Towards a flexible and transparent database evolution

Rui Humberto Pereira[1] and J.Baltasar García Perez-Schofield[2]

[1] Instituto Politécnico do Porto, ISCAP, Portugal,
`rhp@iscap.ipp.pt`
[2] Universidad de Vigo, Departamento de Informática, España,
`jbgarcia@uvigo.es`

Abstract. Applications refactorings that imply the schema evolution are common activities in programming practices. Although modern object-oriented databases provide transparent schema evolution mechanisms, those refactorings continue to be time consuming tasks for programmers. In this paper we address this problem with a novel approach based on aspect-oriented programming and orthogonal persistence paradigms, as well as our meta-model.
An overview of our framework is presented. This framework, a prototype based on that approach, provides applications with aspects of persistence and database evolution. It also provides a new pointcut/advice language that enables the modularization of the instance adaptation crosscutting concern of classes, which were subject to a schema evolution.
We also present an application that relies on our framework. This application was developed without any concern regarding persistence and database evolution. However, its data is recovered in each execution, as well as objects, in previous schema versions, remain available, transparently, by means of our framework.

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

1 Introduction

Applications refactoring is a common activity in programming practices. These practices target the applications' maintenance in order to correct design or implementation mistakes, or, yet, prolong its life cycle introducing new functionalities. Many of them imply updates in the data model of persistent data, requiring a double intervention in the application source code and database. Modern object-oriented databases enable application's objects to be made persistent without a previous schema definition, alleviating that problem. The db4o (http://www.db4o.com), Versant (http://developer.versant.com) and ObjectDB (http://www.objectdb.com) systems provide transparent schema evolution mechanisms. In these systems, some types of updates in object's structure are propagated to database without the intervention of the programmer. However, in these systems, complex schema updates such as class movements in the inheritance hierarchy, field renaming and semantic changes in the content of the field, require the programmer's intervention in order to convert the data from the old structure to the new one. Many of these complex schema updates require helper conversion applications. Thus, the schema updates continue to be time-consuming tasks that programmers are faced with, compromising the programmers' productivity

The aspect-oriented programming (AOP) techniques aim at the modularization of applications' crosscutting concerns, such as logging, auditing and persistence. Earlier research [1][2] works have demonstrated that AOP also enables flexible and pluggable approaches to supporting database crosscutting concerns, such as schema evolution and instance adaptation. However, these works were not totally concerned about the transparency of the whole evolution process from the point of view of the programmer. We argue that the combination of orthogonal persistence and AOP paradigms provide good means to improve programmers productivity, as well as the ease of modification of the application, and therefore its lifetime.

In this paper we provide a short overview of our prototype, the AOF4OOP framework [3][4][5]. This framework supports orthogonal persistence and database evolution following an aspect-oriented paradigm. Applications that rely on our framework are practically oblivious regarding its persistence, as well as the database evolution when changes are applied to its schema. Programmers by means of our pointcut/advice expressions can introduce additional behaviour into the system in order to deal with database evolution. These aspects of persistence are introduced in applications by means of AOP techniques.

We also discuss a case study that demonstrate how our framework prototype can improve the programmer's work in terms of quality and productivity. It is based on a geographical application that, initially, was developed without any concern regarding persistence. In a second development phase, we provide this application with the aspect of orthogonal persistence applying minor changes to its source code. Finally, we evolved its data model in order to accommodate new functionalities. Using our pointcut/advice expressions, the database was adapted, the entire process being transparent to the application in its new database schema version, as well as to the old application, in the initial schema version.

In next section we briefly discuss our framework and pointcut/advice expressions. Section 3 presents a case study of an application, whose persistence aspects were modularized by means of our framework. In Section 4 we present some related studies. Finally, we present our conclusions and address some future work in our research.

2 Framework Overview

The persistence and database evolution are, respectively, aspects of applications and databases. Regarding persistence modularization, in earlier works [3][4][5], we have presented our meta-model and prototype. In these works we discuss how the orthogonal persistence aspect is modularized in applications that rely on our framework. Our meta-model supports database multi-version schemas by means of class versioning based on a class renaming strategy. Thus, several versions of the same application can share one logical schema database. Due to space restrictions, we cannot perform a complete discussion of our meta-model and the implementation of the framework.

