Weaving Aspects in a Persistent Environment

Awais Rashid

Computing Department, Lancaster University, Lancaster LA1 4YR, UK
awais@comp.lancs.ac.uk

Abstract. This paper discusses two mechanisms for weaving aspects in persistent environments founded on object-oriented databases. The first mechanism is based on exploiting existing aspect languages and their associated weavers while the second mechanism is based on building weaving functionality into the database management system (DBMS). The first mechanism has been used to integrate AspectJ and its associated weaver with the Jasmine ODBMS. The second approach has been used to implement a weaver within the SADES object database evolution system.

Keywords: Aspect-oriented programming, aspect weaver, persistent aspects, separation of concerns, object-oriented databases

1 Introduction

One of the prominent aspect-oriented programming mechanisms is the use of an aspect language and its associated weaver. An aspect language offers constructs – the *aspects* – to separate cross-cutting features from existing programming modules e.g. classes in OO languages[1]. It also facilitates specification of reference points – the *join points* – which identify links between the code encapsulated by the aspects and the classes cross-cut by this code. It also supports definition of behaviour to be executed with reference to the join points. An aspect weaver is a tool which merges the aspects and classes with respect to the join points [10]. This merging or weaving can be carried out at two points in time:

- compile-time (static weaving)
  The aspect weaver acts as a pre-processor weaving the aspect definitions into the class definitions before compilation. Alternatively, the aspect weaver acts as a post-processor weaving the aspect definitions into the compiled class code.
- run-time (dynamic weaving)
  The aspect weaver acts as a run-time interpreter or run-time generator [8].

Several aspect languages and their associated weavers have been developed. The most well-known of these is AspectJ [1]: an aspect language for Java. The associated weaver is a compile-time pre-processor weaving aspect definitions into the Java class definitions[2] before they are compiled by the Java compiler. Since the weaver (ajc) is itself written in Java some dynamic weaving is possible by calling ajc.main( ) from within a Java program with aspect and class definitions as parameters. The resulting compiled Java class code can then be loaded using the ClassLoader. An aspect language and a weaver for Smalltalk have been discussed in [6]. In this implementation aspects live beyond compile-time and can be dynamically woven into the classes. Note that aspects in AspectJ can also live beyond compile-time hence facilitating dynamic binding and run-time introspection.

While existing implementations observe the need for aspects to live beyond compile-time they do not take into account the fact that some aspects might even outlive the program execution. In [14] [15] it was argued that several aspects cut across entities in persistent environments such as databases and persistent programming languages. Examples of such aspects include instance adaptation during schema evolution [17], versioning [15], links among persistent entities [16], constraints, access rights, security, data representation [15] and distribution [12]. These aspects are persistent by nature because they cross-cut a range of persistent entities: objects, class definitions (in the schema), meta-class definitions (in the meta-schema), etc. [15].

This paper discusses two mechanisms for weaving aspects in persistent environments founded on object-oriented databases. The first mechanism is based on exploiting existing aspect languages and their associated weavers while the second mechanism is based on building weaving functionality into the database management system (DBMS). The two mechanisms address different application areas. The first mechanism addresses the needs of the database

[1] For simplification, from this point onwards, the more specific term “classes” will be used instead of the general term “programming modules”.

[2] Based on AspectJ 0.7 beta 12

application developer while the second caters for the needs of the database administrator and maintainer.

2 Using Existing Weavers

The obvious choice for defining and weaving aspects in a persistent environment based on an object database is the use of existing aspect languages and their associated weavers. This is because object database management systems offer APIs for main OO programming languages such as C++ and Java, and aspect languages and weavers are also available for such languages. AspectJ [1], for example, has been available for aspect-oriented programming with Java for a few years. However, it is essential that any weaving mechanism based on using existing languages and weavers must not only integrate seamlessly with the persistence model of the underlying ODBMS but also take into account the presently evolving nature of aspect languages and weavers. Any weaving mechanism must, therefore, account for the constraints imposed by the ODBMS, and the aspect language and its weaver. While some of these constraints might be specific to the particular ODBMS, aspect language or weaver, there are several general constraints which have been summarised below:

