Models@run.time for Object-Relational Mapping Supporting Schema Evolution

Sebastian Götz and Thomas Kühn

Institut für Software- und Multimediatechnik
Technische Universität Dresden, Germany,
D-01062, Dresden, Germany
sebastian.goetz@acm.org, thomas.kuehn3@tu-dresden.de

Abstract. Persistence of applications written in an object-oriented language using a relational storage system has been investigated for a long time [4]. In this paper, two problems of current approaches to object-relation mapping are addressed. First, their high configuration effort, and second, their lacking support for continuous development. To address these problems, we introduce a novel object-relational mapping approach, that uses a runtime model of the system. The runtime model is utilized in two ways. First, to derive mapping information from the runtime state of the application, that usually has to be provided by developers. Second, to allow for lossless application schema evolution. That is, we present an approach, that reasons about design time and runtime information to relieve developers from configuration details of the object-relational mapping and show how to utilize the same information to allow for continuous schema evolution of applications.

Keywords: object-relational mapping, object-roles, schema evolution

1 Introduction

The transformation of domain objects between object-oriented systems and relational databases has been investigated for a long time, resulting in industry-wide accepted standards for object-relational mappers (ORM), like the Java Persistence API (JPA). In consequence, software engineers got a sophisticated abstraction layer for persistence. Common ORM relieve software engineers from manually creating database schemata, but instead derive the relational from the object-oriented schema.

In this paper, two problems of current approaches to object-relation mapping are addressed. First, their high configuration effort, and second, their lacking support for continuous development.

The high configuration effort is due to the lack of taking runtime information into account for the automated mapping. Current ORMs either use configuration files or annotations to specify, which classes and attributes have to be persisted in which way. Both approaches only refer to the object-oriented schema of the application, but do not take runtime information into account.

This runtime information can, e.g., reveal the cardinality of an N:M collection. Another example, is the approach to be used for mapping inheritance to the relational schema. Which approach is best, depends on how many objects exist or are instantiated for which classes in the hierarchy. Current ORM require the developer to provide such additional information about the expected runtime of the application.

In other words, ORMs do not support a continuous development process. Changes to the schema of an application typically lead to the loss of data collected during running older versions of the application. This imposes additional efforts on the developers during continuous development, because for each new version of the application, they either have to manually update the database schema or to backup and migrate the data from the old to the new schema.

Thus, this paper aims at a novel approach to object-relational mapping, based on the models@run.time paradigm, (i) to use runtime information in addition to design time information to automatically infer the relational schema and (ii) to support lossless schema evolution.

The paper is structured as follows. In the next section related work is outlined, followed by the main concepts of our approach in Sect. 3. We conclude the paper in Sect. 4.

2 Related Work

Object-relational mapping is a well discussed research topic [4]. The Java Persistence API, part of the EJB 3.0 specification from JSR 220, forms a standard for object-relational mappers in the context of Java. Many implementations of this standard exist, for example Hibernate[1]. Ports of these approaches to the .NET environment — for example NHibernate[2] — and further approaches, like Microsoft's Entity Framework exist. Current object-relational mappers provide transparency to the developers by relieving them of specifying the database schema manually. Instead developers write configuration files or annotate their code to describe, which classes have to be mapped in which way to the database. Some information, like the type of an attribute, is extracted by static code inspection to keep the configuration files lean. Nevertheless, not all mappings can be derived using static code inspection. For example, the kind of a relationship between two classes referencing each other (1:N vs N:M) or the optimal mapping of an inheritance hierarchy (table per class, table per hierarchy, ...) with regard to the trade-off between data redundancy and performance. Our approach utilizes runtime information of the application (i.e., knows about the dynamic object structure) to derive further mapping information, like the two examples described above.

Beside frameworks for object-relational mapping, object-relational DBMS (ORDBMS) [8] (e.g. PostgreSQL) have been developed. They offer a rich querying language and allow to store data in complex, object-oriented structures.

[1] http://www.hibernate.org
[2] http://www.nhforge.org

Many concepts of ORDBMS have been incorporated into SQL99, that is supported only partially by major DBMS. Oracle, IBM, Microsoft and Sybase support (parts of) SQL99 with varying degree (e.g., Sybase ASE does not support table inheritance).