Our framework[3] follows the orthogonal persistence paradigm[6]. The paradigm's principles advocate that objects must be handled in the same manner, despite of its state persistence (persistent or transient), and reachable transitively. We argue that these principles, due to data access transparency and orthogonality, provide means for applications to be *oblivious* [7] regarding persistence aspects, when applying AOP techniques. Furthermore, orthogonal persistence also enables application's schema to be incrementally [6] propagated to database. Our multi-version database schema approach, enabled by our meta-model, is able to accommodate each new class version[5]. Thus, applications that rely on our framework can be developed without major concerns for persistence, schema evolution and instance adaptation. In the next sections we discuss our join point model, as well as how our weaving process enables such *obliviousness* [7]. In Section 3 we present an application, whose persistence aspects are provided by our framework.

2.1 Aspects

In AOP paradigm, pointcuts *quantify* [7] a set of join points where the code in a base program should be affected. By applying pointcut expressions in these join points additional behaviour can be activated through an *advice* that could take place before, after or around the join point. A set of pointcuts and their corresponding advices are collected in a module called *aspect* [8]. In our framework, two kind of aspects were modularized: application and framework/database. The application's aspects were: (1) persistence and (2) data integrity. And, in the framework/database scope, the modularized aspects were: (1) schema evolution, (2) instance adaptation, (3) database data integrity, (4) object storing, (5) system statistics and (6) system debugging. These aspects were constructed by means of the AspectJ [9] programming language.

Regarding the instance adaptation, this is a crosscutting concern of the classes subject to a schema evolution. Due to the requirements posed by our meta-model, that require a persistent definition of the instance adaptation aspect, we developed a database weaver and a new XML based pointcut/advice language. Using this XML based language, programmers can write the instance adaptation aspect, which is weaved at runtime by our dynamic weaver. Conceptually, the default instance adaptation aspect, provided by framework's main module [5], is extended by means of these pointcut/advice expressions. Furthermore, this aspect extension is reified as a collection UBMO meta-objects [5].

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

2.2 Pointcut/advice expressions

We developed a new kind of pointcut/advice expressions that follows the *quantification* definition posited by Filman and Friedman[7].

"AOP is thus the desire to make programming statements of the form In programs P, whenever condition C arises, perform action A."

The conditions to trigger the action are specified through the following matching parameters:

`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 advised object must be an array of this class.

`matchSuperClassName -` Super class canonical name of the advised classes. Matching through a super class is meant to reduce the number of user definitions. If many target classes that share a common super class exist, all of them are advised. This parameter allows the * wildcard.

`matchOldClassVersion -` The class version of the persistent object at database being emulated/converted. This parameter allows the * wildcard.

`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 grant a correct target application version. This parameter allows the * wildcard.

`matchParentClassName -` This contextual parameter enables advising objects which at runtime are pointed as from a certain class. In our application (see Section 3) this parameter advises the `Coordinate` objects which are associated to a `Area` class. This parameter allows the * wildcard and `[]`.

`matchParentMember -` This is another contextual parameter that enables advising objects which at runtime are pointed as from a certain object member. In our application this parameter advises `Coordinate` objects whose reference is in the `bounds` member.

The action is defined in native programming language. Additionally, some parameters can complement the conversion process definition.

`applyDefault -` This parameter, when true, applies the default conversion behaviour, alleviating the programmers' effort in order to write this action (the body's advice). Note that, in many cases just 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 return that superclass.

`conversionClassName -` This is an optional parameter that forces the name of the weaved class which implements the *advice*. It is useful just for debugging purposes. Thus, knowing that name, in case of any runtime error, the programmer can understand the localization of that error.

Figures 4 and 5 present examples of these expressions. They will be discussed in Section 3.4.

Our approach, in order to reify the instance adaptation aspect inside the database, is supported by an XML language. This approach presents two advantages: (1) Enables an easier edition of this aspect - Using a special graphical tool, as well as using a simple text editor. (2) The language extensibility provided by XML - New features can be added to our language without compromising the existing ones.

