Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 77 (2005) 193–207
The Journal of Systems and Software
www.elsevier.com/locate/jss

Model driven generation and testing of object-relational mappings

Stephan Philippi *

Department of Computer Science, University of Koblenz, P.O. Box 201 602, 56016 Koblenz, Germany

Received 14 July 2003; received in revised form 19 July 2004; accepted 26 July 2004
Available online 11 September 2004

Abstract

Object-oriented software development as well as relational data storage are leading standards in their respective areas. The persistent storage of objects in relational tables is therefore a topic of major interest. To do so efficiently, a plethora of problems has to be overcome due to the impedance mismatch between the object-oriented and relational paradigms. Nowadays, dedicated object-relational middlewares are frequently used to decouple relational databases from object-oriented applications. Even if this approach shields the developer from the majority of run-time related aspects, the manual mapping of incrementally evolving complex object models to relational tables still remains as an inherently difficult and error-prone task. Therefore, this article focuses on automation support for the model driven generation and testing of object-relational mappings.

Keywords: Object-oriented software development; Relational database; Impedance mismatch; Object-relational middleware; Object-relational mapping

1. Introduction

Whereas object-orientation has become the standard paradigm of software development with the UML (OMG, 2004) as lingua franca, there seems to be no single best solution for the persistent storage of objects. The most natural approach would be the use of an object-oriented database management system (OODBMS) for these purposes. Although the ODMG standard for object-oriented databases (Cattell et al., 1999) and its implementations have matured over the last decade, relational and object-relational database management systems (RDBMSs/ORDBMSs) are still commonly used and there are almost no indications for a significant change in the future. Due to the popularity of relational data storage with its broad installation base on the one hand and the use of object-oriented concepts for software development on the other, application objects inevitably need to be stored in (object-) relational databases. Unfortunately, the interplay of objects and relations is not trivial, as there is a significant distance between the object-oriented and relational paradigms. This distance is often referred to as *object-relational impedance mismatch* (Smith and Zdonik, 1987).

To be able to access a relational database from an object-oriented application, there are in principle three different possibilities (see e.g. Ambler, 2000). The easiest approach is to directly include SQL statements into the application code. The advantage of this approach is that it allows for the rapid development of prototypes and small systems. For more complex object models and production systems the tight coupling between application and relational schema is unsuitable, as modifications of the schema often require changes in the application source code. The encapsulation of SQL statements into own *data classes* is a slightly better approach from an architectural point of view. But even if this separates concerns in conformance to object-oriented ideas, changes of the relational schema still lead

* Tel.: +49 261 287 2720; fax: +49 261 287 2721.
E-mail address: philippi@uni-koblenz.de

0164-1212/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2004.07.252

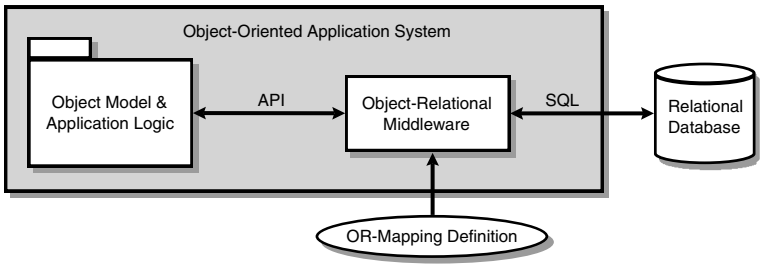

Fig. 1. Encapsulation of a RDBMS with an object-relational middleware.

to modifications of the application source code. An approach to overcome these problems is the use of a dedicated middleware, which decouples relational databases from object-oriented applications. Fig. 1 shows the basic principle of this approach. During the run-time of an application, the object-relational (OR) middleware is configured with application specific definitions on how object structures have to be mapped to a relational schema. The SQL queries needed to access the database are then dynamically generated by the middleware, which thus encapsulates the database from an application point of view. The main advantage of this principle is that the database schema may be changed without the need for application source code modifications. For the application to work properly, only the mapping specification needs to be adjusted.

However, even if an existing middleware is used for the decoupling of object-oriented applications from relational databases, mappings between these different structures need to be specified manually. In this context, *forward engineering* refers to the mapping of object structures to relations, while *reverse engineering* starts from existing relational structures and maps them to object-oriented ones. The perspective taken throughout this article is forward engineering. Due to the inherent complexity of mapping object structures to relational ones, especially the development of object-relational mappings for non-trivial object models takes considerable efforts and is highly error-prone. Engineers in this area, in fact, need an in-depth knowledge of two distinct paradigms and their particular differences. What makes matters even more difficult is that object models evolve during their development and thus change their structure over time. In order to free system engineers as far as possible from the need for manual mapping specifications, this article introduces an approach for the automatic generation of object-relational mappings.

Even if the transformation of design models to relational structures has a long tradition which started with the invention of the entity-relationship model (Chen, 1976), the automatic generation of object-relational mappings is only rarely discussed in the literature. In addition to publications on this topic, another area of related work are automation features provided by existing OR-middlewares [1]. However, 'generation' capabilities of software in this area usually only refers to:

(a) The automatic import of object models from class diagrams or implementation code into a mapping tool to support the manual mapping to relational table structures (forward engineering), and/or
(b) the automatic import of existing relational tables into a mapping tool to support manual mapping to object structures (reverse engineering), and/or
(c) the generation of static code to map objects to relations at run-time. The latter is only necessary for OR-middlewares, which do not dynamically generate SQL at application run-time.

With tools in this category, like 'Object Integrator' (Secant Technologies, 2001), 'EdgeXtend' (Persistence Software, 2002) and 'Cocobase' (Thought Inc., 2002), the development of object-relational mappings is still a manual task. An approach for the automatic mapping of object models to relations is described in (Tesch and Volz, 1999). A drawback of the underlying concepts of this proposal is that only object models with structurally restricted inheritance hierarchies are covered. In contrast, the approach taken by 'TopLink' (Object People, 2002) is restricted to the automatic mapping of attributes from classes, which are already associated to existing tables, i.e. only a small part of the overall mapping task is automated. A more general disadvantage, which to the best of our knowledge holds for all existing approaches in this area, is that the automatic mapping of objects to relations is only supported with single fixed translation patterns, see e.g. (Keller et al., 1993) and (Tesch and Volz, 1999). With a fixed mapping pattern the developer has no control over the generation process and can therefore not customize the outcome to the specific needs of his project. As there is no single best mapping pattern for *all* application systems, such static strategies are bound to be unsuitable to meet the requirements in the majority of more complex projects.

[1] A more detailed comparison of existing object-relational middlewares is given in (Cunningham and Cunningham Inc., 2004).

What can be learned from the problems of existing approaches for the automatic generation of object-relational mappings is that system requirements have to be considered. In order to take the automatic generation of object-relational mappings a step further, therefore, the input to the automatic mapping function to be introduced below are an object model *and* the requirements for the mapping of this model. The mapping requirements are declaratively specified by the developer within the object model. As a prerequisite for the discussion of the underlying principles of this approach, the next section of this article discusses and evaluates the principles of middleware-based object-relational mappings. Afterwards, concepts for the model driven generation and testing of OR-mappings are introduced in sections three and four, followed by a brief description of a prototypical implementation of the presented ideas in an experimental CASE tool. A short summary and an outlook on further perspectives finally conclude the presentation.

2. Principles of middleware-based or-mappings

This section deals with middleware-based application systems development as well as the principles of object-relational mappings. The perspective taken is that of forward engineering, i.e. object-relational mappings are developed starting from an object-oriented application model. Reverse engineering, which starts with a given relational model, is discussed for instance in (Ramanathan and Hodges, 1997; Kwan and Li, 1999).

