# On to Aspect Persistence

Awais Rashid

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

**Abstract.** Over the recent years aspect-oriented programming (AOP) has found increasing interest among researchers in software engineering. *Aspects* are abstractions which capture and localise cross-cutting concerns. Although persistence has been considered as an aspect of a system, persistence of aspects has been largely ignored. This paper identifies several scenarios where aspect persistence is an essential requirement. A model for aspect persistence and an initial prototype based on AspectJ (0.6beta2) are presented. Various open issues are also pointed out.

## 1 Introduction

Over the recent years aspect-oriented programming (AOP) [9] has found increasing interest among researchers in software engineering. It aims at easing software development by providing further support for modularisation. *Aspects* are abstractions which serve to localise any cross-cutting concerns e.g. code which cannot be encapsulated within one class but is tangled over many classes. A few examples of aspects are memory management, failure handling, communication, real-time constraints, resource sharing, performance optimisation, debugging and synchronisation. In AOP classes are designed and coded separately from aspects encapsulating the cross-cutting code. The links between classes and aspects are expressed by explicit or implicit *join points*. An *aspect weaver* is used to merge the classes and the aspects with respect to the join points. This can be done statically as a phase at compile-time or dynamically at run-time [7, 9].

AOP research has identified persistence as an aspect of a system [12, 21]. However, the need for aspect persistence has not been considered. Existing work on lifetime of aspects [7, 9, 11] has argued that at least some of the aspects should live for the lifetime of the program execution and not die at compile-time. In certain cases the need for aspects to outlive the program execution can arise. [17], for example, proposes the use of an aspect repository for managing and reusing aspects in a distributed environment. Aspects in the repository live beyond the execution of the programs using them. Some aspects can be persistent by nature. [18, 19] identify several aspects cross-cutting the schema and the data in object-oriented databases. These include instance adaptation, versioning, clustering, access rights and data representation, etc. Due to the close integration between object-oriented programming languages and object-oriented databases these aspects are seamlessly used by application programs but are persistent by nature and reside in the database.

This paper proposes an approach for aspect persistence. The next section makes a case for aspect persistence by discussing several scenarios involving persistent aspects. A description of the proposed aspect persistence model is presented in section 3. The model is independent of a particular AOP approach and takes into account the evolving nature of aspect languages and representations. An initial prototype implementation of the model using AspectJ (0.6beta2) [2] and the Jasmine (1.21) object database management system is discussed in section 4 while the various open issues are outlined in section 5. Section 6 concludes the paper and identifies directions for future work.

## 2 Why Persistent Aspects?

This section describes some scenarios involving persistent aspects. The discussion demonstrates that aspect persistence is an essential requirement in many cases, hence highlighting the need to provide support for the purpose.

### 2.1 An Aspect Repository for Managing and Reusing Aspects

[17] proposes the use of an aspect repository to manage and reuse aspects in a distributed environment. The approach aims at reducing the additional complexity introduced due to the need to find an aspect in a network of computing units. Aspects in the repository are persistent and applications at different locations can query the repository to retrieve the required aspects at run-time. This is shown in fig. 1 [17].

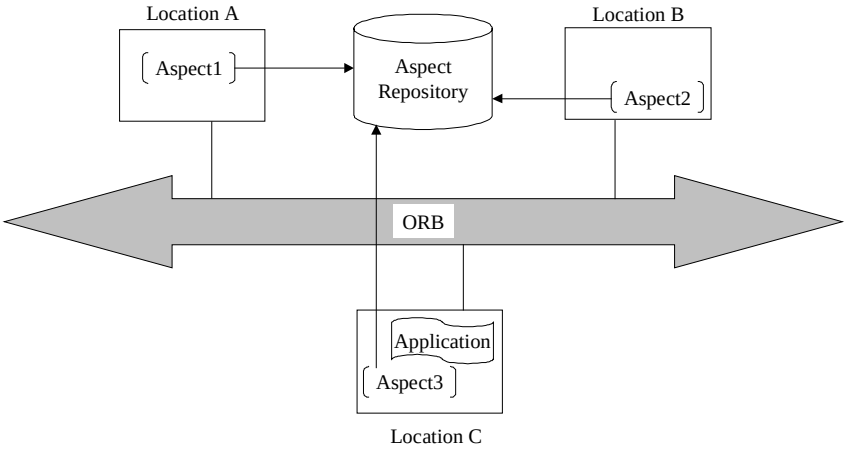



**Fig. 1.** Managing and reusing distributed aspects through an aspect repository [17]

Another scenario is an automated software development environment where both components and aspects reside in a database. The appropriate components and aspects are retrieved by the assembling process which carries out the weaving.

