

From Object-Oriented to Aspect-Oriented Databases

Awais Rashid¹, Elke Pulvermueller²

¹Computing Department, Lancaster University, Lancaster LA1 4YR, UK
marash@comp.lancs.ac.uk

²Wilhelm Schickard Institute for Computer Science, University of Tuebingen, 72076
Tuebingen, Germany
pulvermueller@acm.org

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 aspects in the persistence domain in general and in databases in particular have been largely ignored. This paper brings the notion of aspects to object-oriented databases. Some cross-cutting concerns are identified and addressed using aspects. An aspect-oriented extension of an OODB is discussed and various open issues pointed out.

1 Introduction

Over the recent years aspect-oriented programming (AOP) [18, 19] 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. An *aspect weaver* is used to merge the classes and the aspects.

Aspect-orientation appears to be following the same development phases as object-orientation. Introduced through object-oriented programming in the late 1960s (SIMULA-67) object-orientation is now employed in a wide range of software development activities such as analysis, design, modelling, etc. It has also been successfully applied in the areas of databases and knowledge-bases. Likewise, research in aspect-orientation is now progressing from programming into other areas such as specification [6, 7, 9, 30] and design [15, 17, 41]. Its applicability in the area of databases has, however, not yet been explored. Although persistence has been considered as an aspect of a system [27, 41] aspects in the persistence domain in general and in databases in particular have been largely ignored. This paper brings the notion of aspects to object-oriented databases in order to achieve a better separation of concerns. Reflecting on the fact that AOP is not limited to object-oriented programming languages [19], we are of the view that aspects can be employed in database technology other than OODBs e.g. relational, object-relational, active

databases, etc. This will, however, form the subject of a future paper. The discussion in this paper focuses on extending object-oriented databases with aspects.

The next section provides an overview of aspect-oriented programming. Section 3 identifies some of the cross-cutting concerns in OODBs and discusses how aspects can be employed to address these effectively. This is followed by a description of an aspect-oriented extension of an object-oriented database. Extending object-oriented databases with aspects is a new concept and a number of issues need to be resolved before it can be considered as mature database technology. Some of the key open issues are identified in section 5. Section 6 discusses some related work while section 7 summarises and concludes the paper.

2 Aspect-Oriented Programming

Aspect-oriented programming aims at achieving a better separation of concerns by localising cross-cutting features e.g. code implementing non-functional requirements such as memory management, failure handling, debugging, synchronisation, etc. Code for debugging purposes or for object synchronisation, for instance, is spread across all the classes whose instances have to be debugged or synchronised, respectively. Although patterns [12] can help to deal with such cross-cutting code by providing guidelines for a good structure, they are not available or suitable for all cases and mostly provide only partial solutions to the code tangling problem. With AOP, such cross-cutting code is encapsulated into separate constructs: the aspects. As shown in fig. 1 classes are designed and coded separately from code that cross-cuts them. The links between classes and aspects are expressed by explicit or implicit *join points*. [17] categorises¹ these links as:

- *open*: both classes and aspects know about each other
- *class-directional*: the aspect knows about the class but not vice versa
- *aspect-directional*: the class knows about the aspect but not vice versa
- *closed*: neither the aspect nor the class knows about the other

An *aspect weaver* is responsible for merging 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 [16, 19].

Different AOP techniques and research directions can be identified. They all share the common goal of providing an improved separation of concerns. AspectJ [3] is an aspect-oriented extension to Java. The environment offers an aspect language to formulate the aspect code separately from Java class code, a weaver and additional development support. AOP extensions to other languages have also been developed. [8] describes an aspect language and a weaver for Smalltalk. An aspect language for the domain of robust programming can be found in [11].