Schema evolution is another well discussed research topic. We address a special kind of schema evolution, namely *co-evolution* of object-oriented applications and their generated database schema, which has rarely been discussed. MeDEA [3] is an approach using manual specifications of how application schema changes are translated into database schema changes. In [9] changes to the application are reflected as changes of the mapping between applications and storage, which allows for automatic co-evolution. Our approach uses change descriptions to build the required select, insert and update queries to store and restore objects. Removals and complex changes (e.g., rename, move, ...) are handled by our approach without affecting the database. Only additive changes (added attributes, classes, etc.) are applied to the database schema. In consequence, time-consuming changes, like splitting a class, are not propagated to the storage. Instead the change descriptions are used to translate between application and storage schema. Hence, our approach shifts performance penalties from change to data access time. If part of the data is accessed frequently, our approach allows persisting the changes affecting that data. Nevertheless, the focus of this paper with regard to schema evolution is on using change descriptions to avoid or postpone time-consuming propagation of application changes to the storage by generating the required queries based on the change descriptions. In [5], a language for systematic database evolution is presented. This approach allows to systematically evolve the database including its data. Approaches to automatically evolve database queries such as Prism [2] exist, too. Such approaches provide means to evolve queries in accordance to changes in the database schema. Contrarily, queries in our approach evolve in accordance to changes in the application schema. Moreover, we *generate* queries, whereas approaches like Prism focus on the evolution of manually written queries.

In summary, open challenges for object-relational mapping are (i) effective means to reduce the configuration efforts and (ii) means to handle lossless schema evolution to support continuous development. How we addressed these two challenges is described in the succeeding section.

3 Concepts

For our object-relational mapping approach based on models@run.time with support for schema evolution, we propose a novel architecture as depicted in Figure 1. The principle idea is to enrich applications at load time with code that exposes the existence and state of all objects subject to persistence. By this, in addition to schematic information, the application's runtime state can be covered in a runtime model, that is used to derive configuration information.

The process, as depicted in Figure 1, comprises 5 steps.

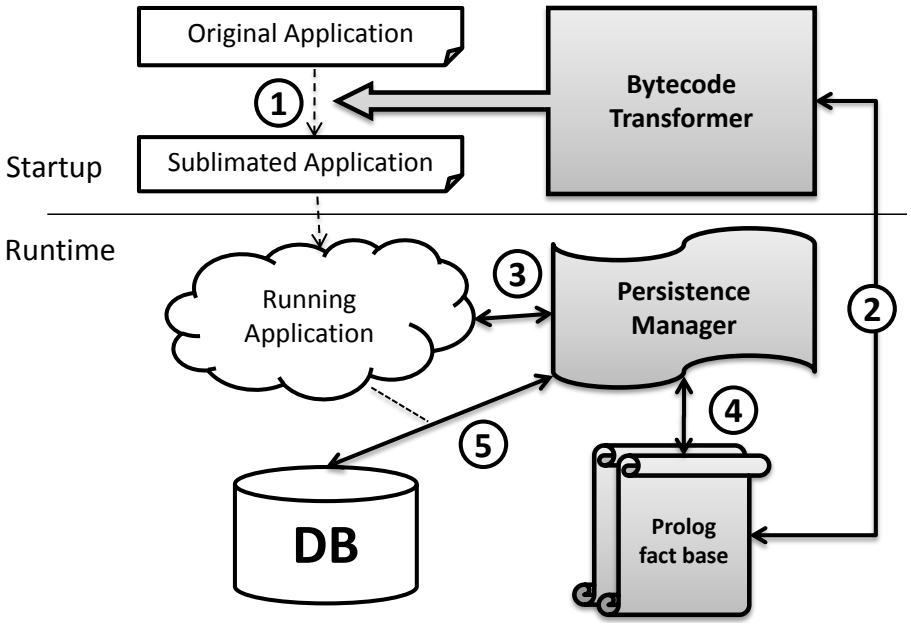

Fig. 1. Overview of Architectural Parts and their Connection.

(1) The application is transformed by a `Bytecode Transformer` that is introducing an explicit persistence-related event stream by weaving notification calls into the application.

(2) The transformer additionally inspects the applications schema and translates it to a Prolog fact base, dedicated to schema information (`schema fact base`). When a new version of the application is started, schema changes are identified by the transformer and are noted in that fact base, too. Using Prolog we are able to derive further information, like the transitive closure of inheritance hierarchies, using logical rules.

(3) If the developer decided to persist application schema changes, adjustments to the database are triggered at load time.

(4) Whenever the value of an attribute is changed or a new object is instantiated, e.g., in a constructor or in getter and setter methods, it will notify the `Persistence Manager` that keeps a runtime model of the current application's state in form of a Prolog fact base (`runtime fact base`).

(5) The `Persistence Manager` furthermore provides means to store, search and restore domain objects. All these activities force the `Persistence Manager` to connect and communicate with the database. When the `Persistence Manager` connects to the database for the first time, it creates the complete database schema. Otherwise only changes are applied or change descriptions are added.