2.1. Middleware-based application systems development

The model driven development process of object-oriented applications based on OR-middleware is sketched in Fig. 2. In the first stages of a project an object model is developed according to the requirements. During the development of the application object model it has to be decided, which classes of the architecture have to be persistence enabled, i.e. classes have to be identified, from which instances have to be stored in a database. If the application development has reached a state where large data sets are needed for testing purposes, the selected classes have to be manually mapped to relational tables. Afterwards, the database structures are created either manually or automatically by the mapping tool (if supported). As the mapping process itself is inherently complex, and also the interplay of object-oriented application, object-relational middleware, and relational database is non-trivial, the developer usually also creates a test suite to automatically test for the correct storage and retrieval of objects from persistence enabled classes.

The process in Fig. 2 does not explicitly reflect that the development of a complex application system is actually an iterative task in which the system incrementally evolves. A consequence of evolutionary systems development is that each iterative step which changes a persistent class or its position in the inheritance hierarchy, inevitably leads to the partial redefinition of the object-relational mapping as well as the relational structures. Moreover, the test suite has to be partially re-implemented in order to reflect these changes.

2.2. Principles of object-relational mappings

The inherent difficulties of mapping objects to relations stem from the multitude of choices for each class, its attributes and associations, as well as its position in the inheritance hierarchy. Decisions in this context are not to be taken lightly, as they have a profound impact on the following aspects:

- *Understandability*: From a forward engineering point of view, a relational schema and the object-relational mapping on top of it are the easier to understand, the closer both resemble the application object model.
- *Maintainability*: An OR-mapping and the relational schema are the easier to maintain, the less changes are needed to adjust both to object model

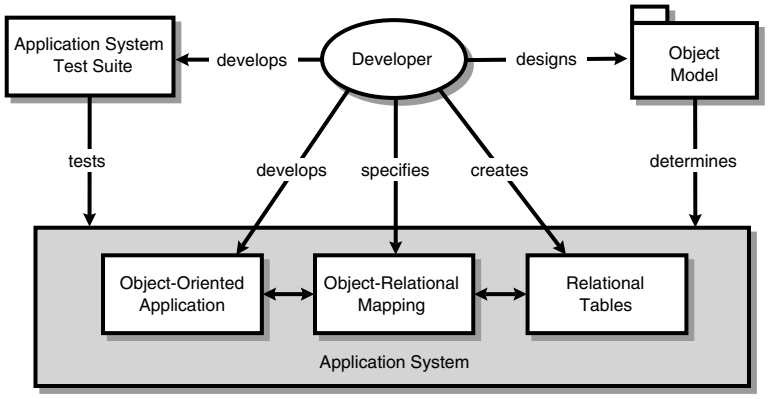

Fig. 2. Application system development process using an OR-middleware.

modifications. Maintainability is therefore a measure of difficulties which arise, if for instance cardinalities change and attributes are added to classes.
- *Storage space*: A relational schema does not waste storage space if redundancy and `NULL` values are minimized.
- *Performance*: The performance of a mapping strategy cannot be measured independently from other aspects like indexing strategy, concrete data sets, query types, DBMS and others. Out of this, the number of joins needed for the retrieval of application objects from a database is used as an abstract measure of performance throughout this article.

As a prerequisite to the discussion of automation concepts for object-relational mappings in section three, the underlying principles and alternatives are introduced in the following together with their impact on the aspects described above. The basics of OR-mappings are also discussed e.g. in Blaha et al. (1988), Agarwal et al. (1995), Brown and Whitenack (1995), Keller (1997), Ambler (2000) and Rahayu et al. (2000). The description of the mapping principles to follow does not take object-relational features of current ORDBMSs into account. The reason for this is that the object-relational extensions described in the SQL3 standard (Melton et al., 2001) are not yet fully implemented in every ORDBMSs. Thus, SQL92 (Darwen and Date, 1993) is still the common ground for object-relational mappings.

2.2.1. Mapping of classes and attributes

If a non-abstract class only contains attributes with basic types, it can be represented by a single relational table. With this simple mapping each field of the relation represents an attribute of the class. The programming language attribute types need to be mapped to equivalent types of the underlying DBMS. Objects of the class are then stored as rows of the corresponding table.

As objects and tuples have differing identity concepts, special attention is needed here. In the object-oriented world, each instance of a class has a unique object identifier (OID). This OID is transparently created as an implicit object property and cannot be changed. A tuple in a purely relational system, instead, needs to have a primary key to be able to be uniquely identified amongst all rows of a table. This primary key consists of one or more fields of a relation. In contrast to OIDs, these fields may carry application specific information and are thus subject to user modifications. With more recent object-relational DBMSs, immutable OIDs may be generated for each tuple. At least from a theoretical point of view this solves the identity problem. However, since not (yet) every DBMS supports this feature reliably, OR-middleware usually provides own functionality for these purposes. A common approach in OR-middlewares to cope with the identity mismatch is that each persistence enabled class needs to have an additional attribute which stores an explicit identifier. This identifier is then mapped as primary key to the corresponding relational table. Fig. 3 gives an example which maps a class 'Address' to a relational table according to this pattern.

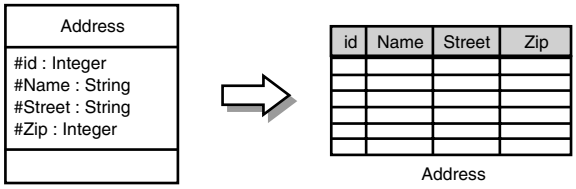

Fig. 3. Simple mapping of classes to relations.

The one-to-one mapping of classes to tables is the most basic pattern which is often sufficient, but not always suitable or even possible. Therefore, classes may generally be mapped to multiple relations and also multiple classes can be mapped to a single relation. The decision to take depends on the position of a class in the inheritance hierarchy as well as the importance of performance, understandability, maintainability, and storage space requirements. Another aspect in this context is the handling of abstract classes, which do not have direct instances in object-oriented applications. Still, the mapping of such structures into separate relations might be advantageous, as will be discussed below. Classes with non-basic attribute types, like lists, cannot be mapped as described above, too. The reason for that is the first normal form of relational databases, which prohibits the storage of structured values in relational fields. Non-basic attribute types are therefore usually handled like associations.

2.2.2. Association mappings

As *n*-ary associations of a conceptual domain model are usually transformed into binary ones in design models, the discussion in this section focuses on one-to-one, one-to-many and many-to-many relationship mappings. Each of these associations may either be uni- or bidirectional. The different choices for mapping associations to relational structures are described in the following:

One-to-one associations: There are several alternatives for the mapping of a one-to-one association as given in Fig. 4a. The first possibility is to map both classes as well as the association to separate relations (Fig. 4b). The advantages of this approach are that it is not difficult to understand and also schema maintenance is supported, as cardinality modifications in the object model can be easily integrated into the relational schema. On the other hand, this mapping pattern may be unsuitable if query performance is important. In such a case, both classes may be mapped to separate tables, where a foreign key reference to the association target is embedded into the source table (Fig. 4c). Performance of association traversal is improved with this mapping, but main-