- **Constraints imposed by ODBMSs**
    - Object database management systems often require that classes whose instances are to be stored in the database extend a system provided *Persistent Root Class*. These classes are then augmented by persistence-related code using a *persistence processor* at the pre-compilation or post-compilation stage. Examples of such systems include the Object Data Management Group (ODMG) standard [7], O2 [3] and Jasmine [2].
    - Due to proprietary restrictions it is not possible to modify the system classes implementing the persistence model of the object database management system being used.
    - Most object database management systems employ the *persistence by reachability* principle. When a transaction commits all objects reachable from a persistent object are transitively made persistent. Examples of such systems include the ODMG standard [7], O2 [3], Jasmine [2] and Object Store [4].
- **Constraints imposed by aspect languages and weavers**
    - The aspect structures can vary considerably across aspect languages.
    - Some aspect languages and weavers might support run-time aspects while others might not.
    - Aspect languages and aspect structures are continuously evolving as AOP technologies mature.

2.1 Integration of Aspect Language and ODBMS API

The proposed model for integrating an aspect language and the ODBMS API for the respective OO language is shown in fig. 1. The model operates within the above constraints and is an adaptation of the aspect persistence model proposed in [14]. The *persistent root class* offered by the ODBMS API is extended by a *surrogate persistent root class*. All application classes whose instances are to be stored in the object database extend the *surrogate persistent root class* and hence, indirectly, extend the *persistent root class*. The *surrogate persistent root class* provides methods which are invoked each time an object is made persistent. This is achieved by providing wrappers around the ODBMS transaction operations. This mechanism has been preferred over using reflective mechanisms, if present in the OO language, to obtain information about the *persistent root class*. This is because one cannot be sure about the methods of the *persistent root class* invoked upon object persistence and the order of their invocation.
The invocation of the *surrogate persistent root class* methods upon object persistence provide suitable join points for the *persistent root aspect*. This aspect intercepts these invocations and ensures that all aspects reachable from the persistent object are transitively made persistent (a more detailed description of aspect persistence by reachability can be found in [14]). All application aspects that cut across persistent objects extend the *persistent root aspect*. The *surrogate persistent root class*, *persistent root aspect* and aspect persistence by reachability provide a non-intrusive, natural extension of the persistence model employed by several ODBMSs. Hence, the restrictions imposed by the nature of ODBMSs and their APIs are effectively dealt with.

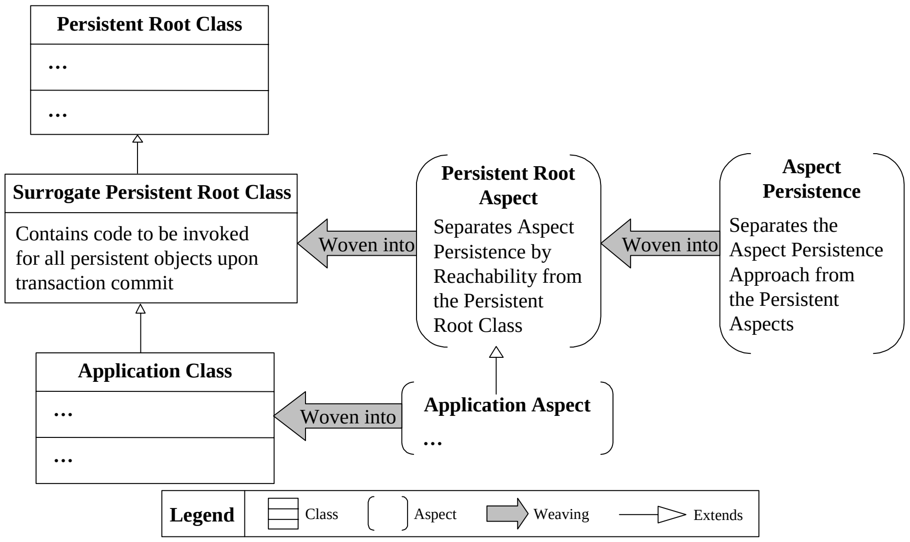

Persistent Root Class
…
…
Surrogate Persistent Root Class
Contains code to be invoked for all persistent objects upon transaction commit
Woven into
Persistent Root Aspect
Separates Aspect Persistence by Reachability from the Persistent Root Class
Woven into
Aspect Persistence
Separates the Aspect Persistence Approach from the Persistent Aspects
Application Class
…
…
Woven into
Application Aspect
…
Legend
Class
Aspect
Weaving
Extends