### 2.2 Instance Adaptation during Class Versioning

The second example is based on instance adaptation during class versioning [3, 14, 20] in object-oriented databases. Class versioning allows several versions of one type to be created during evolution. An instance is bound to a specific version of the type and when accessed using another type version (or a common type interface) is either converted or made to exhibit a compatible interface. This is termed as instance adaptation and is essential to ensure structural consistency. A detailed description of class versioning is beyond the scope of this paper. Interested readers are referred to [3, 14, 20]. The following discussion demonstrates that the instance adaptation code cross-cuts the class versions and can be separated using aspects. It should be noted that the instance adaptation aspects cross-cut persistent entities (the class versions) and are persistent by nature.

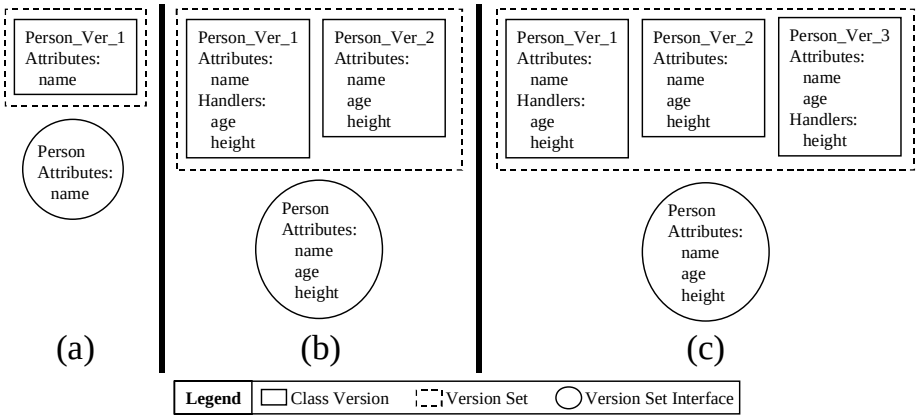



**Fig. 2.** Class Versioning in ENCORE

We first consider the instance adaptation strategy of ENCORE [20]. A similar approach is employed by AVANCE [3]. As shown in figure 2, applications access instances of a class through a *version set interface* which is the union of the properties and methods defined by all versions of the class. Error handlers are employed to trap incompatibilities between the version set interface and the interface of a particular class version. These handlers also ensure that objects associated with the class version exhibit the version set interface. As shown in fig. 2(b) if a new class version modifies the version set interface (e.g. if it introduces new properties and methods) handlers for the new properties and methods are introduced into all the former versions of the type. On the other hand, if creation of a new class version does not modify the version set interface (e.g. if the version is introduced because properties and methods have been removed), handlers for the removed properties and methods are added to the newly created version (cf. fig. 2(c)).

The introduction of error handlers in former class versions is a significant overhead especially when, over the lifetime of the database, a substantial number of class versions can exist prior to the creation of a new one. If the behaviour of some handlers needs to be changed maintenance has to be performed on all the class

versions in which the handlers were introduced. To demonstrate the use of persistent aspects we have chosen the scenario in fig. 2(b). Similar solutions can be employed for other cases. As shown in fig. 3(a) instead of introducing the handlers into the former class versions they are encapsulated in an aspect. Fig. 3(b) depicts the case when an application attempts to access the *age* and *height* attributes in an object associated with version 1 of *class Person*. The aspect containing the handlers is woven into the particular class version. The handlers then simulate (to the application) the presence of the missing attributes in the associated object.