For example, as depicted in Figure 2, imagine the concept `Student` in a university management system. During development, such a concept is likely to change often. In a first version, the *class* `Student` associated to persons comprises the attributes `studentID`, `address` and `curSemester`, which denotes the current semester of the student. The class `Person` contains the attribute `name`. The next development iteration leads to the removal of `address`, due to privacy constraints, and the addition of the attribute `birthday`. The corresponding change operations are denoted as facts in the schema fact base, shown in the right upper side, too.

Thus, the four major architectural parts of our approach are the `Bytecode Transformer`, the `Persistence Manager`, the `runtime` and the `schema fact base`. Figure 3 depicts the interconnection of these parts and the Prolog fact bases, defined by five steps, which will be described in the following.

Sublimate. Persisting domain objects of applications means to store a snapshot of the application in the database. To derive such a snapshot all domain objects need to unfold their state. For that purpose the implementation of all annotated classes is enriched with additional code, so every change of an attribute is signaled as an event at runtime to the persistence manager. The creation of objects is signaled as an event, too. Thus, the implicit dataflow in the application is made explicit in form of an event stream. Technically, event notifications are realized as method calls, which are woven into constructors and after attribute assignment statements[3]. The listener to these events will be described in step *Trace*. The bytecode transformer has been implemented as a Java agent, that permits class modifications at load time, using the Javassist library [1] to analyze bytecode and to insert statements.

Extract and Compare. At load time of the application's classes, their schema is extracted to the schema fact base. If no previously extracted fact base exists, the schema will be extracted completely, whereby the existence of each class and attribute is noted as a separate fact. The relations between classes by inheritance and delegation are noted as predicates, too.

[3] e.g., `this.message = "Hello"` fires a `valueChanged` event for the `message` attribute

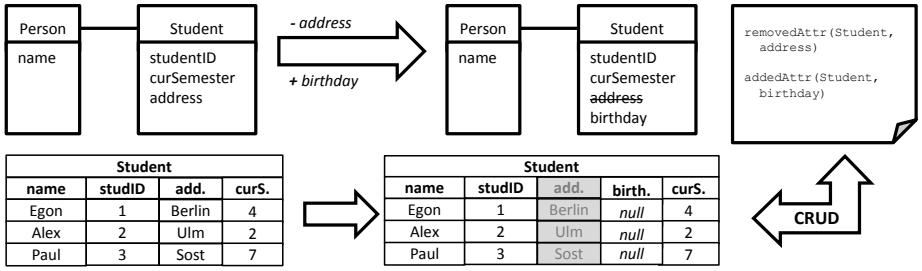

Fig. 2. Schema Evolution Example.

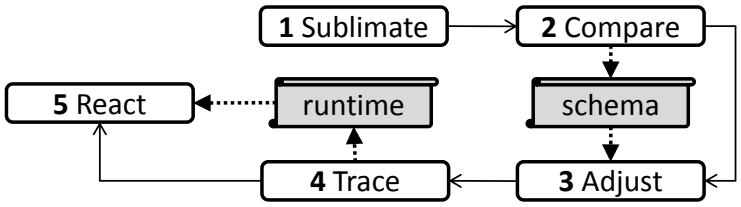

Fig. 3. Cycling Steps of our Approach with Fact-Base Connection.

If a formerly extracted schema already exists at application startup, the facts are compared. That is, for every fact which is extracted from the new schema its existence will be checked. If the fact already existed, there has been no change to this part of the structure. If the fact does not exist, it has either been changed or removed. It is not always possible to determine, whether the fact has been removed or changed. Facts of attributes can be changed in three ways: altering their type, their name or both. If only the type has been changed, the change can be identified by the name of the attribute, which needs to be unique for the class. This does not work, if the name has been changed. The root of these problems is, that the intention of the developer cannot be reconstructed. It might be, that the developer removed the old attribute and added a new one of the same type or he removed an old attribute and added a new one with a different type, but the same name. The same holds for facts about the existence of classes. Thus, changes to facts are recognized as removals and additions. By default the name will be used to identify changes. To cover the developers intent, we provide annotations (`@Renamed(old)`, `@Moved(old)` and `@SplittedClass(old)` to name but a few), that can be used by the developer to demarcate changed methods from newly added ones and thereby expressing the semantic correlation of corresponding removals and additions. To relieve the developer from explicitly stating his intent, the refactoring log of the used development environment could be used for that purpose as, for example, shown in [7].