Fig. 1. Integration of the aspect language and the OO language API offered by the ODBMS

In order to deal with the evolving nature of aspect languages all links between aspects and classes have been kept strictly *class directional* i.e. the aspects know about the classes but not vice versa [9]. This localises changes resulting from the evolution of the aspect language making maintenance and modifications to the persistence model inexpensive. Such changes are further aided by the *aspect persistence* aspect which separates the persistence approach from the persistent aspects; persistence is a cross-cutting concern in a system [11] [18]. It also encapsulates any language or weaver specific features that need to be incorporated into the model. As a result changes that do not affect aspect reachability are localised to the *aspect persistence* aspect. This also makes it possible to keep the aspect language and ODBMS API integration model largely independent of the particular aspect language as language specific features are mainly encapsulated in one single aspect[3].

2.2 The Weaving Process

The model in fig. 1 provides integration between the aspect language and the OO language API offered by the ODBMS. It does not describe the actual weaving process in this persistent environment nor does it discuss how the varying support for run-time aspects across aspect languages and weavers is catered for. The weaving process is shown in fig. 2. The nature of the transformation pipeline in this process automatically accounts for the fact that not all weavers support run-time aspects.

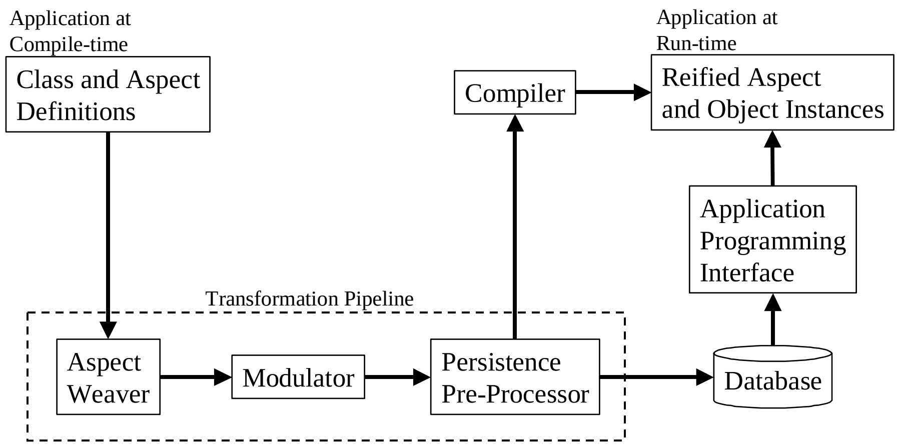
The transformation pipeline is composed of three code generators:

- **The aspect weaver** takes the aspect and class definitions and merges them with respect to the join points (assuming pre-compilation weaving).
- **The modulator** removes any syntactic mismatches between the code produced by the aspect weaver and the code to be supplied to the persistence pre-processor (assuming pre-compilation processing).
- **The persistence pre-processor** takes the modulated woven code and generates persistence capable code and the database schema.

The persistence capable code from the transformation pipeline is fed to the language compiler which compiles the code to an executable format. Note that if the aspect language and weaver support run-time aspects these would be reified as objects together with the woven structures. As a result the persistence pre-processor will automatically generate persistent representations for both woven structures and the reified aspects. The application will, therefore, be able to access these reified aspects together with the objects at run-time through the ODBMS API (integrated with the aspect language as discussed above). If the aspect language and weaver do not support run-time aspects the persistence processor will automatically generate persistent representations for woven structures only (as these will be the only input from the weaver). As a result aspects will not be available to the application at run-time. Note that the persistent representation of an aspect (if it can live beyond compile-time) is automatically

[3] Note that it is not possible to achieve full independence from the language features as the *persistent root aspect* and aspect persistence by reachability will employ these features.

determined by the persistence processor (as the aspect is reified as an object). The application programmer does not need to concern him/herself with this issue.



