Database Evolution on an Orthogonal Persistent Programming System

A Semi-Transparent Approach

Rui Humberto R. Pereira

Instituto Superior Contabilidade e Administração do Porto
ISCAP/IPP
Porto, Portugal
rhp@iscap.ipp.pt

J.Baltasar García Perez-Schofield

Departamento de Informática
Universidad de Vigo
Vigo, España
jbgarcia@uvigo.es

Abstract **— In this paper the problem of the evolution of an object-oriented database in the context of orthogonal persistent programming systems is addressed. We have observed two characteristics in that type of systems that offer particular conditions to implement the evolution in a semi-transparent fashion. That transparency can further be enhanced with the obliviousness provided by the Aspect-Oriented Programming techniques. Was conceived a meta-model and developed a prototype to test the feasibility of our approach. The system allows programs, written to a schema, access semi-transparently to data in other versions of the schema.**

Keywords: schema evolution; aspect-oriented programming; orthogonal persistence

I. INTRODUCTION

Applications that need to persist their data use a well defined data model, termed schema, which has a specific interface to all data types that pertain to that underlying data model. This formalism leads to severe constraints in the changing process of the information systems and organizations itself.

The schema of an application is not immutable due to several factors: The organizations processes are dynamic, consequently their requirements also are dynamic. People that analyses the organization requirements and build the applications make mistakes that must be corrected, sometimes later to the development process. Moreover, there are also the cases, when new features are added to the application, that require new changes at data model level. By all those reasons, frequent changes occur on the data structures, forcing to a constant redesign of the schema with an important impact on the application, with particular complexity on legacy ones.

Traditionally, the database evolution problem has been dealt in object-oriented systems with three different approaches:

- Schema evolution – The schema is simply converted from the old to the new.

- Schema versioning – The schema is maintained in several versions.

- Class versioning – Allows classes to have several versions.

Whatever the chosen approach, any schema state must preserve both structural and behavioral consistency. Zicari [1] identified these two types of consistencies and the anomalies that may occur on the system in their absence. The structural consistency means that the schema must obey to a set of invariants [2][3]. The loss of the structural consistency during a schema change leads to the unavailability of the objects created under the old schema. The behavioral consistency refers to the dynamic part of the schema. Each method must respect their signature and must be aware of all static changes. Otherwise, problems may occur during the normal operation of the application under the new schema.

In OODBMS, schema changes are applied through a set of available primitives provided by the system. Several authors have proposed different taxonomies [2] [3] of modifications to their system schemas. Depending on the evolution approach, changes on the application interface may occur requiring the programmer intervention.

The orthogonal persistence [4] is characterized by three principles: (1) Type orthogonality - All objects can be persistent or transient irrespective of their types, sizes or any other property. (2) Persistence Identification – The form of identifying and persisting object is orthogonal, i.e., all objects are available from one, or more, common root, and all that is related to it, are accessed on the same way. (3) Persistence independence - The same code should be applicable for both transient objects and persistence objects. Due to these three principles, in orthogonal persistent programming systems applications are closely coupled with object database. On the other hand, in orthogonal persistent programming systems the schema is embedded in the application code. We argue that this paradigm of data persistence offers particular conditions to the development of new approaches of database evolution.

The aspect-oriented programming (AOP) [5] consists in a programming technique that allows the separation of application concerns. In an object oriented context, a concern that is transversal to all objects could be segregated from those objects and put in a specialized object called Aspect, while the

remaining concerns, that are specific from each object, will be still maintained in the object class. Applications and programmers are oblivious about them. The authors of other research works [6] have concluded that schema evolution and the instance adaptation are two concerns of the database evolution that could be modularized in aspects. In our research work we explore the AOP techniques to implement the database evolution concerns in an orthogonal persistent programming environment.

In the next section we will debate the particularities of orthogonal persistent programming systems and how they can be applied on the database evolution. Then, we briefly present our research prototype and meta-model giving emphasis to the schema evolution and instance adaptation problem. In the section IV other related works are presented and discussed. Finally, the future work and conclusions are presented.

