SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2017; **47**:193–221
Published online 8 June 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2415

Modularizing application and database evolution – an aspect-oriented framework for orthogonal persistence

Rui Humberto R. Pereira[1,*,†], J. Baltasar García Perez-Schofield[2] and Francisco Ortin[3]

[1]*CICE, Instituto Politécnico do Porto – ISCAP, Matosinhos, Portugal*
[2]*Departamento de Informática, Universidad de Vigo, Pontevedra, Spain*
[3]*Departamento de Informática, Universidad de Oviedo, Oviedo, Spain*

SUMMARY

In the maintenance of software applications, database evolution is one common difficulty. In object-oriented databases, this process comprises schema evolution and instance adaptation. Both tasks usually require significant effort from programmers and database administrators. In this paper, we propose orthogonal persistence and aspect-oriented programming to support semi-transparent database evolution. A default mechanism for instance evolution is defined, but the user may provide modularized solutions using the aspect-oriented paradigm. We present our framework AOF4OOP to test the feasibility of our proposed approach. This prototype allows programmes to transparently access data in other versions of the database schema. We evaluate our framework, comparing it to related approaches using two real applications and measuring the improvement of the productivity of the programmer. Copyright © 2016 John Wiley & Sons, Ltd.

Received 15 July 2014; Revised 28 April 2016; Accepted 2 May 2016

KEY WORDS: schema evolution; instance adaptation; aspect-oriented programming; orthogonal persistent systems

1. INTRODUCTION

Every application is subject to a continuous process of evolution because of many factors, such as changing requirements, new functionalities or even the correction of design mistakes. In many of these cases, there are strong implications for the underlying data model, which has a specific application interface for each of the existing data types. This schema formalism has a significant impact on the entire evolution process, requiring the intervention of programmers and database administrators. This database and application maintenance problem clearly grows with the size and complexity of the schema.

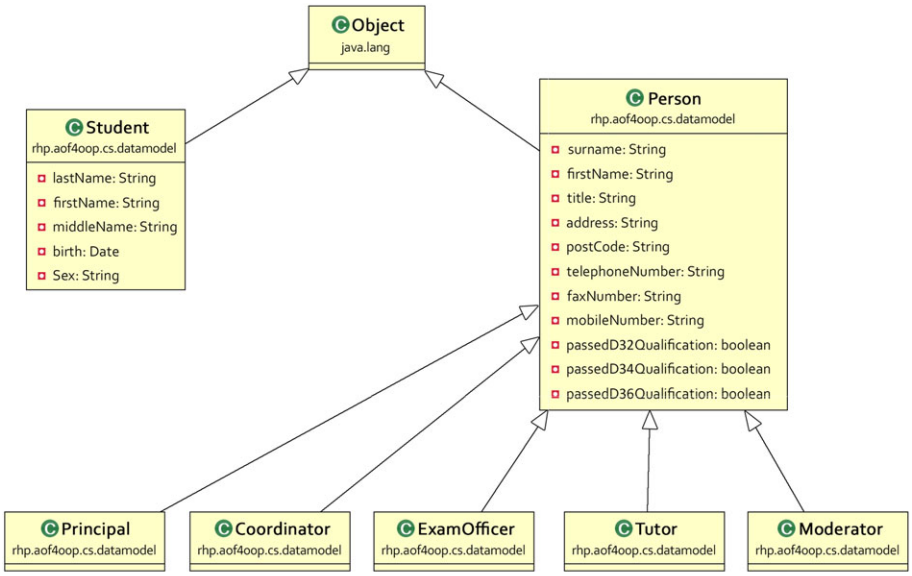
The database evolution problem, in object-oriented databases, has two parts, each in a distinct data layer: (i) schema evolution, at the database metadata layer and (ii) instance adaptation, at the data layer. The schema evolution approach determines application compatibility, which can minimize the negative impact of the evolution process if a multi-version schema approach is chosen. Regardless of the approach to the schema evolution mechanism, the entire database must preserve its structural, behavioural and semantic consistency. Therefore, a proper instance adaptation approach must be applied in order to guarantee that previous existing objects continue to be accessible under the new schema and that no anomalous behaviour occurs during the normal operation of the application.

An example of this problem was presented in an earlier case study [1, 2]. We will use this example as the basis for our discussion in the next sections. Rashid *et al.*'s case study presents a scenario

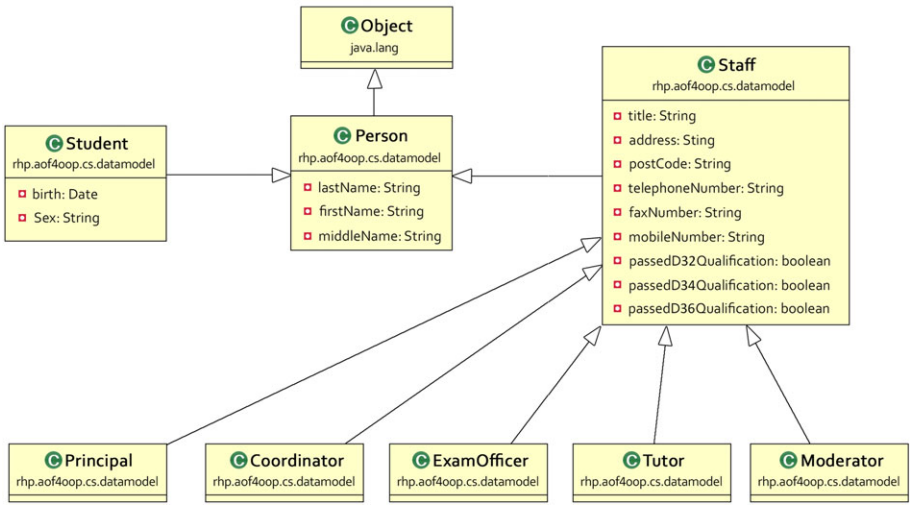
*Correspondence to: Rui Humberto R. Pereira, CICE, Instituto Politécnico do Porto – ISCAP, Matosinhos, Portugal.
†E-mail: rhp@iscap.ipp.pt

of an adult education organization where day-to-day activities are supported on a database, which stores information about students, staff, courses, examinations and results, among others. The management of the organization decided to redesign the database for a variety of reasons, such as fixing design mistakes. In the structure prior to evolution, the class hierarchy did not conform to good object-oriented design principles as the class `Student` was not a subclass of the class `Person`. Thus, in the new structure, `Person` is a superclass of `Student` and `Staff`. Hence, it defines attributes common to both classes (which are transferred to the superclass). Figure 1(a) and (b) shows the structure before and after the evolution, respectively. From now on, we will refer to these two versions of the class structure as version 'A' and version 'B', respectively.

In this example, the database evolution is necessary because the metadata layer (in version 'A') in the object-oriented database system (OODBS) does not follow the new class structure in version 'B'. Additionally, at the data layer, existing instances of the classes must be adapted. This instance adaptation problem is particularly complex in the case of the subclasses of `Person` and `Staff`, in which there are semantic variations.



(a) Before the evolution: Version "A"



(b) After the evolution: Version "B"

Figure 1. Database evolution case study (extracted from [2]).

DOI: 10.1002/spe

The object-oriented database evolution problem has been intensively studied in the last decades but still remains an error prone and time consuming undertaking for programmers and database administrators. Currently, there is some excitement in the scientific community regarding bidirectional transformations (BX) [3]. BX is a mechanism for maintaining the consistency of two (or more) related sources of information. The BX is a concept encompassing many different areas, including software engineering, programming languages, databases and document engineering. As far as we are aware, regarding databases, this research activity has been focused on relational databases, view mechanisms and query translations [4–6], as well as other research areas [7]. Besides other goals, our work intends to fill this gap proposing a new aspect-oriented approach to establish mappings between multiple versions of classes in object-oriented databases.

Object-oriented database management systems like DB4O [8], Versant [9] and ObjectDB [10], provide transparent schema evolution mechanisms. These mechanisms alleviate the programmers' workload by keeping the data and application layers synchronized. Although these modern systems provide such transparent mechanisms, the most frequent types of schema updates identified in earlier works are not supported [11, 12]. In these systems, some types of updates, such as class movements in the inheritance hierarchy, attribute renaming and semantic changes in attributes content, require the programmer's intervention in order to convert the data from the old structure to the new one. Additionally, in the case of a class that is moved within its inheritance hierarchy, these systems also require a temporary class, with a distinct name, which must be renamed at the end of the conversion process. Thus, in real world applications, the use of these systems requires significant effort by the programmers to evolve the database. This is the case with the aforementioned example, in which this type of update occurs.

These database evolution problems motivated us to find solutions to improve the programmer's work in order to be more productive and less prone to errors. This paper addresses the database evolution problem, in the specific context of orthogonal object-oriented persistent programming systems [13, 14] in both its parts: schema evolution and instance adaptation. These systems, due to their intrinsic characteristics, have motivated our research work. We have found that some of these characteristics are distinctive and are not observed in non-orthogonal systems. Therefore, we decided to use a novel approach to deal with the aforementioned problems, applying aspect-oriented programming (AOP) techniques [15]. In this paper, we present our meta-model and its prototype, which explore the benefits of combining the orthogonal persistence paradigm with AOP techniques.

Our aspect-oriented framework enables the modularization of the persistence concern of applications as well as the evolution of the database. Applications rely on this framework in order to be provided with orthogonal persistence mechanisms. Programmers can perform updates of the class structure of the application, because our framework will create a new version of these structures in the OODBS. Our schema evolution approach is incremental, that is, new versions are always added to the metadata layer in the database. Additionally, the framework enables previously created objects (instances of other class versions) to remain available to applications. Each application version only knows its own structure of classes. Our framework adapts the objects loaded from the database according to the class version that is expected by the running application.

Regarding the instance adaptation problem, it is, in many cases, handled autonomously by the framework. Only when structural or semantic variations occur are the default mechanisms unable to perform the instance adaptation autonomously. In these cases, programmers must write conversion code as aspect-oriented modules which extend the default mechanisms of the framework.

The main contribution of our work consists in finding transparent mechanisms for the support of database evolution in orthogonal persistent systems, reducing the programmers' work effort and enabling the coexistence of distinct class versions. In other systems [8–10], programmers must write helper programs to migrate the data when structural or semantic variations occur. In contrast, our framework proposes simple aspect-oriented constructs to extend the default evolution mechanism, making the framework reusable. Our approach focuses on supporting programmers with persistence and database evolution services from a programming perspective rather than a database perspective.

This paper goes beyond the previous publications [16–18] by giving an updated view of our framework and by discussing implementation details that were never published before.

DOI: 10.1002/spe

In this work, we follow the design-science paradigm, which seeks to create innovations that define the ideas, practices, technical capabilities and products through which the analysis, design, implementation, management and use of information systems can be effectively and efficiently accomplished [19, 20]. We follow the guidelines proposed by Hevner *et al.* [21], proposing our framework as an *instantiation* to show that aspect orientation can be used to provide application and database evolution, implemented in a working system [22].

The rest of this paper is structured as follows. In next section, we will discuss two concepts on which our approach is based. Our meta-model and prototype are presented in Section 3. In Section 4, we present an evaluation of our meta-model and prototype. In Section 5, we discuss some related works that inspired our research. Finally, we draw some conclusions from our research.

2. ORTHOGONAL PERSISTENCE AND ASPECT ORIENTATION

2.1. Orthogonal Persistence

According to Atkinson *et al.* [13, 14], an orthogonally persistent system provides applications with abstraction regarding the persistence of their data. In order to achieve such abstraction, systems must follow three principles:

- Persistence independence: the same code should be applicable to both non-persistent and persistent objects;
- Type orthogonality: all objects can be persistent or non-persistent irrespective of their types, sizes or any other property;
- Persistence identification or reachability: a given object is persistent because it is reachable from a persistent root, or through other persistent objects.

Object-oriented databases aim to provide object storing services seamlessly integrated with the programming environment in order to defeat the well-known 'impedance mismatch' problem [23]. Despite these goals, current object database management system (ODBMS) [8–10] interfacing lacks orthogonality, because, in many cases, explicit operations of object storing and activation are required, thus precluding reachability.

2.2. Aspect-oriented programming

Aspect-oriented p rogramming [15] consists in a programming technique that increases the modularity required to follow the separation of concerns principle. Concerns, such as logging, security, or persistence, tend to be scattered and tangled throughout the system modules. The aim of AOP is to improve separation of concerns in a modular way by introducing a new unit of decomposition, called the *aspect*.

According to Filman and Friedman [24], the distinctive characteristics of AOP systems are *Quantification* and *Obliviousness*. *Quantification* relates to making quantified programming assertions over programs [24]. These programming assertions are made through pointcut expressions. Depending on the AOP language, those expressions are capable of identifying a broad type of join point, such as method calls or definitions, read or write properties, object constructors and other strategic application points.

