Evolution of the application and database with aspects

Rui Humberto R.Pereira[1] and J.Baltasar García Perez-Schofield[2]
[1]*ISCAP, IPP, Porto, Portugal*
[2]*Departamento de Informática, Universidad de Vigo, Vigo, Spain*
[1] *rhp@iscap.ipp.pt,* [2] *jbgarcia@uvigo.es*

Keywords: Schema evolution, instance adaptation, aspect-oriented programming, orthogonal persistent systems

Abstract: Generally, the evolution process of applications has impact on their underlining data models, thus becoming a time-consuming problem for programmers and database administrators. In this paper we address this problem within an aspect-oriented approach, which is based on a meta-model for orthogonal persistent programming systems. Applying reflection techniques, our meta-model aims to be simpler than its competitors. Furthermore, it enables database multi-version schemas. We also discuss two case studies in order to demonstrate the advantages of our approach.

1 Introduction

Applications are subject of a continuous evolution process due to many factors such as changing requirements, new functionalities or even the correction of design mistakes. Those applications refactorings may have strong implications on the underlying data model, requiring the evolution of the application and the database, thus, becoming a very complex and time-consuming problem for programmers and database administrators.

Object-oriented databases such as db4o[1] (Paterson et al., 2006), Versant[2], ObjectDB[3], provide transparent schema evolution mechanisms that alleviate programmers in this burden of keeping the data and application layer synchronized. However, in these systems, more complex schema updates such as class movements in the inheritance hierarchy, field renaming and semantic changes in the fields content, require programmer's intervention in order to convert the data from the old structure to the new one.

This evolution problem, of object-oriented applications and databases, is addressed in this paper. We present an aspect-oriented approach, which is focused on the orthogonal persistence paradigm (Atkinson and Morrison, 1995). We found differentiating characteristics in the orthogonal persistent systems, when compared to non-orthogonal ones, that enable our aspect-oriented approach. These characteristics, combined with Aspect-Oriented Programming (AOP) (Kiczales et al., 1997), provide special conditions in order to improve the transparency of the entire evolution.

Our approach is based on a simple meta-model and the reflective capabilities of the programming language. This meta-model also enables the application's data representation in multiple schema versions. Each new schema version is incremental to the previous one, *i.e.*, just new class versions are added to the database.

In the next section we discuss our approach. Section 3 presents a short overview of the AOF4OOP framework, a prototype that implements our meta-model. In Section 4 we discuss a related work that inspired our research. Next, we present two case studies that intend to prove the advantages of our approach, when compared with other systems. Finally, we draw some conclusions and indicate future directions for new research work.

2 Meta-model

The proposed meta-model was designed in order to support database schemas in a multi-version approach, enabling bidirectional application compatibility. Applications, developed to a specific database schema, can access data transparently in another schema version, older or future. The chosen approach to represent multi-version schemas is based on class versioning (Monk and Sommerville, 1993). Issues

[1]http://www.db4o.com
[2]http://www.versant.com
[3]http://www.objectdb.com

such as data consistency, integrity and object identity also were taken into account in our meta-model proposal. The meta-model also aims to be as simple as possible. In order to achieve such goals, a set of meta-classes were defined, as well as reflective techniques.

2.1 Meta-classes

Our meta-model was inspired in Rashid's meta-model that we will present in Section 4. However, it defines a much more limited set of meta-classes. The meta-objects are instances of meta-classes These meta-objects represent the application's classes, its relationships and all data required for emulating objects in every existing schema version. In Section 2.2 we'll discuss how we have reduced the model's complexity by applying the reflection capabilities of the programming language. The set of meta-classes are the following:

- Class Version Meta-Object (CVMO) - This meta-object supports the schema information of an application data class in a specific version.
- Instance Meta-Object (IMO) - Logical representation of an instance of an entity data object. Each IMO meta-object is associated to one or more CVMO meta-objects and to one or more data objects. The set of CVMO meta-objects provides the framework with the historical information of its class, while the set of related data objects provides input information for the emulation process in each of the class versions.
- Attribute Meta-Object (AMO) - Represents the relationships among the data objects. Since composite objects are related through their attributes, this meta-object represents an instance variable that points to another object instance (an IMO meta-object). This meta-object has two distinct forms: as a single object reference or as a reference to an array of objects.
- Root Meta-Object (RMO) - Are database starting points, which give access to all other related objects. Each represents a *root object* (Atkinson and Morrison, 1995), identified by an arbitrary string. Like AMO meta-objects, these also have the same two forms for objects and arrays.
- Update/Backdate Meta-Objects (UBMO) - Each UBMO meta-object defines an explicit relationship between two versions of a class. These meta-objects, together with the default instance adaptation behaviour, define the database instance adaptation aspect. In order to define this database aspect, the programmer applies pointcut/advice constructs which are stored in these meta-objects. In Section 3 we will discuss one example of these expressions.
- Aspect meta-objects (AspMO) - To make the *aspect* code replacement more agile, this type of meta-object represents an *aspect* (in terms of AOP) which can have one of two usages: application *aspects* (e.g. logging) or database management system *aspects* (e.g. security).