II. SCHEMA EVOLUTION IN PERSISTENT ENVIRONMENTS

Applications have embedded in their code the schema. In an object-oriented application, that runs in an orthogonal persistent programming environment, that schema is propagated to the database after the first start. Besides, the closely coupled approach between this type of application and the database, while interacting with the data, can turn that schema propagation in a transparent mechanism. These two characteristics, which have been observed in that kind of applications, shape their schema evolution problem. The orthogonal persistent environment is capable of dealing with the first appearance of an object in order to represent them in the database. We consider this obliviousness in the initial schema propagation to the database metadata layer a simple problem, since the system reflection capability could provide the required information about the persistent objects. However, the same does not happen when that schema, shared by the application run-time, as well the database, is changed by the programmer in the application source code. The structural and semantic differences between the two schema versions compromise the application compatibility leading to its inoperance. Those differences cannot be transparently inferred by the system, requiring additional information provided by the programmer.

Considering the above two characteristics observed in orthogonal persistent programming systems, we can conclude that in this kind of systems the schema evolution happens in the application and it is propagated to the database in a semi-transparent way. That contrasts with other systems where application and database are decoupled components. In such systems, the schema application evolves and simultaneous a set of schema evolution primitives must be applied to the database in order to reconcile these components.

In orthogonal persistent programming environments, the structural and behavioral consistency of the schema embedded in the application, which is propagated to the metadata layer of the database, is validated at compile-time by the compiler of the programming language. This fact, grants a consistent representation of the schema in the database. However, when the schema is changed on the application there are then two versions of the schema: the new on the application, replicated to the database, and the old one yet on the database. All three schema evolution approaches previously presented can be accommodated on this situation in order to solve the presented problem.

Relatively to the other part of the evolution of the database, instance adaptation, no relevant aspect has been identified that characterizes orthogonal systems in contrast to other kind of systems. The objects created under the older schema version must be adapted to the application interface by the system using conversion functions.

III. PROPOSED APPROACH

The developed system [7][8][9] is an aspect-oriented framework capable of providing a Java application with persistence services. In this research prototype we explore the AOP techniques to inject the required persistence behavior in the objects of an application. Both application and programmer stay oblivious [10] about the persistence concern. We argue that our approach applied in our system provides programmers with a powerful tool to an efficient and error prone application development. A full description about the system persistence capabilities can be found in [7][8][9].

Our last research work focused the database evolution as a part of the persistence concern. Thus, we have extended our system with schema evolution and instance adaptation facilities. Despite we use an object-oriented database, the DB4Objects [11], that already provides some schema evolution mechanisms, we have reduced its role to a simple object repository. To support the evolution of the schema in our system, we conceive a meta-model to support the database meta-objects that compose his metadata layer and the application objects. The main goal of our work is to explore the aspect-orientation of database evolution aspects in order to obtain a transparent, pluggable and flexible system.

A. Meta-model

Our meta-model and prototype apply the class versioning approach due to the following reasons: versioning allows both backdate and update application compatibility while class versioning has a better granularity than schema versioning.

Our meta-model has three types of objects: (1) objects, (2) meta-objects, (3) meta-classes. The objects are the application entities. These objects are instances of the versioned classes represented in our model through a meta-object. In turn, the meta-objects are instances of meta-classes. In order to achieve a simple and efficient model, we have defined a small set of meta-classes which are built-in our system. We consider the model simplicity a key issue to performance, since it avoids meta-objects updates and reads. The Class Version Meta-Object (CVMO) supports each class in his version. If a class has several versions, there are as many CVMOs as the number of versions. The Instance Meta-Object (IMO) supports the object instances in all versions. To represent the objects relationships we use Attribute Meta-Objects and Root Meta-Objects (RMO). The former was defined under the assumption that objects refer other objects through their attributes. The RMOs, are the root object database start points to all remaining objects, where each one supports a *root object* [4]. The

