Mixed Generative and Handcoded Development of Adaptable data-centric Business Applications

Pedram Mir Seyed Nazari Alexander Roth Bernhard Rumpe

Software Engineering
RWTH Aachen University
{nazari,roth,rumpe}@se-rwth.de

Abstract

Consistent management of structured information is the goal of data-centric business applications. Model-driven development helps to automatically generate such applications. However, current approaches target full or one shot generation of business applications and often neglect simplicity and adaptability of the code generator and the generated code. Moreover, it is necessary to inspect the generated code in order to add functionality. Thus, here we discuss mechanisms for a code generator to generate a lightweight and highly customizable data-centric business application that is targeted for a variety of users including generated application users, tool developers, and product developers. We achieve simplicity by reducing the mapping of the input model to the generated code to a minimal core of easily understandable concepts. As a consequence, the generated code does not need to be read or understood, since the input model clearly describes what is generated. High customizability is achieved by providing a variety of mechanisms to extend the generator and the generated code. These include template overriding and hook points to extend the code generator. Moreover, to extend the generated code we use hot spots and additional manual extension approach. It is even possible to fully control the code generator and the entire generation process via a scripting language.

Keywords Data-centric Business Application, Generative Development

1. Introduction

Data-centric business applications provide management functionality for structured and consistent information. They offer CRUD (create, read, update, and delete), search, and persistence functionality [16, 17]. Existing model-driven development approaches allow nearly full code generation [14]. Such generators can be powerful tools when used by experienced users. However, developers not familiar with such approaches hardly accept them, because of the their complexity and the loss of control [12, 15]. Consequently, adapting and customizing the code generator or the generated output becomes a labor-intense and time-consuming task.

Even if nearly full code generation is achieved, simplicity (the amount of languages needed to describe the business application and the amount of approaches to integrate handcoded extensions), ease-of-use, and adaptability is not much addressed by current research [2, 3, 9, 18]. Previous work has proposed an infrastructure for generating enterprise applications [11, 13]. This infrastructure consists of multiple code generators and languages, which describe the enterprise applications in a generative way. Nevertheless, the provided code generators and infrastructures employ a variety of modeling languages and may require to develop entire code generators when changes to the generated software system are required.

In this paper, we present a generator that aims at demonstrating the power of the generative software development methodology using the generator framework MontiCore [10]. Our main contribution is a demonstration of easy-to-use generation of almost ready-to-use business applications from abstract models as shown in Fig. 1. This approach is different to existing work as it only requires one input language to describe the data to be managed, provides clear customization approaches for the code generator and the generated systems, and presents a code generator that is designed to automatically integrate handwritten and generated code. In particular, we use a variant of UML class diagrams and produce running Java applications. The generated applications provide a graphical user interface to manage instances of the modeled system. Furthermore, they allow to persist instances in the cloud and share them among users, which may have different roles and rights.

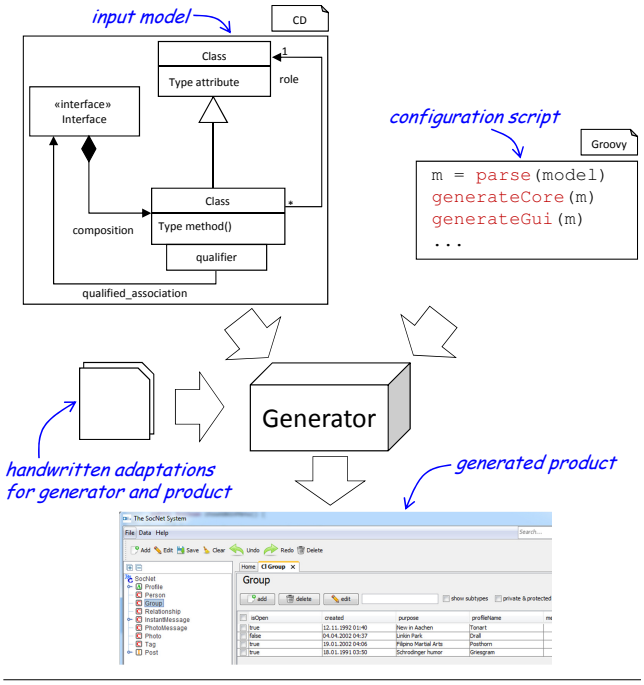

Figure 1. Overview of generation process.

The proposed generator provides an approach to design highly customizable and adaptable code generators by offering a variety of extension mechanisms to even allows to fully control the entire generation process.

The rest of this paper is structured as follows. We first give a brief description of the input language and the generated output in

Sect. 2. Then, we describe how we achieve high customizability by using hook points, hot spots, template overridings, and a control script in Sect. 3. Finally, we conclude our paper in Sect. 4.