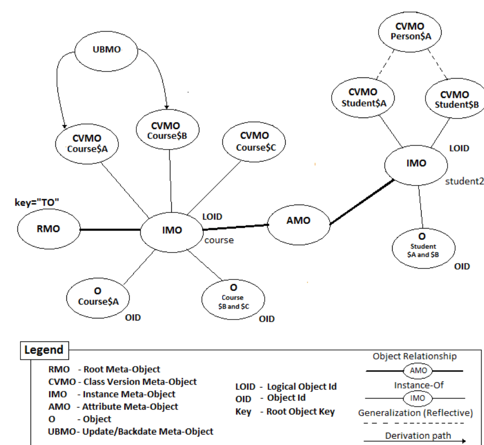

Figure 1: System meta-model

Apart from these meta-objects, there are also the real data objects in the database. Each data object (application's data entities) is under the class version it was created in. Furthermore, it is represented through an IMO meta-object that supports its logical existence, as well as other meta-objects that specify its class versions. Any implementation of this meta-model must emulate the object version as required by the running application. If the emulation to a specific application version is not viable, a copy of the object for that version will have to be maintained. Figure 1 illustrates an example of how these meta-objects are related in order to represent a course related with its students.

2.2 Versioning approach

As our meta-model aims to support the application's schema in many of its versions by following a class versioning strategy, the derivation path of class versions, combined with the classes' inheritance hierarchy, raises the meta-model's complexity to an unsupportable level if many versions exist. Thus, in order to deal with this complexity problem, we extended the meta-model with reflection and mapping techniques. The classes' inheritance hierarchy data is supported through reflection techniques. Regarding the classes' version derivation path, it is handled with an algorithm of mapping, which is based on direct and user-

defined mapping. In order to enable the coexistence of many versions of a same class, a renaming strategy was adopted.

Inside the CVMO meta-object, the class's bytecode is preserved and available for reflection. However, in the class's bytecode, its name and all its superclasses are renamed following a convention. In our implementation, a class `Person` in version `A` is renamed to `Person$A`. This class versioning has two approaches: (1) as a simple object (an object of a primitive type, such as an `Integer` or a `String`), or (2) as a composite object (`Person` or `Invoice`). Composite objects can be composed by simple objects and other composite objects. Thus, simple objects are treated in a non-versioned approach, while the composite objects follow the renaming technique. Using reflection, this renaming technique enables our framework to know, at runtime, the class inheritance relationships.

Regarding the classes' version derivation path, we consider that in many cases an algorithm, based on the direct mapping of classes and attributes, can establish an implicit *n-to-n* connection among all versions of a class. In this direct mapping, the mapping of class version and its attributes is based in their names and equivalence type, by applying default conversion functions. Just when the necessary conditions are not satisfied, is a user-defined connection then required. In these cases, our meta-model enables explicit connections through UBMO meta-objects. These meta-objects contain all mapping information, which follows an aspect-oriented approach. Each UBMO meta-object stores a pointcut/advice expression that quantifies all points in the application code that requires instance adaptation code.

2.3 Object identity

At the meta-model, each IMO meta-object is a logical representation of an instance of an application data entity. As an Object Identifier (OID) identifies an object, a Logical Object Identifier (LOID) identifies an IMO meta-object. Thus, a LOID identifies an application data entity that is supported in the database by one or more objects, each one with its own physical OID. In turn, the OID physically identifies an object or meta-object. This approach simplifies the object update process because it avoids the physical replacement of relationships and it also enables the existence of several versions of the same object. On the other hand, it enables the internal mapping and caching of the application's data objects, which provides means to ensure that two references of the same object share the same identity.

Concerning arrays, these are treated as a collection of individual object references as occurs in the running environment, according to the same programming language variance rules.

2.4 Schema consistency

Applications with a specific schema version should be able of use objects, which reside in the database, irrespectively of their class versions. These requirements raise structural , semantical and behavioural consistency issues that must be addressed.