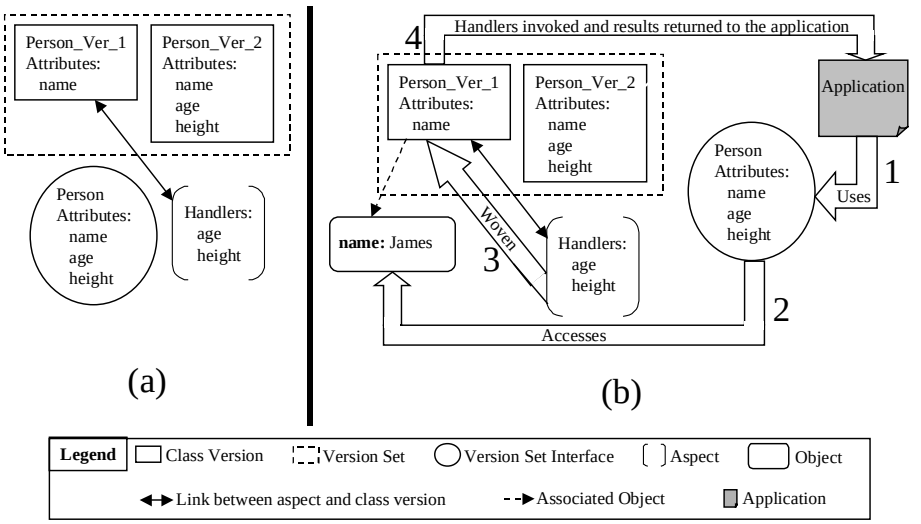



**Fig. 3.** The Aspect-Oriented Instance Adaptation Approach

Encapsulating handlers in an aspect offers an advantage in terms of maintenance as only one aspect is defined for a set of handlers for a number of older class versions. Behaviour of the handlers can be modified within the aspect instead of modifying them within each class version. Aspects also help separate the instance adaptation strategy from the class versions. For example, let us suppose one wants to employ a different instance adaptation approach[1], the use of update/backdate methods for dynamic instance conversion between class versions (as opposed to simulating a conversion) [14]. In this case only the aspects need to be modified without having the problem of updating the various class versions. These are automatically updated to use the new strategy when the aspect is woven. The aspect-oriented approach has a run-time overhead as aspects need to be woven and unwoven (if adaptation strategies are expected to change). However, this overhead is smaller than that of updating and maintaining a number of class versions. The overhead can be reduced by leaving an aspect woven and only reweaving if the aspect has been modified. Details of the aspect-oriented instance adaptation approach have been reported in [18].

[1] Such a need can arise due to application/scenario specific adaptation requirements or the availability of a more efficient strategy.

### 2.3 Clustering

[19] identifies clustering as a persistent aspect which cross-cuts the objects residing in an object-oriented database. Traditionally it is the task of the database application programmer to ensure that related objects are clustered together. However, the task is easy only for a small number of objects. As the number of objects that need to be clustered increases (it should be noted that the clustering reasons could be different for different groups of objects) the programmer's task becomes more and more complicated. The situation worsens if the same object needs to be clustered together with more than one group. Considering clustering as an aspect of data residing in a database allows managing these complex scenarios transparently of the programmer. The programmer can specify clustering as an aspect of a group of objects regardless of whether some objects in the group also form part of another group. The clustering aspects can then be used by the system to work out an efficient storage strategy. Furthermore, if the clustering requirements for an object change the programmer can re-configure the clustering aspect to indicate which group of objects should be clustered with this object. This helps to manage the physical reorganisation of the various clustered objects transparently of the programmer. It should also be noted that clustering is not necessarily an aspect of all the objects residing in the database. Introducing clustering as an aspect allows only those objects having this aspect to be clustered.

### 2.4 Other Persistent Aspects

In addition to the examples presented in section 2.1-2.3 several other persistent aspects can be identified. Versioning is an aspect which cross-cuts both objects and classes in an object-oriented database (assuming the system supports both object and class versioning). Constraints can be considered as an aspect of the object database. Traditionally constraints are specified at the application level or through a DBMS service. Considering constraints an aspect of database entities and providing a concrete abstraction simplifies their specification and management. Access rights, security and data representation can also be regarded as aspects. All these aspects cross-cut persistent entities residing in a database and are persistent by nature.