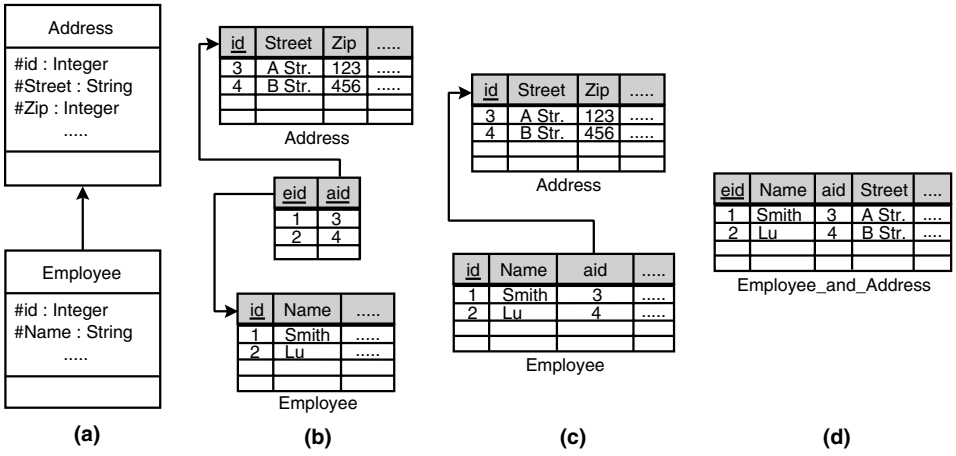

Fig. 4. One-to-one association mappings.

tainability is slightly affected. An even better performance can be achieved on the cost of these two aspects, if both classes and the association are merged into a single relation as given in Fig. 4d. In this case, no joins are needed in order to retrieve objects linked by one-to-one associations. However, if the one-to-one association is optional and the population of at least one of the participating classes is moderate or even high, storage space requirements are negatively affected due to the introduction of `NULL` values into the table.

If there is a bidirectional one-to-one association between two classes, only minor changes to the above described mapping patterns apply. In case of Fig. 4c an additional foreign key would have to be included into the 'Address' table to be able to traverse the association into the reverse direction. As a separate association table as well as a single table with merged classes implicitly enable bidirectional association traversal, no modifications are needed in these cases.

One-to-many associations: There are two choices for the mapping of a one-to-many association as given in Fig. 5a. From an object model perspective the representation of the association in a separate table is the more understandable and maintainable alternative of both (Fig. 5b). If performance is an important issue, this approach may not be advisable. In such a case a foreign key *to* the source table may be embedded into the association target table (Fig. 5c). The technical reason for this reversed foreign key embedding, which does not truly reflect the semantics of the object model, is that redundancy due to repetitive rows can be avoided in the association source table this way. However, even if this alternative is more efficient and in fact implicitly provides bidirectional association traversal, it is also less maintainable, as cardinality changes from one-to-many to many-to-many are not integrated easily.

Theoretically, a one-to-many association can also be merged into a single table as described above for

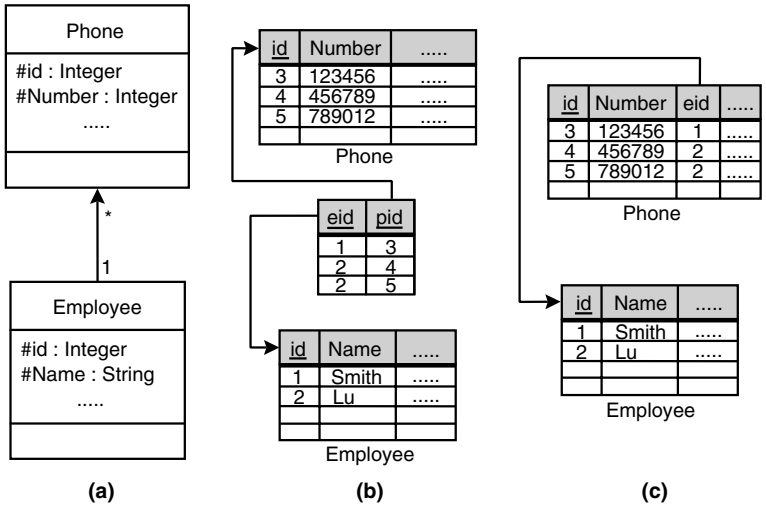

Fig. 5. One-to-many association mappings.

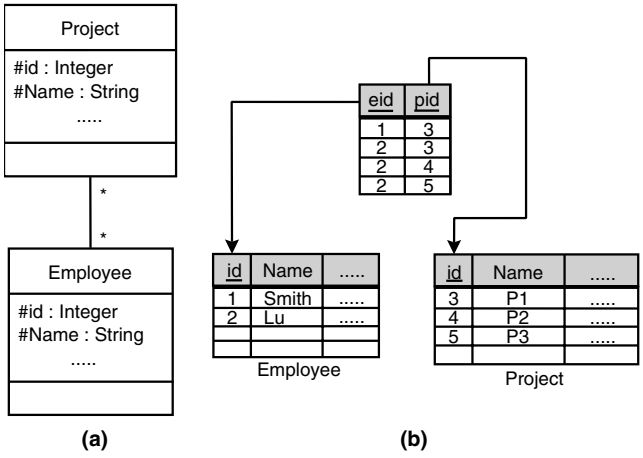

Fig. 6. Many-to-many association mapping.

one-to-one associations (4d). As this pattern inevitably leads to redundancy in the target table, it is not further considered here.

Many-to-many associations: The representation of a many-to-many association demands for a separate relational table which holds foreign keys to the participants. An example of a many-to-many mapping is given in Fig. 6. As already mentioned above, a separate table for the representation of associations implicitly covers their bidirectional traversal.

Many-to-many associations can also be represented with embedded foreign keys as well as a merged table. However, as these patterns introduce redundancy into the database, they are not further considered here.

Table 1 summarizes the impacts of the described association mapping patterns to a resulting relational schema and the OR-mapping on top of it in terms of understandability, maintainability, storage space requirements and performance. Only the described mapping alternatives which do *not* introduce redundancy into the underlying database are given. The evaluation scale ranges from *good* (+) over *moderate* (o) to *poor* (−). The perspective taken for the evaluation is that of an object-oriented application systems developer. As understandability and maintainability are qualitative aspects which also depend on the individual background of a developer, they cannot be measured objectively. Out of this, the table entries reflect at least partially our personal experience in this domain. Moreover, from a strictly relational point of view, some entries in the table would have been different. For instance, the use of 'reverse foreign key embedding' for one-to-many association mappings is a common strategy in relational databases, and thus easy to understand from this perspective. However, as the reason for the reversed foreign key is purely technical, the original semantics of the object model is not truly reflected in the relational structure anymore. The object model perspective therefore gives slightly different ratings in comparison to a relational point of view.

An additional factor which influences the performance of a system is the retrieval policy for associated objects. If a source object is retrieved from the database, the two extremes are to transitively retrieve *all* associated objects or *none* of them. Due to inter-object connections, the first policy might easily lead to the retrieval of the whole database with a single query. The latter approach is also suboptimal, as objects, which are frequently used by a source object, have to be retrieved on a one-by-one basis from the database. This access pattern may result in a heavy load on the database connection and poor performance. The solution to this problem is a developer-specified compromise between these extremes. In detail, object-relational middlewares usually provide *proxies* to be used instead of associated objects (Gamma et al., 1995). If such a proxy is accessed, the middleware transparently retrieves the corresponding object from the database. Less frequently used associations can be traversed on demand with this concept, which in turn helps to improve performance and to reduce database traffic.

2.2.3. Mapping of inheritance relationships

There are several strategies for the mapping of inheritance hierarchies to relational structures. The different possibilities are discussed in the following with the help of the example given in Fig. 7a.