Other AOP approaches aiming at achieving a similar separation of concerns include subject-oriented programming [13], composition filters [1] and adaptive programming [23, 28]. In subject-oriented programming different subjective perspectives on a single object model are captured. Applications are composed of “subjects” (i.e. partial object models) by means of declarative composition rules. The composition filters approach extends an object with input and output filters. These filters are used to localise non-functional code. Adaptive programming is a special case of AOP where one of the building blocks is expressible in terms of graphs. The other building

¹ The categorisation determines the reusability of classes and aspects

blocks refer to the graphs using traversal strategies. A traversal strategy can be viewed as a partial specification of a class diagram. This traversal strategy cross-cuts the class graphs. Instead of hard-wiring structural knowledge paths within the classes, this knowledge is separated.

AOP also bears a close relation to generative programming [10] and reflection [41]. Experience reports and assessment of AOP can be found in [17, 31].

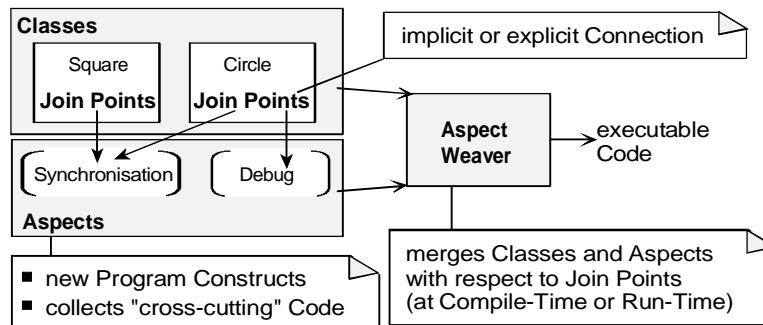


Fig. 1. Aspect Oriented Programming

3 Aspects in Object-Oriented Databases

In this section we identify some of the cross-cutting concerns in object databases and discuss how these can be addressed effectively using aspects. The first three examples are from the evolution domain. Our previous work on object database evolution [32, 33, 34, 35, 36, 37, 39] formed the motivation behind exploring the applicability of aspects in this area. This is followed by an example based on clustering. Some other cross-cutting concerns have also been pointed out. It should be noted that the following discussion regards aspects as persistent abstractions residing in the database. This is a natural extension of existing work on lifetime of aspects [16, 19, 26] which argue that at least some of the aspects should live for the lifetime of the program execution and not die at compile-time.

3.1 Instance Adaptation

Our first example is based on instance adaptation during class versioning [5, 29, 40]. 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 essential to ensure structural consistency. A detailed description of class versioning is beyond the scope of this paper. Interested readers are referred to [5, 29, 40].

We first consider the instance adaptation strategy of ENCORE [40]. A similar approach is employed by AVANCE [5]. 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)).

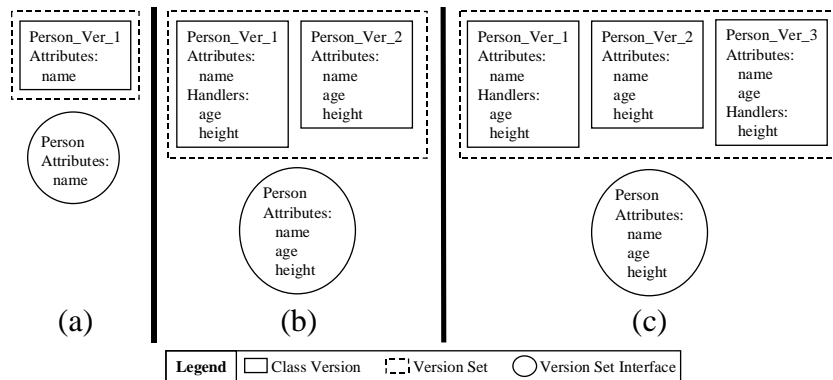


Fig. 2. Class Versioning in ENCORE

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. Besides, 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 our solution 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. Links between aspects and class versions are *open* [17] as an aspect needs to know about the various class versions it can be applied to while a class version needs to be aware of the aspect that needs to be woven to exhibit a specific interface. 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.