Our approach, in order to deal with these issues, benefits from the orthogonal persistence paradigm: (1) closed integration of application and database environments, (2) sharing the same data model and (3) incremental schema evolution. The static type checking done by the compiler, at compile time, grants that each object instance inside the application runtime environment unconditionally pertains to, and follows, a specific application schema. The incremental schema evolution approach enables the propagation of all new class versions to the database, making them persistent. Likewise, all objects are created under only one valid schema, ensuring structural consistency.

The semantic consistency is granted through update/backdate conversion code and additional metadata added into the application schema. Our approach, based on UBMO meta-objects (see Sections 2.1 and 2.2) was inspired in the *update/backdate methods* (Monk and Sommerville, 1993). The user-defined conversion code, inside these meta-objects, can adapt a class that suffered a semantic update. As an example, consider a class `Product` where, in version `A`, the product's weight is defined as pounds and, in version `B`, kilograms is used.

Behavioural consistency must be analysed in two contexts: (1) the application's run-time and (2) the framework's conversion-time. Considering that all objects in the application environment pertain to its schema, naturally, behavioural consistency is always granted. However, at conversion-time this type of consistency can raise complex issues since two versions of a class coexist in the framework's environment. For non-versionable data types, the problem is not relevant. On the other hand, for versionable data types, version conflicts must be avoided. Let $C(\alpha)$ and $C(\beta)$ be, respectively, class C in versions α and β, being α the current class version. If C has a property P, then $C(\beta)$ has $P(\gamma)$, being γ the property's class version. Thus, an object of class C that is being converted is renamed to $C(\beta)$ to avoid name conflicts, as well as all its return and argument values, of its methods, and

attributes to P(γ). Within this approach, we guaranty that all class properties are synchronized with its version state. Thus, each method respects its signature and its code does not result in run-time errors or unexpected results.

3 Implementation of Meta-model

In this section we present a short overview of the implementation of our meta-model (Pereira and Perez-Schofield, 2010)(Pereira and Perez-Schofield, 2011)(Pereira and Perez-Schofield, 2012). This prototype is an aspect-oriented Java framework capable of providing an application with persistence and database evolution services. In the current version of the implementation, a db4o database (Paterson et al., 2006) is used as the object and meta-object repository. Although this object-oriented database already provides object persistence and also has some transparent database evolution capabilities, its role is reduced to a simple object store.

Our framework is a persistent programming environment that follows the three principles of orthogonal persistence formulated by Atkinson (Atkinson and Morrison, 1995). Furthermore, it applies AOP techniques in its internal operations, as well as providing applications with persistence and evolution services. The system's aspects were modularized at two distinct levels: application and framework/database. Thus, aspects of an aspect were modularized in another level. The system's default instance adaptation aspect can be extended through our pointcut/advice constructs, which follow the (Filman and Friedman, 2000): *Quantification* definition posited by Filman and Friedman: *"In programs P, whenever condition C arises, perform action A"*. Thus, each expression is structured as two groups: trigger conditions and action. At run-time, when the required conditions are satisfied, user-defined code is weaved and injected into the system, establishing an explicit mapping between two distinct class versions.

Figure 2 an example of our XML based expressions that reify the instance adaptation aspect in the database. Each `<ubmo>` XML node specifies an UMBO meta-object. The XML language provides extensibility and enables easier editing, through a graphical tool of the framework or using a simple text editor. Due to space restrictions were we cannot describe the entire syntax of those constructs. However, this example will be reused in the next sections, helping its understanding.

The set of pointcut/advice constructs form the instance adaptation aspect, which is rendered at run-

```
<ubmo matchClassName="rhp.aof4oop.cs.datamodel.*"
      matchCurrentClassVersion="B"
      matchOldClassVersion="A"
      matchSuperClassName="rhp.aof4oop.cs.datamodel.Staff">
    <conversion applyDefault="true"
          outputClassName="rhp.aof4oop.cs.datamodel.Staff"
          conversionClassName="ConvStaff$A_to_StaffB">
        String[] parts=oldObj.getSurname().split(" ");
        String middleName="";
        for(int i=0;i<parts.length-1;i++) {
            middleName+=(i>0?" ":"")+parts[i];
        }
        newObj.setMiddleName(middleName);
        newObj.setLastName(parts[parts.length-1]);
        return newObj;
    </conversion>
</ubmo>
```

Figure 2: Pointcut/advice XML expression

time by the framework's weaver. In the database, several persistent versions of a class can exist. The framework's classloader makes those class versions available at conversion-time. The classes pertaining to current schema version are defined in the application's source code.