following diagram exemplifies the use of these meta-objects in the scenario of a 1-n relationship.
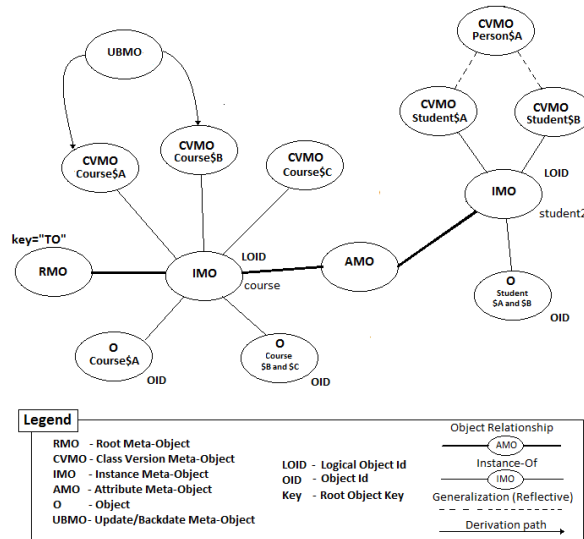

Figure 1 - Meta-model

The meta-model supports two types of class relationships: The class inherence lattice and the class version derivation path. To reduce the resulting complexity of the combination of those two class dimensions, we explore the reflective capabilities of the programming language to obtain the class inherence lattice. The version derivation path can reach a high level of complexity. That information is relevant only to support the semantic differences among the classes. If the instance conversion process could be supported only by default conversion functions, that derivation path information is useless. In our approach, we take advantage of the data schema that is embedded in the application and the closely coupled interaction between application and the database system manager. At run-time, at the first appearance of a new class, the schema manager in the framework core detects that new class, or a new version of an existing class, proceeding to the creation of the corresponding CVMO. An integral copy of the class is saved in that meta-object. If the class has some super-class, that same work is done recursively until the root of the inherence lattice is reached. This procedure grants a full representation of the schema in both dimensions: inherence lattice and version derivation path.

All information about data domain integrity could be defined in Java annotations [12] which are preserved in the CVMO and accessible through reflection. In Java, and other modern object-oriented languages, the referential integrity information is maintained at run-time through the objects attributes. Our approach transparently propagates the objects relationships to the database preserving that information. All references to objects are deleted in the run-time and then on the database. The memory garbage collector removes effectively from the memory the unreferenced objects. In our system we have implemented a DB garbage collector that does the same tasks inside the database.

To support those semantic differences we have conceived the Update/Backdate Meta-Object (UBMO) that implements the conversion function over two or more class versions.

Finally, the Aspect Meta-Object (AspMO) supports AOP aspects. These aspects could be application aspects or DBMS aspects. In the first case, our meta-model allows that aspects could be persistent in the same way that objects can. In our approach, the aspects and objects are distinct entities in the database until the moment that they are instantiated to be delivered to the application. In that moment, the database weaver merges object and aspect. This approach provides applications with an integrated and consistent aspect-oriented environment. On the other hand, there are several database management concerns that could also be modularized as aspects. That DBMS aspects can turn the system highly pluggable and customizable [13].

The object identity has two layer identifiers. The Logical Object Identifier (LOID) identifies an object despites his version. In turn, the Object Identifier (OID) has the real reference to the stored object. Since it may be necessary to preserve more than one version object, an application object has a LOID and may have more than one OID. The root objects are identified by their keys, an arbitrary text string.

B. The aspect-oriented framework

The developed system implements the meta-model previously presented, allowing parallel versions of the schema. Application with different versions of the schema can run sharing the same data.

The system is composed by four components:

- Preprocessor – This component is optional allowing some dynamic typing features that are not compliant with standard Java compiler rules. More information about this module can be found here [8].