## 3 A Model for Aspect Persistence

Section 2 presented several scenarios where aspect persistence is an essential requirement. This section proposes a model for aspect persistence. The model is based on the following observations:

1. Object database management systems often require that classes whose instances are to be stored in the database extend a system provided *Persistent Root Class*. Examples of such systems include the Object Data Management Group (ODMG) standard [4], O2 [15] and Jasmine [6].

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.
3. Most object database management systems employ the *persistence by reachability* principle. When a transaction commits all objects reachable from a persistent object are made persistent. Examples of such systems include the ODMG standard [4], O2 [15] Jasmine [6] and Object Store [16].
4. Persistence is a cross-cutting concern in a system [12, 21].
5. There are various AOP approaches available e.g. AspectJ [2], Composition Filters [1], HyperJ [5], Adaptive Programming [10, 13], etc. All these approaches aim at achieving a better separation of concerns. However, the aspect structures employed for the purpose vary considerably.
6. Aspect languages and aspect structures are continuously evolving as AOP technologies mature.

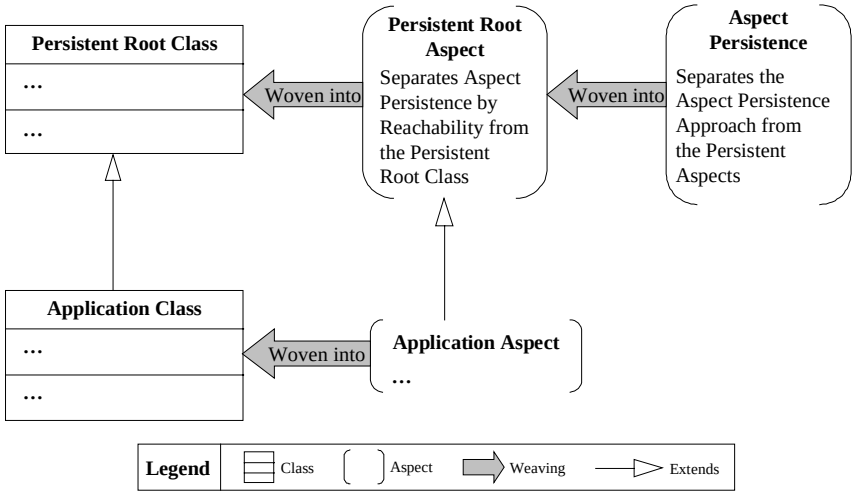



**Fig. 4.** The Aspect Persistence Model

The persistence model is shown in fig. 4. All application classes extend the *Persistent Root Class* offered by the particular object database management system. A similar mechanism is employed for making application aspects persistent. All application aspects extend a *Persistent Root Aspect*. This is a natural extension of the persistence model employed by several object database management systems (observation 1). Since it is not possible to modify the system classes implementing the persistence model of the object database management system (observation 2), all links between aspects and classes have been kept strictly *class directional* i.e. the aspects know about the classes but not vice versa [8]. This also facilitates the modification and reuse of the aspect code [8] (as discussed later) and avoids introduction of additional evolution complexity.

When a transaction commits all aspects reachable from a persistent object are made persistent (observation 3). As shown in fig. 5 *Aspect 1* is an instance of an aspect extending the *Persistent Root Aspect* and reachable from a persistent object *Object 1*. It is, therefore, made persistent. Although *Aspect 2* is also an instance of an aspect extending the *Persistent Root Aspect* it will not be made persistent as it is not reachable from any persistent objects in the scope of the transaction. *Aspect 3* poses an interesting scenario. It is reachable from a persistent object but is not an instance of an aspect extending the *Persistent Root Aspect*. It will be coerced into persistence in order to preserve persistence by reachability. The coercion strategy can be to force the aspect to extend the *Persistent Root Aspect* or to annotate the particular aspect instance (*Aspect 3*) with persistence-related code. The choice of coercion strategy has been left to the implementation of the model. It should be noted that aspect persistence by reachability is transitive in nature i.e. not only aspects reachable from persistent objects are made persistent but also aspects reachable from persistent aspects are transitively made persistent.