2. Generated Applications from UML Class Diagrams

The input language of our generator is a variant of UML class diagrams that allows to focus on data modeling (without addressing methods). From this input the generator produces (parts of) business applications. The generated product is an executable application for managing data that conforms to the UML class diagram description. It offers CRUD (create, read, update, and delete) management functionality to manage objects and associations. Additional support is provided by the graphical user interface that allows browsing, searching, and filtering. On invalid input the generated interface offers instant feedback. In addition, database and multi-user support with role-based access control is generated to allow specification of users, roles, and CRUD operations. Both functionality is provided without employing additional modeling languages.

To simplify the usage of the code generator, we identified three roles with different requirements. First, *end users* of the generated product are unaware of the technical details but simply want to use the generated application. Their main focus is a user interface that is systematically structured and easy to use. In contrast, *product developers* need to handle and manage the generated code to provide extensions and customizations. However, they are usually not interested in the implementation details but mainly in the interfaces and provided APIs. Finally, *tool developers* are highly interested in the implementation and how the overall architecture of the generated application is to be able to adapt and extend functionality of the generated code. As a consequence, we provide a highly customizable code generator and additional concepts to adapt the generated code without the need for a detailed inspection.

2.1 Input Models

The input language is a reduced variant of UML class diagrams and provided in textual form designed using current understanding of semantics and domain-specific design guidelines [7, 8]. Certainly, it does not provide much application-specific functionality. Therefore, various extension and adaption mechanisms are introduced to extend the functionality of generated products. Nevertheless, the input language is sufficient to describe the managed data and generate a working application.

```
1 classdiagram SocialNetwork {
2   abstract class Profile {...}
3
4   class Person extends Profile {...}
5
6   association Person -> (friend) Profile [1] ;
7 }
```

Listing 1. Input model example

The input language focuses on the most important concepts of UML class diagrams especially suited for documenting analysis results. An example is given in Lst. 1. It contains classes, interfaces, and abstract classes. Classes may extend other classes and implement interfaces. We use associations with navigation directions and cardinalities. An association as well as each of its role ends may have names. Associations can be ordered or qualified. Ordered associations are marked using the ordered stereotype and qualified associations require a qualifier. Classes have attributes with associated types. Compositions are omitted in the model, but they are supported as a special form of associations.

2.2 Generated Applications

The generated application is a typical 3-layered architecture composed of the graphical user interface, the application core, and the persistence management to structure its products. The application core realizes only business functionality. As illustrated in Fig. 2, the layers are independent and can easily be exchanged by different implementations. Each leayer has its own runtime environment and standard components for accessing predefined not generated functionality.

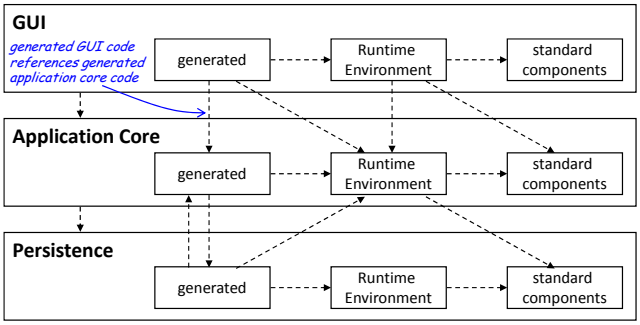

Figure 2. Overview of the generated appliction's architecture.

Since not every functionality can and needs to be generated from the input model, already existing code is reused. There are many sources for this kind of reuse. In order to make use of this kind of reuse, the generated architecture relies on a run-time, which is deployed with the generated application and provides access to external libraries. An example for the need of external libraries is role-based access control. The generated applications allow for create users, roles, and to define the CRUD operations of each role. Since generating role-based access control from abstract models is hard, we generate code that is compatible with Apache Shiro [1]. Hence, no need for introducing a new modeling language is given and developers can rely on existing infrastructure. A major benefit of employing Shiro is the possibility to be very fine grained and for example to define rights on attribute or association level.

3. An Intelligent and Customizable Generator

A code generator becomes helpful, when it effectively assists developers to speed up their work. This is only possible, when the generator actually takes some burden from the developer. For example, by making certain decisions and generating corresponding functionality. Our generator for example targets desktop applications with a layered architecture. Based on that choice, it embodies a variety of additional functionality that can be generated automatically.

Besides taking some burden from the developer, it is an intrinsic property of a good generator to be able to adapt either the generation process or the generated code. In particular, for algorithms that usually cannot be described in a more abstract form than the implementation of the algorithm itself, manual implementation is necessary. Due to this, we provide a variety of extension mechanisms to allow for high customizability of the code generator and the generated code.

For the code generator, we provide explicit hook points, which are dedicated spots in templates that are intended to be customized and extended. Additionally, a more detailed level of customization is provided by allowing to replace every templates of the code generator with a custom template. Finally, in order to give developers full control of the generation process, which includes parsing models, checking context conditions, and generating code, we employ Groovy [6] as a scripting language to control the generator. Hence,

the generator becomes an active library [4], where only parts of the code generator can be executed and the generated code can be adapted.