One table for each class: From a semantical point of view, the most natural approach to map a class hierarchy to relational structures is to assign a single table to each class, which then contains a single field for each attribute (Fig. 7b). This strategy closely resembles the

Table 1
Summary of association mapping impacts

Association type	Mapping pattern	U	M	S	P
One-to-one	Own association table	+	+	+	−
One-to-one	Foreign key embedding	+	o	+	o
One-to-one	Merging into single table	−	−	+/o/−	+
One-to-many	Own association table	+	+	+	−
One-to-many	Reverse foreign key embedding	o	o	+	o
Many-to-many	Own association table	+	+	+	−

U = understandability, M = maintainability, S = storage space, P = performance.

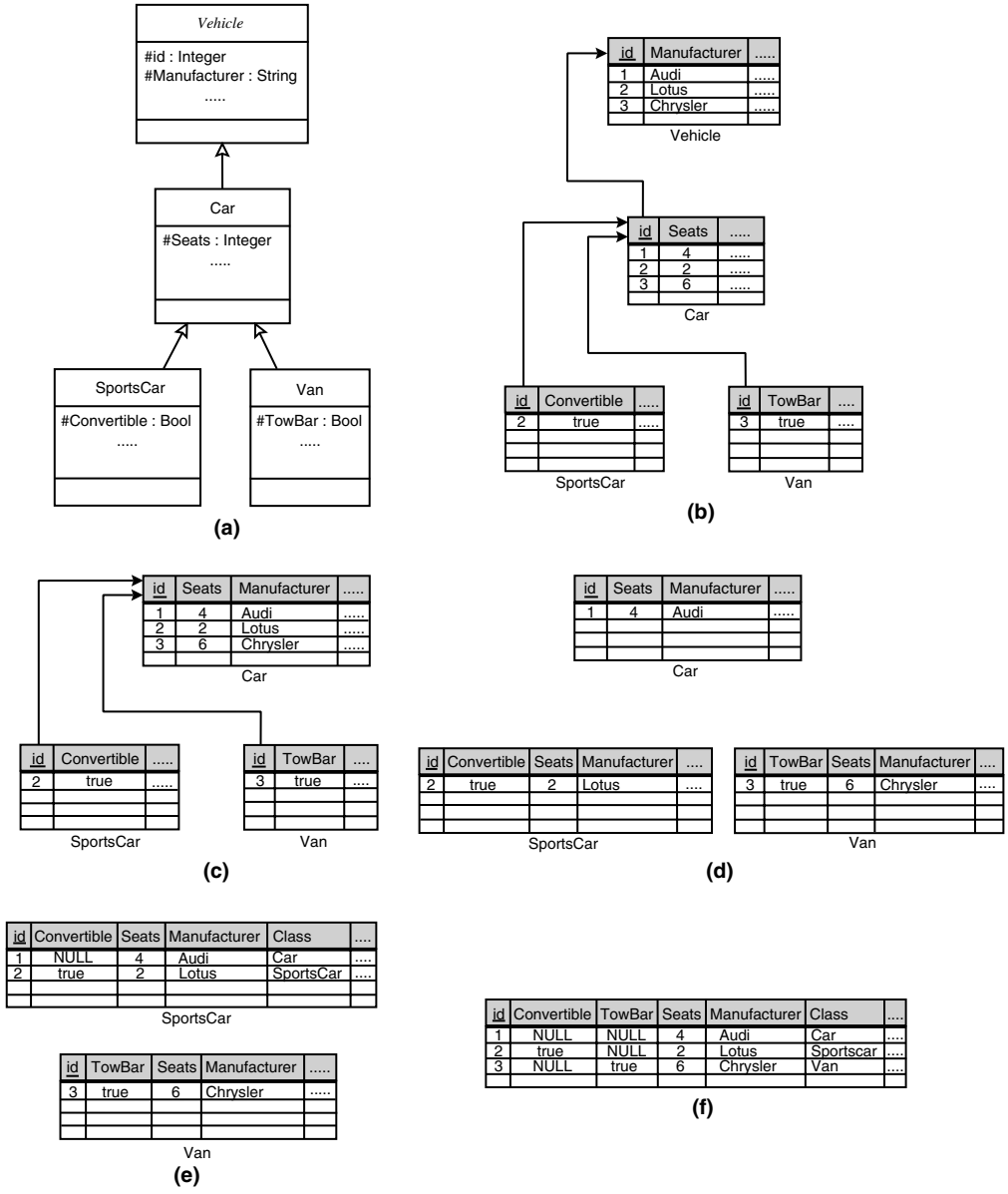

Fig. 7. Alternatives for inheritance hierarchy mappings: (a) initial class hierarchy, (b) one table for each class, (c) one table for each concrete class, (d) one flattened table for each concrete class, (e) one flattened table for each concrete leaf class and (f) one table for all classes.

object model of the application and is thus easy to understand. Maintainability is also driven to its best, as changes in a class only affect the corresponding table. However, as every table only contains attributes which are defined in the associated class, and not the ones inherited from its superclasses, objects are actually scattered over several rows in different tables of the hierarchy. In turn, the retrieval of a single object may require several joins which might lead to performance bottlenecks in inheritance hierarchies with a modest depth.

One table for each concrete class: The second choice for the mapping of a class hierarchy is to assign tables only to concrete classes. In this case each table contains the attributes of the corresponding class as well as the attributes which are inherited to this class from its abstract superclasses (Fig. 7c). In comparison to the mapping described before, the retrieval performance of this approach may be improved considerably, depending on the use of abstract classes in the application class hierarchy. Even if the table structures which result from this mapping strategy are usually not too difficult to understand, there is also a drawback. In contrast to the mapping pattern described before, the modification of an abstract class leads to adjustments of not only a single table, but of *all* tables which correspond to subclasses of the changed class.

One flattened table for each concrete class: The 'one table for each concrete class' approach may be driven further, if each of the tables created with this mapping contains *all* attributes inherited by the corresponding classes from its parents, no matter if they are abstract or not (Fig. 7d). The main advantage of this mapping strategy is that in contrast to the approaches described before, no joins between tables of an inheritance hierarchy are needed to retrieve individual objects. Also, the understandability of the resulting table structures is not negatively affected. On the downside, the modification of a persistence enabled class leads to the need to modify *all* subclass tables.

One flattened table for each concrete leaf class: Another mapping strategy is to only create tables for concrete leaf classes of the hierarchy. These tables contain the attributes of the leaf class as well as *all* attributes inherited to this class (Fig. 7e). Such a table then stores objects from the leaf as well as instances from its transitive superclasses. As the attributes of a branching class are included in all of its leaf classes, branching class instances are stored in one of the corresponding tables. If different types of objects are stored in the same table, an additional attribute is needed in order to indicate the class a stored object belongs to. The advantages of this approach are that no joins are needed for the retrieval of single objects and also the amount of tables is significantly reduced. However, a drawback is that the storage of non-leaf class objects in leaf class tables introduces `NULL` values for attributes, which are only defined for leaf classes. In addition to this storage space issue, understandability and maintainability are not driven to their best, as the application object model is not truly reflected in the table structures anymore.

One table for all classes: The last alternative is to map all classes of an inheritance hierarchy into a single relational table. This table then contains *all* attributes of *all* classes (Fig. 7f). Also, an additional attribute is needed which stores the actual type of an object stored in a row of the table. The only advantage of this approach is that no joins are needed for the retrieval of single objects. Disadvantages are that the resulting relational structure is difficult to understand and maintain for complex object models. Moreover, also a lot of storage space is wasted in the table, since `NULL` entries are needed for attributes of the table structure, which are not contained by stored objects.