- Framework Core – In this component is implemented the interface used by the application through the `CPersistentRoot` class and all basic services like object cache, classloader, DB garbage collector, persistence manager, schema manager and default instance adaptation code.

- Aspects – There are three types of aspects: At the application, management system and database level. The former are the application persistence aspect and data integrity aspect. These aspects use the framework core services. The second types are the management aspects and pluggable modules that could be developed. The management aspects currently implemented are the storage aspect that implements the low level object persistence, the instance adaptation aspect and the DB integrity aspect. At the third level we have plans to store specific application aspects inside the AspMO meta-objects.

- Database – It is the object and meta-object repository.

C. Schema evolution

The two previous characteristics referred above, observed in orthogonal persistent programming systems (for instance, the system presented here), are able to free the programmer, under certain conditions, of replicating the schema changes in the database. That is, when the changes obey a set of invariants, the default conversion functions can deal with the instance adaptation avoiding any additional information. In this scenario our system transparently replicates the new version of the schema to the database.

1) Invariants for a transparent schema evolution
- The name of the class must be the same.

- The name of the attribute must be the same.

- The type of an attribute can be modified but to a compatible type.

- The semantic of the attribute must be preserved.

- An attribute can be moved to a super-class or to a sub class, but must maintain the name. In this case, additionally the type can also be altered to a compatible type.

Other changes to the schema can be freely applied: A new class, the drop of a class or attribute. For the new attributes, the programmer can introduce directly in the application code an annotation that defines the default value of that attribute in each version.

2) Additional schema information
Our goal is to make the schema evolution as transparent as possible. To achieve that goal we have defined a set of special annotations that can complete the schema information directly in the application code. Currently, we are developing those defined annotations:

```
@Aof4oopConstraintCheck(expression)
@Aof4oopConstraintNotNull(message)
@Aof4oopDefault(value)
@Aof4oopVersionAlias(alias)
```

The following listing gives an example of use of these annotations.

```
1   @Aof4oopVersionAlias(alias = "A")
2   class Person
3   {
4       @Aof4oopConstraintNotNull("Invalid name")
5       private String name;
6       @Aof4oopConstraintCheck(expression = "age>0")
7       private int age;
8       @Aof4oopConstraintCheck(expression = "weight>0")
9       private float weight;
10  }
11  @Aof4oopVersionAlias(alias = "B")
12  class Student extends Person
13  {
14      private int studentNumber;
15      @Aof4oopConstraintNotNull(message="Invalid Type")
16      @Aof4oopDefault(value = "Ordinary")
17      private String studentType;
18  }
```

Figure 2 - Class example

The system is capable of automatically define the name of the versioned class, through a hash function, however, the annotations in lines 1 and 11 defines alias to their versions facilitating the future class version identification as showed in Figure 3. The annotations in lines 4, 6, 8 and 15, define the domain integrity policies.

In the example, we also can see the version "B" of the class `Student` that has a new attribute called `studentType`. In the previous version "A" of the class, this attribute does not exist. The annotation, in line 16, defines the default value of this attribute when objects instances are converted to that class version. This type of annotation extends the system default conversion functions with some user defined policies. When they are not enough, the programmer can introduce user defined functions in the specialized meta-objects UBMO of our meta-model. In the next listing we show an example of a semantic conversion of the person weight from kilograms in old version to pounds in the new.

```
Person$B convert(Person$A old)
{
    Person$B self=new Person$B();
    self.setName(old.getName());
    self.setWeight(old.getWeight()*2.20462262);
    return self;
}
```

Figure 3 - User defined function

These update/backdate user defined functions can be simple or complex [14].

3) Class version identification
From the point of view of an application the classes have only one version, their own. However, the user defined conversion functions must see any version of the class. Both code, application and conversion function, run inside the same environment (the virtual machine). To allow this simultaneous existence of all class versions, the system schema manager creates a copy of each version of the class. The class `Person` in version A is `Person$A`. In the framework scope only those classes are used. Before the object delivery to the application, it is renamed to the known name. The application classes are available in their classpath (usually the file system) while the versioned classes are obtained from the database, inside the CVMO meta-objects, through the special framework classloader.