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, the use of update/backdate methods for dynamic instance conversion between class versions (as opposed to simulating a conversion) [29]. 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 [38].

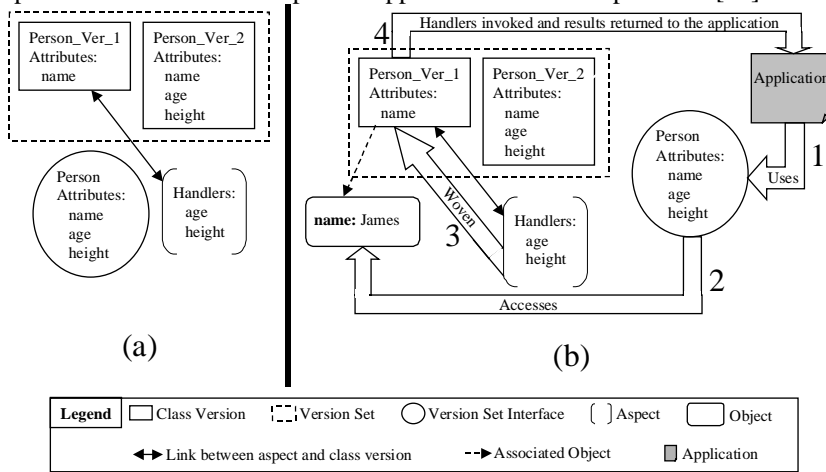


Fig. 3. The Aspect-Oriented Instance Adaptation Approach

3.2 Inheritance

The modification of the inheritance hierarchy in an object database requires rules to resolve conflicts between locally defined and inherited class members [4, 34]. If the system supports multiple inheritance additional rules need to be defined for duplication resolution due to the multiple-path problem [4, 34]. These rules are implemented in a separate system component which is delegated the responsibility of enforcing these rules when new classes are introduced or existing inheritance relationships modified. Alternatively, in a self-descriptive object-oriented model they are implemented in the system class used to instantiate the class meta-objects. In either case modifying the behaviour of the rules introduces an additional evolution problem as consequences for existing classes and their instances can be severe. Besides, there is an overhead in terms of checking for duplication resolution when the system supports multiple inheritance but the inheritance chain does not include any multiple inheritance links. These problems can be addressed by considering inheritance as an aspect of the system.

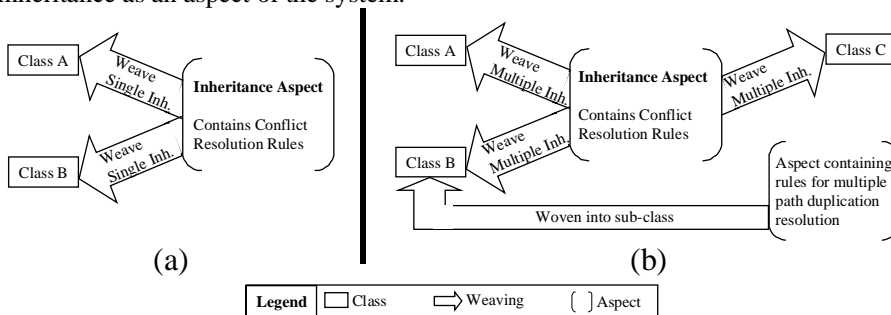


Fig. 4. Inheritance as an aspect of a system

As shown in fig. 4 inheritance relationships can be woven into classes using an *inheritance aspect*. Existing inheritance relationships can be unwoven and new ones woven in order to modify the inheritance hierarchy. Conflict resolution rules need to be woven or rewoven only if they have not previously been woven or have been modified respectively. It should be noted that the conflict resolution rules can be encapsulated in a separate aspect. For simplicity we have included them in the inheritance aspect. If the conflict resolution rules are modified only the conflict resolution aspect would need to be rewoven. Provided the system offers an evolution framework with a flexible instance adaptation strategy such as the one described in section 3.1 changes to instances of existing classes can be propagated relatively easily. As shown in fig. 4(b) duplication resolution rules to counter the multiple-path problem need to be introduced into the classes using multiple inheritance only². This reduces the overhead of checking for duplication resolution when multiple inheritance is not being used. This also makes it possible to swiftly transform a single inheritance system to a multiple inheritance one and vice versa.