A concern, in the form of code written in the *aspect*, can be introduced or even replace an existing implementation of a specific behaviour of an application without being previously prepared for that procedure. Neither the programmers, who write the base code, nor the applications need that knowledge. This second characteristic present in the AOP is called *Obliviousness* [24].

For Friedrich Steimann [25] aspect-orientation is also *Quantification* and *Obliviousness*. Based on his observations of the generalized accepted definitions for domain model and aspect, and what he considers to be the fundamental properties of aspect-oriented software development, the author argues that aspects are always second-order statements and domain models are first-order [26]; therefore it could be argued that aspects (in the aspect-oriented sense) are technical and extrinsic to the modelled problem domain and its models. That is, aspects are technical and they are few [27].

DOI: 10.1002/spe

Some researchers have questioned the benefits of these two characteristics, stating that the most distinctive characteristics of AOP are others such as abstraction, modularity and composability [28]. For Rashid and Moreira, these are the fundamental characteristics of an aspect-oriented software development approach [28]. In the authors's perspective, the modularization of a system's aspects requires a global reasoning over the entire system and its requisites, which must begin in the early phase of the design process. Modular reasoning means that one can *make decisions about a module while looking only at its implementation, its interfaces and the interfaces of modules referenced in its implementation or interface* [29].

The problem with AOP is that while base modules are syntactically oblivious to aspects, they are not semantically oblivious to aspects [30]. Therefore, when studying a module, a programmer has to consider all aspects that can possibly interfere with and change the module's logic. To mitigate this problem, several extensions to AOP have been proposed. Most of them restrict obliviousness by introducing various forms of interface or contract.

Gudmundson and Kiczales [31] proposed placing named pointcuts in the class definition and then exporting them in the class's interface (the so called pointcut interface). The pointcuts are then used by aspects when defining advices which apply to the class. By having a pointcut interface, the programmer is aware that the class may be influenced by aspects. Exported pointcuts form a contract between a class and its client aspects, allowing the class to be evolved independently of its clients so long as the contract is preserved. The work of Gudmundson and Kiczales was continued by Aldrich [32] and Hoffman and Eugster [33], while the idea of explicit interfaces was further developed by Kiczales and Mezini [29].

Kiczales and Mezini [29] introduced the concept of *aspect-aware interfaces* to annotate method declarations with the aspects that may apply to them. In their approach, a class's aspect-aware interface is automatically computed using reverse engineering through a global analysis of the aspects and the classes once a system has been composed.

Pointcut fragility [34] is another challenge for the aspect-oriented research community. Current AOP languages rely on referencing the structural properties of the program. These structural properties are used by pointcuts to define intended conceptual properties of the program. Thus, maintenance changes that conflict with the assumptions made by the pointcuts may introduce defects [35]. These defects occur when a pointcut unintentionally captures or misses a given join point as a consequence of seemingly safe modifications to the base code [34].

Clifton and Leavens [36, 37] and Przybylek [30] proposed distinguishing between harmful and harmless aspects/advices. Their proposals allow programmers to ignore harmless aspects when reasoning about the base code. Clifton and Leavens call the aspects that can change the behaviour of a module *assistants* [38]. The term *assistant* is intended to connote a participatory role for these aspects. On the other hand, *spectator* aspects are limited in that they may not change the behaviour of the modules they apply to (in a way to be made concrete later). Because this type of aspect does not change the behaviour of other modules, the authors in [38] say that one *spectator views* (rather than advises) methods. Hence, they preserve modular reasoning even when applied without explicit reference by the modules they view [38].

The effect of AOP on software development, maintainability and comprehensibility has been the subject of several studies. Bartsch and Harrison [39] conducted experiments in which no advantages were observed when applying AOP techniques. The experiments of Hanenberg and Endrikat [40] suggest that the use of AOP is only beneficial in situations where the crosscutting concerns refer to a large number of places in the code. Another experiment [41] also suggests that *AOP is a potential rewarding investment into future code changes - but it has risks.* Mortensen *et al.* [42] replace crosscutting concerns with aspects in three industrial applications to examine the effects on properties that affect the maintainability of the applications. Their conclusions suggest that aspect-oriented refactoring reduced the size of the code and improved both change locality and concern diffusion.

Persistence is a pure crosscutting concern, meaning that not only would its implementation in an object-oriented language be tangled but also scattered. Pure crosscutting concerns are rare, because *domain models are aspect free* [26]. From the point of view of applications, persistence is a technical

DOI: 10.1002/spe

problem and not a domain problem [26]. It is a technical concern that cuts across any application domain. Therefore, the implementation of persistence in aspect orientation requires quantified statements about the behaviour of programs.

2.3. Aspect orientation for orthogonal persistence

In orthogonal persistent systems, there is a close integration between the application and the database management system. Another distinctive characteristic is that data schemas are shared by both the application and the database. During the development of applications, programmers may not be concerned with persistence, because the orthogonal persistent programming environment will handle that transparently. The persistence aspect does not violate the specified behaviour of the base modules, which makes it a *spectator* [30, 38, 43]. These observations allow us to argue that *obliviousness* [24] is desirable in orthogonal persistent environments.

Regarding the *Persistence Identification* principle, the persistence state of the object just depends on its reachability from a persistent root (another persistent object). Using AOP techniques in these seamlessly integrated systems, objects can be tracked in order to monitor their persistent state through the analysis of their relationships. Using simple `get` and `set` pointcut expressions, access operations to the attributes of the objects can be easily quantified. In our prototype, we just use two AspectJ pointcut expressions (see Figure 3 in Section 3.5). Thus, all objects can share a common persistence *aspect*, which adjusts its behaviour to each situation, depending on the reachability of the object.

In the context of orthogonal persistence, those two pointcuts (`get` and `set`) are effective. If a class, method, attribute or another property is renamed, moved, added or deleted, these two pointcuts continue to be effective. Thus, for the very particular cases before, the pointcut fragility problem [34] is not present. Notice that one can reason about the persistence concern without interfering with any other application concerns, because it is a technical problem that does not depend on the application. Thus, these observations allow us to argue that *quantification* [24] is also desirable in this category of systems.

This crosscutting concern refers to a large number of places in the code of an application. Besides, a change in the aspect code does not change the way it crosscuts the application. Thus, based on previous research results [40, 41], we believe that the aspect orientation of the orthogonal persistence concern may enable large gains in terms of reusability, compensating for an initial higher development effort. We base our argument on the intrinsic *obliviousness* of orthogonal persistent systems.

For the same reasons, the two observed characteristics mentioned before also shape the database evolution problem, making it very different from the one found in non-orthogonal systems. Programmers can change the structure of the classes, because orthogonal persistence enables an incremental propagation of the schema into the database [14]. Consider the example of the class `Person` in version ‘A’. Before this step in the schema evolution, the database only knows the class `Person` in version ‘A’. Next, the programmer updates the structure of class to version ‘B’ within the source code of the application. From that moment on, version ‘A’ is unknown to the application. The first time `Person` in version ‘B’ appears in the ODBMS, it can be automatically registered in the metadata layer of the database. In each subsequent step of the schema evolution, the propagation of new versions of this class is repeated incrementally. From the point of view of applications, this is transparent, being a technical problem in the ODBMS.

With regard to the instance adaptation problem, it can be handled by default and user-defined mechanisms [44]. The former is a technical aspect of the system because objects can be adapted autonomously by the ODBMS. On the other hand, the latter is dependent on the problem domain of the application, that is, the *role aspects* of the classes [26]), thus requiring user-defined settings about the transformation semantics. In the example given, the update in `Person` is a problem that belongs to the domain of the application. Hence, in this case the ODBMS requires such user-defined settings in order to be able to convert the attribute `surname` to `middleName` and `lastName`.

The technical aspects can benefit from all the advantages in terms of flexibility and customization, as demonstrated in earlier works [45–47] (Section 5). Regarding the domain aspects, we argue that they can be added to the system as modules (containing user-defined settings for conversion). That

DOI: 10.1002/spe

is to say, one can extend the mechanisms of the ODBMS (the ODBMS evolution aspect) with new behaviour (*role aspects* [26] of classes). Hence, the aspect-oriented engine of the ODBMS can be reused without modifications. Our approach follows this paradigm.

3. THE ASPECT-ORIENTED FRAMEWORK FOR OBJECT ORTHOGONAL PERSISTENCE

Our framework [16, 17][‡] mediates the interaction between applications and the OODBS, following an aspect-oriented approach. In the next section, we will discuss the programming environment of our prototype, the aspect-oriented framework for object orthogonal persistence (AOF4OOP) framework. In the following sections, the framework and meta-model are discussed. This meta-model is used by the framework to support the data models of applications.

3.1. The programming environment

The application programming interface (API) of the framework is based on a special class (`CPersistentRoot`), which is used in the development of applications. This API follows the principles of orthogonal persistence. Hence, applications manage persistent and non-persistent data without having a full notion of its persistence state. The API enables access to persistent objects through named references, as well as query-based mechanisms. In fact, independently of the manner in which instances are loaded from the OODBS, each of these persistent objects are roots that enable access to all the other objects that are reachable from them. That is to say, objects are transparently loaded into the memory when accessed the first time by the application. When an application updates an object loaded in the memory, then the update is transparently propagated to the OODBS.

The code listing presented in Figure 2 depicts how applications interact with their data using this API. In this example, we use the `Student` class structure presented in Figure 1(b), as well as an additional `Course` class. In line 9, the mechanism based on named references delivers a `Course` object to its variable without the need of any cast operation. This object is identified by its named reference, which is passed as argument to the `getRootObject()` method. In line 10, the course reference is tested, because the course instance with that key may not exist as a persistent object. In such cases, the framework will return a null value. Consequently, in line 12, a new course object is instantiated in memory and then made persistent using the `setRootObject()` method and its key, in line 13. Notice that in the next executions of the program, the reference obtained in line 10 will be not null. In general, the `setRootObject()` operation is used at the first program execution in order to initialize the persistent root objects of the applications.

In line 16, a query-based mechanism is used for loading objects from the database. The implementation of the `CQuery` class permits querying all instances of a class, which is passed as argument in its constructor. Invoking this query through the API returns the collection of all existing objects, of that class, in the database (students already enrolled in other courses). In line 19, the intended `Student` object is found. In this program execution phase, the variables `course` and `student` contain memory references of the respective persistent objects. Then, these two persistent objects are associated, because the `student` reference is added to an internal array of student references inside the `course` object. Notice that this association is durable because both objects are persistent.

In line 22, a new student object is created; however, this new one (`student2`) is non-persistent. Notice that this instance is created using the constructor of the class. Only in the line 25 is the object pointed to by the reference `student2` made persistent. The framework detects that the `student2` becomes reachable from the array of students of the course, consequently activating its persistence mechanisms in order to make this non-persistent object be a persistent object.

The querying mechanism of the framework was inspired by the one implemented in the DB4O ODBMS based on *Native Queries* [8, 48]. In the current version of our framework, only the presented type of query is available, which does not allow query predicates. Other types of more sophisticated queries are planned as future work. Additionally, the API of the framework also pro-

[‡]Its source code is available in the following repository: https://github.com/rhrp/aof4oop.

DOI: 10.1002/spe

```
CPersistentRoot psRoot;
Student student2;
Course course;

// Initiates the API
psRoot=new CPersistentRoot();

// Get the course from the database
course=psRoot.getRootObject("TOC");
if(course==null)
{
    course=new Course("TOC","Course TOC"),new Student[0],new Coordinator(...));
    psRoot.setRootObject("TOC",course);
}
// Find students by their first and last name
List<Student> students=psRoot.query(new CQuery(Student.class));
for(Student student:students)
    if(student.getLastname().equals("Smith") && student.getFistname().equals("John"))
        course.addStudent(student);

// Instantiates a new Student object (still non-persistent)
student2=new Student(1234,"Paul","C","Williams",...);

// Makes the student2 persistent because it is associated to another persistent object
course.addStudent(student2);
```

Figure 2. Application example.

vides a view mechanism based on saved queries. Users invoke a view in the same manner as the SQL views in relational database management system.

Using the API, the users can impose data constraints and control transactions. Additionally, other operations such as system statistics and debugging are also available in the API.

3.2. *The system architecture*

Figure 3 depicts an overall view of the three layers of the system: application, framework and database. In this figure one can observe how applications are provided with persistence and data integrity mechanisms through woven aspects. These application aspects consume basic services that are implemented in the framework.

The framework is composed of several subsystems, such as the application programming interface, caching of objects and meta-objects, schema manager, object mapping (manages object relationships and object identity), classloading (provides applications and the framework with classes), database garbage collecting (for detecting lost object references at the database layer) and database weaver (Section 3.6). Several aspects of the framework are also modularized in terms of AOP. In Section 3.5, we will discuss these aspects, as well as aspects of the applications.