For the generated applications, we offer hot spots as a dedicated spot in the generated code, which is usually known from frameworks as provided methods that have to be overridden, and concepts to extend the generated classes [5]. We strictly separate handcoded artifacts from generated artifacts to allow complete regeneration without loss of the customizations and adaptations. This provides from overriding handcoded extensions by generated code and requires developers to only version the input model and the handcoded artifacts. The code generator detects handwritten extensions and adapts the generated code accordingly to regard it. As Fig. 3 shows this kind of extensions are supported on each layer of the generated architecture.

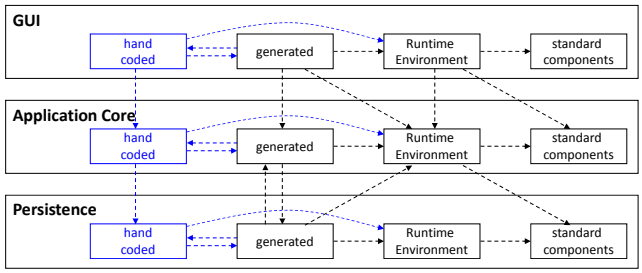

Figure 3. Overview of the generated appliction's architecture with handwritten extensions.

4. Conclusion

Generating data-centric business applications is a complex task and currently requires deep knowledge of multiple modeling languages and the underlying code generators. To tackle the low acceptance of model-driven development approaches to generate business applications, a simplified approach to generated business applications, which requires only one UML class diagram and provides clear customization concepts, is helpful.

In this demonstration, we present a code generator that uses a variant of UML class diagrams as input to generate lightweight but feature rich business applications. The code generator's main focus is on simplicity and adaptability. This is achieved by reducing the input language to one simplified language, adapting the code generator to take some decisions from the developer, and simplifying the mapping of input language concepts to output language concepts. However, since no application specific logic as well as behavior can be expressed, we provide a variety of adaptation mechanisms to adapt the code generator and the generated applications. It is even possible to customize the complete code generation process.

References

[1] Apache Shiro website `http://shiro.apache.org/`.

[2] W. L. d. S. Carlos Eduardo Cirilo, Antonio Francisco do Prado and L. A. M. Zaina. *Interactive Multimedia*, chapter Building Adaptive Rich Interfaces for Interactive Ubiquitous Applications. 2012. ISBN 978-953-51-0224-3.

[3] A. Cicchetti, D. Di Ruscio, and A. Di Salle. Software customization in model driven development of web applications. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 1025–1030, New York, NY, USA, 2007. ACM. ISBN 1-59593-480-4.

[4] K. Czarnecki, U. Eisenecker, R. Glck, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-41090-4.

[5] T. Greifenberg, K. Hölldobler, C. Kolassa, M. Look, P. Mir Seyed Nazari, K. Müller, A. Navarro Perez, D. Plotnikov, D. Reiss, A. Roth, B. Rumpe, M. Schindler, and A. Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In S. Hammoudi, L. F. Pires, P. Desfray, and J. F. Filipe, editors, *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pages 74–85, Angers, Loire Valley, France, February 2015. INSTICC and ESEO, SciTePress.

[6] Groovy Programming Language website `http://www.groovy-lang.org/`.

[7] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[8] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*.

[9] S. Kelly and J. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008. ISBN 978-0-470-03666-2.

[10] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proceedings of Tools Europe*, 2008.

[11] V. Kulkarni and S. Reddy. Model-driven development of enterprise applications. In N. Jardim Nunes, B. Selic, A. Rodrigues da Silva, and A. Toval Alvarez, editors, *UML Modeling Languages and Applications*, volume 3297 of *Lecture Notes in Computer Science*, pages 118–128. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25081-4.

[12] V. Kulkarni and S. Reddy. A model-driven approach for developing business applications: Experience, lessons learnt and a way forward. In *Proceedings of the 1st India Software Engineering Conference*, ISEC '08, pages 21–28, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-917-3.

[13] V. Kulkarni and S. Reddy. A model-driven approach for developing business applications: Experience, lessons learnt and a way forward. In *Proceedings of the 1st India Software Engineering Conference*, ISEC '08, pages 21–28, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-917-3.

[14] V. Kulkarni, R. Venkatesh, and S. Reddy. Generating Enterprise Applications from Models. In J.-M. Bruel and Z. Bellahsene, editors, *Advances in Object-Oriented Information Systems*, volume 2426 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-44088-8.

[15] V. Kulkarni, S. Reddy, and A. Rajbhoj. Scaling up model driven engineering experience and lessons learnt. In D. Petriu, N. Rouquette, and y. Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*, pages 331–345. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-16128-5.

[16] R. Mohan and V. Kulkarni. Model driven development of graphical user interfaces for enterprise business applications experience, lessons learnt and a way forward. In A. Schrr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 307–321. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-04424-3.

[17] A. Schramm, A. Preuner, M. Heinrich, and L. Vogel. Rapid ui development for enterprise applications: Combining manual and model-driven techniques. In D. Petriu, N. Rouquette, and y. Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 271–285. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-16144-5.

[18] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.