Maintaining Vaadin Legacy Applications using DSLs based on Xtext

Marcel Toussaint and Thomas Baar

Hochschule für Technik und Wirtschaft (HTW) Berlin
(University of Applied Sciences)
Treskowallee 8, 10318 Berlin, Germany
Emails: m.toussaint@web.de, thomas.baar@htw-berlin.de

***Abstract*—Vaadin, as a framework for the development of web applications, enables programmers to develop web applications purely in Java. The Vaadin framework has a clean architecture and enjoys a vibrant community. The popularity of Vaadin is certainly also due to numerous tutorials and small examples that illustrate certain aspects of the framework. Vaadin applications, once they became more complex than the appealing tutorials, might run - as well as many other software projects - into maintenance problems. In this paper, we report on a database application, whose programmers followed the suggestions from the tutorials rather strictly. Over time, it became harder and harder to accommodate changes of the database structure since the Java code made certain assumptions on the structure of database tables at many different locations. The classical approach to handle such a situation would be to refactor the entire Java code, which can be very costly. An alternative approach is to use a domain-specific language (DSL) to (a) capture those parts of the application that might vary in future in form of a language, (b) to create a model using this language that reflects the current application, and (c) to change the model due to new requirements and to regenerate those parts of the application that need to be adapted. The last step (c) automates the work of a human software maintainer who would adapt the application code manually due to new requirements (e.g., a new database structure). In this paper, we report on our initial experience when implementing the DSL-based approach using the framework Xtext.**

Keywords–Software design; Metamodelling; Data models; Software maintenance; Graphical user interfaces; Database systems.

I. INTRODUCTION

Web applications became increasingly popular over the last decade due to the multitude of different web browsers, and the convenience for the user of utilizing a web browser in a working environment. One of the main reasons for their popularity is the ability to operate and to update web applications without the need for distributing and installing the software on every single client target platform.

A. *The Open Source Web Application Framework* Vaadin

Vaadin [1] is an open source web application framework for Rich Internet Applications (RIA). Vaadin provides a server side architecture in contrast to JavaScript libraries and browser plugin based solutions. This means, that the majority or even the entire internal program logic is executed on the server. On the client side Vaadin supports Ajax and is based on the framework Google Web Toolkit (GWT) [2]. One of the major advantages of Vaadin is the possibility for the software developer to write the code completely in Java. The framework includes event driven programming and offers Java classes for UI elements such as buttons and lists. In practice, this means writing Vaadin applications is more similar to the development of desktop applications than the traditional Web development with HTML and JavaScript.

Vaadin uses a container-based concept to store and process data objects from external sources (e.g., tables from a database or input files). In this context, a container is a simple entity containing a defined set of items. Each item again possesses a defined set of properties together with their current values.

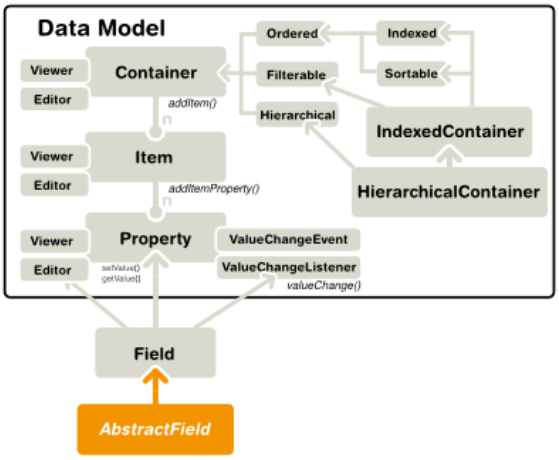

Figure 1. Vaadin's Container Concept [1]

Figure 1 shows Vaadin's architecture to bind UI elements (e.g., subclasses of `AbstractField`) to a `Property` which represents the corresponding attribute value of a data object (e.g., an entry in a database table).