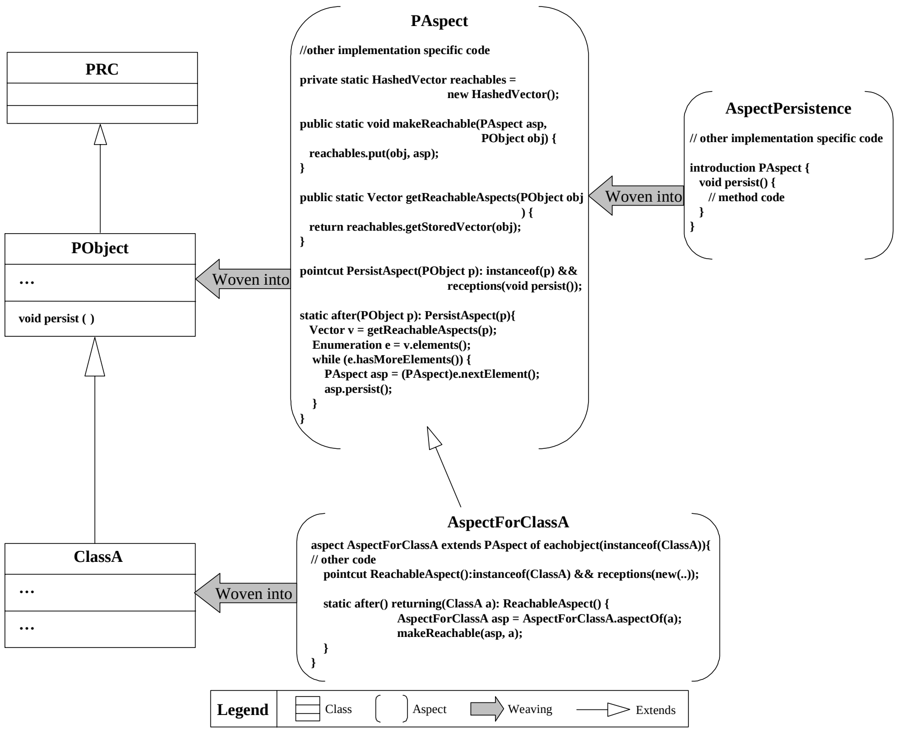
Application at Compile-time
Class and Aspect Definitions
Compiler
Application at Run-time
Reified Aspect and Object Instances
Application Programming Interface
Transformation Pipeline
Aspect Weaver
Modulator
Persistence Pre-Processor
Database

Fig. 2. Weaving in an object database based persistent environment using an existing weaver

2.3 Implementation based on AspectJ and Jasmine ODBMS

The above mechanism has been employed to use AspectJ 0.7 beta 12 and its associated weaver for aspect definition and weaving in a Jasmine object database environment. The implementation of the integration model is shown in fig. 3. *PRC* is the *persistent root class* in the Jasmine Persistent Java binding (pJ). *PObject* is the *surrogate persistence root class* and has a special instance-level method called *persist( )* which is invoked for all persistent objects (identified through persistence by reachability) just before a transaction commits. *PAspect* is the *persistent root aspect* and determines all reachable aspects from a persistent object after the *persist( )* method has been invoked (upon transaction commit). The reachability from an object is explicitly specified in each sub-aspect of *PAspect* through a static advice. All reachable aspects are made persistent through a call to the *persist( )* method for the aspect instance. The *persist( )* method is introduced into *PAspect* by the *AspectPersistence* aspect. The modulator in this implementation only replaces any $ signs in the code generated by the AspectJ weaver as this is regarded a reserved character in Jasmine, and hence, its persistence pre-processor.

