A Template-Based Method to Create Efficient and Customizable Object-Relational Transformation Components

Igor Lihatsky, Anatoliy Doroshenko, and Kostiantyn Zhereb

Institute of Software Systems of National Academy of Sciences of Ukraine,
Glushkov prosp. 40, 03187 Kyiv, Ukraine
igor_md@ukr.net, {doroshenkoanatoliy2,zhereb}@gmail.com

Abstract. We describe a method to create object-relational transformation components by using the code generation system to automatically generate a persistence layer based on the database structure. A text template engine is used to generate SQL queries, business classes and APIs to access data from the application code. Provided default implementations are sufficient to quickly obtain a working persistence layer. However the real power of proposed solution lies in extensive customization capabilities. Therefore the developed system provides a high developer productivity because of automation, as well as high performance and flexibility due to the possibility of customization. Performance measurements demonstrate the high efficiency of the generated code, both in terms of execution speed and code size.

Keywords: object-relational paradigm mismatch, persistence layer, code generation, text template engine, object-relational mapping.

1 Introduction

Currently most business applications are created using object-oriented languages such as Java, C#, C++, and store persistent data in relational databases such as Oracle, Microsoft SQL Server or MySQL. Both object-oriented and relational paradigms provide a well-established, reliable foundation for solving applied problems. They are supported by numerous tools and methodologies that simplify the development process and improve the product quality.

However, the underlying data models for object-oriented and relational systems differ significantly. Therefore applied developers face the need to implement a conversion between object-oriented and relational representation of the same data. Sometimes such conversion is implemented manually, in which case obtained code is more efficient, but developers need to perform quite significant routine work of manually transforming each database object. In other cases automated tools (such as ORM systems [9]) are used, improving the developer productivity, but often sacrificing runtime performance and flexibility. There is a need of solution that can combine the flexibility and performance of manual implementations with automation achieved by using ORM systems.

H.C. Mayr et al. (Eds.): UNISCON 2012, LNBIP 137, pp. 178–184, 2013.

In this paper we describe our approach to object-relational transformation by using the code generation system to automatically generate a persistence layer based on the database structure. A text template engine is used to generate SQL queries, business classes and APIs to access data from the application code. Provided default implementations are sufficient to quickly obtain a working persistence layer. However the real power of proposed solution lies in extensive customization capabilities. Therefore the developed system provides a high developer productivity because of automation, as well as high performance and flexibility due to the possibility of customization. Performance measurements demonstrate the high efficiency of the generated code, both in terms of execution speed and code size.

The contribution of this paper is not some absolutely new technique or method, but rather a new combination of existing methods, such as using code generation and text templates, in order to improve the runtime efficiency of object-relational transformation, as well as increase the developer's productivity.

The rest of the paper is organized as follows: first we briefly analyze the current state of object-relational mismatch problem and describe existing solutions. Then we discuss our approach in detail, describe our implementation of this approach, illustrate capabilities of the developed system on simple examples and evaluate the performance of our system. Finally we provide the conclusions and directions of future research.

2 Object-Relational Mismatch: Problems and Approaches

Currently most business applications are created using two mainstream programming paradigms: object-oriented programming (OOP) [3] and relational database management systems (RDBMS) [2]. However these two paradigms use different and incompatible data representations: an object graph in OOP and tables in RDBMS, and the data access approaches also differ significantly. Because of such differences between OOP and RDBMS (often referred to as a *paradigm mismatch*, or an *impedance mismatch* [9]) there is a need of components that perform transformation between these representations. While developing such components, two main goals are reducing runtime overhead for each transformation and reducing development effort.

One approach to connect OOP and RDBMS representations is hand-coding the persistence layer. In this case the developer has to manually implement saving and loading data using SQL queries and low-level database access APIs (e.g. ADO .NET for Microsoft .NET framework or JDBC for Java). Advantages of this approach include maximum performance, complete control over the process of saving and loading data and a possibility to use advanced features of particular database systems. The main drawback is a large amount of routine and error-prone work, during both implementation and maintenance of the persistence layer.

Another approach is to use object-relational mapping (ORM) systems that automatically save and restore the object graph into RDBMS using a formal description of a mapping [1]. Popular ORM solutions include Hibernate for Java [1], NHibernate and Entity Framework for .NET [8]. The advantages of ORM solutions

include significantly reduced development efforts, increased maintainability and independence from the database provider. Their main drawback is the runtime overhead: automatically generated SQL queries can be less efficient compared to those created and optimized by hand. Also ORM systems can generate unused features, such as queries for updating a table that should be read-only. Some ORM solutions may impose restrictions on the domain classes, such as inheriting from the base class provided by ORM or restrictions on the types of collections and associations.