We also note that our approach does not require another programming language like Vegal [1]. Thus, the *advice* can be written in the same base program's programming language.

2.3 Conversion and database dynamic weaving

The weaver is the central element of any aspect-oriented tool. According the pointcut/advice expressions content, it inserts new behaviour or replaces the one existing in the base program. Our database dynamic weaver takes this approach, extending the framework's default behaviour.

The framework's default instance adaptation behaviour already is capable of performing conversion, in many cases using our *direct mapping* algorithm. This mechanism supports all schema updates that can be autonomously inferred. In our prototype, the *direct mapping* deals with field changes for compatible types (e.g. `int` to `long`, `float` to `String`), movement of fields across the class inheritance structure, field removing and new fields initialized at zero or `null`. When semantic or structural updates occur in the schema, a *user-defined conversion* is required. This instance adaptation aspect, which is stored at the database as a collection of UBMO meta-objects, is weaved dynamically at runtime by our database weaver extending the default one. Inside the framework's main module, when required, the user-defined conversion code is called through the `doConversion()` system call. At runtime, our weaver compiles the *advice* source code (inside the `<Conversion/>` node) and loads a special class that implements that system call. The following example presents this function instantiated for a `Product` object in version "A" being converted to the actual one.

```
Product doConversion(Product$A oldObj,Category$A oldObjParent,Product newObj,Category newObjParent)
{
     newObj.setWeight(2.2046*oldObj.getWeight());
     return newObj;
}
```

This function receives four arguments. Their names are reserved words, which are references to the old object (in version `"A"`), the object being converted (in the current schema version) and their parent objects (in the context of the conversion). The former provides data about the object in its old form, in version `"A"`. This object pertains to that class version renamed with a suffix `$A`, having exactly the same structure (see [5]). The `newObj` provides a pre-initialized new object, or a pre-converted by means of the *direct mapping* mechanism. This option is defined in the parameter `applyDefault`. The other two provide non-local data, which improves the richness of the user-defined conversion functions. In this example, the products' weight data at the database (in version `"A"`) is represented as pounds. However, the running application expects this data in kilograms. This conversion function receives a `Product` object pre-converted through its `newObj` argument. Thus, just the `weight` member needs to be converted, alleviating the programmer's effort. In this example, the product's category information is not needed for this conversion. We highlight the expressiveness of conversion code that is enabled by our approach: (1) full programming language features, (2) access to non-local data and (3) application classes behaviour can be reused in these functions.

3 Geographical Application

In order to demonstrate the benefits of this framework, in terms of productivity and quality, we developed a real application. This proof of concept application uses data obtained from the online OpenStreetMap[4] geographical database. This online tool allows exporting a user-defined area through its coordinates. The OSM export files, in XML format[5], contains all data related with that geographical area. Each contains the coordinates of the boundaries and it is structured as a set of objects such as *Nodes*, *Ways*, *Relations* and *Tags*.

Regarding to hardware and software, application and framework were tested in a personal computer with 4GB of RAM running Linux OS and Java 1.6.

3.1 User interface

Figure 1 depicts two screenshots of the application's user interface (UI): one presenting Madeira Island and another choosing an OSM file to be imported to the local database. This UI is organized as three panels: a menu on top, object browser on the left and maps on the right. The object browser enables the selective hiding of map objects such as points of interest and roads. The map and its objects are presented on the right side of the UI.

3.2 Data model

The application's data model follows the same structure as the OSM files, which is presented in Figure 2. An `Area` object corresponds to an imported area through an OSM file. First the file is imported as an `OSMFile` object. Then, its contents are copied to an `Area` object. Just `Area` objects are made persistent. Thus, each `Area` contains sets of `Node`, `Way` and `Relation` as arrays. Since a `Node` object can pertain to more than an `Area`, `Way` or `Relation`, they are shared, occupying just a unique slot in the database, and having a unique object identity.