Note that this implementation was initially carried out using AspectJ 0.6 beta 2 and later ported to AspectJ 0.7 beta 12. This provided an opportunity to reflect on the effectiveness of the integration model in coping with changes to the aspect language. The class directional nature of the aspects made it possible to make all the changes without affecting existing application classes. The AspectJ 0.6 beta 2 implementation relied on the explicit instantiation of an aspect followed by its addition to the aspect list of an object to determine reachability (by calling the *getAspects( )* method on an object). These features were dropped in AspectJ 0.7 beta 12 in favour of automatic maintenance of links between aspects and objects and disallowing of explicit aspect instantiation. Although changes were localised to the aspects it was decided to maintain reachability links explicitly in the model implementation to further reduce the evolution complexity should such changes occur in the language in the future. All other changes to the language were localised to the *AspectPeristence* aspect. This effective localisation of changes indicates that the proposed mechanism is capable of effectively coping with the presently evolving nature of aspect languages and weavers.



```
PAspect
//other implementation specific code

private static HashedVector reachables =
                              new HashedVector();

public static void makeReachable(PAspect asp,
                                     PObject obj) {
  reachables.put(obj, asp);
}

public static Vector getReachableAspects(PObject obj
                                          ) {
  return reachables.getStoredVector(obj);
}

pointcut PersistAspect(PObject p): instanceof(p) &&
                              receptions(void persist());

static after(PObject p): PersistAspect(p){
  Vector v = getReachableAspects(p);
   Enumeration e = v.elements();
   while (e.hasMoreElements()) {
      PAspect asp = (PAspect)e.nextElement();
      asp.persist();
   }
}
```

```
AspectPersistence
// other implementation specific code

introduction PAspect {
  void persist() {
    // method code
  }
}
```

```
AspectForClassA
aspect AspectForClassA extends PAspect of eachobject(instanceof(ClassA)){
// other code
   pointcut ReachableAspect():instanceof(ClassA) && receptions(new(..));

   static after() returning(ClassA a): ReachableAspect() {
                  AspectForClassA asp = AspectForClassA.aspectOf(a);
                  makeReachable(asp, a);
   }
}
```

Fig. 3. Integration of AspectJ 0.7 beta 12 with the Jasmine Java API

3 Building a Weaver into the DBMS

The mechanism discussed in section 2 provides a means for database application programmers to use existing aspect languages and weavers in an object database environment. However, there are other roles in a database environment e.g. database administrators and maintainers[4]. These roles often perform maintenance or maintenance-related activities. It is, therefore, imperative that they benefit from the effectiveness of AOP in localisation of changes to cross-cutting features (hence reducing maintenance overheads). Very often advanced maintenance features are available through programming interfaces based on proprietary languages. For instance, both O2 [3] and Jasmine [2] offer advanced schema evolution functionality through proprietary languages O2C and ODQL respectively. As a result use of existing aspect languages and weavers is not an option. If the DBMS developer chooses to offer the benefits of aspect-orientation to these roles then aspect languages and weavers for proprietary languages need to be built into the DBMS.

[4] Note that the same person can play the different roles.

When building an aspect weaver into a DBMS the following factors must be taken into account:

- The aspect weaver must be aware of the fact that aspects may be persistent. This differs from the mechanism in section 2 where an aspect language integration mechanism and a transformation pipeline shield the weaver from the persistent nature of aspects.
- The aspect weaver needs to provide a persistent structure for the aspects (instead of relying on a transformation pipeline for the purpose).
- The aspect weaver might need to retrieve and weave the aspects before information is delivered to an application. Examples of such aspects are instance adaptation and links among persistent entities. While the DBMS uses these aspects to reduce maintenance overhead the application does not need to be aware of their existence and can simply use the woven structures.
- Aspects might be modified after they have been woven.