The database comprises two areas: an OODBS (implemented with DB4O [8] in the current prototype) and an XML file. The OODBS feeds the storaging aspect with *data objects* and meta-objects (excluding the UBMO ones). In Section 3.3, we will discuss these two concepts of object. As one can observe in Figure 3, conceptually, the management system of the OODBS is part of the framework. Notice that our framework also provides database management mechanisms, which complement the ones already available in the DB4O OODBS. Thus, the DB4O is reduced to a simple data repository. The XML file feeds the weaver of the framework with *aspects* (in the AOP sense). This implementation strategy has enabled a rapid development of the framework. *Data objects* and meta-objects are easily stored in an OODBS like DB4O. On the other hand, for our UBMO meta-objects, because of the special editing and extensibility requirements of our pointcut/advice constructs, an XML format was considered quite adequate.

The framework also provides a preprocessor. This component is optional, allowing some dynamic typing features that are not compliant with standard Java compiler rules. It acts as a static weaver, allowing the introduction of additional code into applications that enables the framework to perform a correct persistence of parametric classes[49]. More information about this module can be found in [50].

DOI: 10.1002/spe

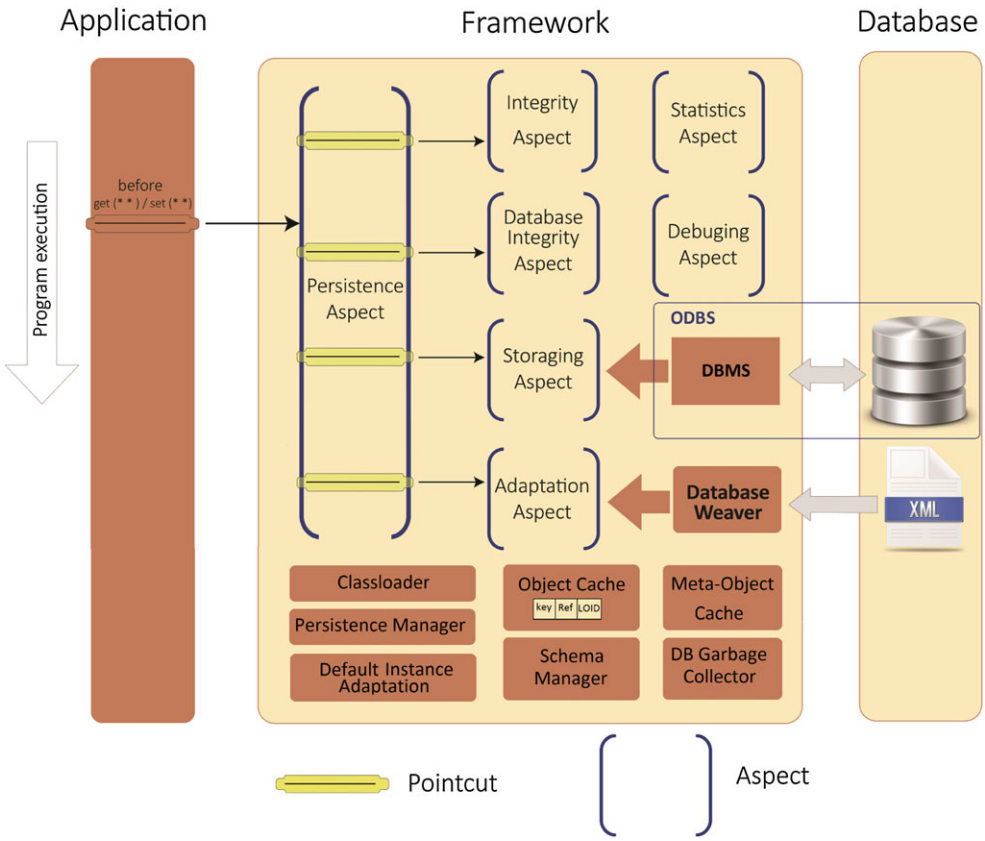


Figure 3. System architecture.

3.3. *The meta-model*

In this section, we discuss our proposal for a new meta-model based on class versioning [51, 52]. Our meta-model is used for representing in the database the data models of applications through its basic entities: *data objects* and meta-objects.

In our meta-model, there are two concepts of object: (i) *Logical Object* and (ii) *Data Object*. The first represents an instance of an application data entity, for example, a student that is represented through the `Student` class in the application scope. As result of several steps in the application evolution, the structure of the `Student` class may have distinct versions. Our class versioning meta-model was designed to support a class in the several versions that it may have. Thus, each instance of the `Student` class can be stored in the OODBS according to the structure definition known by the application that created it. From the point of view of the application, a class has only one version, which is the one defined in the source code. However, in the OODBS, each object of that `Student` class may be an instance of another version of this application class. These `Student` instances in the OODBS are supported through *data objects*. The *data objects* form the data layer of the system.

Such a level of abstraction, regarding the class version, is enabled by *Logical Objects*. This kind of object encapsulates the class version and structure details from applications. When an application requires a *data object* associated to a *logical object* which does not follow the expected class version, the framework emulates the (logical) object in the correct version.

The *logical objects* are supported in the metadata layer using the meta-objects of our meta-model. The framework uses this metadata stored in meta-objects and data in *data objects* to emulate (logical) objects as they are expected by applications.

The meta-objects that form the metadata layer of the system are instances of meta-classes. In our meta-model, there are the following six meta-classes: (1) Class Version Meta-Object (CVMO); (2)

DOI: 10.1002/spe

Instance Meta-Object (IMO); (3) Attribute Meta-Object (AMO); (4) Root Meta-Object (RMO); (5) Update/Backdate Meta-Objects (UBMO); and (5) Aspect meta-objects (AspMO).

Figure 4 illustrates an example of how these meta-objects are related in order to represent one course instance associated with its students and coordinator. In this example, we reuse the structures

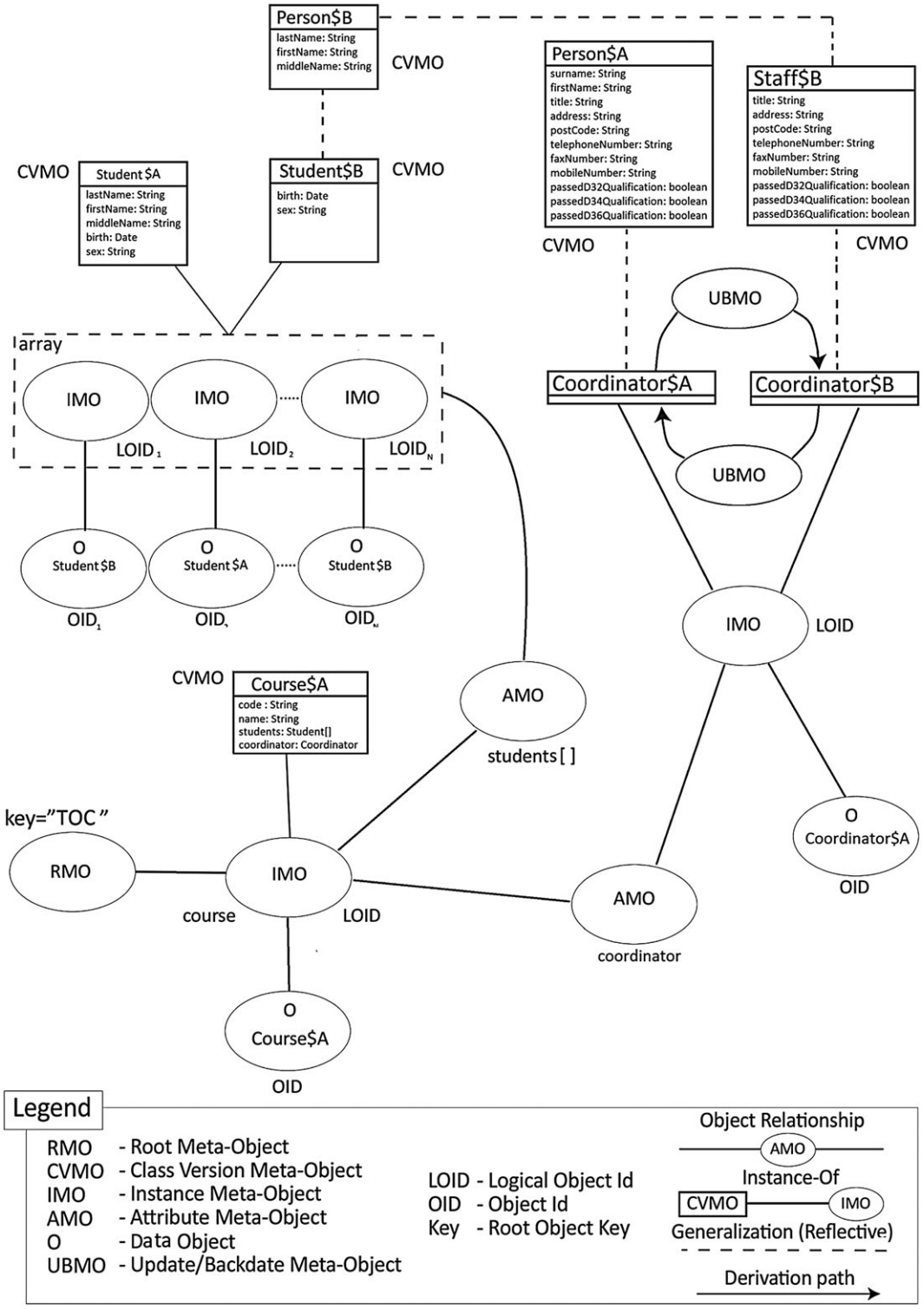


Figure 4. Meta-model: Example of students enrolled in a course.

DOI: 10.1002/spe

of the classes `Student` and `Coordinator` in order to discuss how the meta-model represents them in versions 'A' and 'B', in, respectively, Figure 1(a) and (b).

The persistent root is an instance of `Course`, which is pointed through a RMO meta-object and identified through an arbitrary string key value (a named reference). In the presented example, this key has the value '`TOC`'. In Figure 2, we discussed how applications use the API of the framework in order to obtain the `course` object using its named reference. The course instance, pointed to by the RMO meta-object, has an IMO meta-object that supports its logical identity.

Additionally, this IMO meta-object is also associated with two AMO meta-objects. These AMO meta-objects support the relationships to the coordinator and enrolled students of the course. That is to say, they represent the attributes `coordinator` and `students[]` of the `course` logical object. In turn, the AMO meta-object of the coordinator is associated with another IMO meta-object, which supports the `coordinator` logical object. On the other hand, the AMO meta-object of the students is associated with a set of IMO meta-objects. Each one of them is a cell of an array of students. Notice that other AMO or RMO meta-objects may refer to these logical objects (cells of the array).

In Figure 4, a class in a specific version is represented using the notation `ClassName$Version`. Our framework recognizes new classes and versions through the generation of this version identifier using a hash function. As a result of the hash function, these automatically generated; identifiers are not very programmer-friendly. That is why the `@Aop4oopVersionAlias` Java annotation [53] has been added to the framework. This annotation can be associated to a class in order to have its versions easily named. Figure 5 depicts how this annotation is used to identify the class versions presented in the Figure 1(b). Notice that each class can have a distinct version identifier. In this case, all classes were redesigned.

At runtime, our prototype provides each application with the objects pertaining to its class versions, even though its persistent version in the database may be different. Additionally, in the same runtime environment, the internal mechanisms of the framework require access to objects that could exist in another version. Thus, a strategy is required in order to allow multiple versions of the same class to exist simultaneously at runtime. Regarding the unique name invariant that any structure of classes must obey, the name of a class must be unique. Java, as well as most programming languages, does not provide type versioning support.

In order to accommodate these multiple versions, we followed a class renaming strategy that fulfils the unique class name invariant. Consider, as an example, the application class `Person` in version 'C'. Their *data objects* are managed in the framework and stored in the database as `Person$C`. Regarding the other versions of `Person`, for example, 'A' and 'B', these are managed as `Person$A` and `Person$B`. That is to say, from the point of view of the application, there is a class `Person`; whereas from the point of view of the framework, there are three classes: `Person$A`, `Person$B` and `Person$C`.

The class `Person` is defined in the source code of the application. On the other hand, these renamed classes (`Person$A`, `Person$B` and `Person$C`), as well as their bytecode, are stored inside the database (as CVMO meta-objects). By using the classloader of the framework, these three classes are available at runtime for the conversion mechanisms. This approach overcomes those programming language limitations, while enabling programmers to write conversion code without using any other programming language like Vejal [54].

```
1  @Aof4oopVersionAlias(alias = "B")
2  class Person { ... }
3
4  @Aof4oopVersionAlias(alias = "B")
5  class Student extends Person { ... }
6
7  @Aof4oopVersionAlias(alias = "B")
8  class Staff extends Person { ... }
```

Figure 5. Identifying class versions.