D. Instance adaptation

The objects resident in the repository are created under a schema version. However, they must be available to any application in their schema. Using the historical information about the class, the framework emulates the required class version by the application, even its current version. That is, in the object repository the objects are stored in their renamed version. An application that uses a class `Person` on version A has their objects stored as `Person$A`. This approach in fact requires the emulation in all operations of reading or writing. Yet, the approach allows the coexistence of the same class in more than one version in the run-time environment accessible by the conversion function.

On write operations the classes are converted to version of the corresponding class version of the application. This lazy conversion puts the class in most pertinent version accelerating the emulation [15] process requiring only the renaming of the object class. The instance adaptation was implemented as an aspect in terms of AOP. This means that a future alternative approach can easily replace this one, turning the framework into a highly customizable system.

The used emulation process suffers of the same problem that screening [16] technique requiring in some cases the maintaining of more than one object copy. In the Figure 1 we illustrate this scenario with a class `Course` that exists in three versions and requires the maintenance of the version A and one of the other two versions, B or C, since any these last two can be generated from each other.

IV. RELATED WORK

The PJama System [17][18][19] is an experimental persistent programming, like our system, based in the Java language programming and in same principles of orthogonal persistence and reachability. To support persistence, the standard Java Virtual Machine (JVM) was modified. In this point, PJama contrasts with ours since we do not apply any change to the Java platform. The fact of the system being aspect-oriented enabled the access to the strategic joint point of the system, thus avoiding any JVM modification.

The PJama system does only support one schema versioning process at a time, while the application is offline. The schema evolution occurs outside the store in the application. After that, the programmer rebuilds the application and passes a list of the changed classes to an evolution tool that performs the adaptation of the existing data. Once again, differs from our system since it does not transparently detect the new versions of the class nor produces all required metadata or does not perform any instance adaptation.

The concept of default and custom conversion is also present in this system. On custom conversion, the programmer must write their own conversion functions by following a convention of method signature.

This research work addresses the development of a persistent store coupled to the Java virtual machine while our work focus the modularization of the application persistence concern and database evolution concerns in AOP aspects.

In SADES [20] system the concept of aspect-oriented database [6] was tested. The authors of this research work have identified a set of database concerns that could be modularized in order to obtain a customizable and pluggable system. They applied AOP techniques to implement database evolution concerns in the SADES system and later in AspOEv [13].

The main focus of this research work was to demonstrate that the aspect-orientation of the database provides means of, in each case, apply the most adjusted strategies of schema evolution or instance adaptation, contrasting with other proposals where those strategies are fixed.

To support schema evolution and instance adaptation concerns, the authors of those systems have proposed a meta-model based in the same three types of entities that we now also propose. The main difference between this older proposal and the one presented here is focused on the type of meta-objects and on what both models try to represent. Rashid et al [21] meta-model represents the application data structures. A meta-object can be a class, a member or even a parameter. In our opinion, on a multi-versioned schema the resulting complexity of this model is huge. In this meta-model, the meta-objects are separated in different *virtual spaces* so that object relationships can be *inter* or *intra space*. One innovative type of meta-object in this approach is the aspect meta-object that represents an application entity aspect or a database aspect. These aspects can also persist to the several application executions. Our meta-model also has this type of meta-object in which we have been inspired.

The AspOEv framework provides its own language, Vejal [22], which is Java enriched with versioned type semantics as well as aspect-oriented features. This language consists in a powerful tool to construct user defined conversion functions. Vejal integrates aspect capabilities within its versioned type system. However, we argue that our approach, based in renamed classes, provides the same semantics richness while avoids the introduction of a new language. On the other hand, Vejal must be interpreted while in our system we use native code.