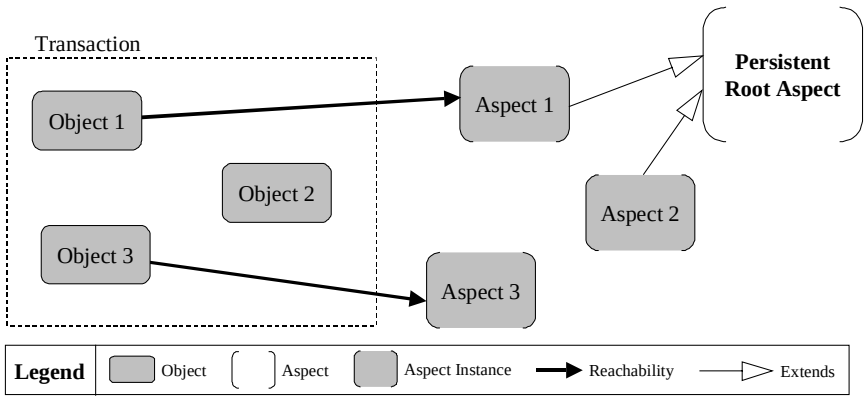



**Fig. 5.** Aspect Persistence by Reachability

As shown in fig. 4 the *Persistent Root Aspect* encapsulates the persistence by reachability code. This code defines the behaviour of an aspect transitively reachable from a persistent object upon transaction commit. Since all application aspects extend the *Persistent Root Aspect* the behaviour is propagated down the hierarchy. Reflecting on observations 4, 5 and 6 persistence has been regarded as an aspect of the persistent aspects (cf. fig. 4). The persistence approach is separated from the persistent aspects through the *Aspect Persistence* aspect. This makes the model independent of a particular AOP approach because the persistent aspects do not encapsulate the knowledge about their storage structure (which largely depends on the particular AOP approach being employed). It also localises the changes resulting from the evolution of the aspect language or the aspect structure making maintenance and modifications to the persistence model inexpensive. Such changes are further aided by the class-directional nature of links between aspects and classes.

## 4 PersAJ: An AspectJ Prototype

This section presents PersAJ (Persistent AspectJ), an implementation of the persistence model using AspectJ (0.6beta2) from Xerox PARC and the Jasmine (1.21) object database management system from Computer Associates. The implementation is shown in fig. 6. Application classes and aspects from the instance adaptation scenario in section 2.2 have been used as an example. It should be noted that the fig. only depicts the code relating to the description in section 3. Other implementation code has been omitted.