3.1 Persistent Aspect Structures

Three different persistent aspect structures may be employed by a weaver in an object database environment. Each of these has its advantages and disadvantages:

1. Aspects are reified as first-class persistent objects. Instead of modifying the code of classes the weaver simply delegates control to the aspects when a join point is encountered [6] [8]. The advantage of this approach is that full reflective information about the persistent aspects is available. However, it is most suitable in situations where aspects may be repeatedly modified. Otherwise the delegation overhead can be significant because, unlike weaving approaches based on code modification, code optimisations cannot be applied.
2. Reify the aspect as a first-class persistent object but weave it into the code when required. The advantage of this approach is that full reflective information is available about the persistent aspect prior to weaving. Code optimisations can be applied when the aspect is woven. However, post-weaving reflective information is not available.
3. Simply store the aspect code as described by the aspect language in the database. When required, the code can be retrieved and woven into the class code. This mechanism is very efficient as no reflection overheads are involved and code optimisations can be applied. However, this also means that no reflective information is available prior to or after weaving.

3.2 Weave-on-demand and Weave Histories

As mentioned earlier aspects may be modified after they have been woven. As a result the weaver needs some mechanism to propagate the changes in the aspect definition to its woven state. If the weaving mechanism is based on code modification (structures 2 and 3 in section 3.1) the reweaving overhead can be significant, hence compromising the advantages gained from applying code optimisation. The following two mechanisms can be employed, in conjunction, to deal with the changing nature of aspects and propagate these changes in an efficient manner in weaving approaches based on code modification:

- **Weaving on-demand:** An aspect is woven only if it has not been previously woven. An aspect is rewoven only if it has been modified since the last weave.
- **Maintaining weaving history:** Once an aspect is modified its previous definition is saved. This is referred to during reweaving by the weave-on-demand process as the previous definition needs to be unwoven before the new one is woven.

A weaving process using weaving on-demand and weaving histories is shown in fig. 4.

As shown in fig. 4 each aspect maintains the following information:

- Current weave state: the present aspect definition.
- Previous woven state: the last aspect definition that was woven.
- A timestamp indicating the last point in time the aspect was woven.
- A timestamp indicating the last point in time the aspect was modified.

Weaving on-demand and weave histories are complemented by *selective weaving* which further improves the efficiency of the weaving process by only unweaving and reweaving the modified parts of an aspect instead of unweaving and reweaving the whole aspect. The input to the weaving process is, therefore, ΔWeave which is a measure of the extent to which an aspect has been modified since the last time it was woven. As a result ΔWeave can range from *null* (aspect not modified since last weave) to *current weave state* (aspect not woven before or completely modified since last weave).

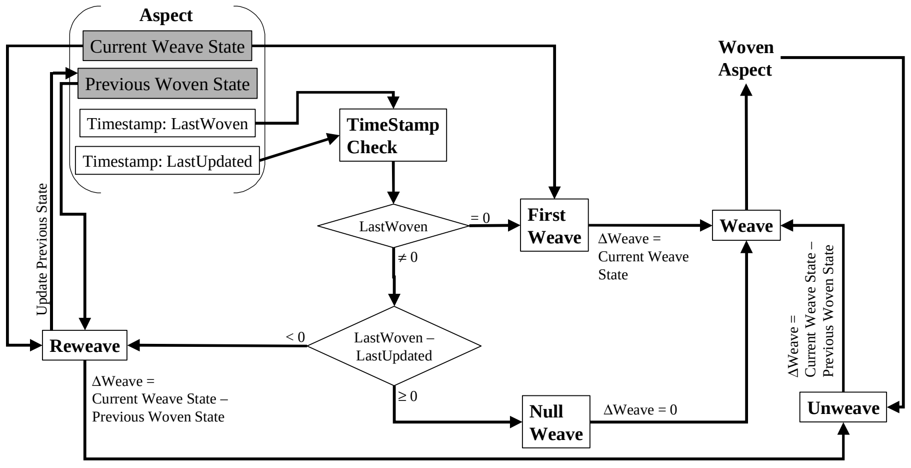


Fig. 4. A weaving process using weave-on-demand, weave histories and selective weaving