Table 2 summarizes the impact of the inheritance mapping patterns to performance, understandability, maintainability and storage space requirements. Additional factors, like the depth of the inheritance hierarchy, the use of abstract classes in an object model as well as typical query patterns, also influence the consequences to these aspects, but are not considered here since their impact depends on particular object models and their use.

As can be seen from the multitude of alternatives for the mapping of objects to relations, many factors influence the decision for the 'right' mapping. The choice for the mapping of a class and its associations has a profound impact not only on the retrieval performance of objects, but also on the understandability and maintainability of a relational schema as well as its storage space requirements. In a classical application development process as sketched in Fig. 2, object-relational mappings have to be re-evaluated and partially redefined with each evolutionary step of the object model. Due to the inherent complexity of the manual mapping process, the constant re-evaluation and re-specification of object-relational mappings is highly error-prone. A test suite which checks for correct storage and retrieval of objects from persistence enabled classes is therefore inevitable. Unfortunately, the test suite also has to be adjusted accordingly after each evolutionary step of the application system. As the test cases for a specific class also have to take inherited attributes as well as (inherited) associations into account, the (partial) re-implementation of such a test suite is a non-trivial task on its own. To be able to offer an adequate CASE tool support for these problems, the next section introduces a pragmatic approach for the model driven generation of object-relational mappings on the basis of the above mapping evaluation.

3. Model driven generation of object-relational mappings

The primary goal of the approach to be presented in this section is to free developers of object-oriented application systems from the need to manually map objects

Table 2
Summary of inheritance mapping impacts

Inheritance mapping pattern	U	M	S	P
One table for each class	+	+	+	−
One table for each concrete class	+	o	+	o/−
One flattened table for each concrete class	+	−	+	+
One flattened table for each concrete leaf class	−	−	o/−	+
One table for all classes	−	−	−	+

U = understandability, M = maintainability, S = storage space, P = performance.

to relations. Ideally, the model driven generation of object-relational mappings enables developers to concentrate on their main interest, i.e. the application logic of a system. As to whether an approach in this area is accepted by developers heavily depends on the way it is integrated into the object-oriented software development process. Thus, the starting point for the automatic generation of object-relational mappings is an application object model, i.e. a UML class diagram. As a prerequisite for mapping generation, persistence enabled classes have to be identified first. This identification by default leads to the automatic mapping of all attributes of a class into relational structures. If this default handling is too coarse-grained, the developer may choose not to map particular attributes of persistence enabled classes. With this information provided, an object-relational mapping can already be automatically generated with a predefined default settings for a trade-off between performance on the one hand and understandability, maintainability and storage space requirements on the other. However, since the choice between the trade-off alternatives depends on domain specific requirements, the use of a single fixed pattern for automatic mapping generation is generally not advisable. In case the predefined default settings for the automatic mapping generation are not fulfilling all system requirements, developers therefore need to be able to specify which trade-off between the above aspects is appropriate in a particular stage of a project for the persistent parts of the object model or subsets thereof. Ideally, the specification of a particular trade-off for (parts of) a given object model is supported in a declarative way, in order to shield developers from the underlying mapping details and the resulting relational structures. Out of this, trade-off levels are defined in the following as a basis for declarative mapping specifications. With these pre-defined levels, developers simply have to choose the desired trade-off for the mapping of (parts of) an object model according to given system requirements.

The below definitions of the trade-off levels are based on the impacts of the described mapping patterns on understandability, maintainability, storage space requirements and performance as summarized in Tables 1 and 2. In detail, the inheritance hierarchy mapping patterns serve as starting point for the trade-off level specification range. However, from the five alternatives for inheritance mappings described above, only four are actually used. The 'one table for all classes' approach is excluded in the following, as it does not provide any benefits in comparison to the 'one flattened table for each concrete leaf class' strategy. Thus, four inheritance mapping patterns remain, which are combined with different variations of association mappings. The trade-off levels introduced in the following are constructed to provide developers with a variety of choices ranging from structures which are easy to understand and maintain on the one hand to performance optimized ones on the other.

(1) The first trade-off level focuses on high understandability and maintainability as well as low storage space requirements. No measures are taken for performance improvements. Due to the high understandability of the resulting relational schema and the mapping on top of it, this trade-off level should be the default setting for the automatic generation of object-relational mappings. The inheritance hierarchy of the persistence enabled parts of the object model is mapped at this level using the 'one table for each class' pattern. Associations are mapped to own tables, irrespectively of their particular cardinality types.
(2) Level two slightly improves the first one with respect to performance issues. The inheritance mapping pattern of choice is 'one table for each concrete class', i.e. attributes of abstract classes are merged into their descendants. As with level one, all association types are mapped to own tables. Maintainability is affected at this level, since changes of abstract classes in the object model lead to modifications of tables, which correspond to subclasses of the changed abstract class.
(3) The third trade-off level further optimizes the mapping with respect to performance. Whereas the inheritance hierarchy is still mapped with the 'one table for each concrete class' pattern, one-to-one associations now use foreign key embedding. Other association types are mapped to own tables. Apart from the positive effects on systems performance, maintainability is slightly affected at this level, as cardinality changes e.g. from one-to-one to one-to-many are not easily integrated into the schema anymore.
(4) Level four changes the inheritance hierarchy mapping to 'one flattened table for each concrete class'. All (inherited) attributes of a class are stored within a single table this way, which prevents the need for joins in order to retrieve objects. The association types are mapped as described in level three. The performance improvements achieved at this level negatively influence the maintainability of mapping and schema, as each change of a class in the object model leads to changes of all tables which correspond to its subclasses.
(5) The fifth level still uses the 'one flattened table for each concrete class' inheritance hierarchy mapping pattern. One-to-one associations are represented with embedded foreign keys. One-to-many associations are mapped using the reversed foreign key embedding pattern. In consequence, these measures do not only improve performance, but also negatively affect understandability and maintainability.

(6) The sixth and last trade-off level exclusively focuses on performance. In consequence understandability, maintainability and also storage space requirements are affected. The inheritance mapping pattern used at this level is 'one flattened table for each concrete leaf class'. One-to-one associations are merged together with their participating classes into a single table. One-to-many associations use the reverse foreign key mapping pattern, whereas many-to-many associations are still mapped to own tables.

This list of trade-off levels intentionally omits the combination of 'one flattened table for each concrete leaf class' with (reverse) foreign key embedding for one-to-one and one-to-many associations. This combination is not advantageous, as it does not provide performance benefits in comparison to trade-off level five, but negatively affects the other aspects. Moreover, combinations of mapping patterns which introduce redundancy into the database also have been excluded from further consideration. Table 3 summarizes the impacts of the remaining trade-off levels on the various aspects. As with the tables before, the perspective taken is that of an object-oriented application systems developer. The detailed impact of these trade-off levels to the above aspects in fact also depends on a given application object model and its properties. In addition, the qualitative aspects maintainability and understandability cannot be measured objectively. Therefore, the combinations of inheritance hierarchy and association mapping patterns, as well as their estimated impacts, reflect at least partially our own preferences and experiences. Out of this, these levels are intended as a starting point for further discussions, and at the same time as a basis for the implementation of a prototype.

From a developer's point of view the specification of an appropriate trade-off level for a whole project is not fine-grained enough to support performance optimizations only in specific parts of an application object model. It has to be possible, therefore, to refine a globally defined trade-off level for UML packages and classes. In this context, local specifications overrule global ones for the actual generation of object-relational mappings. Different levels of granularity are thus provided for the developer to identify parts of an object model, which are crucial for the overall performance of a system. In addition to the different trade-off specifications, the developer may also choose if objects, which are related to a source object, have to be retrieved together with this source object, or on demand only. Such specifications may be given at project, package and association level. They are used to generate mappings, which make use of proxies in the designated areas as described in Section 2.2.2.