This architecture is very generic and widely applicable, but has the shortcoming that `Property` objects do not have a specific type. The information, which tables exist in the database and what columns they have has to be stored in the application code, together with boilerplate-code to access the attribute values of database entries in a type-safe way. Once the database table structure changes, specific parts of the application code have to be changed as well.

B. *Domain-Specific Languages &* Xtext

Domain-specific languages are specialized computer languages matching a particular problem domain. They permit software design solutions to be expressed using the same

 ISBN: 978-1-61208-367-4

terminology and level of abstraction as the specific problem domain. Over the last few years there has been increased interest in domain-specific languages due to their potential of improving the productivity, quality and especially efficiency of software engineering.

The open-source Xtext framework [3] provides a solid toolkit for developing textual domain-specific languages. To specify a language, a grammar written in Xtext's grammar language has to be created. Once this is done, the user can create models in the specified language. Xtext supports this concept even by generating grammar specific editors. In many cases, the user wishes to generate other artifacts (e.g., source code) from these models. Xtext provides special support for implementing generators to produce such artifacts.

One of the key features of the Xtext framework is the possibility of seamlessly integrating it into the Eclipse IDE in form of a plugin. This plugin provides syntax highlighting for the DSLs created combined with code-folding and -checking. This toolset becomes handy in our scenario, because the Vaadin framework is supported by a corresponding plugin in the Eclipse IDE as well.

C. Tackling Software Maintenance with DSLs

Vaadin's architecture for bridging the gap between UI- and persistence-layer with the very generic classes `Container`, `Item` and `Property` causes the problem, that the programmer has to keep certain information on the database structure elsewhere in the Java code. Ideally this this information is kept in a single place. In practice, however, this is often not the case and the Java code suffers from the smell of *Solution Sprawl* [4].

There are refactoring techniques to eliminate the smell from the code, but refactoring is generally costly [4][5]. Furthermore, it is often not evident that the refactored versions are really better with respect to *performance*, *readability*, *maintenance* of the code. Note that many different quality criteria can be applied to assess the code and some of them might be complementary, i.e., the implementation code cannot match all these criteria perfectly.

Once we accept that some legacy systems cannot be made 'perfect' by refactoring due to the lack of financial and time resources or the lack of skilled programmers we can look for alternatives to deal with the current situation. One observation is, that maintenance requests, e.g., to change the data structure of the underlying database, requires a multitude of adaptations in the application code and thus is considered to be costly. However, these adaptations, that are traditionally done by a programmer, could be automated, if we succeed in

(a) capturing the possible impact maintenance requests might have on the application code (e.g., changing the database structure also implies changing the UI) and in

(b) generating the necessary code changes automatically.

Note that (a) is done by a suitable domain-specific language and that (b) substitutes the work of a human programmer and thus makes a maintenance operation for the legacy code much less costly.

The rest of the paper is organized as follows: In Section II, we dig into details of the legacy application we started from. Throughout the paper, this application will serve as a running example. In Section III, we present our approach on maintaining legacy applications using Xtext DSLs. In Section V, we outline future steps and summarize lessons learned so far.

II. A Motivating Example

At our university, a Web Application is used for management, revision and versioning of course programs. The application has been developed with the Vaadin framework and the developers strictly followed the recommendations presented in several Vaadin showcase projects.

Figure 2 shows a simplified excerpt from the architecture of this application. The database covers a multitude of entities, which are represented by the application via specific domain classes (e.g., the database table `Users` is represented by the domain class `TableUser`). The corresponding application logic and functionality is encoded in associated service classes. Note that in the given example, only a subset of table columns is represented by attributes of the domain class (i.e., the column `password` is not represented). This is due to the fact that some technical columns are not relevant in the context of the applicable service object. The modalities of how and especially the decision, which data should be presented in the user interface, is encoded in a view class `UserView`.