[4] http://www.openstreetmap.org
[5] http://wiki.openstreetmap.org/wiki/OSM_XML

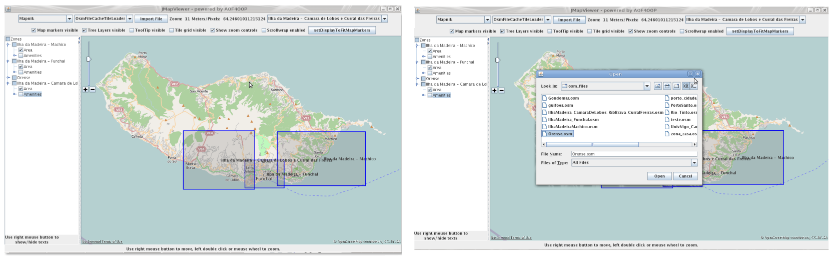


Fig. 1. User interface

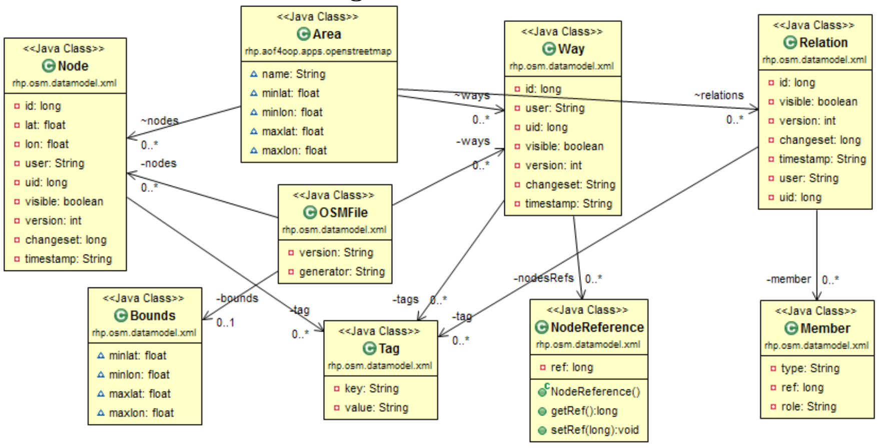


Fig. 2. Application's data model with class `Area` at version `"A"`

3.3 Modularizing data persistence

At the initial phase of the development of the application, we just imported map areas to memory without any concern regarding data persistence. Map areas were imported from an OSM file and then put in a `Hashtable` collection. Thus, during the application's execution time all the imported data remained available, but is lost in any further application's execution after its restart. The following Java code listing illustrates that first stage of development.

```
Hashtable<String,Area> areas=new Hashtable<String,Area>();       /*Non persistent application*/
Area importedArea=importFile(name,file.getAbsolutePath());
areas.put(area.calcKey(),area);
```

The next listing shows how our framework alleviates the programmer's effort in order to provide applications with the persistence concern.

```
CPersistentRoot psRoot=new CPersistentRoot();                         /*Persistent application*/
Area importedArea=importFile(name,file.getAbsolutePath());
psRoot.setRootObject(area.calcKey(),area);
```

We note the simplicity of the entire process. The framework's API provides a collection of persistent root objects following the same approach as the `Hashtable` collection. The `Area` objects, and its related objects, are made transitively persistent after being put in that persistent root. After that, any update in these objects, and any other that is reachable, are transparently reflected on the database.

3.4 Updating application's schema

In this section we discuss an update applied to application's data model. Before the update, each imported `Area` represents a rectangle in the map. After the update, the data model enables merging of areas forming a polygonal area. When the user imports an area in the map, it can be merged if it overlaps other existing areas. Thus, the class `Area` evolved from version `"A"` to `"B"`. Figures 2 and 3 depict the class `Area` in versions `"A"` and `"B"`, respectively. In version `"A"` class `Area` is a rectangle, just with a pair of coordinates, while in `"B"` it is an irregular polygon with several edges, requiring a collection of `Coordinate`.