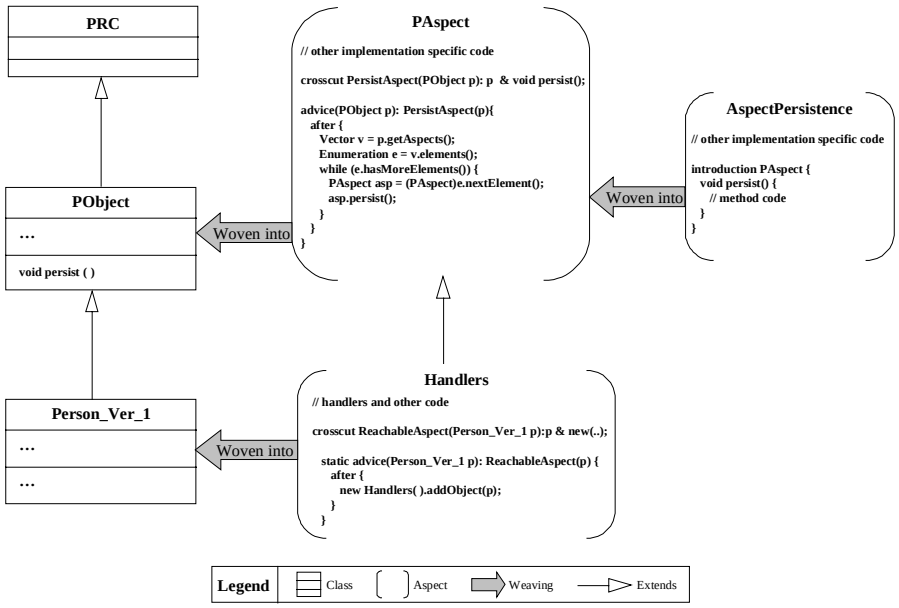



**Fig. 6.** PersAJ: An implementation of the persistence model using AspectJ (0.6beta2)

*PRC* is the *Persistent Root Class* in the Jasmine Persistent Java binding (pJ). This class is extended by all Java classes whose instances are to be stored in the Jasmine database. Since no information is available about the PRC class, we have introduced the class *PObject* which acts as the *Persistent Root Class* for all application classes in PersAJ (in this case *Person_Ver_1*). *PObject* 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. This is achieved by providing wrappers around the Jasmine transaction commit operation. *PAspect* is the *Persistent Root Aspect* in PersAJ and is extended by all persistent application aspects (in this case *Handlers*). The application aspects implement a special static advice making the aspect instance reachable from the associated class object being instantiated. The advice shown in *PAspect* (cf. fig. 6) determines all reachable aspects from a persistent object after the *persist( )* method has been invoked (upon transaction commit). 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 which separates the persistence approach and storage structure from the aspects being made persistent.
The Jasmine Persistent Java binding (pJ) offers a Java Persistence Processor (JPP): a code generator which adds persistence capability to a Java class by adding code to its definition. It also generates a corresponding schema definition for the underlying Jasmine database system. PersAJ employs a simple script which runs the AspectJ compiler in the preprocess mode. The code generated by AspectJ is passed to a custom built parser which parses the generated class definitions and replaces any $ signs with two underscores. This is essential as AspectJ uses $ signs to differentiate generated code from the Java code supplied by the programmer while Jasmine regards $ as a reserved character. The class definitions produced by the parser are passed to the Jasmine Persistence Processor which adds the persistence related code and generates the database schema. Instances of the processed classes and aspects can now be stored in the database.

## 5 Open Issues

This paper introduces the idea of persistent aspects. One of the open research issues is the persistent representation of an aspect. Due to the different aspect representations e.g. AspectJ, Composition Filters, etc. used in application programs, persistent representation of aspects needs careful exploration.
Persistent aspects and dynamic weaving introduce additional overhead at run-time and can be feasible only with efficient weaving mechanisms. The development of efficient weavers is therefore an important research issue.
One of the issues identified during development of PersAJ is the need for parameterised aspects. This is not supported in the AspectJ implementation (0.6beta2) due to the lack of parametric polymorphism in Java. If aspect parameterisation is available the implementation can be more generic and maintainable. An aspect can be parameterised by classes in which it is to be woven, hence, making the join points and crosscuts generic. Special *weave parameters* can be used to provide a generic reconfiguration mechanism during dynamic weaving. It can also provide a solution for the different aspect representations problem identified above. The representation to be used can be determined by the aspect passed as a parameter making the persistence model more transparent to the programmer.

## 6 Conclusions and Future Work

This paper has proposed an approach for aspect persistence. Aspect persistence is a natural extension of existing work on lifetime of aspects. Although existing AOP research has considered persistence as an aspect of a system, the need for aspect persistence has been largely ignored. The novelty of this work is in identifying concrete scenarios where aspect persistence is an essential requirement and providing support for the purpose. The proposed aspect persistence model is independent of the

aspect language and the aspect representation employed by it. Persistence is considered an aspect of the persistent aspects hence providing support for inexpensive changes to the persistent representation of aspects due to changes in the aspect language or the aspect structure. Class-directional links allow implementation of the model without modifying the persistence model of the object database management system being used. A prototype implementation of the model based on AspectJ (0.6beta2) and Jasmine (1.21) object database management system validates the concepts proposed in the paper.