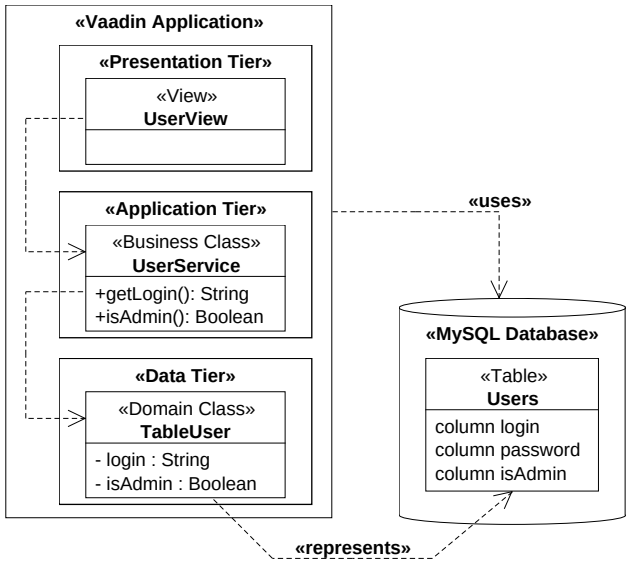

Figure 2. Simplified Excerpt from the Architecture

This implementation leaves us with one major disadvantage concerning the maintainability: Java classes from each tier are rendered deprecated artifacts as soon as the database definition is changed or extended. This produces a large overhead during the development process due to the fact that significant parts of the Java code have to be kept in sync with the database structure.

To illustrate this, consider the following example: if the `Users` table in the database is to be extended by an additional attribute, the Java representation by the domain class `TableUser` has to be updated accordingly to reflect such change. To ensure that the new attribute can be used properly, new functions to manipulate and to retrieve its value(s) must be implemented in `UserService`. Furthermore, the behavior of the corresponding UI class `UserView` concerning the new attribute (i.e., whether the attribute should be displayed or be editable in case of a list object) has to be defined manually.

 ISBN: 978-1-61208-367-4

To counter this problem, the usage of object-relational mapping could provide a proper solution. O/R mapping allows us to automatically convert data from the database content to the Java objects and vice versa. By using object-relational mapping, the concrete domain classes can be adapted automatically, once the database structure has been changed. However, since the UI classes depend on the domain classes, we still would have to manually adapt them to conform to the altered domain classes (and thus to the altered database structure).

III. Using DSLs to Maintain Vaadin Applications

We started to explore an alternative approach to solve the maintenance problems described above by using a DSL-based infrastructure. The idea is to generate the domain classes, which represent certain database tables, as well as the corresponding UI classes by using specific code generators. These generators have to regenerate all parts of the application, that need to be adapted due to a maintenance request.

A. Combining multiple DSLs

As maintenance requests often concern several different parts of the application (data, UI), we find it necessary to spread the modeling layer across multiple, interrelated DSLs. Figure 3 illustrates the concept of the 'integration model'.

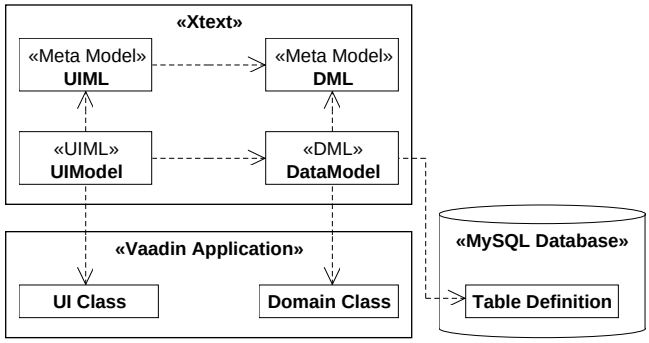

Figure 3. Overview of the Dependencies between Involved Artifacts

With the help of Xtext's grammar language, we created the abstract syntax (i.e., the meta model) for both a UI model language (`UIML`) and a data model language (`DML`) for the existing application. While the data model describes how the data objects are composed and how they are stored, the UI model describes the way these objects are presented to the user. The (`DML`) provides a modeling language for the data objects used in the MySQL context and their representation in the Vaadin application. The concrete code fragments for both the MySQL table definitions and the Java domain classes can be automatically created by using a code generator. Analogously, the Java classes for the UI are regenerated by code generators based on the `UIML`.