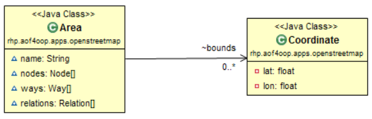
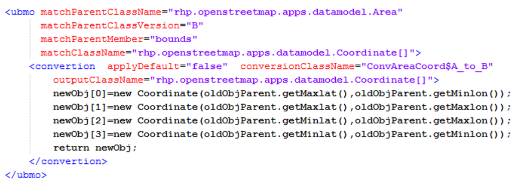


Fig. 3. Class `Area` in version `"B"`

Database schema evolution - As referred in Section 2, an incremental schema evolution is enabled [5] [6]. Thus, the entire process is transparent from the programmer's point of view. More information can be found in [5].

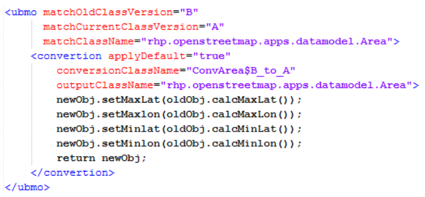
Database instance adaptation - The presented case, due to structural changes in class `Area`, requires user-definitions in order to enable the conversion of its instances in two directions: version `"A"` to `"B"` and `"B"` to `"A"`. Although the AOF4OOP framework enables bidirectional application compatibility, an `Area` defined as polygon in version `"B"` must be presented as a rectangle in version `"A"`. Thus, to enable that compatibility direction, the conversion procedure should produce a rectangle that includes the entire polygon or the major rectangle should fit inside the polygon. In version `"A"`, the `Area` edges coordinates are calculated from the polygon edges in version `"B"`. Figure 5 presents the required aspect definition.

In the opposite conversion direction, an array of coordinates that define the four edges of the polygon (rectangle) are required. In Figure 4 this interesting case is presented. The member `bounds` does not exist in version `"A"` of the `Area` class. The `matchParentMember` parameter enables the matching when this object member is accessed in version `"B"`. Inside the body's *advice*, the user-defined Java code produces the array of `Coordinate` using non-local information from `Area` object in version `"A"`.



```
<ubmo matchParentClassName="rhp.openstreetmap.apps.datamodel.Area"
      matchParentClassVersion="B"
      matchParentMember="bounds"
      matchClassName="rhp.openstreetmap.apps.datamodel.Coordinate[]">
    <convertion  applyDefault="false"  conversionClassName="ConvAreaCoord$A_to_B"
        outputClassName="rhp.openstreetmap.apps.datamodel.Coordinate[]">
        newObj[0]=new Coordinate(oldObjParent.getMaxlat(),oldObjParent.getMinlon());
        newObj[1]=new Coordinate(oldObjParent.getMaxlat(),oldObjParent.getMaxlon());
        newObj[2]=new Coordinate(oldObjParent.getMinlat(),oldObjParent.getMaxlon());
        newObj[3]=new Coordinate(oldObjParent.getMinlat(),oldObjParent.getMinlon());
        return newObj;
    </convertion>
</ubmo>
```

Fig. 4. Conversion from A to B



```
<ubmo matchOldClassVersion="B"
      matchCurrentClassVersion="A"
      matchClassName="rhp.openstreetmap.apps.datamodel.Area">
    <convertion applyDefault="true"
        conversionClassName="ConvArea$B_to_A"
        outputClassName="rhp.openstreetmap.apps.datamodel.Area">
        newObj.setMaxLat(oldObj.calcMaxLat());
        newObj.setMaxlon(oldObj.calcMaxLon());
        newObj.setMinlat(oldObj.calcMinLat());
        newObj.setMinlon(oldObj.calcMinlon());
        return newObj;
    </convertion>
</ubmo>
```

Fig. 5. Conversion from B to A

In both cases, as discussed in Section 2.3, at runtime, the user-defined conversion Java code is weaved as a `doConversion()` function. By means of our aspect-oriented approach, the right *advice* is triggered when a class needs to be converted. That takes place lazily only when objects are accessed. Then, that function is called at conversion-time, returning the converted object as applications expect. Thus, `Area` objects inside the database, in any version, are transparently accessed by applications regardless of the schema version.