3.3 Versioning

Versioning of objects has always been a key requirement for databases especially systems aiming at providing integrated support for applications such as computer aided design (CAD), computer aided software engineering (CASE), etc. Existing work on versioning e.g. [4, 14] recognises that all objects residing in the database do not need to be versionable. If versioning features are encapsulated in an aspect these can be woven into those objects only that need to be versioned and not others. This also reduces the overhead to check whether an object is versioned or not. Besides, if the system supports class versioning, a generic versioning aspect could encapsulate the versioning functionality while specific aspects could provide additional object versioning and class versioning semantics to objects and classes respectively.

3.4 Clustering

Traditionally it is the task of the 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 would allow 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. When the clustering aspects are woven an efficient storage strategy can be worked out by the system³. 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 will help 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 Issues relating to weaving an aspect as a consequence of another aspect being woven have been discussed in [20]

³ This can be done by the weaver.

3.5 Other possible aspects

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 would simplify their specification and management. Access rights, security and data representation can also be regarded as aspects.

4 An Aspect-Oriented Extension of an OODB

In [36] we described a higher level model of an object-oriented database. The model is based on the observation that three types of entities exist within an object-oriented database. These are:

- Objects
- Meta-objects
- Meta-classes

As shown in fig. 5(a) entities of each type reside within a specific *virtual space* within the database. Objects reside in the *object space* and are instances of classes. Classes together with defining scopes, class properties, class methods, etc. form the *meta-object space* [35]. Meta-objects are instances of meta-classes which constitute the *meta-class space*.

For prototyping the aspects presented in section 3 we have extended the model with an *aspect-space* (fig. 5(b)). Aspects reside in the aspect space and are instances of the meta-class aspect. For simplicity we have shown only the versioning aspect.

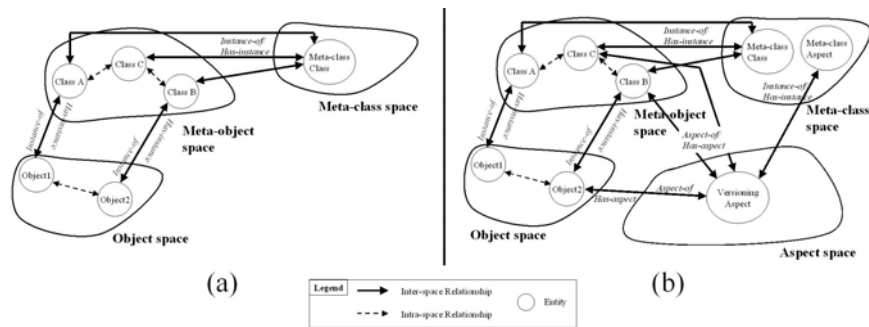


Fig. 5. Virtual Spaces in an Object Database

Fig. 5(b) also shows that the notion of inter-space⁴ and intra-space relationships⁵ introduced in [36] seamlessly extends to the aspect space. Aspects in the aspect space bear *aspect-of/has-aspect* relationships with entities residing in other virtual spaces. These relationships indicate an *open link* [17] between an aspect and the corresponding entity. *Class-directional* and *aspect-directional* links [17] can also be

⁴ Inter-space relationships are those where participating entities reside in different virtual spaces e.g. relationships among meta-objects and objects.

⁵ Intra-space relationships are those where the participating entities belong to the same virtual space e.g. relationships among objects, relationships among meta-objects.

created as the system supports uni-directional relationships [36]. Implementation of the semantics of *closed* links has not yet been explored. Although not shown in figure 5(b) intra-space relationships can exist within an aspect space. For example, a specific class versioning aspect can have a specialisation relationship with the versioning aspect. Note that the specialisation does not need to have semantics similar to the specialisation relationship (inheritance) in object-oriented models. It can simply be a conceptual specialisation.