The information derived from the comparison needs to be noted, too. This is because either the database schema (if the developer decided to persist changes) or queries against that schema need to be adjusted to the changed application schema, as will be described in the *Adjust* step.

Changes to the application are collected until the developer decides to *commit* them, i.e., removed attributes, classes, roles and so on are not removed from the database until the developer decides to apply the changes in a non-recoverable manner by marking the application as a milestone. At the next application startup all changes aggregated in the schema fact base are applied to the database schema. Additive changes to the application schema (new class, attribute, role, ...) are immediately applied to the database schema.

Leaving columns of removed attributes in the database schema could lead to problems, if for example, an attribute of the same name is added to the same class or (in case of table per hierarchy mappings) any subclass of it. Our

approach avoids such problems, because the fact, that there was an attribute that has been removed, is stated in the schema fact base and, thus, can be taken care of, e.g., by (internally) using a different name for the new attribute.

Adjust. As all changes can be seen as additions, removals or combinations of both, the fact base is extended with facts, representing the changes in this way. This knowledge enables to react to changes at runtime in a flexible way. When restoring objects, the projection of the respective select queries has to be adjusted. Storing objects requires default values for removed attributes. For example, knowing, that an attribute has been removed, is reflected by removing it from the projection part of all related select queries and adding a default null value for the removed attribute, when storing the object. Splitting a class C into C_1 to C_N leads to new select queries, that project only attributes required by each new class, respectively. Storing objects is reflected by N insert statements, one for each C_i.

The *addition* of classes leads to the extension of the corresponding relational schemata. The *removal* of them does not lead to their deletion from the database schema. They will be kept in the database, like attributes, until the developer marks the application version as milestone release.

Trace. As described before, the implicit dataflow is transformed into an explicit event stream, capturing the state of the domain objects and their life cycle. Furthermore, schema information is noted in the schema fact base. This way design and runtime information is available for static and dynamic structure analysis. In consequence, lots of information, that usually has to be provided in configuration files, can be derived using logical rules like, e.g., multiplicities of relationships (1:1, 1:N or N:M) or an optimal mapping of an inheritance hierarchy for the current data. We will illustrate this by example in the following paragraph.

Consider a class `StudentGroup`, referencing a collection of students. Class `Student` declares a reference to the group. Whether a student can be a member of more than one group is "hidden" in the code to manage the collection. Depending on the kind of relationship (1:N vs. N:M) a different relational mapping is used. The more general case leads to a separate relation pointing to `Student` and `StudentGroup`, whereas else only a column is added to `Student`, pointing to the `StudentGroup`. To detect the kind of relation one can look at the dynamic object structure. Two instances of `Student` pointing to the same `StudentGroup` indicate an N:M relation. This approach requires a set of objects to analyze, which might not exist when the system is started. Hence, if not enough data is available to defer valuable information from the dynamic object structure, the more specific case is used in the first place. In particular, in the example above, only an additional column is added to class `Student`. If it turns out later, that the system allows for the more general case, the mechanism for schema evolution of the approach is utilized. That is, the required table for the mapping of students to groups and vice versa is added, but the data is not migrated. The additional column remains in the table for class `Student`. The runtime utilities

take care of accessing the mapping relation, handling new mappings and editing existing ones by generating the corresponding queries appropriately.

A further benefit of having static and runtime information as Prolog fact base is, that it allows for *automatic normalization*. To normalize a schema, its functional dependencies need to be identified. For this purpose, more than just the schema is needed, but runtime information, too. Unfortunately, fully automatic normalization of real-world schemata is very time-consuming. For example, classes, having 20 or more attributes, have an enormous amount of possible functional dependencies[4], that have to be evaluated. Optimizations to the normalization algorithm allow deriving the normalized schema with limited memory space and in shorter time, but still the required processing time is far away from viable usage at runtime.

React. This final step contains the listener of the event stream from the step *Sublimate*. Besides forwarding the events to the trace utility, described in the previous step, it provides utilities to store and restore domain objects.

When an object is to be stored, can be chosen from a set of *persistence strategies* by the developer. They effect data consistency in multi-user environments as well as transactional security in terms of checkpoints:

1. Manual — the developer proactively handles transactions
2. Application Level — all objects are stored at application shutdown
3. Instance Level — all objects are stored, whenever a new object is instantiated
4. Value Level — single objects are stored, whenever one of its values is changed

First, the developer may manually invoke the `store`-process. In other terms, he proactively commits a transaction. The transformer injects this method, if the developer annotated a method for that purpose. The other three strategies weave this call into the appropriate places, e.g. into the handler of value changes for value-level persistence.

