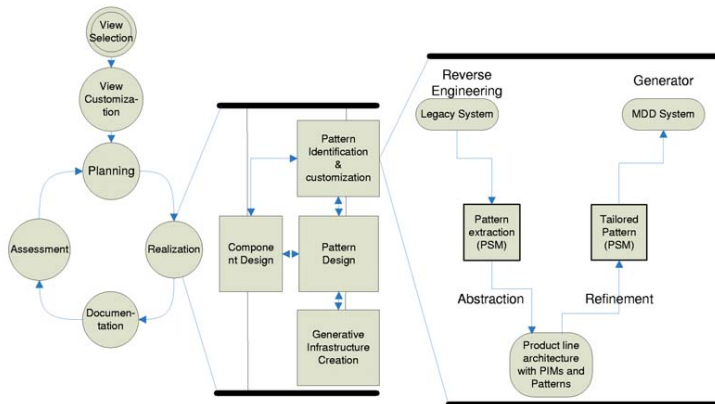


Fig. 1. PuLSE-MDD's Iterative Process

tion into an application that is based on the same (product line) architecture but underlying a MDD tool (e.g., Optimal/J [3]). To achieve this, reverse engineering must (see right side of Figure 1):

- Identify the parts in the existing source code that corresponds to the code that will be generated by a MDD tool when transforming the PIM into the platform-specific model (PSM) and the concrete implementation.
- Abstract the platform-independent model (PIM) hidden in the existing application (hidden because often there is no or only non-consistent or outdated documentation available).

The first bullet can be addressed by available architecture recovery approaches and reverse engineering technologies. The second bullet is addressed by pattern detection approaches. Here, the challenge is that manually implemented patterns usually are variants of general patterns adapted for the particular context or working environment. Pattern matching [9] reveals such pattern instances. While transforming an existing application into an MDD-based application, firstly, an abstraction of the instances (variants) to identify the clean, general pattern and, secondly, a formalization of the relationship between general and instantiated pattern are required. Then, either the instance of the pattern is replaced by (a series of) pattern(s) executed by a MDD tool solving the identical problems, or the detected patterns extends the existing pattern catalogue as an organization-specific customization.

The PIMs build then the foundation for the migration towards MDD, since conceptually the legacy systems and the system constructed with MDD are just two variant realizations of the same platform independent architecture. Reverse engineering thereby extracts the information required to build higher-level abstractions and representations of applications, which correspond to

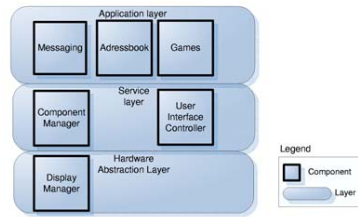


Fig. 2. GoPhone Layered Architecture

MDD's platform-independent models (PIM), as well as reverse engineering matches patterns in existing applications with patterns used by a MDD tool to transform the PIM in a platform-specific implementation. Hence, reverse engineering based on architecture and product-line competence supports the exploitation and migration of existing systems, and makes sure that knowledge embedded in the existing applications merges with the concepts of the new MDD approach.

3 GoPhone Case Study

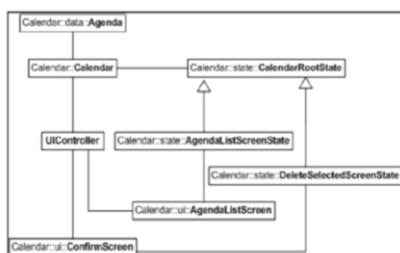
The case study presented in this section is based on a hypothetical mobile phone company, the GoPhone Inc, a publicly available [12] test bed developed especially for the purpose of validating and illustrating product line methods, techniques, or tools. In the case study, PuLSE-MDD was applied in the reengineering-driven mode. The architecture and the manual implementations of existing components have been analyzed to identify patterns, which in turn were applied to generate code frames for further components in the application layer. In several iterations an initial pattern set was identified and implemented for the chosen implementation technology (J2ME).

In the following subsections, a rough sketch of the GoPhone architecture is first provided, and then the concrete application of PuLSE-MDD is exemplified.

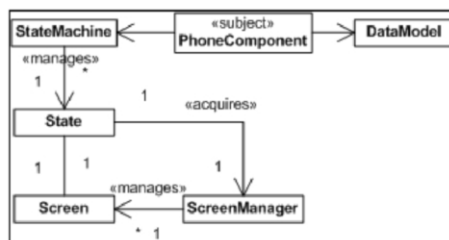
3.1 The GoPhone Architecture

The layers of the GoPhone architecture are sketched in Figure 2. On the bottom the hardware abstraction layer contains the DisplayManager component which handles visualization issues as well the according variations that result from the variety of mobile devices. The UserInterfaceController as part of the service layer wraps the DisplayManager and provides additional user interface services for components of the application layer. Also located in the service layer, the ComponentManager introduces the component orientation facet as another predominant architectural style into the product line architecture. The ComponentManager component handles mainly the communication and

life cycle of components in the application layer. This manager component is an integral part of a component framework which adds generality to the architecture as it enables the product lines' extension with further application layer components. Whereas the components in service and hardware abstraction layer can be seen as the underlying infrastructure the application layer components realize the business functionality of a GoPhone product (i.e., for instance, the mobile phone depicted in Figure 2 realizes messaging, address book, and calendar functionality). Each of these components is an instance of a generic PhoneComponent, which defines the common structure of application layer components.