DOI: 10.1002/spe

Two types of change can be applied to a class: at the class level (adding, removing or modifying attributes and methods), or changes to its inheritance structure. Each time that any of these changes occur, a new class version is created. If a superclass is updated, then its subclasses must evolve, requiring a new version identifier for each one of them. For example, let us consider a superclass that has five attributes, from `a` to `e`. This superclass, as well as its subclasses, has the same identifier in version 'A'. Subsequently, the superclass is updated, and a new attribute `f` is added to it. As a consequence, all classes in the hierarchy get a new version identifier 'B'. In a second update, the attribute `a` is replaced by `z`. As in the previous update, all classes in the hierarchy get a new version identifier `C`. Notice that if a superclass is updated, all its subclasses are affected.

Our schema manager automatically detects new class versions when they are not recognized by the classloader. Thus, the first time a class is made persistent, an additional operation is performed. The schema manager registers that new class version into the metadata layer as a CVMO meta-object. From that moment on, the class version is registered in the database, and it is recognized by the classloader as well. This incremental evolution of classes frees the programmer from having to register classes in two places: the application and the database. Programmers just need to perform the update in the source of the application code because the framework will produce a new class version each time a class is modified.

Each IMO meta-object is associated with one or more CVMO meta-objects. Each of these CVMO meta-objects supports the metadata regarding the structure of the class of the logical object in a specific version. Besides the metadata, CVMO meta-objects preserve the bytecode of the class. Using this preserved bytecode, our framework knows the internal structure of each class (members, methods and annotations), as well as the hierarchical relationships. The hierarchical relationships are not explicitly represented in the metadata because they can be obtained from the bytecode of the class through reflection. By searching CVMO meta-objects in the database, it is possible to find all subclasses of a class. Figure 4 presents some examples of instance-of relationships that are supported through CVMO meta-objects. The class `Student` in version 'A' has no user-defined superclass; however, in version 'B', it is a subclass of the class `Person`. As the `Coordinator` class, its superclass is replaced from one version to another. Additionally, the bytecode of the class stored in these meta-objects feeds the classloader of the framework.

Each one of these IMO meta-objects is also associated to one *data object*. Notice that in our prototype, we just maintain one *data object* version per logical instance. In our example, the `course` (logical) object in the database is maintained through one single *data object* in class version 'A', as well as the `coordinator` object in its class version 'A'. Regarding the array of students, some cells, that is, IMO meta-objects) have their data objects in class version 'A' and others in class version 'B'.

Our approach based on *logical object* and *data object* concepts requires a similar perspective regarding the identifiers. As an object identifier (OID) identifies physical (data) objects in the database, a logical object identifier (LOID) identifies a logical object (an IMO meta-object). That is to say, an LOID is the logical identity of an application object which encapsulates the physical structure of data in the database. In turn, the OID physically identifies a *data object* or meta-object. In the example depicted in Figure 4, one can observe these two types of identifier.

This approach simplifies the object update process because it avoids the physical replacement of relationships. Furthermore, it also allows an application class to have instances in distinct versions.

Our framework has an internal data structure that provides the mapping between LOID identifiers and references to objects in the memory. Thus, two objects with the same LOID inside the running environment have a common reference in the memory. That is to say, for applications, they are the same object instance.

The aspect meta-objects represent *aspects* (in terms of AOP), which can have one of two usages: application *aspects* (e.g. logging) or database management system *aspects* (e.g. security). This type of meta-object is just part of our meta-model, not being implemented in our framework, as well as being outside of the scope of this paper.

The UBMO meta-objects support the instance adaptation mechanisms. In Section 3.4, we will continue this discussion, presenting this type of meta-object.

DOI: 10.1002/spe

3.4. Supporting instance adaptation

The simplicity of class evolution, addressed with an incremental approach, contrasts with the complexity of the adaptation of existing data in the database. Therefore, this instance adaptation problem raises complex implementation issues, especially in a multi-version schema, which is our case. In order to address this problem, our framework performs two types of mapping: *direct mappings* and *user-defined mappings*.

3.4.1. Direct mappings. A direct mapping is applied in the absence of any user-defined mapping. This mechanism supports all class updates that can be autonomously inferred by default conversion functions [44]. In our framework, a direct mapping deals with changing attributes to compatible types (`int` to `long`; `int` to `double`; any numerical type to `String`), movement of attributes across the class inheritance structure and new attributes initialized to zero or `null`. Additionally, our framework provides an `@Aof4oopDefault(value)` annotation that is used for passing default values to conversion functions. When objects are being converted by direct mapping, their new attributes can be initialized with user-defined values passed by this annotation.

In our example presented in Figure 1(a) and (b), the `Student` class can be converted between versions 'A' and 'B' using a direct mapping. Despite the structural changes in the hierarchical relationships of the classes, a direct mapping between their attributes is possible. No user definitions are required: in our framework, the default instance adaptation mechanism can handle it autonomously.

3.4.2. User-defined mappings. User-defined mappings are required when direct mapping is unable to deal with the conversion. In Figure 4, two examples of this type of mapping are necessary in case of the `Coordinator` class. Notice that structural changes that occurred in its superclasses have affected the semantic consistency between these two class versions. One UBMO meta-object supports user definitions for converting `Coordinator` objects from class version 'A' to 'B', and another one supports converting from class version 'B' to 'A'. As we will discuss in Section 3.4.3, these two UBMO meta-objects can handle all the five subclasses of `Person` in version 'A' and `Staff` in version 'B'.

These meta-objects are built using our pointcut/advice constructs. That is to say, programmers write these pointcut/advice constructs, which are stored as UBMO meta-objects in the metadata layer.

3.4.3. Pointcut/advice expressions. Instance adaptation is a crosscutting concern of classes that are the subject of evolution. In order to quantify these join points, we developed a new kind of pointcut/advice construct, which follows the *Quantification* definition posited by Filman and Friedman [24]: "In programs P, whenever condition C arises, perform action A."

These expressions are structured into two parts: trigger conditions and action. In Table I, we present a quick reference guide for these constructs, improved because our last publication [17].

At runtime, an action is triggered when the object instance being loaded from the database and the one required by the application satisfy the conditions of a construct. The trigger conditions establish an explicit mapping between these two class versions. The action information is used to extend the instance adaptation aspect of the framework with the required conversion behaviour.

These constructs are written in XML to enable easier editing using a graphical tool or simply a text editor. Furthermore, our approach also benefits from the extensibility of XML for the future implementation of new features. In our implementation, the action is written in Java, which is embedded into the `<sourceCode>` XML node. The conditions for triggering the action are specified through matching parameters. In an aspect-oriented sense, these matching parameters quantify the objects subject to conversion, while the conversion function is an *advice*.

Continuing with our example in Figure 4, the adaptation of all those five subclasses from version 'A' to 'B' only requires two UBMO meta-objects. In Figures 6 and 7, one can observe the XML-based constructs supporting these two meta-objects. From the point of view of the framework, users intend to adapt objects from one version to another, which does not mean that this has to be the real

DOI: 10.1002/spe

Table I. Pointcut/advice constructs quick reference guide.

Parameter/node	Description
Conditions	
`name`	Mandatory parameter which enables pointcut identification. It is applied in the generation of the name of woven classes. This name is very useful for debugging purposes because in case of a runtime error, the programmer can determine its localization within the source code.
`matchClassName`	Class canonical name of the advised classes. Emulated objects whose classes match this parameter are advised. This parameter allows the * wildcard. If the class name is followed by `[]`, the adapted object must be an array of this class.
`matchSuperClassName`	Superclass canonical name of the advised classes. Matching through a superclass is meant to reduce the number of user definitions. If there are many target classes that share a common superclass, all of them are advised. This parameter allows the * wildcard.
`matchOldClassVersion`	The class version of the persistent object in the database being emulated/converted. This parameter allows the * wildcard and `or` operator.
`matchCurrentClassVersion`	Defines the class version identifier of the running application. This parameter is required when many versions of the class already exist. In these cases, we must ensure a correct target application version. This parameter allows the * wildcard and `or` operator.
`matchParentClassName`	This contextual parameter enables advising objects which at runtime are pointed to from a certain class. This parameter allows the * wildcard.
`matchParentMember`	This is another contextual parameter that enables advising objects, which at runtime are pointed to from a certain object member.
Action	
`applyDefault`	This parameter, when true, applies the default conversion behaviour, alleviating the programmers' effort in order to write this action. Note that in many cases, only some members of a class require user-defined code.
`outputClassName`	This parameter specifies the type of return value. This is very useful when we intend to apply an expression to several classes that share a common superclass. In these cases, the *advice* can convert that superclass or any other.
`sourceCode`	Conversion code in plain programming language. Must be finished with a return statement.

```
1  <ubmo name="ConvSubClassesStaff$A_to_$B"
2     matchSuperClassName="rhp.aof4oop.cs.datamodel.Staff"
3     matchClassName="rhp.aof4oop.cs.datamodel.*"
4     matchOldClassVersion="A"
5     matchCurrentClassVersion="B">
6    <conversion applyDefault="true" outputClassName="rhp.aof4oop.cs.datamodel.Staff">
7      <sourceCode>
8          <![CDATA[
9        String middlename=.....;
10       newObj.setLastName(oldObj.getSurname());
11       newObj.setMiddleName(middlename);
12       return newObj;
13         ]]>
14     </sourceCode>
15   </conversion>
16 </ubmo>
```

Figure 6. Conversion of all subclasses of `Staff` from version 'A' to 'B'.

chronology of the evolution of the classes. Thus, in the scope of the conversion source code, the 'old' and 'current' versions only represent the point of view of the construct.

In these five subclasses, the problem arises from structural changes made to their superclass. In version 'A', the name attributes of the `Person` class are `firstName` and `Surname`. In version 'B', these attributes were replaced by others in the `Person` class, as well as having `Staff` introduced as a subclass of `Person`.

DOI: 10.1002/spe

```
1  <ubmo name="ConvSubClassesPerson$B_to_$A"
2        matchSuperClassName="rhp.aof4oop.cs.datamodel.Person"
3        matchClassName="rhp.aof4oop.cs.datamodel.*"
4        matchOldClassVersion="B"
5        matchCurrentClassVersion="A">
6     <conversion applyDefault="true" outputClassName="rhp.aof4oop.cs.datamodel.Person">
7       <sourceCode>
8        <![CDATA[
9        //conversion code
10       ...
11       return newObj;
12       ]]>
13      </sourceCode>
14    </conversion>
15 </ubmo>
```

Figure 7. Conversion of all subclasses of `Staff` from version 'B' to 'A'.

Figure 6 presents the required user definitions in order to adapt all `Person` subclasses from version 'A' to 'B'. Each construct has a name as mandatory attribute. Besides the name, other optional ones are required in order to establish the target objects for adapting. From lines 2 to 5, matching conditions for target identification are defined as: superclass, class, old and new class versions. The `matchClassName` attribute defines which is the application class of the target objects. The `matchOldClassVersion` restricts the target objects to the ones stored in the database in the class version 'A'. In order to restrict the target object regarding the class version required by the running application, the attribute `matchCurrentClassVersion` specifies the class version 'B'. Additionally, the operator '`|`' (`or`) can be applied in order to match with several versions.

In line 6, the parameter `outputClassName` tells the framework what the return type is. Notice that in this case, the conversion mechanism returns an object of class `Staff`, hence allowing a single construct to handle all five subclasses of `Staff`. The code in lines 9 to 12 performs the semantic adaptation between the two class versions.

Figure 7 depicts the second UBMO meta-object. As in the previous one, this expression handles all five subclasses of `Person` class when they are being converted from class version 'B' to 'A'.

In the current state of development of our framework, programmers must edit these constructs by hand. For them, this represents an additional cognitive overhead. Notice that we chose XML only to represent UBMO meta-objects. We argue that a graphical tool for editing XML will eliminate that overhead. Thus, we plan to include such a new graphical tool to edit UBMO meta-objects in the future versions of our framework.

3.5. Aspects supporting persistence and database evolution

In this section, we discuss our approach to addressing the database evolution problem using AOP. Our meta-model discussed in Section 3.3 and the orthogonal persistence paradigm, as well as class versioning, are the bases of this approach.

The aspects of the system were modularized at two distinct levels: application and framework. Therefore, the specific cross-cutting concerns of the persistence aspect are also modeled as aspects, that is, aspects of aspects). These aspects are woven with the components of the applications and the framework using AspectJ. Figure 3 illustrates a program execution that is intercepted at join points that require persistence behaviour. In this figure, one can observe join points (`get(*/*)` and `set(*.*)`), in the execution of the program, which are intercepted by the persistence aspect.