Second, with application level persistence all objects are only stored, when the application shuts down. This leads to the lowest runtime-penalties, because there is no database interaction while the application is running. In contrast, multiple users working concurrently on the same data do not see each others changes until they restart the application. Hence, data consistency cannot be ensured. A transaction in this strategy is of little use, as it spans the time from startup to shutdown of the application. Hence, the user has only two checkpoints during his work, which in case of a system crash leads to the loss of all changes since the application has been started. Notably, storing all objects does not need the data to be fully available in main memory. This is because the fact bases containing the data are (text) files and iteratively storing each single object (or groups of them) requires only part of them to be held in main memory.

Third, instance level persistence updates the database, whenever a new object is instantiated. Multiple users, working concurrently on the same data, see

[4] Each subset of the 20 or more attributes may functionally depend on each other subset of the 20 or more attributes.

course-grain changes of each other. Also course-grain transactions are supported. That is, checkpoints exist, whenever an object was created. This comes for the cost of lower application performance, because now the application interacts with the database while the application is running.

Finally, value level persistence is the most fine-grained strategy. Whenever a value is changed, this change will be reflected in the database, leading to high performance penalties, but fine-grained multi-user support and transactional security. To restore objects a search mechanism is required. That is, a query language like RSQL [6] is needed to search for objects to be restored.

The reconstruction of found objects is technically challenging, which is mostly due to the absence of explicit setter methods for all attributes of a class. The bytecode transformer weaves in special constructors and setters for that purpose.

In summary, our approach reduced the configuration efforts for developers by collecting static as well as runtime data of the application in Prolog fact bases to derive information about the dynamic structure, which else has to be provided by the developer. Our approach allows for application schema evolution by using change descriptions to generate the required select, update and insert queries. The only changes, that need to be directly propagated to the database are additive changes, posing no problem to the majority of RDBMS.

4 Conclusion

We presented a novel approach for object-relational mapping. We reduced the configuration efforts for developers by extracting the application schema and, using bytecode transformations, the dataflow of the application's domain objects into Prolog fact bases, that allow to reason about the *static* and *runtime* structure of the application. Support for schema evolution is achieved by using change descriptions to generate the required queries to store and restore objects. This shifts performance penalties from change time to data access time. In consequence, changes to those parts of the schema defining frequently accessed data can become performance bottlenecks. To handle such cases, our approach utilizes (existing) mechanisms to persist changes (i.e., applies the changes to the database schema).

As future work, the practicality and scalability of the approach have to be investigated using case studies. Moreover, an empirical analysis to decide when schema changes should be persisted and until when the overhead due to not persisting the changes is acceptable, should be conducted.

Acknowledgments

This work is funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907) and in the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing".

References

1. S. Chiba. Javassist — a reflection-based programming wizard for java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.
2. C.A. Curino, H.J. Moon, and C. Zaniolo. Graceful database schema evolution: the prism workbench. *Proceedings of the VLDB Endowment*, 1:761–772, August 2008.
3. E. Dominguez, J. Lloret, A. Rubio, and M. Zapata. Evolving the implementation of isa relationships in eer schemas. In *Advances in Conceptual Modeling - Theory and Practice*, volume 4231 of *Lecture Notes in Computer Science*, pages 237–246. Springer Berlin / Heidelberg, 2006.
4. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 5th edition.* Addison-Wesley, 2007.
5. Kai Herrmann, Hannes Voigt, Andreas Behrend, and Wolfgang Lehner. Codel - a relationally complete language for database evolution. In *Proceedings of the 19th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, 2015.
6. Tobias Jäkel, Thomas Kühn, Hannes Voigt, and Wolfgang Lehner. Rsql - a query language for dynamic data types. In *Proceedings of IDEAS*, pages 185–194, New York, NY, USA, 2014. ACM.
7. Ilie Savga, Michael Rudolf, and Sebastian Götz. Rigorous and practical refactoring-based framework upgrade. In *Proceedings of 7th International Conference on Generative Programming and Component Engineering (GPCE'08)*, 2008.
8. M. Stonebraker, D. Moore, and P. Brown. *Object-Relational DBMSs: Tracking the Next Great Wave.* Morgan Kaufmann, San Francisco, 1998.
9. J. Terwilliger, P. Bernstein, and A. Unnithan. Automated co-evolution of conceptual models, physical databases, and mappings. In *29th International Conference on Conceptual Modeling (ER 2010)*, volume 6412 of *Lecture Notes in Computer Science*, pages 146–159. Springer Berlin / Heidelberg, 2010.