Kuppuswami et al [23], also explored the AOP techniques proposing a flexible instance adaptation approach. In this work, the authors developed a system that supports instance adaptation with two aspects: (1) The Update/backdate Aspects that implements the concern of emulation of object versioning. The object is retrieved from the store in their physical version and then it is converted to the expected interface of the application. (2) The second aspect is a Selective Lazy Conversion Aspect. This aspect is responsible for physically converting the stored objects when they are accessed and the need of conversion is determined. This aspect operates the conversion under a deferred approach to avoid the disturbance on the normal system functioning. On the other hand, this aspect is selective about the instances to be converted. In some cases, when necessary, the physical conversion occurs and on other cases the objects are kept indefinitely in their version. For any object instance that remains in an older version, its structure and behavior is emulated with the former aspect. In this work, the flexibility provided the evolution concerns as database aspects, is specially highlighted by the authors. They conclude that the encapsulation of the concerns in aspects enables the easy replacement of the adaptation strategy and code, contrasting with other existing systems that introduce code directly on the class versions.

V. FUTURE WORK

The proposed meta-model in the context of the pool of experiments has responded satisfactory, but should be object of more research work to accommodate static attributes, all types of composite objects and relationships complexity.

Although our meta-model previews the existence of aspect meta-objects, the current version of our system does not have that implemented. We are planning the development of a new module, a run-time weaver, that introduces the aspect code in the objects and meta-objects.

Our system was not developed with performance in mind nor has any transaction mechanism or error handling, either. The transaction management will be the subject of a future line of research.

The current DB classloader module allows transparent access from the java virtual machine to all version classes through its name variation. It would also be interesting to develop an integration plug-in with any development tool (like Eclipse or Netbeans) allowing the inclusion of those class versions resident in the database as a special classpath.

VI. CONCLUSIONS

The developed system offers a substantial level of transparency and obliviousness on the evolution of the database. Under certain conditions, the programmer can freely introduce changes directly on the embedded application schema which are automatically reflected on the database metadata. Besides, the existing data on the object repository is transparently adapted to that new schema version or any other. Our approach uses standard language mechanisms to extend the class definition with additional information that enhances the efficiency of the default conversion functions. In many situations the need of any intervention on the database is totally avoided with the introduction of those annotations associated to the classes. Other annotations can also reinforce the data domain integrity in their schema version.

When a semantic change occurs on attributes or classes, those annotations are not enough. In these cases, we have previewed in our meta-model a special type of meta-object introduced manually by the programmer that contains the user defined conversion function. As result, the system is a semi-transparent database evolution system.

The proposed meta-model allows the existence of parallel versions of the schema granting backdate and update application compatibility. Comparatively with other systems, our meta-model uses a small set of meta-classes built-in in the system. Some of the meta-models introduced in the related work section are very detailed, providing a full description of the underlying data model. However, this level of detail could theoretically introduce serious performance penalties because of the need of maintaining a high number of metadata items. The meta-model presented here uses the information about classes in the application and the meta-objects in the database metadata layer. The main mission of these few meta-objects is to provide the system with historical information about other versions of the schema (past or future).

The characteristics defining orthogonal persistent programming systems, discussed in the introduction section, are central to the inception of the meta-model and framework presented here. On the other hand, the use the AOP techniques took advantages of the seamless integration between application and database. Thus, the modularization of the database evolution concern has facilitated avoiding any modification on the virtual machine.

The meta-model and framework are very dependent of the class versioning approach for schema evolution. However, we do not consider that inflexibility a disadvantage. This schema evolution approach comparatively with schema versioning has a better granularity.

REFERENCES

[1] Zicari, R.. A framework for schema updates in an object-oriented database system. In *Data engineering, 1991. proceedings. seventh international conference on*. 1991.

[2] Banerjee, Jay;Chou, Hong-Tai;Garza, Jorge F.;Kim, Won;Woelk, Darrell;Ballou, Nat;Kim, Hyoung-Joo. Data model issues for object-oriented applications. ACM Trans. Inf. Syst. (1987) 5: pp. 3-26.