B. Designing a DSL for Maintaining Implementation Code

When switching from the traditional approach of maintaining implementation code to our DSL-based approach, one of the biggest challenges is to design an appropriate DSL. This DSL must take into consideration both all information a maintaining request consists of and the current structure of the implementation.

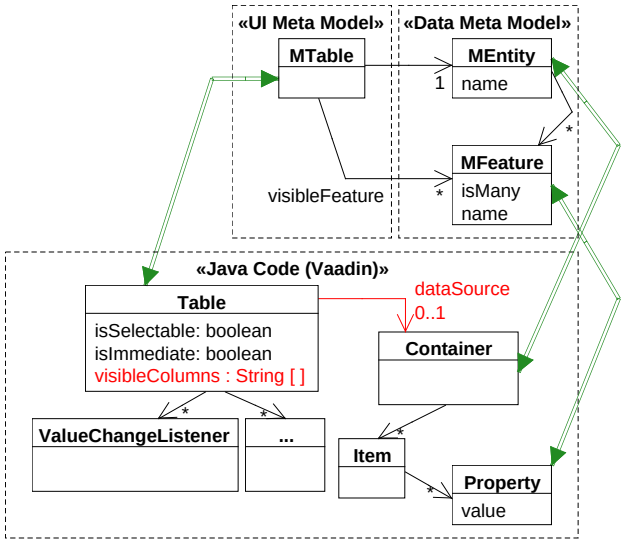

Figure 4. Relationship between DSL and Implementation Code (Excerpt)

The upper part of Figure 4 depictures some content of the meta models for our DSLs `UIML`and `DML`. The lower part presents relevant parts of the application code in form of a UML class diagram. Our meta classes in the meta models always start with a capital 'M'. An instance of `MEntity` consists of many features (`MFeature`) and represents a database table with its columns. The `UIML` consists of those concepts that allow to specify how information is presented in the application UI. For example, the concept `MTable`, which refers to a concrete instance of `MEntity`, represents how entities are displayed in the UI by the Vaadin class `Table`.

The lower part of Figure 4 shows the implementation classes used for displaying database contents. Whenever the content of a database table has to be displayed, an instance of the Vaadin class `Table` is created and configured. The configuration is done in terms of setting attributes such as `isSelectable`, `isImmediate`, `visibleColumns` or in terms of adding configuration objects, e.g., of type `ValueChangeListener` or others. One observation is, that the configurations of `Table` objects remain largely the same, no matter what database entity is to be presented. In our example, only the values for `visibleColumns` and the `dataSource` (marked in red in Figure 4) differ among the instances of `Table`. The `visibleColumns` is a list of strings, containing all columnnames to be shown by the table. The `dataSource` is a `Container` which holds a set of instances of `Item`, which in turn consists of instances of `Property`.

The connection between the upper and lower part of the figure is marked by green arrows. They represent a concrete mapping from model concepts to implementation classes. This mapping encompasses the process of code generation. For example, whenever our code generator processes an `MTable` instance, it produces Java code by which a `Table` instance is created and configured appropriately.

IV. Related Work

The idea of substituting parts of an application written in a conventional programming language such as C, C++ or Java

 ISBN: 978-1-61208-367-4

by one or multiple DSLs in order to increase productivity and to reduce maintenance costs is not new [6]. The goal of *Language-oriented programming (LOP)* [7][8] is to decompose software systems into orthogonal parts that can be described sufficiently detailed by using a DSL. Since the syntax of a DSL is optimized towards a certain purpose, the hope is that maintaining a DSL model is less demanding than maintaining code written in a programming language.