3.5.1. Application aspects. Two aspects were modularized at the application level: (i) data persistence and (ii) data integrity. The former regards all join points, where persistent data is read or written. Each accessed object is advised according to its state of persistence. The latter enables all integrity checks defined through the Java annotations enabled by our framework. A discussion of these integrity mechanisms is beyond the scope of this paper.

DOI: 10.1002/spe

3.5.2. Framework aspects. The main module of the framework implements all the components that do not require flexibility or that are not transversal. Thus, five aspects were modularized at this level: (i) Instance adaptation: This is the base aspect that implements default mechanisms for instance adaptation. In Section 3.6, we will discuss our approach to extend this base aspect of the framework. (ii) Database data integrity: Ensures constraint maintenance (introduced in the application source code) during the conversion/emulation process from one version to another. (iii) Object storing: This aspect deals with the persistence concern of the framework. It enables a flexible and pluggable persistence implementation. (iv) System statistics: Generates the internal statistics of the system. Using the API of the framework, users can get this information about the system. (v) Debugging: Generates debugging data.

3.6. Extending the instance adaptation aspect of the framework

Our framework is based on a hybrid procedure of static and runtime weaving. The instance adaptation aspect is statically attached to the framework by means of the AspectJ weaver, as well as the remaining aspects of the framework and the persistence aspect in applications. The runtime weaving enables the base instance adaptation aspect to be extended in order to be able to deal with the specific issues of adaptation of each class version (its *role aspects*). In this section, we discuss this runtime weaving mechanism of the framework.

When adaptation is not required, the *data object* just requires a dynamic typing mechanism in order to change its type (e.g. `Person$B` to `Person`). Thus, our framework instantiates the class (e.g. `Person`) using its default constructor and then copies all attributes from the *data object* to the newly instantiated one. At the end, the framework maps the converted (logical) object using its memory reference and LOID. After this moment, the object is available in the application scope.

On the other hand, if adaptation is required, additional steps are performed. The framework checks if there is a pointcut/advice construct (i.e. an UBMO) that matches the object being adapted. If there is a match, the action part of that construct (action `sourceCode` of the UBMO meta-object) is delivered to the runtime weaver of the framework. This weaver compiles the source code, producing a new *aspect* as a special class that implements the `doConversion()` function. Notice that this function is an internal mechanism to invoke the aspects added to the framework. In terms of optimization, the runtime weaving procedure is carried out once at first, or when the UBMO meta-object is updated.

The `doConversion()` function has an interface which is known by the mechanisms of the framework. This system call receives two pairs of arguments defined in its interface: `oldObj`, `oldObjParent`, `newObj` and `newObjParent`. Their names are reserved words, which are references to the target object, in two distinct versions, and their parents in the context of the conversion. After the runtime weaving has taken place, this function is invoked.

In Figure 8, an example of a `doConversion()` function is presented. The class that implements this function was woven at runtime having its origins in the pointcut/advice construct presented in Figure 9. A simple example is presented in these two figures. Consider version 'A' of class `Person`, where the person's name is a single property, that is, its attribute `name`; however, in the current version of the application, assume it is 'B', the person's name is split into two parts: the first and last names. At conversion time, the `doConversion()` function receives a `Person` object pre-converted through its `newObj` argument (loaded from the database and pre-converted as discussed previously through direct mapping). Thus, only the `firstName` and `lastName` attributes need to be converted, alleviating the programmer's effort.

```
1  Person doConversion(Person$A oldObj, Family$A oldObjParent,
2                      Person newObj, Family newObjParent)
3  {
4       String tmp=oldObj.getName().substring(0,oldObj.getName().lastIndexOf(" "+newObjParent.getFamilyName()));
5       newObj.setFirstName(tmp);
6       newObj.setLastName(newObjParent.getFamilyName());
7       return newObj;
8  }
```

Figure 8. Example of a woven `doConversion()` function.

DOI: 10.1002/spe

```
<ubmo name="ConvPerson$A_to_Person$B"
      matchClassName="rhp.aof4oop.cs.datamodel.Person"
      matchOldClassVersion="A"
      matchCurrentClassVersion="B"
      matchParentClassName="rhp.aof4oop.cs.datamodel.Family">
  <conversion applyDefault="true"
              outputClassName="rhp.aof4oop.cs.datamodel.Person">
    <sourceCode>
        <![CDATA[
    String tmp=oldObj.getName().substring(0,oldObj.getName().lastIndexOf(" "+newObjParent.getFamilyName()));
    newObj.setFirstName(tmp);
    newObj.setLastName(newObjParent.getFamilyName());
    return newObj;
        ]]>
     </sourceCode>
   </conversion>
</ubmo>
```

Figure 9. Pointcut/advice construct for adapting a person's last name.

Notice that in this example, the remaining attributes are handled by direct mapping, because `applyDefault` is set to true. In the conversion context, a `Family` object is the parent of the `Person` object (the one that is being converted). This is ensured due to the matching parameter `matchParentClassName="rhp.aof4oop.cs.datamodel.Family"`. The conversion code takes advantage of the `Family` object, which contains the family's name, in order to split the person's name into those two parts. In this example, `doConversion()` is fed with the target object `Person` in the scope of the application, its *data object* `Person$A` at the database and their parent objects (`Family` and `Family$A`).

3.7. *Structural, semantic and behavioural consistency*

The proposed meta-model, based on class versioning, raises consistency issues. In this section, we briefly discuss how these requirements are addressed in our approach.

The static type checking performed by the compiler ensures that each object instance inside the application runtime environment unconditionally pertains to, and follows, the schema version of the application. Inside the user-defined conversion code, such classes are statically checked during the compilation of the weaving process, ensuring that they belong to a known class version. Thus, in both cases, anomalous behaviour is avoided at runtime because there are no invalid references to methods or attributes. Because each new class version is incrementally propagated to the database, in this layer, all object instances also pertain to one of those multiple versions of the classes. Hence, the schema evolution invariants are propagated to the database.

While structural and behavioural consistency is verified at compile time, semantic consistency is addressed with update/backdate conversion code and additional metadata added into the application classes. Our UBMO meta-objects ensure this type of schema consistency.

Several database evolution taxonomies for object-oriented databases have been proposed [2, 55]. The taxonomy of our framework is closely tied to the programming language because programmers can freely perform updates in the class structures of their applications. Our proposal to address this problem is, theoretically, able to deal with all kinds of updates at the class level in which there is equivalence. For example, consider class `C` in versions α and β, referred to, respectively, as Cα and Cβ. We define a conversion function f, which converts an object in version α to β, by

$$f : C\alpha \rightarrow C\beta. \tag{1}$$

Hence, consistency is always achievable at the class level if there is such a function f.

On the other hand, if there is no equivalence, our framework cannot handle the problem. As an example, consider a class in which an attribute is added in version 'A'. In version 'B' of this class, this attribute is removed and no other attribute providing the same information is added or exists in the class. In this example, there is an equivalence from version 'A' to 'B'. However, no equivalence exists from version 'B' to 'A'. This lack of equivalence breaks the continuity of the versions. Applications based on version 'A' of that class structure will lose information if objects are updated to version 'B'.

DOI: 10.1002/spe

Our framework detects this lack of equivalence ('B' to 'A' conversion in our example), and then, by default, AOF4OOP does not allow the conversion. In that case, during the startup of the application in version 'A', a message notifies the user that an exception is going to be thrown when it accesses the field deleted by 'B'. However, if there is a solution in the specific domain for performing such a conversion, the programmer may specify the conversion routine following the proposed aspect-oriented adaptation mechanism.

In order to reduce the number of such cases, our pointcut/advice constructs still address another scenario. Consider a class `C` in version α and class `D` in version β, referred to, respectively, as Cα and Dβ. The equation for this case is

$$f : C\alpha \rightarrow D\beta. \tag{2}$$

Thus, if there is such a function f, then it is possible to convert objects in class Cα to class Dβ. This evolution scenario occurs when in the class structure α, there is a class `C` which is replaced by `D` in the class structure β. From the point of view of our meta-model and framework, the replacement of a class and its update produces the same problem because a function f will perform the conversion. Notice that our framework invokes an appropriate function f when applications require an object according to its class structure.

As discussed in Section 3.6, the internal conversion mechanism of the framework is provided with non-local information. That is to say, besides information regarding the target object, its parents' information is also available for this mechanism. Thus, Equation (2) can be generalized as follows:

$$f : (C\alpha, PC\alpha, PD\beta) \rightarrow D\beta, \tag{3}$$

where PCα and PDβ are, respectively, the parent object of `C` in version α and the parent object of `D` in version β. As an example, consider class `Person` in class structure α, which has its address information in three attributes: `street`, `city` and `zipCode`. Because several persons (belonging to a family) can share the same address, it makes sense to create a class `Address` with these attributes. Thus, in the new class structure β, objects of class `Address` are obtained from

$$f : (null, \texttt{Person}\alpha, \texttt{Person}\beta) \rightarrow \texttt{Address}\beta. \tag{4}$$

Notice that in this scenario, the class `Address` does not exist in version α. Hence, it is passed on as a null value to the function f.

Based on the previous arguments, one can conclude that our approach to maintaining consistency is based on equivalence through the use of conversion functions. Hence, if the class structures (before and after the evolution) are not equivalent, then the evolution problem cannot be fully addressed within our approach. In some cases, the lack of a direct equivalence between two versions of a class may be overcome using non-local information at the immediate object relationships level.

3.8. Transactions, failures and locking

Transactions and failures have been identified by many authors [56–59] as being impossible to totally aspectize. In our framework, only transactional mechanisms are aspectized. The semantics of the transactions remains in the application scope.

The orthogonal persistence paradigm has implications for the way that concurrency is handled in our framework. These implications have to do with Atkinson's principles of *Persistence Independence* and *Reachability*. In order to address the challenges imposed by this paradigm, we implemented a multithreaded environment within a single Java Virtual Machine (JVM).

By default, our framework works in autocommit mode. In this mode, when one object attribute is updated, the persistence mechanisms work atomically, that is, both data and metadata are saved within a unique database transaction. Each time an application object is updated, its LOID remains locked. During the time that the normal program execution is intercepted by the persistence aspect, the application loses control in favour of the framework. The application remains locked until the operation is successfully finished, or a runtime exception is thrown. If an exception is thrown, the attribute of the object in the memory as well as its data and metadata in the database are rolled back. In such cases, the framework throws a runtime exception of the type `EFrameworkFault`.

DOI: 10.1002/spe

Non-persistent objects are treated in the same manner. Although these operations on non-persistent objects do not produce runtime exceptions, because they do not interact with the database, exceptions may occur. Note that the constraint checking mechanisms of the framework might prevent an attribute update. If the new values are not in accordance with the integrity definitions introduced in the source code as annotations, then the framework throws a runtime exception of type `EIntegrityFault`.

In order to achieve a more sophisticated transactional model, we implemented three new system calls: `beginTransaction()`, `commitTransaction()` and `rollBackTransaction()`. The system call `beginTransaction()` switches off the autocommit mode. While autocommit mode is turned off, changes are not committed in the database. That only occurs at the end of the transaction, by calling `commitTransaction()`. Calling `rollBackTransaction()` will end the transaction and roll back all changes in the database, as well as in the instantiated objects in memory. At the end of a transaction, the system returns to autocommit mode. These mechanisms of the framework ensure the atomicity, consistency, isolation and durability properties in a multithreaded environment.

Because of the orthogonal persistence principles, the framework must include non-persistent objects in the transactions. During a transaction, the framework saves the state of the object as it was at the beginning of the transaction. Thus, if an exception occurs, the initial state is restored.

Two instantiated objects belonging to two distinct JVMs do not share the same memory. Hence, if one of them is updated, the other one is not synchronized. Currently, only problems regarding memory synchronization and locking across several threads are taken into account. This problem limits the use of our framework to a single JVM. These issues, as well as mechanisms for nested transactions, are currently under development.

4. EVALUATION

4.1. Proof of concept application

In order to test our framework in a real scenario, a real world application was developed. It provides means for users to manage personal data regarding students and staff members of an adult education organization. Our proof of concept application[§] is based on the same case study in Rashid *et al.* [1, 2] and is presented in Figure 1(a) and (b).

The user interface is organized into two tabs, one for students and another for staff. Both tabs provide the same operations for students and staff members. Figure 11(a) and (b) depicts two screenshots of the user interface of the application when the staff tab is selected. In these figures, our application is working according to the version ‘B’ of the class structure. As we will discuss, our application exists in two versions, one in each state of the class structure: ‘A’ and ‘B’, that is, before and after the evolution.

During the initial phase of the development of the application, data was managed in memory without any concern for data persistence. At the execution time of the application, all data introduced by users remains available. Although the application is totally operational at this phase, all that data is lost in any further execution after its restart.

In order to provide the application with persistence mechanisms, we use our framework. Figure 10 presents the listing with the new simple lines of code which are required in order to attach the application to the framework.