[3] Ferrandina, Fabrizio;Meyer, Thorsten;Zicari, Roberto;Ferran, Guy;Madec, Joelle. Schema and Database Evolution in the O2 Object Database System. In *Proceedings of the 21th international conference on very large data bases*. 1995.

[4] Atkinson, Malcolm;Morrison, Ronald. Orthogonally persistent object systems. The VLDB Journal (1995) 4: pp. 319-402.

[5] Kiczales, Gregor;Lamping, John;Mendhekar, Anurag;Maeda, Chris;Lopes, Cristina;Loingtier, Jean-marc;Irwin, John. Aspect-oriented programming. In *Ecoop*. 1997.

[6] Rashid, Awais;Pulvermueller, Elke. From Object-Oriented to Aspect-Oriented Databases. In *Proceedings of the 11th international conference on database and expert systems applications*. 2000.

[7] Pereira, Rui Humberto;Perez-Schofield, J.B.G.. An aspect-oriented framework for orthogonal persistence. In *Information systems and technologies (cisti), 2010 5th iberian conference*. 2010.

[8] Pereira, Rui Humberto;Perez-Schofield, J B G. Orthogonal persistence in Java supported by Aspect- Oriented Programming and Reflection. In *Information systems and technologies (cisti), 2011 6th iberian conference*. 2011.

[9] http://www.iscap.ipp.pt/~rhp/aof4oop

[10] Filman, R.;Friedman, D.. Aspect-Oriented Programming is Quantification and Obliviousness. (2000) : .

[11] Paterson, Jim;Edlich, Stefan;Hörning, Henrik;Hörning , Reidar. The Definitive Guide to db4o. . Apress, 2006.

[12] Bloch, Joshua. JSR 175: A Metadata Facility for the Java Programming Language. (2004) : .

[13] Rashid, Awais;Leidenfrost, Nicholas A.. Supporting Flexible Object Database Evolution with Aspects.. In *Gpce*. 2004.

[14] Ferrandina, Fabrizio;Meyer, Thorsten;Zicari, Roberto. Implementing Lazy Database Updates for an Object Database System. In *Proceedings of the 20th international conference on very large data bases*. 1994.

[15] Clamen, Stewart M.. Type Evolution and Instance Adaptation. (1992) : .

[16] Banerjee, Jay;Kim, Won;Kim, Hyoung-Joo;Korth, Henry F.. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the 1987 acm sigmod international conference on management of data*. 1987.

[17] Dmitriev, Misha. The First Experience of Class Evolution Support in PJama. In *The third international workshop on persistence and java*. 1998.

[18] Dmitriev, M;Atkinson, M. Evolutionary Data Conversion in the PJama Persistent Language. In *In 1st ecoop workshop on objectoriented databases*. 1999.

[19] Dmitriev, Misha;Hamilton, Craig. Towards Scalable and Recoverable Object Evolution for the PJama Persistent Platform. In *Proceedings of the international symposium on objects and databases*. 2001.

[20] Rashid, Awais. SADES - a Semi-Autonomous Database Evolution System. In *Workshop ion on object-oriented technology*. 1998.

[21] Rashid, Awais;Sawyer, Peter. Dynamic Relationships in Object Oriented Databases: A Uniform Approach. In Lecture Notes in Computer Science. Bench-Capon, Trevor and Soda, Giovanni and Tjoa, A. Vol. 1677.1999.

[22] Rashid, Awais;Leidenfrost, Nicholas. VEJAL-An Aspect Language for Versioned Type Evolution in Object Databases. (2006) : .

[23] Kusspuswami, S.;Palanivel, K.;Amouda, V.. Applying Aspect-Oriented Approach for Instance Adaptation for Object-Oriented Databases. In *Proceedings of the 15th international conference on advanced computing and communications*. 2007.