After the developer has specified the persistence enabled parts of an application object model, an object-relational mapping can be automatically generated on the basis of assigned trade-off levels. As it is not our intention to develop yet another object-relational middleware as a target for the automatic mapping generation, the question arises what is actually generated? In order to simplify the software development process given in Fig. 2, the actual output of the model driven generation consists of:

Table 3
Summary of trade-off levels and their consequences for the automatic generation of OR-mappings

Level	U	M	S	P	Inheritance mapping	Association mapping
1	+	+	+	–	One tables for each class	$1:1 \rightarrow$ own table $1:n \rightarrow$ own table $m:n \rightarrow$ own table
2	+	+/o	+	o/–	One tables for each concrete class	$1:1 \rightarrow$ own table $1:n \rightarrow$ own table $m:n \rightarrow$ own table
3	+	o	+	o	One tables for each class	$1:1 \rightarrow$ key embedding $1:n \rightarrow$ own table $m:n \rightarrow$ own table
4	+	o/–	+	+/o	One flattened table for each concrete class	$1:1 \rightarrow$ key embedding $1:n \rightarrow$ own table $m:n \rightarrow$ own table
5	o	–	+	+	One flattened table for each concrete class	$1:1 \rightarrow$ key embedding $1:n \rightarrow$ reverse key embedding $m:n \rightarrow$ own table
6	––	––	–	++	One flattened table for each concrete leaf class	$1:1 \rightarrow$ merged table $1:n \rightarrow$ reverse key embedding $m:n \rightarrow$ own table

U = understandability, M = maintainability, S = storage space, P = performance.

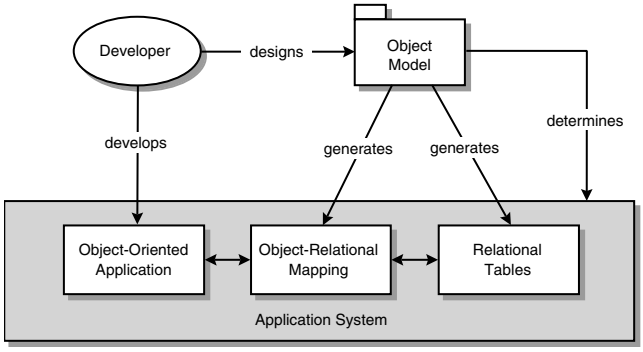

Fig. 8. Simplified application systems development process using model driven generation of OR-mappings.

(a) SQL DDL statements to construct relational structures and
(b) specifications of the object-relational mapping in the format of a particular OR-middleware.

Advantages of this approach are that it is independent from the implementation language of the application system, the object-relational middleware to use, and the underlying relational database. The impact of the model driven generation of object-relational mappings on the software development process as described in this section is given in Fig. 8. In contrast to the original process in Fig. 2, the developer is freed from the mapping specification and the creation of relational structures. Both is now automatically generated from the application object model on the basis of declarative trade-off level specifications. As a consequence, mapping details and the underlying relational structures are hidden from the developer by means of declarative specifications in the application object model. If we further assume that the component for the generation of OR-mappings and relational structures works correctly, a test suite to check for structural mapping failures is not needed any longer. Especially the incremental development of complex application systems benefits from the presented approach, as there is no need to manually develop OR-mappings and relational structures as well as test suites throughout the software development process. Consequently, the efforts needed for the development of object-oriented application systems based on RDBMs are lowered considerably. Furthermore, also developers not too familiar with RDBMs technology are enabled with the presented approach to build object-oriented application systems on top of RDBMSs.

4. Automated testing of object-relational mappings

As motivated above, a test suite to check for the correct storage and retrieval of objects in a relational database is not needed if mapping definitions and relational tables are automatically generated from an application object model. However, despite the advantages of the presented approach, the proposed concepts for the model driven generation of object-relational mappings may not be sufficient to fulfill *all* requirements for *every* application. In addition to the combinations of mapping patterns provided by the trade-off levels, classes may for instance also be represented in fewer tables. Moreover, a splitting of classes into several separate tables might be advantageous in certain circumstances. As these are highly application specific decisions, which cannot be automated, it has to be possible for the developer to manually refine a generated OR-mapping according to specific application needs. As the output of the generation process contains the object-relational mapping specification in the format of an existing OR-middleware, the developer is free, if necessary, to manually refine the generated mapping with the tools provided by the middleware vendor. At this point, the importance of understandability and maintainability of a generated OR-mapping and its underlying relational schema becomes obvious from an object-oriented application systems developer point of view. If mapping and schema are exclusively generated, these qualitative aspects are not as important as with the manual refinement of both. In the latter case the developer starts from the result of the last generation step, thus the non-functional requirements understandability and maintainability immediately influence the efforts needed to manually refine such a mapping.

Due to the encapsulation of relational databases by an OR-middleware, manual optimizations of generated mappings are transparent to the object-oriented application. However, the manual mapping of incrementally evolving complex object models to relational structures is inherently difficult. Therefore, the manual tuning of an object-relational mapping should be postponed as long as possible, or even better, completely avoided. In case manual modifications are inevitable, a test suite is needed in order to ensure the quality of an application system from a persistence point of view. Such a test suite automatically checks for the correct storage and retrieval of objects from persistence enabled application classes. The usefulness of test cases to check for the correct storage of application objects in a relational database stems from the separation of both through the use of an OR-middleware. As the test cases are based on the application object model, they are not affected by modifications of the object-relational mapping and the relational structures (see Fig. 1). If the developer chooses to manually refine the automatically generated OR-mapping, an object model based test suite which checks for structural conformance is therefore well suited to reveal mapping failures. The actual importance of such a test suite in case of a manual refinement of

OR-mappings for complex object models cannot be overestimated, as even subtle modification of a given mapping might easily lead to persistence failures. As the manual development of suitable test cases for object-relational mappings in an iteratively evolving environment also takes considerable efforts, concepts for their automatic generation from application object models are introduced in the following.

The main concept of test suite generation is the structural one-to-one mirroring of persistence enabled parts of an application object model. In consequence, persistence enabled classes, as well as their transient superclasses, have a counterpart in the test suite, which separates the application logic from the persistence test cases. An additional class is needed in the test suite, which triggers the start of all tests and protocols the results. Fig. 9 gives an example which illustrates this structure. Here, the application object model classes 'Vehicle', 'Car' and 'Driver' (Fig. 9a) are mirrored, and also a class 'TestCaseDriver' has been added to control and protocol test runs (Fig. 9b).

In order to test the persistence enabled parts of the application model with the described structure, each class of the test suite contains a single test method. This method checks, if all attributes of the corresponding class in the object model, as well as its associations, are correctly stored and retrieved from the database by the OR-middleware via the given mapping specification. To be able to do so, an instance of the particular object model class is instantiated in its corresponding test class first. The attributes and associations of this object are then initialized with test values. If the class to test is a descendant of a superclass, it has to be checked as to whether the inherited features are correctly stored and retrieved, too. If the values finally retrieved from the database correspond to the ones the object was initialized with, the mapping does not suffer from structural deficiencies.

To be able to benefit from the introduced simplifications of the development process as given in Fig. 8 and the generation of test suites in case of manual refinements, the underlying automation concepts for the generation of OR-mappings and test suites have to be supported by an integrated development environment. Therefore, the next section sketches the integration of the described concepts into a prototypical CASE tool.