In the former version of the application, only the array initialization in line 6 existed. In line 3, the application tries to get the reference to an array of `Student` from the database using its named reference ‘students’. If no reference is found in the database, a new empty one is created and then put into the database (code in lines 6 and 7). After that point in the execution line of the program, all the objects that are reachable from this array will be persistent by means of the transparent mechanisms of the framework.

[§]Its source code, as well as the framework, is available in the following repository: https://github.com/rhrp/aof4oop.

DOI: 10.1002/spe

```
1  CPersistentRoot psRoot=new CPersistentRoot();
2
3  Student[] students=psRoot.getRootObject("students");
4  if(students==null)
5  {
6     students=new Student[0];
7     psRoot.setRootObject("students",students);
8  }
```

Figure 10. Providing the application with persistence.

Regarding the update of the objects, the user interface provides a button that is activated when the fields of the form are changed by the user. When this button is clicked, the content of the fields is copied to the corresponding attributes of objects. No additional work is required to update the database because these objects are reachable from a persistent root.

The application was initially developed using as its base version 'A' the class structure in Rashid *et al.* In order to make our case study more realistic, we populated the database with real data regarding 3500 students and 212 staff members. Next, the decision of the management of the organization, to redesign the database, was simulated by updating the class structure of the application according to version 'B'. Consequently, some changes had to be made in the application and its user interface.

The database structure was no longer compatible with this new version of the application because the class structures of versions 'A' and 'B' are not equal. In order to deal with this structural and semantic incompatibility in the five staff classes, we applied the pointcut/advice constructs in Figures 6 and 7, which were previously discussed. Regarding the students, no user intervention was required. The default mechanisms of the framework for instance, adaptation were capable of dealing with the problem. Notice that no semantic changes occurred in class `Student`. In this case, only attribute movements across the class hierarchy were observed (from `Student` to `Person`).

After the framework was extended using those two pointcut/advice constructs, the application (in version 'B') was initiated. The entire database of both versions was available to the application. Objects in version 'A' are emulated only when required (the persistence mechanisms of the framework are lazy). Users may update the data about the students and staff because an object in memory after being updated is saved into the database as the class structure in version 'B' (objects are stored as the class structure of the application that performed the last update). All the objects continued to be available from the application in version 'A', even though they were updated by the application in version 'B'.

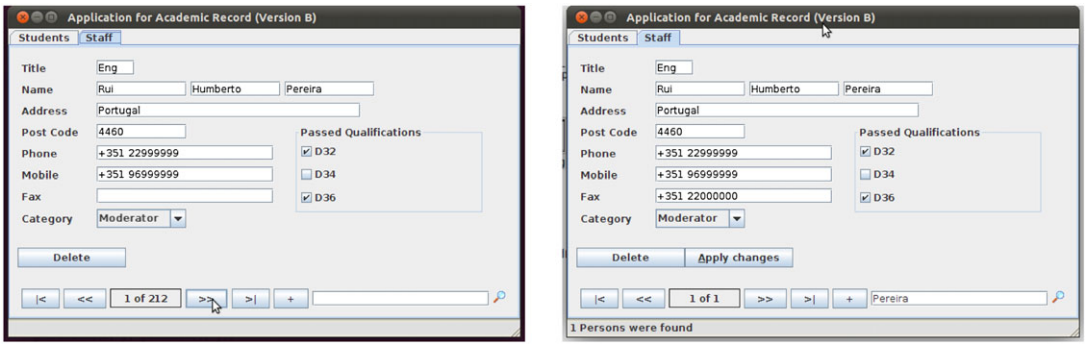
4.2. Volume of changes in the source code

In order to evaluate the gains in terms of work productivity that our framework offers to programmers, we conducted another comparative case study intended to assess the added value of the framework to its object repository based on a DB4O OODBS. The DB4O is a reference system in object-oriented databases that provides object persistence with a considerable level of transparency and orthogonality. It could be considered as the system most compatible with the AOF4OOP framework, and hence, we chose it compare it with our system.

The same redesign scenario used in the previous section was considered again. In this study, we adapted two applications that use the data model in version 'A' represented in Figure 1(a) before their redesign. The first application was presented in the previous section. The second one provides the same features but stores its data directly in a DB4O OODBS. Notice that they are the same application using two distinct persistence mechanisms.

Because both applications share a common data model, the code defining the class structure was implemented as a common library. The task of performing the source code adaptations in this common structure of classes was named Task-1. Additionally, both applications have some lines of source code that depend on the class structure, which requires programmer intervention (in particular in the graphical user interface (GUI) presented in Figure 11(a) and (b)). In version 'A', the `Coordinator` class (and the remaining `Staff` subclasses) has a `surname` member that does

DOI: 10.1002/spe



(a) Navigating through records (b) Searching and editing

Figure 11. User interface of the application.

not exist in the new version 'B'. It also has two new members, `middleName` and `lastName`, in version 'B'. We have structured both applications in order to share the source code of the GUI components. Thus, this second adaptation task is also identical in both applications, which we named Task-2. The orthogonality present in the persistence mechanisms of the DB4O and AOP4OOP systems has eliminated the need of adaptation in other parts of the applications.

An additional task was still necessary in order to adapt the existing class instances in version 'A', stored in the database, to the new version 'B'. We named this third task Task-3. The first challenge lies in a limitation of DB4O, which does not transparently support inserting and removing classes from the inheritance hierarchy. Both types of class transformation occur in this study, so an additional application is required to adapt the object instances (`Student` and all `Staff` subclasses) from the old class version to the new one. The process suggested[¶] by the authors of DB4O is the following one: (i) Create the new hierarchy with different names, preferably in a new package; (ii) Copy all values from the old classes to the new classes; (iii) Redirect all references from existing objects to the new classes. Additionally, there are still semantic and structural differences between the two versions. In order to adapt the objects belonging to the DB4O based application, we developed two helper applications[‖] that perform the process described earlier. The first application performs the copy of the objects to a temporary class structure in version 'B', in a distinct package in order to avoid class names colision. It also performs the semantic adaptation of the copied objects. The second helper application copies all the objects in this temporary class structure to the definitive one and deletes the temporary class structure. Task-3 in the case study application based on DB4O is then finished by running these two applications in the right order.

In contrast with the DB4O case study application, the AOF4OOP-based application does not require, in Task-3, any additional helper programs, which saves on the programmer's effort. As discussed in the previous section, only one of those two pointcut/advice constructs is required for adapting from version 'A' to 'B'. After introducing the construct in Figure 6, the programmer may run the application in version 'B' because the framework handles the object adaptation problem.

For each task, we conducted an analysis of the impact in terms of lines of code, classes/aspects and atomic changes. Regarding the atomic changes, we adopted the Chianti taxonomy [60]. Table II depicts our results for the three tasks in both case study applications.

The presented results show measurable benefits in the case of the application based on our framework. Using our framework for accomplishing Task-3, programmers need only write 16 new lines of code necessary for the database evolution *aspect* defined through the pointcut construct depicted in Figure 6. On the other hand, 315 new lines of source code are required for those two helper applications. In terms of global work effort for writing new code, the AOF4OOP-based application

[¶]Accessed April 2, 2014 in the following URL: http://community.versant.com/documentation/reference/db4o-8.1/net/reference/Content/advanced_topics/refactoring_and_schema_evolution/refactoring_class_hierarchy.htm.

[‖]Their source code is available in the following repositories: https://github.com/rhrp/aof4oop_cs_db_step1 and https://github.com/rhrp/aof4oop_cs_db_step2.

DOI: 10.1002/spe

Table II. Volume of changes.

	Task-1		Task-2		Task-3		**Total**	
Metric	DB4O	AOF4OOP	DB4O	AOF4OOP	DB4O	AOF4OOP	**DB4O**	**AOF4OOP**
NLOC	106	106	27	27	315	16	448	149
MLOC	9	9	13	13	0	0	22	22
DLOC	74	74	14	14	0	0	88	88
NCA	1	1	0	0	3	1	4	2
MCA	2	2	1	1	0	0	3	3
DCA	0	0	0	0	0	0	0	0
AC	90	90	14	14	47	3	151	107

NLOC: new lines of code; MLOC: modified lines of code; DLOC: deleted lines of code; NCA: new classes/aspects; MCA: modified classes/aspects; DCA: deleted classes/aspects; AC: atomic changes; AOF4OOP: aspect-oriented framework for object orthogonal persistence.

only consumes one-third of the effort that is necessary with the DB4O application. These metrics of effort cannot be proportionally converted to working time, because working time depends on the programmers' skills and experience. Notice that each programmer may find a distinct solution for each problem, with a distinct effort in terms of lines of code and working time. In order to obtain an estimate of the working time, we invited two programmers to perform these three tasks. The results showed the same proportion: 15 and 46 min of working time, for the AOF4OOP- and DB4O-based applications, respectively.

Regarding the presented scenario, these experimental results have shown the advantages of the framework in terms of programmer productivity over a reference system in the state-of-the-art of object-oriented databases. However, there are two other additional advantages that must be considered: (i) In order to perform the schema updates, DB4O must follow an eager stop-the-world approach, while the proposed framework does not and (ii) The DB4O approach does not allow the previous application to continue being operational, while AOF4OOP only requires an additional pointcut/advice construct.

4.3. Meta-model analysis

In order to evaluate our meta-model, we extended the case study of Rashid *et al.* [1, 2]. This earlier case study compares the authors' system with three others, when faced with the design correction scenario used as an example in our proof of concept application discussed in Section 4.1.

We reused this scenario in order to extend the previous work with our results and to enable a more direct and easier comparison. This comparative case study was structured into two parts. The first analyzes the impact on meta-objects, at the schema level, while the second provides an analysis of the instance adaptation process. The systems compared were Orion, Encore, TSE and SADES [61].

4.3.1. Schema. The relationships between classes are modified in each schema update. Depending on how the corresponding meta-model represents relationships, the impact could be more or less complex, and would affect a distinct number of meta-objects. Classes, attributes, methods and other types of model entities are represented in the meta-model through their corresponding meta-objects. Thus, all these meta-objects are interconnected according to the data structure represented. For example, a class meta-object has an inheritance relationship with its superclass meta-object, and an aggregation relationship with all its subclasses, attributes and methods. Orion, Encore and TSE use attributes at the meta-object level to implement those relationships between meta-objects. In SADES, those connections are represented as relationship objects [2]. Our approach, presented in Section 3, is simpler because none of those relationships require meta-objects.

In the comparative case study in Rashid *et al.* [1, 2], the introduction of a non-leaf class `Staff`, which has a different impact on each system, affecting several meta-objects, is given as an example. Table III presents their results merged with our own. As we can observe, with regards to this correction scenario, our meta-model presents the best results in terms of affected meta-objects. Regarding

DOI: 10.1002/spe

Table III. Meta-model comparison.

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

the SADES system, it surpasses Orion, Encore and TSE because of the usage of relationship objects. These first class objects, which encapsulate the information regarding the connections among the meta-objects, contrast with these three systems that have that information embedded within the meta-objects. A detailed description of the implications of introducing the `Staff` superclass in each system can be found in the case study referred to the aforementioned section.

Because the updates in our meta-model are incremental, only new class versions are added to the schema. Thus, just eight new CVMO meta-objects are inserted: seven new class versions for the existing classes and the new `Staff` class. Additionally, because of update/backdate compatibility, two UBMO meta-objects provide the means for conversion in both directions. In this scenario, all remaining changes are transparently treated by the direct mapping mechanism, avoiding any extra information.

This design correction scenario, while being very simple, enables a good understanding of the advantages present in our meta-model. Although this earlier case study does not cover the AspOEv system (an evolution of the SADES system, see Section 5), we can extrapolate these results because SADES and AspOEv follow the same approach in terms of the meta-object layer [46].

4.3.2. Instance adaptation. The instance adaptation problem is addressed in the second part of [1, 2]. It provides a detailed comparative description of the adaptation mechanisms in the four systems. Those systems were also described in that earlier work, as well as our own in Section 3. Considering the existing descriptions, here, we just discuss the positioning of our approach in relation to those studied systems.

SADES, in relation to Orion, Encore and TSE, stands out because of its customization capability and the flexibility of the instance adaptation process. The aspect-oriented process allows the customization and flexibility by dynamically applying the most suitable technique. Those advantages will be discussed in Section 5. In this regard, SADES, AspOEv and our framework are equivalent because the three systems follow the same aspect-oriented approach.

4.4. Limitations of the framework

4.4.1. Performance. Our framework emulates objects according to the version expected by the application. This approach introduces a significant delay if the objects are not yet in cache. Although performance degradation was not perceptible in our tests, it exists because of the additional mechanisms of versioning implemented. However, in our geographical application[**] [17], because of the use of large arrays of objects, this performance degradation was observable.