The weave-on-demand process checks the *LastWoven* and *LastUpdated* timestamps when an aspect needs to be woven. If the *LastWoven* timestamp is zero this means that the aspect has never been woven before. The complete aspect definition (*ΔWeave = Current Weave State*) is passed to the weaving process. If the *LastWoven* timestamp is not zero but newer than the *LastUpdated* timestamp, this means that the aspect has not been modified since it was last woven and, hence, does not need to be rewoven. A null weave state (*ΔWeave = 0*) is passed to the weaving process. If the *LastUpdated* timestamp is newer than the *LastWoven* timestamp, this means that the aspect has been modified since it was last woven. The reweaving process then calculates the extent of modification (*ΔWeave = Current Weave State – Previous Woven State*) and passes this information to the unweaving process which unweaves the modified parts before the weaving process reweaves the modified parts. Note that the unweaving and weaving processes, in this case, operate under the control of the reweaving process which updates the previous woven state of the aspect once it has been successfully rewoven. This ensures atomicity of the reweave operation.

3.3 A Weaver for the SADES Object Database Evolution System

The above mechanism has been employed to implement an aspect weaver as part of the SADES object database evolution system [13] [16] [17]. SADES has been implemented as a layer on top of the Jasmine object database management system [4] and makes extensive use of its proprietary language ODQL to obtain low-level access to Jasmine functionality. ODQL is not the only language used in SADES. The system, in fact, has been built from a combination of Java, C, C++ and ODQL. Consequently existing aspect weavers cannot be employed in SADES and a custom solution is the only choice.

The integrated aspect weaver facilitates the implementation of a cost-effective, customisable instance adaptation mechanism in SADES. Instance adaptation is the process of simulating object conversion or physically converting objects across historical class definitions during schema evolution. In [17] it was demonstrated that the instance adaptation behaviour in an object database system is cross-cutting in nature. This is because traditionally the same adaptation routines are introduced into a number of class versions. Consequently, if the behaviour of a routine needs to be changed maintenance has to be performed on all the class versions in which it was introduced. Adaptation routines for a particular class version often reference the structure of other class versions hence resulting in code tangling across various versions of a class. In SADES this cross-cutting behaviour is separated using aspects. The aspects are defined using a declarative aspect language modelled on AspectJ [1]. It provides three simple constructs facilitating:

- identification of join points between the aspects and class versions
- introduction of new methods into the class versions
- redefinition of existing methods in the class versions

The maintainer specifies the instance adaptation aspects as declarative statements passed as strings to methods in the SADES Java API. The aspect specification is parsed to generate the persistent aspects which are in turn associated with the class versions. The persistent representation simply stores the aspect code in the database (aspect structure 3 in section 3.1). This choice is driven by the fact that the instance adaptation code needs to be efficient. Hence, code optimisation has been preferred over reflective information.

The aspect weaver fully supports on-demand weaving and maintenance of weave histories. Selective weaving is, however, only supported in a limited fashion. This is because calculating ΔWeave is a very complex operation and, at present, has only been implemented for specific scenarios. The on-demand weaving process is invoked using a composition filters mechanism [5] and a *weaver interface object* which exposes the weaver functionality to the rest of the system. As shown in fig. 5 an output dispatch filter intercepts any interface mismatch messages and delegates them to the weaver which then dynamically weaves (or reweaves) the required instance adaptation aspect. The appropriate instance adaptation routine is then invoked to return the results to the application.

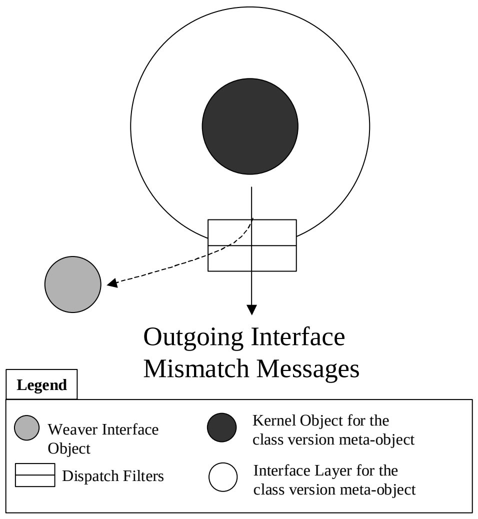

Outgoing Interface
Mismatch Messages
Legend
Weaver Interface Object
Kernel Object for the class version meta-object
Dispatch Filters
Interface Layer for the class version meta-object