4 Related work

Rashid and Leidenfrost [1] in the AspOEv system, an evolution framework, made experiences in order to customize the evolution in the commercially available Jasmine object-oriented database. Their framework supports flexible adaptation and introduction of evolution mechanisms enabling a dynamic evolution of the schema and adaptation of existing instances, governing an object database by means of AOP techniques. The framework's aspects support such flexibility by capturing crosscutting hot spots (customization points in the framework) and establishing their causality relationships with the custom evolution approaches.

The AspOEv framework employs its own application programming language, Vejal [1][10], an object-oriented language with a versioned type system. Vejal has a two-level versioning identifier system: `C<1>` indicates version 1 of class `C` and `C<s=1>` implies the class version of `C` that occurs in schema version 1. In Vejal, one version of a class may be present in multiple schema versions.

Kuppuswami *et al.* [2] have also explored AOP techniques proposing a flexible instance adaptation approach. In this work, the authors have developed a system that supports instance adaptation with two *aspects*: update/backdate and selective lazy conversion *aspects*. The authors also highlight the flexibility provided by the AOP techniques to support database evolution

These earlier works proved that concerns encapsulation as *aspects* enables the easy replacement of the adaptation strategy and code, contrasting with other existing systems that introduce code directly into the class versions. They were inspiring for us. On the other hand, we argue that our approach provides programmers with transparency along with that flexibility, due to orthogonal persistence and our aspect-oriented language for database evolution. Since it does not requires the knowledge of a new programming language and enables a total modularization of database evolution.

5 Conclusions and future work

We presented a short overview of our framework. Applications that rely on this framework can be developed without major concerns for persistence aspects. Applying AOP techniques, the framework provides applications with that additional behaviour. Regarding database evolution, the framework, due to its orthogonality and multi-version schema approach, also enables a transparent evolution of schema and object instances. Such transparency is achieved by means of the orthogonal persistence principles and AOP techniques.

We also discussed our aspect-oriented language that extends the framework's default instance adaption aspect. This language captures the application cross-cutting hot spots and establishes a relationship with user-defined conversion functions. Inside these functions, objects of a class in distinct versions can be handled through the same programming language used in the program's base.

The presented geographical application served as proof of concept, showing the gains in terms of productivity for programmers' work. Regarding this case study, we argue that our framework overcomes much of the existing limitations found in modern systems, in terms of transparency and flexibility.

Despite the benefits for programmers, our framework just provides a single and simple transaction model. Furthermore, classes that belong to the Java packages cannot be persistent due to JVM restrictions. These two issues are the subject of our current research.

References

1. Rashid, A., Leidenfrost, N.A.: Supporting flexible object database evolution with aspects. In Karsai, G., Visser, E., eds.: GPCE. Volume 3286 of Lecture Notes in Computer Science., Springer (2004) 75–94
2. 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, Washington, DC, USA, IEEE Computer Society (2007) 35–40
3. Pereira, R.H., Perez-Schofield, J.: An aspect-oriented framework for orthogonal persistence. In: Information Systems and Technologies (CISTI), 2010 5th Iberian Conference. (2010) 1 –6
4. Pereira, R.H., 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. (june 2011) 1 –6
5. Pereira, R.H., Perez-Schofield, J.B.G.: Database evolution on an orthogonal persistent programming system - a semi-transparent approach. In: Information Systems and Technologies (CISTI), 2012 7th Iberian Conference. (june 2012) 1 –6
6. Atkinson, M., Morrison, R.: Orthogonally persistent object systems. The VLDB Journal **4**(3) (1995) 319–402
7. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. Technical report (2000)
8. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Aksit, M., Matsuoka, S., eds.: ECOOP'97 Object-Oriented Programming. Volume 1241 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1997) 220–242
9. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: Proceedings of the 15th European Conference on Object-Oriented Programming. ECOOP '01, London, UK, UK, Springer-Verlag (2001) 327–353
10. Rashid, A., Leidenfrost, N.A.: Vejal: An aspect language for versioned type evolution in object databases. Workshop on Linking Aspect Technology and Evolution (held in conjunction with AOSD) (2006)