Although most of this performance degradation is unavoidable and inherent in the class versioning approach, we argue that the flexibility and customization enabled by the aspect-orientation of the database evolution provides means for other approaches. For example, maintaining multiple versions of the *data objects*. Another example consists of a lazy update of the *data objects* to the class

[**]This previous proof of concept application uses data obtained from the online OpenStreetMap (http://www.openstreetmap.org) geographical database. The OpenStreetMap online geographical database allows a user-defined area to be exported through its coordinates. The OSM export files, in XML format (http://wiki.openstreetmap.org/wiki/OSM_XML), contain all the data related with that geographical area. Each one contains the coordinates of the boundaries and is structured as a set of objects, such as *Nodes*, *Ways* and *Relations*. Our geographical application enables users to import those OSM files into the local database. Users can browse these locally stored geographical areas, thus having access to all the information (their points of interest).

DOI: 10.1002/spe

version of the most recent (and most used) application version. Such flexibility and customization provides the necessary means to apply the adjusted strategies to each case.

In terms of performance, some of the current limitations of the prototype derive from the use of an OODBS as an object repository (DB4O). This OODBS provides high-level interaction mechanisms in a single version schema, and therefore, it is not well adapted to the requirements of the framework. Theoretically, a new specialized object store for our framework and meta-model would enhance its overall performance and orthogonality, avoiding many additional operations. We will consider this specialized object repository as a future improvement to be made to our framework.

4.4.2. Transactions and failures. In our framework, each member of a persistent object is updated within a single database transaction. During the life cycle of transactions, the application remains locked. If any error occurs, an exception of the framework is thrown and the entire transaction, involving objects and meta-objects, is reverted. At the end of the transactions, the framework returns the control to the application. Using this approach, the atomicity, consistency, isolation and durability properties are always ensured. We consider this as a minimalist approach in addressing the problem of transactions, because it does not serve the requirements of complex application transactions. Thus, we are developing additional mechanisms to manage transactions and failures. In Section 3.8, we have discussed these new features of our framework.

4.4.3. Concurrency and process model. As discussed in Section 3.8, our framework provides a safe multithreading environment for concurrent programming. It enables concurrent programming within a single JVM, sharing memory and other resources. Examples of these types of environments are embedded systems and personal devices.

Although a safe multithreading environment is enabled by the prototype, it is restricted to a single JVM. This limitation arises from the orthogonal persistence paradigm, where the persistence state of objects depends on its reachability. Two instantiated objects in two distinct JVMs do not share memory. Hence, if one of these objects is updated, the other one does not get synchronized. Memory synchronization and locking across several JVM are still unresolved issues that we intend to address in our future work.

4.4.4. Type orthogonality. In order to allow different class versions, a class renaming strategy was adopted. Besides the impact on the performance of the system, such a solution raises implementation issues due to JVM restrictions. Note that classes pertaining to the standard Java library cannot be manipulated by reflection. Thus, in the current version of our prototype, important classes such as `ArrayList` are limited in terms of versioning. As a workaround, these classes are treated as non-versionable objects, that is, their structure cannot be changed. Although an LOID is assigned to each, they are saved into the database as is.

5. RELATED WORK

5.1. Persistence as an aspect

Al-Mansari *et al.* [58, 62] discussed the benefits of AOP combined with orthogonal persistence and proposed a solution based on *path expressions pointcut* (PEP) and *persisting containers*. An orthogonal persistent system based on these new pointcut expressions will provide *aspects* with information about non-local objects, solving the problem of determining the reachability of objects. However, the authors only propose a denotational semantics for PEP. Their contribution is a guide for future PEP developments and its integration with AOP systems.

Soares *et al.* [63] present their experience while refactoring a web application, a Health Watcher system, modularizing all code related with distribution and persistence concerns using AspectJ.

Rashid *et al.* [57, 64] addressed the persistence modularization as an aspect. Rashid and Chitchyan [57] demonstrate that, in the context of a database application, persistence can be effectively aspectized. However, these authors argue that only partial obliviousness is desirable. They argue that

DOI: 10.1002/spe

persistence has to be taken into account as an architectural decision during the design of data-consumer components [57]. For example, GUI components need to be aware of large volumes of data so that they may be presented to users in manageable chunks.

Yet these two previous studies did not focus on orthogonal persistence. Although persistence has been successfully modularized, the reusability of the *aspects* is poor, because in those systems, the persistence is a *role aspect* of classes [26]. In developing our framework, we followed the best practices for designing aspect-oriented systems [33]. Thus, we chose a favourable balance between obliviousness and modularity. In consequence, the presented framework has enabled an easy and non-invasive implementation of the persistence concern in our proof of concept application. As discussed in Section 4.1, the application was initially developed without any implementation of the persistence concern. Applying a minimal change in its source code, we provide the objects of the application with a persistence aspect. All remaining code of the application is kept unchanged, being oblivious to persistence concerns and decoupled from the framework. Moreover, the persistence aspect does not violate the specified behaviour of the base modules, which makes it a *spectator* [30, 43]. Despite providing applications with additional behaviour, they continue working if turned off, being thus oblivious to that lack of persistence. Furthermore, the orthogonal persistence paradigm followed by our framework still enables transparent data access while promoting programming quality.

5.2. Database evolution as an aspect

As far as we are aware, the main research work carried out in the field of object-oriented databases schema evolution, applying AOP techniques, points to solutions that partially, and in some cases, totally solve the schema evolution problem. The research work carried out by Rashid *et al.* [2, 46, 61, 65] has been an important contribution to the improvement of the flexibility and customization features of known evolution techniques. Those authors have proposed an approach that allows the combination of isolated techniques into one unique, powerful and integrated solution, applying AOP techniques.

In the SADES system [61], Rashid introduced the concept of *aspect* in object-oriented databases, turning the system, in itself, into an aspect-oriented database [65]. In this research work, some crosscutting concerns that were identified could be aspectized in terms of AOP. Those crosscutting concerns exist at the database management system and database levels. Constraints checking, access rights, as well as all related aspects of the database evolution domain, such as instance adaptation, inheritance and versioning, are good examples of those concerns.

In the SADES system and later in the AspOEv system [46], the AOP techniques were explored in order to implement database mechanisms of schema evolution and instance adaptation. Rashid *et al.* also propose a meta-object model [66] based on three types of entities to represent data and schema structures: objects, meta-objects and meta-classes. This meta-model was an inspiration for our work, presented in Section 3.3. However, as discussed in Section 4.3, our meta-model has fewer meta-classes and hence is much simpler. Our meta-model supports schemas based on class versioning but does not represent all the derivation paths of class versions. Additionally, class hierarchy relationships are also not explicitly represented in our meta-model.

The authors of that paper remark on the high degree of flexibility provided by their approach when compared with other systems. They argue that traditionally object-oriented databases only offer a single schema evolution approach coupled with a specific instance adaptation mechanism. Hence, they do not effectively meet the needs of the applications [45]. They concluded that the aspect-orientation of a database allows for a pluggable and customizable system, which can provide database administrators and programmers with powerful means to do their work [45]. At the database management system level, it enables the access to strategic system internal points to introduce customizable components. In the scope of database evolution, the most appropriate approach can be chosen or even a combination of them [45, 46].

In that approach, the stored entities, both data and metadata objects, can have their own *aspects* at the database level. Additionally, these *aspects* themselves can be made persistent entities and also stored in the database. In those authors' opinion, with which we agree, this approach opens a

DOI: 10.1002/spe

broad spectrum of customization solutions. On objects, the specific concerns of application logic and entities could be aspectized. On the other hand, the concerns of metadata objects, such as update/backdate routines, are likely to be aspectized. This approach avoids the need to introduce new code in classes every time conversion methods are updated (when a new version of the class occurs).

The AspOEv system introduced its own language, Vejal [54], capable of manipulating the objects in their multiple class versions. An application based on the AspOEv system is written in Vejal programming language. With this approach, Rashid and Leidenfrost intend to overcome the generalized limitation of object-oriented programming languages, which do not support type versioning. In the Vejal language, the class `Person` in version `1` is represented as `Person< 1 >`. In another syntax, `Person<s=1>`, which means the class version of `Person` that occurs in version `1` of the schema. In Vejal, a class in its version is a type. We concur with the authors' opinion that this approach improves type-checking safety by avoiding incorrect type inferences when evolving the database with the adaptation of types and instances.

In AspOEv, the Vejal language provides programmers with the means to deal with variations in class structure, as version types. Therefore, applications must be aware of the semantic and structural variations among class versions. This contrasts with our approach, which offers the possibility of modularizing these *role aspects* of classes [26]. Notice that in our framework the details of how to convert a class between its versions are completely separate from the applications and the framework itself. Each application only knows its own structure of classes.

The AspOEv system was inspired by SADES. However, SADES uses aspects to directly plug-in the instance adaptation code into the system. This system requires programmers to be aware of low-level technical details. Database consistency is ensured by *Reflective Handlers* [54], which handle the mismatch exceptions. Consequently, the complexity of the instance adaptation hot spots has to be exposed to the programmer. With this exposure, there is the risk of unwanted interference with the database from the aspect code. This system also contrasts with our framework, in which programmers just write the appropriate conversion functions to adapt the objects. Furthermore, SADES supports customization of the instance adaptation approach only: the schema evolution strategy is fixed. It does not support version polymorphism either.

Kuppuswami *et al.* [47] have also explored AOP techniques proposing a flexible instance adaptation approach, developing a system that supports instance adaptation with two *aspects*: update/backdate and selective lazy conversion *aspects*. These *update/backdate aspects* support the concern regarding the emulation of object versions. The object is retrieved from the store in its physical version, and it is then converted to the expected structure by the application.

In their work, those authors also highlight the flexibility provided by AOP techniques to support database evolution concerns. They conclude that the encapsulation of concerns within *aspects* enables the easy replacement of the adaptation strategy and code. Thus, this contrasts with other systems that introduce code directly into the class versions.

The approach proposed by Kuppuswami *et al.* [47] implements *role aspects* of the classes as update/backdate *aspects*. Although this approach is, in its basis, similar to the one proposed in the present paper, it does not offer a systematic interface to declare and implement aspects. Programmers must implement those aspects using AspectJ or another AOP tool.

These earlier works showed that aspect-orientation of the database evolution enables a high-level of flexibility and customization. Like these systems, our approach also enables the same flexibility and customization, in terms of applied strategies.

6. CONCLUSIONS

We propose a new approach based on a meta-model and pointcut/advice constructs to provide applications with orthogonal persistence in a database based on class versioning. The implemented AOP4OOP framework enables persistence and database evolution concerns of the applications to be modularized in terms of AOP. It provides programmers with semi-transparent mechanisms to deal with the evolution of classes. The class structure of the application can be freely changed in its source code, being then incrementally reflected into the database, autonomously producing a

DOI: 10.1002/spe

new version of each modified class. Only when semantic updates occur must programmers apply the proposed XML-based pointcut/advice constructs. The existing applications that recognize a previous class structure version continue to work, unchanged, oblivious to schema changes at the database level.

In terms of database evolution support, our framework offers more transparent mechanisms which affect positively the programmers' productivity. Our framework avoids the need for helper programs in many scenarios, like the one discussed in this paper (as well as a geographical application presented in [17]). The development of such helper programs penalizes the workload in terms of productivity, because they are additional tasks for the programmers.

Besides the same benefits found in related work [46, 47, 54], the AOP4OOP framework is reusable because of our aspect-oriented approach supported in our XML-based pointcut/advice constructs. The details of how to convert a class between its versions are modularized apart from the applications and the framework itself. We argue that our system contrasts with others, which require technical knowledge to deal with the inconsistencies among class versions. It allows programmers to write applications and conversion code using the same programming language, without requiring a specific framework language. Additionally, these pointcut/advice constructs allow programmers to use local and non-local information regarding the target object being converted. We argue that our approach empowers the richness and expressiveness of the user-defined conversion functions. The presented examples highlight the following benefits: (i) full programming language features; (ii) access to non-local data; and (iii) the application class behaviour can be reused in conversion functions.

Compared with other approaches [1, 2], the evaluation presented has shown that our meta-model is simpler than the existing approaches, because of the implicit relationships among its class versions and class hierarchies, thereby requiring a less complex meta-object layer. As shown in the presented comparative case study, it reduces the number of affected meta-objects.

We exploit the characteristics of the orthogonal persistence paradigm. Despite its many design challenges, we think that this persistence paradigm provides particular means for modularity, applying AOP techniques. Our approach incorporates the benefits of those two paradigms. With our proof of concept application, we demonstrated these benefits within a real scenario.