5. Prototypical CASE-tool integration

The above described automation concepts for the generation of object-relational mappings and their test cases are integrated into NEPTUN (Philippi, 2002), a modular CASE tool platform for the development of object-oriented systems. Amongst other features, NEPTUN provides capabilities for multiperspective systems modeling. As UML class diagrams are used for architectural specifications, this framework is well suited as a basis for the prototypical integration of the introduced automation concepts.

Starting point of the simplified development processes for OR-middleware based systems (Fig. 8) is an application object model. Fig. 10 gives an example object model as UML class diagram in NEPTUN. Furthermore, this figure contains the project inspector with the global setting for the trade-off level, which enables the developer to control the OR-mapping generation with respect to the needs of his project. As described above, he may choose between six different levels and also he may specify a global setting which defines if proxies have to be generated for association mappings as described in Section 2.2.2. In the next development stages the classes to be persistence enabled have to be identified. If necessary, package and/or class-based mapping strategies may be specified to overrule global settings. In order to be able do so, the tool provides inspectors similar

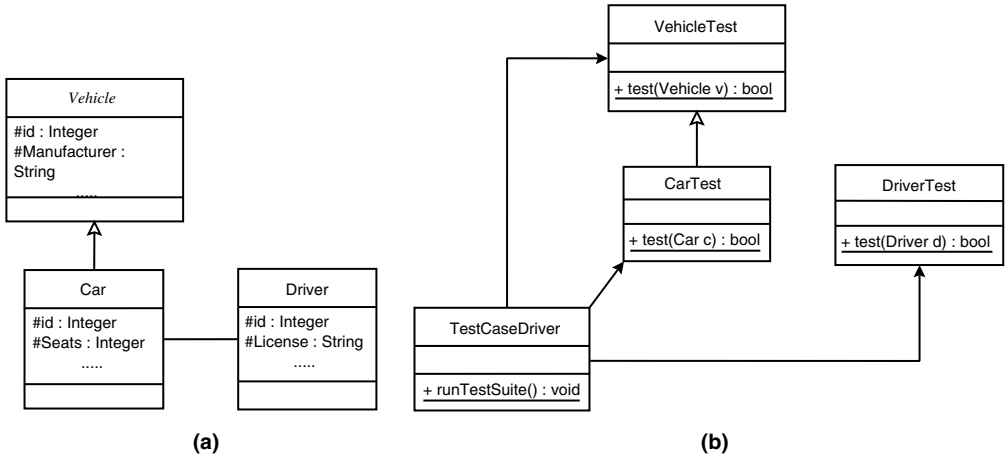

Fig. 9. Test suite generation concept (static view): (a) initial class hierarchy and (b) test suite class hierarchy.

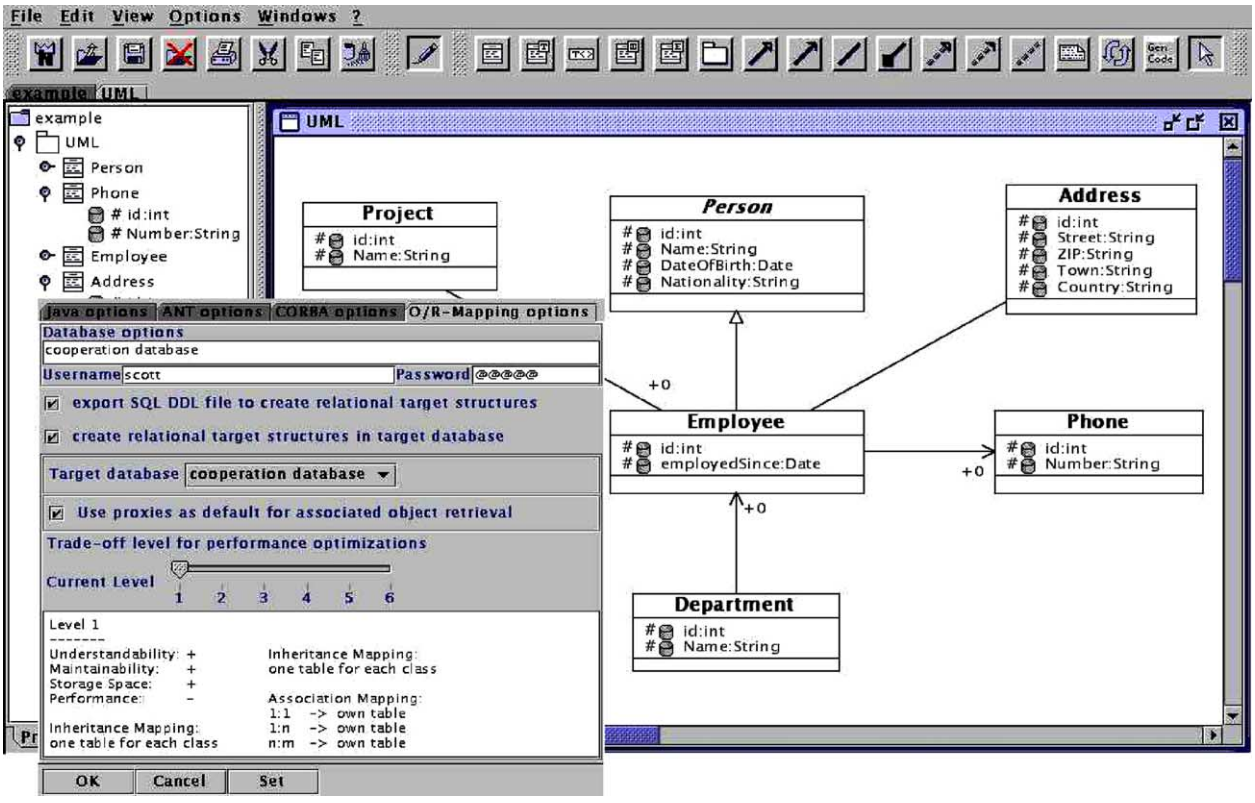

Fig. 10. Screenshot from the NEPTUN CASE tool with an example object model

to the one in Fig. 10. The class inspector in addition allows for the (de-)selection of attributes which do not have to be mapped to relational structures, i.e. which are not to be stored persistently. Within the object model of the example all attributes are marked as persistent, which is indicated by an additional symbol resembling a data store.

If the developer decides to generate the persistence related parts of the application object model, the first result is a SQL DDL script for the creation of relational structures. This script may either be directly sent to the database, or exported to the file system. Fig. 11 gives the visualization of the generated relational structure for the example. The default global setting to control the generation is trade-off level one, which optimizes for understandability, maintainability, and storage space requirements. This global setting is refined for classes 'Department' and 'Phone' with trade-off level five and for class 'Address' with level six. Consequently, there is a conflict for the mapping of e.g. the association between 'Employee' and 'Address'. Whereas the trade-off level assigned to 'Employee' demands for the mapping of the association to a separate table, the level of 'Address' requires the association to be mapped as a merged table. For the automatic mapping generation, a higher trade-off level also has a higher priority. The rationale behind this is the fact that we assume the default setting to be a lower trade-off level, which is locally refined for performance optimizations. In choosing higher trade-off levels for conflict resolution, the developer is freed from the need to multiply refine local trade-off levels for both classes being part of an association.