Yet another approach to avoid a paradigm mismatch is to use object-oriented or object-relational databases [11]. Such solutions include a direct support for object-oriented features such as the object composition, link navigation, encapsulation, inheritance and polymorphism. Popular object databases include Caché [6] and Google App Engine Datastore [10]. Also object-oriented features are included in traditional relational databases, such as Oracle and Microsoft SQL Server [11]. By using object databases the transformation overhead can be avoided entirely, thus increasing both the developer productivity and the runtime performance. Another advantage is a simpler data model that is common for the data storage and data manipulation. Major drawback of this approach is reliance on less popular, and therefore less optimized and standardized database engines. Therefore currently object databases are used in some specific applications [11], but they cannot replace RDBMS as a mainstream data storage technology.

The analysis shows that current solutions of the object-relational paradigm mismatch force the application developer to make a choice between the automation and flexibility. Therefore our goal is to find effective ways to store the object graph aimed at achieving maximum performance and flexibility, as well as the sufficient degree of automation.

In this paper we use the following approach. We create the custom persistence layer, similar to the manual approach, by using code generation tools to automate its creation and reduce amount of routine coding. RDBMS are used as the data storage (instead of object databases) because of their high reliability and performance. The generated persistence layer consists of business objects (classes) that interact with the database and are generated from the database structure. The persistence layer provides safe interaction with the database by default, to prevent attacks such as SQL injections. Persistence layer operates with strongly typed objects that allow catching many errors and typos during compilation and not during query execution. All aspects of generated code can be customized without changing the structure of the application.

There are existing systems that use the similar approach, such as a NetTiers Application Framework [7]. However it is based on a commercial CodeSmith template engine [5] which increases the cost of the system. Also the NetTiers system does not generate strongly typed objects for all stored procedures, and supports only C# language – there is no possibility to generate the code in different programming languages.

3 The C-Gen System

In this section we describe the C-Gen – our code generation system that can be used as a solution of the object-relational paradigm mismatch.

The C-Gen system generates a persistence layer of the application using a database structure as an input. The database describes the entities in a subject domain and therefore can be used to create business objects. Currently the system is unidirectional: we can generate business objects from the database structure, but not vice versa.

The C-Gen uses a code generation approach, i.e. the automatic generation of source code from the given input data. As a code generation tool we use a Text Template Transformation Toolkit (T4 [4]). Templates are used to generate a program source code based on the model (database structure). The generated file can use an arbitrary text format, in particular, it can be a program source code in any language.

The generation process starts from existing database that is designed manually. A domain model is represented as a set of tables and views with relations. The database developer takes all responsibility for creating database. This approach supports maximum performance and flexibility of the created application. The developer has a full control over the process of designing and creating the database. As a result the developer can create a high-quality and efficient SQL code.

When the domain model is implemented in the database, we use the C-Gen system to generate a persistence layer. It consists of stored procedures inside the database, as well as business objects and access methods in a source code. The C-Gen supports two types of stored procedures: *simple stored procedures* support CRUD (create, retrieve, update, delete) operations and are generated automatically; *custom stored procedures* are specific to a concrete situation, and their design is fully controlled by the developer. Thus, by generating simple stored procedures automatically the developer is relieved from writing them by hand. On the other hand we reserve the possibility of implementing performance-critical stored procedures for the developer.

Custom stored procedures are also used to represent the relations between the tables. The tables can be related in one of three different ways: one-to-to one, one-to-many, many-to-many. Therefore one object in OOP can be represented as several connected RDBMS tables. There are other differences between RDBMS and OOP data model: tables in RDBMS don't support encapsulation, inheritance and polymorphism which are basic properties of OOP paradigm; tables have explicit identity provided by primary keys while objects can rely on implicit identity of memory location; objects support link navigation while tables rely on joins to access related properties.

Custom stored procedures that represent the relationships between the tables and views contain logic of joining the tables and views between each other. As the result we have a new object that represent such relation. This object presented in souse code as strongly new class, and custom procedure as the method. In this way we can map database relations on objects.

The next step after generating stored procedures is creating business objects. For each entity in the database, the C-Gen system generates a corresponding class. Each

database field is represented as a strongly-typed property. For each custom stored procedure the C-Gen system also generates a corresponding class based on the procedure name and return fields. The final step is the generation of access methods for simple and custom stored procedures; their parameters and return type depend on stored procedure signature.

After the C-Gen system completed its work, the application developer can work with business classes as required in application. Working with database is completely hidden behind generated methods. If some changes are made to the database structure, the persistence layer can be regenerated. Therefore the C-Gen combines the automation of ORM systems with flexibility and performance of hand-coded persistent layer.

Notice that the C-Gen system encapsulates many features common to the database access code. The generated code retrieves connection strings from application configuration files, establishes connection to the database, sends and retrieves data, manages transactions, etc. Of particular importance is built-in validation that prevents SQL injections and similar types of attacks, making the generated code more secure by default. All these features are provided without any efforts from application developers. However, if the requirements of particular applications conflict with the default implementation, it can be easily changed by editing templates. This is another advantage of the C-Gen system compared to ORM solutions.