As shown in figure 5(b), versioning is an aspect of Object 2, Class B and Class C. These are the entities that will be versioned. Class A and object 1 will not be versioned. It should be noted that object 1 could have a versioning aspect despite Class A not having one.

The aspect-oriented extension has been layered on top of the Jasmine object database management system⁶. Weaving is carried out by weavers developed for the specific purposes. Development of a more sophisticated weaver is in progress. It is worth mentioning that the aspect-oriented instance adaptation strategy has been incorporated into the SADES evolution system [32, 33, 34, 37, 39] and has proved to be a more flexible implementation as compared to instance adaptation in existing systems [38].

5 Open Issues

This paper introduces the idea of persistent aspects. One of the open research issues is the persistent representation of an aspect. This is because, besides aspects in the persistence domain described in this paper, application programs employing aspect-oriented programming techniques and run-time aspects will require aspects to outlive the program execution i.e. there will be a need to make aspects persistent. An example 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. [31] also identifies the need of an aspect repository when employing aspects in a distributed environment. 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.

Other issues that need to be addressed include the applicability of the concepts in relational, object relational, active and deductive databases, etc. Another area worth exploring is whether querying support should be extended to aspects residing in the database. *Aspect-based querying and retrieval* can also pose interesting research questions.

6 Related Work

Although separation of concerns in object-oriented databases has not been explicitly considered, some of the existing work falls in this category. The concept of object version derivation graphs [25] separates version management from the objects. A similar approach is proposed by [32, 33, 34, 37, 39] where version derivation graphs manage both object and class versioning. Additional semantics for object and class versioning are provided separately from the general version management technique. [24] employs propagation patterns [21, 22] to exploit polymorphic reuse mechanisms

⁶ <http://www.cai.com/>

in order to minimise the effort required to manually reprogram methods and queries due to schema modifications. Propagation patterns are behavioural abstractions of application programs and define patterns of operation propagation by reasoning about the behavioural dependencies among co-operating objects. [36] implements inheritance links between classes using semantic relationships which are first class-objects. The inheritance hierarchy can be changed by modifying the relationships instead of having to alter actual class definitions. In the *hologram approach* proposed by [2] an object is implemented by multiple instances representing its many faceted nature. These instances are linked together through aggregation links in a specialisation hierarchy. This makes objects dynamic since they can migrate between the classes of a hierarchy hence making schema changes more pertinent.

7 Conclusions

The use of AOP to achieve a better separation of concerns has shown promising results [17]. Although some of the existing work in object-oriented databases implicitly addresses cross-cutting concerns, no attempts have been made to capture these explicitly. The novelty of our work is in extending the notion of aspects to object-oriented databases in order to capture these concerns explicitly. We have identified a number of cross-cutting features and discussed how these can be effectively addressed using aspects. Some examples have been discussed in order to demonstrate the effectiveness of aspects in localising the impact of changes, hence making maintenance easier. We have presented an aspect-oriented extension of an object database which is used to prototype the various examples. The extension is natural and seamless and does not affect existing data or applications. Our work in the immediate future will focus on persistent representations of aspects and development of efficient weaving mechanisms. We believe that these issues will be crucial for the effective application of aspects in object databases. Development of an aspect-oriented evolution framework for object databases is another area of interest. We are also interested in exploring the applicability of aspects in database technology other than OODBs.