REFERENCES

1. Rashid A. Aspect-oriented schema evolution in object databases: a comparative case study. *Workshop on Unanticipated Software Evolution*, Spain, 2002.
2. Rashid A, Sawyer P. A database evolution taxonomy for object-oriented databases. *Journal of Software Maintenance and Evolution: Research and Practice* 2005; **17**(2):93–141.
3. Czarnecki K, Foster J, Hu Z, Lmmel R, Schrr A, Terwilliger J. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, vol. 5563, Paige RF (ed.)., Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2009; 260–283.
4. Terwilliger JF. Bidirectional by necessity: Data persistence and adaptability for evolving application development. In *GTTSE*, vol. 7680, Lmmel R, Saraiva J, Visser J (eds)., Lecture Notes in Computer Science. Springer, 2011; 219–270. Available form: http://dblp.uni-trier.de/db/conf/gttse/gttse2011.html#Terwilliger11 [last accessed April 2014].
5. Bernstein PA, Jacob M, Pérez J, Rull G, Terwilliger JF. Incremental mapping compilation in an object-to-relational mapping system. *SIGMOD Conference*, New York, New York, USA, 2013; 1269–1280.
6. Beine M, Hames N, Weber JH, Cleve A. *Bidirectional transformations in database evolution: A case study at scale*.
7. Curino CA, Moon HJ, Zaniolo C. Graceful database schema evolution: The prism workbench. *Proc. VLDB Endow.* 2008; **1**(1):761–772.
8. Paterson J, Edlich S, Hrning H, Hrning R. *The Definitive Guide to db4o*. Apress: Berkely, CA, USA, 2006.
9. Corporation V. *Versant object database fundamentals manual*, 2010. Available from: http://developer.versant.com/developer/resources/objectdatabase/documentation/VODFundamentals.pdf [Accessed on 2 April 2014].
10. Ltd OS. *Objectdb 2.3 developer's guide*. Available from: http://www.objectdb.com [Accessed on 2 April 2014].
11. Advani D, Hassoun Y, Counsell S. Extracting refactoring trends from open-source software and a possible solution to the 'related refactoring' conundrum. *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, ACM, New York, NY, USA, 2006; 1713–1720.
12. Piccioni M, Oriol M, Meyer B. Schema evolution for persistent object-oriented software: model, empirical study, and automated support. *CoRR* 2011; **39**(2):184–196.

DOI: 10.1002/spe

13. Atkinson MP, Bailey PJ, Chisholm KJ, Cockshott PW, Morrison R. An approach to persistent programming. *The Computer Journal* 1983; **26**(4):360–365. Available from: http://comjnl.oxfordjournals.org/cgi/content/abstract/26/4/360 [last accessed April 2014].
14. Atkinson M, Morrison R. Orthogonally persistent object systems. *The VLDB Journal* 1995; **4**(3):319–402.
15. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J. Aspect-oriented programming. In *ECOOP'97 ? Object-Oriented Programming*, vol. 1241, Aksit M, Matsuoka S (eds)., Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 1997; 220–242.
16. Pereira RH, Perez-Schofield JBG. Database evolution on an orthogonal persistent programming system - a semi-transparent approach. *2012 7th Iberian Conference Information Systems and Technologies (CISTI)*, Madrid, Spain, 2012; 1–6.
17. Pereira RH, Perez-Schofield JG. Towards a flexible and transparent database evolution. In *New Perspectives in Information Systems and Technologies, Volume 2, Advances in Intelligent Systems and Computing*, Vol. 276, Rocha l, Correia AM, Tan FB, Stroetmann KA (eds). Springer International Publishing, 2014; 23–33.
18. Pereira RH, Perez-Schofield JG. Evolution of the application and database with aspects. In *ICEIS 2014 - Proceedings of the 16th International Conference on Enterprise Information Systems*, Vol. 1, Hammoudi S, Maciaszek LA, Cordeiro J (eds). SciTePress: Lisbon, Portugal, 2014; 308–313.
19. Denning PJ. A new social contract for research. *Commun. ACM* 1997; **40**(2):132–134.
20. Tsichritzis D. The dynamics of innovation. In *Beyond Calculation: The Next Fifty Years of Computing*, Dinning P, Metcalfe R (eds). Copernicus: New York, NY, USA, 1997; 259–265.
21. Hevner AR, March ST, Park J, Ram S. Design science in information systems research. *MIS Q* 2004; **28**(1):75–105.
22. March ST, Smith GF. Design and natural science research on information technology. *Decis. Support Syst.* 1995; **15**(4):251–266.
23. Dearle A, Kirby GNC, Morrison R. Orthogonal persistence revisited. In *Object Databases*, Norrie MC, Grossniklaus M (eds). Springer: Berlin Heidelberg, 2010; 1–22.
24. Filman RE, Friedman DP. Aspect-oriented programming is quantification and obliviousness. *Workshop on Advanced Separation of Concerns*, OOPSLA, Minneapolis, USA, 2000.
25. Steimann F. The paradoxical success of aspect-oriented programming. *SIGPLAN Not* 2006; **41**(10):481–497.
26. Steimann F. Domain models are aspect free. In *MODELS/UML 2005 (SPRINGER*. Springer, 2005; 171–185.
27. Steimann F. Aspects are technical, and they are few. *Proceedings Of The European Interactive Workshop On Aspects In Software (EIWAS04)*, Berlin, Germany, 2004.
28. Rashid A, Moreira A. Domain models are not aspect free. In *Model Driven Engineering Languages and Systems*, vol. 4199, Nierstrasz O, Whittle J, Harel D, Reggio G (eds)., Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2006; 155–169.
29. Kiczales G, Mezini M. Aspect-oriented programming and modular reasoning. *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, MO, USA, 2005; 49–58.
30. Przybylek A. Quasi-controlled experimentations on the impact of AOP on software comprehensibility. *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*, IEEE Computer Society, Washington, DC, USA, 2013; 253–262.
31. Gudmundson S, Kiczales G. Addressing practical software development issues in aspectj with a pointcut interface. In *Advanced Separation of Concerns*, 2001.
32. Aldrich J. Open modules: A proposal for modular reasoning in aspect-oriented programming. In *Workshop on Foundations of Aspect-Oriented Languages*, 2004; 7–18.
33. Hoffman K, Eugster P. Trading obliviousness for modularity with cooperative aspect-oriented programming. *ACM Transactions on Software Engineering and Methodology* 2013; **22**(3):22:1–22:46.
34. Störzer M, Koppen C. Pcdiff: Attacking the fragile pointcut problem, abstract. *European Interactive Workshop on Aspects in Software*, Berlin, Germany, 2004.
35. Przybylek A. Systems evolution and software reuse in object-oriented programming and aspect-oriented programming. *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns (TOOLS ,11)*, Springer-Verlag, Berlin, Heidelberg, 2011; 163–178. Available from: http://dl.acm.org/citation.cfm?id=2025896.2025909 [last accessed April 2014].
36. Clifton C, Leavens GT. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *FOAL Workshop*, 2002.
37. Clifton C, Leavens GT. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *SPLAT*, 2003.
38. Clifton C, Leavens GT. *Spectators and assistants*. Enabling Modular Aspect-Oriented Reasoning, 2002.
39. Bartsch M, Harrison R. An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Journal* 2007; **16**(1):23–44.
40. Hanenberg S, Kleinschmager S, Josupeit-Walter M. Does aspect-oriented programming increase the development speed for crosscutting code? an empirical study. *3rd International Symposium on Empirical Software Engineering and Measurement (ESEM 2009)*, Lake Buena Vista, Florida, USA, 2009; 156–167.
41. Hanenberg S, Endrikat S. Aspect-orientation is a rewarding investment into future code changes - as long as the aspects hardly change. *Information and Software Technology* 2013; **55**(4):722–740.
42. Mortensen M, Ghosh S, Bieman J. Aspect-oriented refactoring of legacy applications: an evaluation. *IEEE Transactions on Software Engineering* 2012; **38**(1):118–140.

DOI: 10.1002/spe

43. Leavens GT, Clifton C. Multiple concerns in aspect-oriented language design: A language engineering approach to balancing benefits, with examples. *Proceedings of the 5th Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT '07)*, ACM, New York, NY, USA, 2007.
44. Ferrandina F, Meyer T, Zicari R, Ferran G, Madec J. Schema and database evolution in the o2 object database system. *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995; 170–181. Available from: http://dl.acm.org/citation.cfm?id=645921.673314 [last accessed April 2014].
45. Rashid A, Sawyer P. Aspect-orientation and database systems: an effective customisation approach. *IEE Proceedings - Software* 2001; **148**(5):156–164.
46. Rashid A, Leidenfrost NA. Supporting flexible object database evolution with aspects. In *GPCE*, vol. 3286, Karsai G, Visser E (eds)., Lecture Notes in Computer Science. Springer, 2004; 75–94. Available from: http://dblp.uni-trier.de/db/conf/gpce/gpce2004.html#RashidL04 [last accessed April 2014].
47. Kusspuswami S, Palanivel K, Amouda V. Applying aspect-oriented approach for instance adaptation for object-oriented databases. *Proceedings of the 15th International Conference on Advanced Computing and Communications*, IEEE Computer Society, Washington, DC, USA, 2007; 35–40. Available from: http://dl.acm.org/citation.cfm?id=1333633.1333708 [last accessed April 2014].
48. Cook WR, Rosenberger C. *Native queries for persistent objects, a design white paper*, 2005.
49. Alagić S, Royer M. Genericity in java: persistent and database systems implications. *The VLDB Journal* 2008; **17**(4):847–878.
50. Pereira RH, Perez-Schofield JBG. Orthogonal persistence in java supported by aspect-oriented programming and reflection. *2011 6th Iberian Conference Information Systems and Technologies (CISTI)*, Chaves, Portugal, 2011; 1–6.
51. Monk S, Sommerville I. Schema evolution in oodbs using class versioning. *SIGMOD Rec.*, Vol. 22, September 1993; 16–22.
52. Clamen SM. Type evolution and instance adaptation. *Technical Report*, Pittsburgh, PA, USA, 1992.
53. Bloch J. *JSR 175: A metadata facility for the java programming language*, Vol. 30, 2004. Available from: http://jcp.org/en/jsr/detail?id=175 [last accessed April 2014].
54. Rashid A, Leidenfrost NA. Vejal: An aspect language for versioned type evolution in object databases. *Workshop on Linking Aspect Technology and Evolution (held in conjunction with AOSD)*, Bonn, Germany, 2006.
55. Banerjee J, Kim W, Kim HJ, Korth HF. Semantics and implementation of schema evolution in object-oriented databases. *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87)*, ACM, New York, NY, USA, 1987; 311–322.
56. Kienzle J, Guerraoui R. Aop: Does it make sense? the case of concurrency and failures. *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, Springer-Verlag, London, UK, UK, 2002; 37–61. Available from: http://dl.acm.org/citation.cfm?id=646159.680038 [last accessed April 2014].
57. Rashid A, Chitchyan R. Persistence as an aspect. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, ACM, New York, NY, USA, 2003; 120–129.
58. Al-Mansari M, Hanenberg S, Unland R. Orthogonal persistence and AOP: A balancing act. *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '07)*, ACM, New York, NY, USA, 2007.
59. Soares S, Borba P. Towards reusable and modular aspect-oriented concurrency control. *Proceedings of the Towards ACM Symposium on Applied Computing (SAC '07)*, ACM, New York, NY, USA, 2007; 1293–1294.
60. Ren X, Shah F, Tip F, Ryder BG, Chesley O. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Not* 2004; **39**(10):432–448.
61. Rashid A. Sades - a semi-autonomous database evolution system. *Workshop ion on Object-Oriented Technology (ECOOP '98)*, Springer-Verlag, London, UK, 1998; 24–25.
62. Al-Mansari M, Hanenberg S, Unland R. On to formal semantics for path expression pointcuts. *Proceedings of the ACM Symposium on Applied Computing (SAC '08)*, ACM, New York, NY, USA, 2008; 271–275.
63. Soares S, Borba P, Laureano E. Distribution and persistence as aspects. *Software: Practice and Experience* 2006; **36**(7):711–759.
64. Rashid A. On to aspect persistence. In *Generative and Component-Based Software Engineering*, vol. 2177, Butler G, Jarzabek S (eds)., Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2001; 26–36.
65. Rashid A, Pulvermueller E. From object-oriented to aspect-oriented databases. *Proceedings of the 11th International Conference on Database and Expert Systems Applications (DEXA '00)*, Springer-Verlag, London, UK, 2000; 125–134. Available from: http://dl.acm.org/citation.cfm?id=648313.755689 [last accessed April 2014].
66. Rashid A, Sawyer P. Dynamic relationships in object oriented databases: A uniform approach. In *Database and Expert Systems Applications*, vol. 1677, Bench-Capon T, Soda G, Tjoa A (eds)., Lecture Notes in Computer Science. Springer: Berlin / Heidelberg, 1999; 816–816.

DOI: 10.1002/spe