In some cases there is a need to customize the generation process. The C-Gen system provides a variety of options aimed at customization. The system provides capabilities to choose which code is generated (partial generation) and customize business objects. Using partial generation prevents the system from generating an unused code. It allows the developer to customize the generation process by specifying which features should be generated.

Another customization option affects generated classes representing entities from the domain model. In some cases there is a need to extend generated classes, e.g. add some properties that are not saved to database, provide additional methods or override generated methods. In order to support such scenarios, the C-Gen generates two classes per entity: abstract and final sealed class. The abstract class contains automatically generated implementation of all the methods. Final sealed class inherits from an abstract class. Therefore it is possible to override some method in final sealed class, implement some interface, define custom logic etc. Final sealed class will not be overwritten during the next code generation.

4 Performance Evaluation

To evaluate advantages of our approach, we have compared the performance of the persistence layer generated by the C-Gen system with a code generated by NetTiers [7]. For an 18Mb sample database (representing an Internet shop) we have generated all relevant code using both systems. Then we measured the performance of *Select* and *Insert* operations for different query loads. Measurements were performed on a server with Core i5-2500k 3.3 GHz CPU and 8Gb RAM, running Windows 7 x64 SP1 and Microsoft SQL Server 2008 R2 x64 Express. The results of performance measurements are shown on Fig. 1.

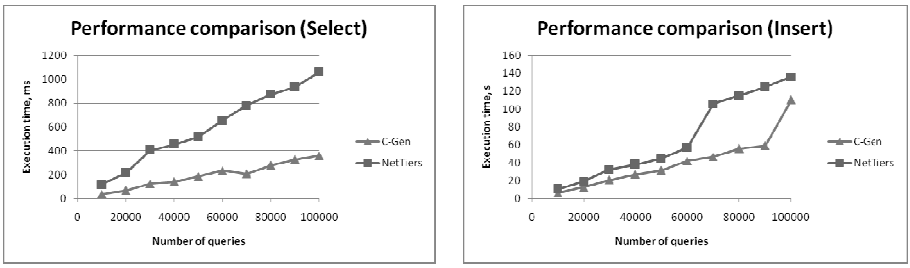

Fig. 1. Performance comparison of C-Gen and NetTiers systems

As can be seen from measurement results, the C-Gen system generates more efficient code: for *Select* operations it is about 3 times faster compared to NetTiers, and for *Insert* operations – 1.5-2 times faster. Another advantage of the C-Gen system is much smaller size of the generated code: all generated code for C-Gen is 1.2Mb, compared to 13.3Mb for NetTiers.

5 Conclusion

In this paper we have described our approach to creating a persistence layer to connect object-oriented and relational data. Our approach focuses on providing high-performance and flexible solution with high degree of automation. To this end we use a text template system to generate SQL queries, business classes and data access methods. We have implemented the C-Gen code generation system that creates easy to use and efficient persistence layer based on the database structure. Code generation mechanisms based on T4 text templates allow to generate the source code in any programming language. As a result the developer can focus on implementing more complex logic in custom stored procedures. Also the overall performance of application is improved. The system has built-in data validation mechanisms that prevent SQL injections and similar attacks. The application developer can interact with the database through strongly-typed objects that avoid runtime errors. The system is easily extensible and customizable by modifying templates, writing custom stored procedures, removing unneeded features and extending generated classes. Performance evaluation demonstrates high efficiency of the code generated by the C-Gen system compared to the similar system (NetTiers), both in terms of code size (10 times smaller) and execution speed (1.5 – 3 times faster).

Further research directions include the development of templates to automate other routine development tasks, such as the creation of user interface components for data editing. Another possible direction is extraction of templates from existing code to simplify the automation of development process and capture domain knowledge.

References

1. Bauer, C., King, G.: Java Persistence with Hibernate. Manning, New York (2007)
2. Booch, G., Maksimchuk, R.A., Engle, M.W.: Object-Oriented Analysis and Design with Applications. Addison-Wesley (2007)
3. Coad, P., Nicola, J.: Object-Oriented Programming. Prentice Hall, Upper Sadder River (1993)
4. Code generation and T4 Text Template, http://msdn.microsoft.com/en-en/library/bb126445.aspx
5. CodeSmith tools, http://www.codesmithtools.com/
6. Kirsten, W., Ihringer, M., Rudd, A.: Object-Oriented Application Development Using the Cache Postrelational Database, 2nd edn. Springer (2003)
7. .netTiers Application Framework, http://nettiers.com/
8. O'Neil, E.J.: 2008. Object/relational mapping 2008: hibernate and the entity data model (edm). In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008), pp. 1351–1356 (2008)
9. Russell, C.: Bridging the Object-Relational Divide. Queue 6(3), 18–28 (2008)
10. Sanderson, D.: Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure, 1st edn. O'Reilly (2009)
11. Tamer A-zsu, M., Valduriez, P.: Principles of Distributed Database Systems, 3rd edn. Springer (2011)