Fig. 5. Interception of interface mismatch messages and their delegation to the weaver using composition filters

4 Conclusions and Future Work

This paper has discussed two mechanisms for weaving aspects in an object database environment: using existing languages and weavers, and building custom weavers into the ODBMS. A model for integrating an aspect language and an ODBMS API has been discussed. It has been demonstrated that the model can provide a seamless integration and can effectively cope with changes in the aspect language. A transformation pipeline for weaving aspects and automatic generation of their persistent structures has been discussed. It has been shown that the transformation pipeline can automatically cope with the varying support for run-time aspects in different aspect languages and their weavers.
The paper has also discussed some possible persistent structures for aspects when building a weaver into the ODBMS. The advantages and disadvantages of each structure have been highlighted. Mechanisms such as weave-on-demand, weave histories and selective weaving have been proposed to effectively deal with changes to aspect definitions after they have been woven. Selective weaving can significantly reduce the reweaving overhead in environments where aspects can live beyond compile-time and may be modified after they have been woven.
The work in the immediate future will concentrate on developing effective mechanisms for calculating ΔWeave during selective weaving. New persistent representations for aspects will also be investigated.

References

[1] Xerox PARC, USA, "AspectJ Home Page", http://aspectj.org/, 2000
[2] *The Jasmine Documentation*, 1996-1998 ed: Computer Associates International, Inc. & Fujitsu Limited, 1996.
[3] *The O2 System - Release 5.0 Documentation*: Ardent Software, 1998.
[4] *Object Store C++ Release 4.02 Documentation*: Object Design Inc., 1996.
[5] M. Aksit and B. Tekinerdogan, "Aspect-Oriented Programming using Composition Filters", ECOOP '98 AOP Workshop, 1998
[6] K. Boellert, "On Weaving Aspects", AOP Workshop at ECOOP '99, 1999
[7] R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russel, O. Schadow, T. Stenienda, and F. Velez, *The Object Data Standard: ODMG 3.0*: Morgan Kaufmann, 2000.
[8] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools and Applications*: Addison-Wesley, 2000.
[9] M. A. Kersten and G. C. Murphy, "Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming", OOPSLA, 1999, ACM, SIGPLAN Notices, 34(10), pp. 340-352.
[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", ECOOP, 1997, Springer-Verlag, Lecture Notes in Computer Science, 1241
[11] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales, "Aspect-Oriented Programming Workshop Report", ECOOP '97 Workshop

Reader, 1997, Springer-Verlag, Lecture Notes in Computer Science, 1357, pp. 483-496.

[12] E. Pulvermueller, H. Klaeren, and A. Speck, "Aspects in Distributed Environments", Generative and Component-Based Software Engineering (GCSE), 1999, Springer-Verlag, Lecture Notes in Computer Science, 1799

[13] A. Rashid, "A Database Evolution Approach for Object-Oriented Databases", in *Computing Department*: Lancaster University, UK, 2000.

[14] A. Rashid, "On to Aspect Persistence", 2nd International Symposium on Generative and Component-based Software Engineering (GCSE part of proceedings of NetObjectDays), 2000, pp. 453-463.

[15] A. Rashid and E. Pulvermueller, "From Object-Oriented to Aspect-Oriented Databases", 11th International Conference on Database and Expert Systems Applications (DEXA), 2000, Springer-Verlag, Lecture Notes in Computer Science, 1873, pp. 125-134.

[16] A. Rashid and P. Sawyer, "Object Database Evolution using Separation of Concerns", *ACM SIGMOD Record*, Vol. 29, No. 4, pp. 26-33, 2000.

[17] A. Rashid, P. Sawyer, and E. Pulvermueller, "A Flexible Approach for Instance Adaptation during Class Versioning", ECOOP 2000 Symposium on Objects and Databases, 2000, Springer-Verlag, Lecture Notes in Computer Science, 1944, pp. 101-113.

[18] J. Suzuki and Y. Yamamoto, "Extending UML with Aspects: Aspect Support in the Design Phase", 3rd AOP Workshop held in conjunction with ECOOP '99, 1999