Experience reports on applying LOP for the sake of reducing maintenance costs have been published rarely. Klint et al. report in [9] on a benchmark for the maintenance of different DSL implementations. Some implementation (called vanilla implementations) have been realized using conventional programming languages (Java, JavaScript, C#) while others utilized various DSL tools (ANTLR, OMeta, Microsoft Modeling Platform). The results indicate that the usage of DSL tools is (slightly) advantageous.

Fehrenbach et al. describe in [10] their system *SugarJ* [11] and how a user can embed an external DSL in existing programming code. As an example, they present how the Java Pet Store application can be partly rewritten using four DSLs, what makes the code more readable, type-safe and maintainable. The difference to our work is that we use Xtext instead of SugarJ and that we do not design our DSLs for forward engineering the application from scratch. Instead, our DSLs take the source code from existing legacy applications heavily into account.

V. Conclusion and Future Works

This paper addressed maintaining problems of legacy applications. As an example, we have chosen a Vaadin application, that displays database contents mainly in form of tables. Whenever the structure of the database has changed, the programmer has to adapt the Java classes implementing the application's UI accordingly. Doing this process manually is tedious and error-prone.

We report on the experiences we made, when rewriting the existing web application using DSLs. We focused on those parts of the application, that need to be adapted whenever the structure of the underlying database has changed. We make some suggestions on how a DSL can be designed, such that this DSL exactly covers those parts of the application, that might be affected when adapting the application to new requirements.

In our opinion, the DSL has to be aligned to the existing legacy code. To achieve this, we had to inspect the code and to create corresponding UML class diagrams manually. In future, this might be done automatically by appropriately tailored reengineering techniques.

The main advantage of the proposed solution is that it overcomes potential deficiencies of existing code by replicating and regenerating the code in form of a comprehensible model rather than having to refactor the internal structure of an application. We have chosen the Xtext framework to define DSLs and code generators. This decision has been made due to Xtext's excellent Eclipse integration including syntax highlighting, code completion and static analysis.

References

[1] Vaadin, "Vaadin homepage," retrieved: October 2014. [Online]. Available: http://www.vaadin.com

[2] Google, "Gwt project homepage," retrieved: October 2014. [Online]. Available: http://www.gwtproject.org

[3] Itemis, "Xtext homepage," retrieved: October 2014. [Online]. Available: http://www.eclipse.org/Xtext

[4] J. Kerievsky, Refactoring to Patterns. Addison-Wesley, 2004.

[5] H. M. Sneed, "Planning the reengineering of legacy systems," IEEE Software, vol. 12, no. 1, 1995, pp. 24–34.

[6] P. Hudak, "Modular domain specific languages and tools," in Proceedings of International Conference on Software Reuse (ICSR). IEEE, 1998, pp. 134–142.

[7] M. P. Ward, "Language oriented programming," Software - Concepts and Tools, vol. 15, 1994, pp. 147–161.

[8] M. Fowler, "Language workbenches: The killer-app for domain specific languages?" 2005, retrieved: July 2014. [Online]. Available: http://www.martinfowler.com/articles/languageWorkbench.html

[9] P. Klint, T. van der Storm, and J. Vinju, "On the impact of dsl tools on the maintainability of language implementations," in Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications, ser. LDTA '10. New York, NY, USA: ACM, 2010, pp. 10:1–10:9. [Online]. Available: http://doi.acm.org/10.1145/1868281.1868291

[10] S. Fehrenbach, S. Erdweg, and K. Ostermann, "Software evolution to domain-specific languages," in Proceedings of Software Language Engineering (SLE), ser. LNCS, M. Erwig, R. F. Paige, and E. V. Wyk, Eds., vol. 8225. Springer, 2013, pp. 96–116.

[11] S. Erdweg, "Sugarj homepage," retrieved: July 2014. [Online]. Available: http://www.sugarj.org

 ISBN: 978-1-61208-367-4