(a) Implementation view



(b) Conceptual view

Fig. 3. Architectural views of an application layer component

3.2 The Calendar Component

We started with a comparative analysis of existing application layer components implementation models. Figure 3 shows on the left side the implementation view of a Calendar component. During this analysis step implementation models were generated for different components. In parallel, the comprehension for the different component parts duty had to be developed. The commonalities were identified and abstracted. Although this is maybe the most challenging task in general, it was comparatively easy for this case study as all components had a similar package structure and consistent naming conventions. As these components are designed to fit into the component framework they provide standardized interfaces to hook them up with the component manager. All components follow a similar pattern for structures that manage user interfaces and behavior user prompt to enable interaction with the service components. Data models and data storage is also a concern that cuts through all application layer components and that is managed in a similar way. The discovered commonalities can be generalized into the conceptual model shown on the right side of Figure 3.

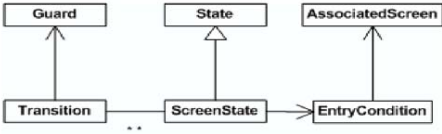
| | |
|------------------------------|--|
| Intent | Due to the small screen size of mobile devices the amount of information displayable and editable in the business components at a time is restricted. An intelligent menu prompt and screen management is necessary. |
| Pattern | Introduction of a state machine per component that handles each active screen and the according data in an associated state. |
| Tailoring at component level | Each state manages an associated screen. |
| Pattern Structure |  |
| Translation Rules | <ol style="list-style-type: none">1. Within the package of the PhoneComponent a package "state" is created.2. Each state specified becomes a class. A class that encapsulates all specified states is created. The name of this class is a concatenation of the PhoneComponent's class name and the string "Statemachine"3. Each state class specifies an action method where the guards at the transition are evaluated. If there are more outgoing transitions from one state if-statements are created.4. Each states' entry condition is evaluated and if necessary a new Screen for the user is displayed. ... |
| Automation | The state design pattern can be generated from an abstract description. |

Fig. 4. Pattern Description for the State Pattern

In the following the specific implementation models are merged into a generic component implementation model. Having now both -the conceptual and the generic implementation models- patterns are extracted from source code that describe, how conceptual architectural elements are mapped onto their implementation counter parts. The main difference to other MDD approaches here is that the specific GoPhone architecture is the key driver for realizing the translation patterns. In other approaches it is the other way round: a generic third party J2ME pattern set may influence or even prescribe a systems architecture. In the worst case even technical details constrain the envisioned architecture. This doesn't mean that such a pattern set couldn't be helpful if the patterns used there don't impose restrictions on the planned architecture.

The pattern extraction process leads to a rule set that describes how the conceptual parts of a component are mapped onto a J2ME based implementation. Through analyzing each conceptual component part in greater detail we develop a language which can be seen as a formalization and extension of the conceptual model shown in Figure 3. This language enables us to specify new components and to describe how the architectural concepts should be used. <<subject>> denotes that the given model as a whole specifies a component where the PhoneComponent entity has the role of the component interface. <<Manages>> denotes that there is some logic so the non navigable entity can control the target entity, for example the StateMachine that controls the component states and their transitions. <<Acquires>> means, that a State accesses a further system component, i.e. the ScreenManager component at runtime to request a user screen associated with that state. Figure 4 exemplarily shows the analysis results for the state machine concept in a model.

The first row describes why the pattern was used in the architecture. The row "pattern structure" shows part of a meta-model for the state machine concept. This will -with minor changes- be used as an (UML notation based) platform independent language to model the state machine concept for other components. The translation-rules section describes the mappings of a platform independent model to physical entities. Generally these rules are mainly a result of the reverse engineering activity. In our example it turned out, that the state machine was realized using the state pattern [13]. So in this case a well known design pattern was tailored to the needs of the GoPhone architecture. Through the reverse engineering activity and abstraction it is prepared for automated reuse within other components.

Figure 5 shows the application of the tailored state pattern using an UML notation to describe the behavior of the calendar component as input for automation. In this section we developed a platform independent language to specify application layer components of a product line architecture in a platform independent manner. Through reengineering and generalization we extracted rules that can be used as a generator input that automatically translate conceptual entities (i.e. a state machine) into different J2ME based implementations. The generator is therefore a means to support variability at architectural level. Generation techniques could also be used to realize variability at component level. One possibility would be to generate code that itself is generic or configurable; for example a state machine where variant state classes can be included or excluded through configuration management. Platform independence is supported as the models state concepts and therefore could be used as input for an .NET compact Framework generator.