At present AspectJ 0.7beta4 has been released. Our work in the immediate future will involve porting the PersAJ implementation to the new release. We are also interested in providing implementations of the model for other AOP approaches. At present the persistent representation of aspects is handled by the Jasmine Java Persistence Processor. We are interested in investigating various persistent representations of aspects. We are particularly interested in developing an infrastructure mapping the various aspect structures on to a common persistent representation and vice versa. Such an infrastructure will serve as middleware between implementations of the persistence model for different aspect languages and the common persistent representation of aspects.

**Note:** The work reported in this paper is part of the Aspect-Oriented Databases initiative at Lancaster which aims at bringing the notion of separation of concerns to databases and providing aspect persistence mechanisms. Further information can be found at: http://www.comp.lancs.ac.uk/computing/aod/

## References


[1] Aksit, M., Tekinerdogan, B., "Aspect-Oriented Programming using Composition Filters", *Proceedings of the AOP Workshop at ECOOP '98, 1998*

[2] AspectJ Home Page, http://aspectj.org/, *Xerox PARC, USA*

[3] Bjornerstedt, A., Hulten, C., "Version Control in an Object-Oriented Architecture", *In Object-Oriented Concepts, Databases, and Applications (eds: Kim, W., Lochovsky, F. H.), pp. 451-485, Addison-Wesley 1989*

[4] Cattell, R. G. G., *et al*., "The Object Database Standard: ODMG 2.0", *Morgan Kaufmann, c1997*

[5] "Multi-dimension Separation of Concerns using Hyperspaces", http://www.research.ibm.com/hyperspace/

[6] "Jasmine 1.21 Documentation", *Computer Associates International, Inc., Fujitsu Limited, c1996-98*

[7] Kenens, P., *et al*., "An AOP Case with Static and Dynamic Aspects", *Proceedings of the AOP Workshop at ECOOP '98, 1998*

[8] Kersten, M. A., Murphy, G. C., "Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming", *Proc. of OOPSLA 1999, ACM SIGPLAN Notices, Vol. 34, No. 10, Oct. 1999, pp. 340-352*

[9] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., "Aspect-Oriented Programming", *Proceedings of ECOOP '97, LNCS 1241, pp. 220-242*

[10] Lieberherr, K. J., "Demeter", *http://www.ccs.neu.edu/research/demeter/index.html*

[11] Matthijs, F., et al., "Aspects should not Die", Proceedings of the AOP Workshop at ECOOP '97, 1997

[12] Mens, K., Lopes, C., Tekinerdogan, B., Kiczales, G., "Aspect-Oriented Programming Workshop Report", *ECOOP '97 Workshop Reader, LNCS 1357, pp. 483-496*

[13] Mezini, M., Lieberherr, K. J., "Adaptive Plug-and-Play Components for Evolutionary Software Development", *Proceedings of OOPSLA 1998, ACM SIGPLAN Notices, Vol. 33, No. 10, Oct. 1998, pp. 97-116*

[14] Monk, S. & Sommerville, I., "Schema Evolution in OODBs Using Class Versioning", *SIGMOD Record, Vol. 22, No. 3, Sept. 1993, pp. 16-22*

[15] "The O2 System - Release 5.0 Documentation", *Ardent Software, c1998*

[16] "Object Store C++ Release 4.02 Documentation", *Object Design Inc., c1996*

[17] Pulvermueller, E., Klaeren, H., Speck, A., "Aspects in Distributed Environments", *Proceedings of GCSE 1999, Erfurt, Germany*

[18] Rashid, A., Sawyer, P., Pulvermueller, E., "A Flexible Approach for Instance Adaptation during Class Versioning", *Proceedings of ECOOP 2000 OODB Symposium (in publication as an LNCS volume by Springer-Verlag)*

[19] Rashid, A., Pulvermueller, E., "From Object-Oriented to Aspect-Oriented Databases", *To Appear in Proceedings of DEXA 2000*

[20] Skarra, A. H. & Zdonik, S. B., "The Management of Changing Types in an Object-Oriented Database", *Proceedings of the 1st OOPSLA Conference, Sept. 1986, pp. 483-495*

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