The second result of the generation is the mapping specification in the format of the chosen OR-middleware. Even if the introduced concepts for the automatic

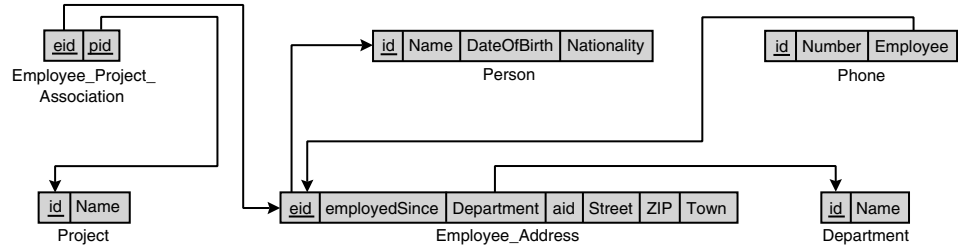

Fig. 11. Visualization of the generated structure for the example object model.

generation of OR-mappings are independent from a particular middleware, their actual generation is not. As there is no standardized mapping description format, a port has to be provided by the tool for each middleware to support. At present, our prototypical implementation supports 'TopLink' (Object People, 2002) and 'Object/Relational Bridge' (OJB) (OJB, 2002), as prominent commercial and open source OR-middlewares.

As the introduced automation concepts might not fulfill *all* requirements of *every* application system, the developer may decide at some point to manually refine the generated mapping. In case the chosen object-relational middleware provides a mapping tool, the generated mapping specification may be imported into this tool for manual refinements. Thus, the developer does not have to start from scratch, but benefits from the mapping which has been generated according to his preferences. Due to the flexibility of the introduced mapping generation approach and its multiple levels of developer control, only minor mapping changes in small parts of the application object model are ideally needed to customize a generated mapping to specific application needs, if at all. Since the manual refinement of object-relational mappings for complex application models is very demanding and thus prone to error, the prototypical implementation of the automation concepts in NEPTUN also includes the generation of test suites along the lines of the pattern described above.

6. Conclusions and further perspectives

In order to bring object-oriented applications and relational databases together, the so-called object-relational impedance-mismatch has to be overcome. A common approach to bridge the gap between the object-oriented and the relational paradigm is to use an object-relational middleware. Such a middleware is configured with a mapping specification which defines how application objects are stored in a relational database. This way, both parts of an application system are largely decoupled from each other. Changes in one part of a system do not necessarily lead to changes in the other, often only the mapping specification needs to be adjusted. However, with existing approaches to object-relational mapping, such a specification needs either to be defined manually or is generated according to a fixed translation pattern. While the manual specification of mappings for incrementally evolving non-trivial object models takes considerable efforts and is highly error-prone, the automated mapping with fixed translation patterns is unsuitable, since there is no single best mapping pattern for *all* applications.

From this perspective, the approach presented in this paper goes beyond existing solutions, since the introduced automation concepts for the model driven mapping of objects to relations specifically take project related requirements into account. The concepts itself are independent from the underlying relational database, the object-relational middleware and the (object-oriented) implementation language of a project. The mapping principles are transparently encapsulated by the introduced trade-off levels, which allow to control the mapping generation at multiple levels of object model detail along the lines of given system requirements. Ideally, the developer is thus freed from the manual development of object-relational mappings. However, in case the automation concepts are not sufficient to fulfill all requirements of a project, the outcome of the generation may be further refined manually. In order to guarantee that the manual refinement of a generated mapping does not affect the quality of a system from a persistence point of view, a test suite is automatically generated from the application object model. This test suite checks if objects from persistence enabled classes of the application object model are correctly stored and retrieved from the relational database via the OR-middleware and the given mapping specification. As the mapping and the underlying relational structures can be modified without having to alter the application object model, structural mapping failures can be automatically detected this way. The overall impact of the concepts introduced in this article on application systems development is summarized in Fig. 8 with the simplified development process. In comparison to the original process in Fig. 2, most of the tasks are automated with the presented approach. Consequently, the efforts needed to develop object-oriented application systems on top of RDBMs are lowered considerably. As a proof of concept the model driven automatic generation and testing of object-relational mappings has been prototypically integrated into an experimental CASE tool platform.

Future work on the presented approach includes customizable trade-off levels, which will provide the opportunity for user-specified combinations of inheritance and association mapping patterns. In order to further optimize generated mappings and underlying relational structures with respect to understandability and maintainability, future versions will eventually support SQL3 features, e.g. to map inheritance hierarchies and complex attribute types. However, as a prerequisite for developments into this direction, database and OR-middleware vendors have to support these features more broadly, first.

Acknowledgments

The author would like to thank Jacob Köhler and Dion Whitehead for commenting an earlier version of this article as well as Christian Schmitt for working on the NEPTUN prototype.

References

Agarwal, S., Keene, C., Keller, A.M., 1995. Architecting object applications for high performance with relational databases. Technical Report, Persistence Software. Available from <http://www.persistence.com>.

Ambler, S., 2000. Mapping object to relational databases. Technical Report, Ambysoft. Available from <http://www.ambysoft.com>.

Blaha, M.R., Premerlani, W.J., Rumbaugh, J.E., 1988. Relational database design using an object-oriented methodology. Communications of the ACM 31 (4).

Brown, K., Whitenack, B.G., 1995. Crossing chasms—a pattern language for Object-RDBMS integration. Technical Report, Knowledge Systems Corporation. Available from <www.ksc.com>.

Cattell, R., Barry, D., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., Velez, F., 1999. The Object Database Standard—Release3.0. Morgan Kaufmann Publishers.

Chen, P., 1976. The entity-relationship model: toward a unified view of data. ACM Transactions on Databases 1 (1).

Cunningham & Cunningham Inc., 2004. Object Relational Tool Comparison. 'c2.com/cgi-bin/wiki?ObjectRelationalTool Comparison'.

Darwen, H., Date, C.J., 1993. A Guide to the SQL Standard: A User's Guide to the Standard Relational Language SQL. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns. Addison Wesley, Reading, MA.

Keller, W., 1997. Mapping objects to tables—a pattern language. Technical Report, Objectarchitects. Available from <http://www.objectarchitects.de>.

Keller, A.M., Jensen, R., Keene, C., 1993. Persistence software: bridging object-oriented programming and relational databases. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data.

Kwan, I., Li, Q., 1999. A hybrid approach to convert relational schema to object-oriented schema. Information Sciences, 117.

Melton, J., Simon, A.R., Gray, J., 2001. SQL 1999—Understanding Relational Language Components. Morgan Kaufmann Publishers.

Object People, 2002. TopLink Documentation. Available from <http://www.objectpeople.com>.

OJB, 2002. Object/Relational Bridge Documentation. Available from <jakarta.apache.org/ojb>.

Object Management Group, 2004. UML 2.0. Available from <http://www.omg.org/uml>.

Persistence Software, 2002. EdgeXtend Documentation. Available from <http://www.persistence.com>.

Philippi, S., 2002. A CASE-tool for the development of concurrent object-oriented systems based on Petri-Nets. Petri-Net Newsletter 62.

Rahayu, J.W., Chang, E., Dillon, T.S., Tanier, D., 2000. A methodology for transforming inheritance relationships in an object-oriented conceptual model to relational tables. Information and Software Technology 42.

Ramanathan, S., Hodges, J., 1997. Extraction of object-oriented structures from existing relational databases. ACM SIGMOD Record 26 (1).

Secant Technologies, 2001. Object Integrator Documentation. Available from <http://www.secant.com>.

Smith, K.E., Zdonik, S.B., 1987. Intermedia: a case study of the differences between relational and object-oriented database systems. In: Proceedings of the OOPSLA'87 Conference on Object-oriented Programming Systems, Languages and Applications.

Tesch, T., Volz, M., 1999. A lightweight object manager for group-aware applications. GMD Report 47. Available from <http://www.darmstadt.gmd.de/oasys>.

Thought Inc., 2002. Cocobase Documentation. Available from <http://www.thoughtinc.com>.