4 Conclusion

When systematically migrating organizations to an MDD-approach, a successful and efficient transition benefits from selected software product lines, software architectures and reverse engineering into account. Software Architecture contribute by view-based documentation of architecture (including meta-models and instantiations of patterns), platform independent models for patterns and their collaboration. Product line engineering contributes by concepts on how to manage the commonalities and variations between manual and generated implementations (e.g., business logic, translation) and explicit variation points for existing and abstracted patterns to be generated, and application engineering to support different platforms (e.g., J2EE). Reverse Engineering brings in techniques for pattern identification, the abstraction of existing pattern instances and explicit documentation and identification of

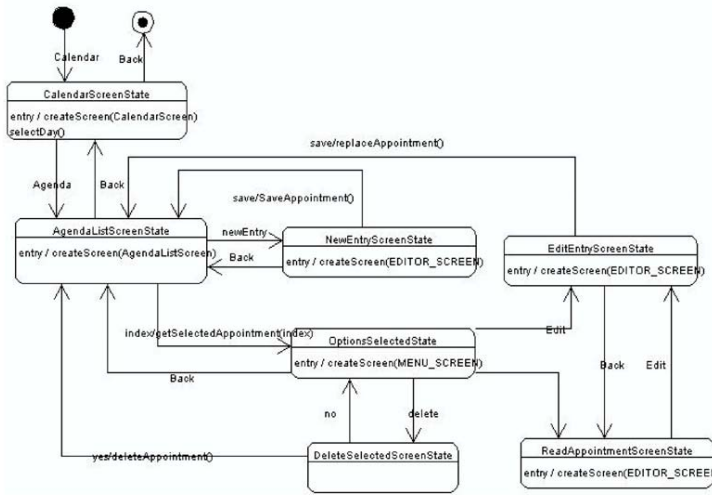


Fig. 5. State Machine of Calendar Component

existing translation rules. In short, reverse engineering driven by software architecture development and product-line competence supports the exploitation and migration of existing applications, and makes sure that technical concepts and knowledge embedded in the existing applications merges with the concepts of the new MDD approach.

Nevertheless, such a migration still requires guidance and support by experts on methodologies and guidance how to select the appropriate customizations, how to tailor MDD to the current practices of an organization, how to migrate to MDD by using concepts out of the three technologies (software architectures, product line engineering, reverse engineering) and thus, how to ensure a successful and efficient migration. Our experiences from product line engineering show that appropriate customizations and tailoring to organizational environments and current practices is a prerequisite for the successful introduction of a new development paradigm. Future work will address the semi-automation to reduce the expert effort and the tool support by integration with a particular MDD-tool. Next to software architectures, product line engineering and reverse engineering, there are other software engineering areas, which have to be investigated in future with respect to their impact on an efficient migration to model-driven development. However, we think that technologies and concepts as presented in this work help to facilitate the migration and to lower technology skeptics that may occur when introducing MDD in a software development organization. The guidance gained from well understood software engineering areas ease the overhead of the migration.

References

- [1] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., and DeBaud, J.-M.: *PuLSE: A Methodology to Develop Software Product Lines*, in Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), ACM, Los Angeles, CA, USA, p. 122-131, May 1999.
- [2] Bayer, J., Forster, T., Ganesan, D., Girard, J.-F., John, I., Knodel, J., Kolb, R., Muthig, D., *Definition of Reference Architectures based on Existing Systems*, Fraunhofer IESE, March 2004..
- [3] OptimalJ: <http://www.compuware.com/products/optimalj/>.
- [4] Chikofsky, E.J., J.H. Cross, *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, January 1990, pp. 13-17.
- [5] GoPhone Case Study (in German), <http://www.software-kompetenz.de/?21629> .
- [6] Hofmeister, C., Nord, R., and Soni, D., *Applied Software Architecture*, Addison-Wesley, 2000.
- [7] IEEE Std 1471-2000, *IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems*, IEEE Computer Society, IEEE Computer, 2000.
- [8] Kruchten, P., *The 4+1 View Model of Architecture*, IEEE Software, 12(6):42-50, November 1995.
- [9] Sartipi, K., Kontogiannis, K., and Mavaddat, F., *A Pattern Matching Framework for Software Architecture Recovery and Restructuring*, in iwpc, ieeepress, p. 37-47, June 2000.
- [10] Weiss, D. M., and Lai, C.T.R., *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley, 1999.
- [11] To appear: M. Anastasopoulos, T. Forster, D. Muthig: Optimizing Model-driven Development by Deriving Code Generation Patterns from Product Line Architectures, submitted to ICSE 2005, St. Louis, USA
- [12] GoPhone Case Study (in German), <http://www.software-kompetenz.de/?21629>
- [13] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995