4 Related work

Rashid, in the SADES system (Rashid, 1998), introduced the concept of *aspect* in object-oriented databases turning the system, in itself, into an aspect-oriented database. In the SADES system and later, in the AspOEv system (Rashid and Leidenfrost, 2004), the aspect-oriented programming techniques were explored in order to implement database mechanisms of schema evolution and instance adaptation. The AspOEv system has its own language, the Vejal (Rashid and Leidenfrost, 2006), capable of manipulating the objects in their multiple class versions. This research work was inspiring to us, being the starting point for our work.

Kuppuswami *e*t al. (Kusspuswami et al., 2007) have also explored AOP techniques proposing a flexible instance adaptation approach.

5 Case studies and discussion

5.1 Meta-model evaluation

An earlier case study (Rashid, 2002) provides a comparative evaluation of four different systems based on a design correction scenario. This scenario consists of a data structure of seven classes that are redesigned,

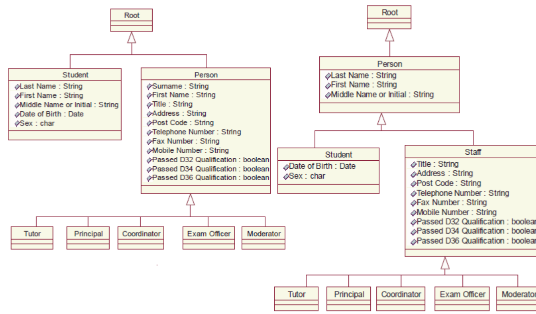

Figure 3: Schema evolution (extracted from (Rashid, 2002))

Table 1: Meta-model complexity comparation.

Studied system	Affected meta-objects
Orion	33
Encore	54
TSE	56
SADES	23
AOF4OOP	10

leading to the appearance of a new class `Staff`. Figure 3 presents that same structure before and after the evolution.

We reused this scenario in order to extend the previous work with our results, enabling a direct and easier comparison. This comparative case study was structured into two parts: schema evolution complexity and instance adaption The former analyzes the impact on meta-objects relationships, at the schema level, while the latter provides an analysis of the instance adaptation process. Here we just discuss our approach. A detailed comparison of those four systems can be found in that earlier case study.

Regarding the complexity in schema evolution, we argue that the incremental approach enabled by the orthogonal persistence paradigm and our meta-model reduce the complexity problem leading to better results, as shown in Table 1. This table combines the results of that work with ours, presenting the respective number of affected meta-objects for each compared system.

In our approach, as schema updates are incremental, eight new CVMO meta-objects are inserted: seven new class versions for the existing classes and the new `Staff` class. Additionally, due to update/backdate compatibility, two UBMO meta-objects provide the conversion means in both directions: `Person` class `Surname` attribute in the former version and the new one with a distinct attribute called `LastName`. The XML expression in Figure 2 presents one of these two user definitions. All remaining changes are transparently treated by the default instance adaptation mechanism, avoiding any extra information. Thus, our approach enables a semi-transparent schema evolution.

This design correction scenario, while being very simple, allows for a good understanding of the advantages present in our approach: (1) smaller overhead in terms of effected meta-objects and (2) avoids human intervention at the database, providing schema evolution primitives.

Regarding instance adaptation, SADES presents the best results when compared with the other three systems. Since our aspect-oriented approach is inspired in the author's work, their results and ours are equivalent in terms of flexibility.

5.2 Programmers' productivity

In this section we will present another comparative case study which intended to assess the framework's added value of its underlying object repository, a db4o object database. The same redesign scenario of previous section was used. For that, we invited two distinct programmers to adapt two simple applications, which use that data model before redesign. The former application stores data directly into a db4o database, while the latter uses our framework to manage its persistence. Since both applications share a common class structure, this code was implemented into a common library. Furthermore, both base applications must be adapted to support the occurred changes in class `Person`. Given that both tasks are exactly the same in both applications, we measure just one time and consider the same value for both. This strategy guarantees a fare measure, avoiding time variations. Notice that the second application would benefit from the programmer's knowledge acquired during the first application, solving the exact same problem.

After that, both programmers were concerned with the db4o application. Their first challenge lies in a db4o database limitation, which does not support the following two schema changes transparently: (1) inserting classes into an inheritance hierarchy; and (2) removing classes from an inheritance hierarchy. Both class transformation types occur in our study, so an additional application is required to adapt object instances (`Student` and all `Staff` subclasses) from an old class version to the new one. The authors of the db4o database suggested an eager stop-the-world model application, which sequentially scans the entire database, reads, converts and stores data in objects that pertain to a different class name. In the last phase, the conversion application uses a db4o API to rename all new classes to their original name.

In contrast to the db4o application case study, the one based on our prototype does not require any ad-

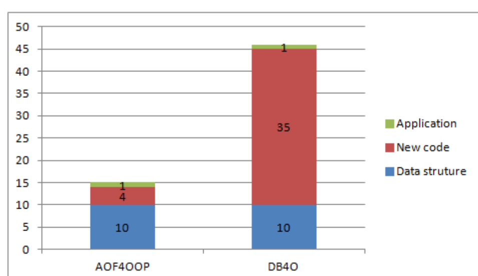

Figure 4: Effort comparison (work minutes)

ditional conversion application, saving programmers time. Only the semantic consistency cannot be transparently dealt with in the framework, requiring the programmer's intervention in order to write two XML constructs. One of them is presented in Figure 2.

Due to performance reasons, this aspect-oriented construct is applied just to `Staff` subclasses. Notice that, `Student` objects can be adapted by default mechanism with less overhead. On the other hand, it is applied once in the `Staff` class, avoiding individual definitions for its subclasses. Thus, just one meta-object is required with the best possible performance.

The graph in Figure 4 illustrates the framework's benefits for the programmer in terms of work productivity in this specific scenario.

6 Conclusions

We presented a novel aspect-oriented approach based on a meta-model in order to provide applications with orthogonal persistence in a multi-schema database. Enabled through orthogonal persistence and our XML expressions, the database evolution among its schema versions is semi-transparent. Thus, schema evolution is performed directly into the application's source code.

Regarding instance adaptation, it provides the means for flexible and agile support that just requires human intervention when semantic updates occur in the application schema.

Our experimental results, supporting the evolution, have demonstrated the advantages of our approach over a reference system in the state-of-the-art of object-oriented databases.

At current state of our framework the transactional model is very simple. Object's properties are updated in individual transactions. Due to JVM restrictions some classes cannot be renamed limiting its persistence and the framework's orthogonality. The overall framework's performance can be improved with a specific engine of object storing. Our research will continue focused on these implementation issues.

REFERENCES

Atkinson, M. and Morrison, R. (1995). Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–402.

Filman, R. E. and Friedman, D. P. (2000). Aspect-oriented programming is quantification and obliviousness. Technical report.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Aksit, M. and Matsuoka, S., editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg.

Kusspuswami, S., Palanivel, K., and Amouda, V. (2007). Applying aspect-oriented approach for instance adaptation for object-oriented databases. In *Proceedings of the 15th International Conference on Advanced Computing and Communications*, pages 35–40, Washington, DC, USA. IEEE Computer Society.

Monk, S. and Sommerville, I. (1993). Schema evolution in oodbs using class versioning. *SIGMOD Rec.*, 22:16–22.

Paterson, J., Edlich, S., Hrning, H., and Hrning, R. (2006). *The Definitive Guide to db4o*. Apress, Berkely, CA, USA.

Pereira, R. H. and Perez-Schofield, J. (2010). An aspect-oriented framework for orthogonal persistence. In *Information Systems and Technologies (CISTI), 2010 5th Iberian Conference*, pages 1 –6.

Pereira, R. H. and Perez-Schofield, J. B. G. (2011). Orthogonal persistence in java supported by aspect-oriented programming and reflection. In *Information Systems and Technologies (CISTI), 2011 6th Iberian Conference*, pages 1 –6.

Pereira, R. H. and Perez-Schofield, J. B. G. (2012). Database evolution on an orthogonal persistent programming system - a semi-transparent approach. In *Information Systems and Technologies (CISTI), 2012 7th Iberian Conference*, pages 1 –6.

Rashid, A. (1998). Sades - a semi-autonomous database evolution system. In *Workshop ion on Object-Oriented Technology*, ECOOP '98, pages 24–25, London, UK. Springer-Verlag.

Rashid, A. (2002). Aspect-oriented schema evolution in object databases: A comparative case study. In *Workshop on Unanticipated Software Evolution*.

Rashid, A. and Leidenfrost, N. A. (2004). Supporting flexible object database evolution with aspects. In Karsai, G. and Visser, E., editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 75–94. Springer.

Rashid, A. and Leidenfrost, N. A. (2006). Vejal: An aspect language for versioned type evolution in object databases. Workshop on Linking Aspect Technology and Evolution (held in conjunction with AOSD).