References

- [1] Aksit, M., Tekinerdogan, B., "Aspect-Oriented Programming using Composition Filters", *Proceedings of the AOP Workshop at ECOOP '98, 1998*
- [2] Al-Jadir, L., Leonard, M., "If We Refuse the Inheritance ...", *Proceedings of DEXA 1999, LNCS 1677, pp. 560-572*
- [3] AspectJ Home Page, <http://aspectj.org/>, Xerox PARC, USA
- [4] Banerjee, J. et al., "Data Model Issues for Object-Oriented Applications", *ACM Transactions on Office Information Systems, Vol. 5, No. 1, Jan. 1987, pp. 3-26*
- [5] 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
- [6] Blair, L., Blair, G. S., "The Impact of Aspect-Oriented Programming on Formal Methods", *Proceedings of the AOP Workshop at ECOOP '98, 1998*
- [7] Blair, L., Blair, G. S., "A Tool Suite to Support Aspect-Oriented Specification", *Proceedings of the AOP Workshop at ECOOP '99, 1999*
- [8] Boellert, K., "On Weaving Aspects", *Proc. of the AOP Workshop at ECOOP '99*
- [9] Clarke, S., et al., "Separating Concerns throughout the Development Lifecycle", *Proceedings of the AOP Workshop at ECOOP '99, 1999*
- [10] Czarniecki, K., "Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-based Component Models", *PhD Thesis, Technical University of Ilmenau, Germany, 1999*

- [11] Fradet, P., Suedholt, M., "An Aspect Language for Robust Programming", *Proceedings of the AOP Workshop at ECOOP '99, 1999*
- [12] Gamma, E. et al., "Design Patterns - Elements of Reusable Object-Oriented Software", Addison Wesley, c1995
- [13] Harrison, W., Ossher, H., "Subject-Oriented Programming (A Critique of Pure Objects)", *Proceedings of OOPSLA 1993, ACM SIGPLAN Notices, Vol. 28, No. 10, Oct. 1993, pp. 411-428*
- [14] Katz, R. H., "Toward a Unified Framework for Version Modeling in Engineering Databases", *ACM Computing Surveys, Vol. 22, No. 4, Dec. 1990, pp. 375-408*
- [15] Kendall, E.A., "Role Model Designs and Implementations with Aspect Oriented Programming", *Proceedings of OOPSLA 1999, ACM SIGPLAN Notices, Vol. 34, No. 10, Oct. 1999, pp. 353-369*
- [16] Kenens, P., et al., "An AOP Case with Static and Dynamic Aspects", *Proceedings of the AOP Workshop at ECOOP '98, 1998*
- [17] 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*
- [18] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J., Lopes, C., Maeda, C., Mendhekar, A., "Aspect-Oriented Programming", *ACM Computing Surveys, Vol. 28, No. 4, Dec. 1996*
- [19] 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*
- [20] Klaeren, H., Pulvermueller, E., Rashid, A., Speck, A., "Supporting Composition using Assertions, Computing Department, Lancaster University, Technical Report No: CSEG/4/00
- [21] Lieberherr, K. J., Huersch, W., Silva-Lepe, I., Xiao, C., "Experience with a Graph-Based Propagation Pattern Programming Tool", *Proc. of the International CASE Workshop, IEEE Computer Society 1992, pp. 114-119*
- [22] Lieberherr, K. J., Silva-Lepe, I., Xiao, C., "Adaptive Object-Oriented Programming using Graph-Based Customization", *CACM, Vol. 37, No. 5, May 1994, pp. 94-101*
- [23] Lieberherr, K. J., "Demeter", <http://www.ccs.neu.edu/research/demeter/index.html>
- [24] Liu, L., Zicari, R., Huersch, W., Lieberherr, K. J., "The Role of Polymorphic Reuse Mechanisms in Schema Evolution in an Object-Oriented Database", *IEEE Transactions of Knowledge and Data Engineering, Vol. 9, No. 1, Jan.-Feb. 1997, pp. 50-67*
- [25] Loomis, M. E. S., "Object Versioning", *JOOP, Jan. 1992, pp. 40-43*
- [26] Matthijs, F., et al., "Aspects should not Die", *Proceedings of the AOP Workshop at ECOOP '97, 1997*
- [27] Mens, K., Lopes, C., Tekinerdogan, B., Kiczales, G., "Aspect-Oriented Programming Workshop Report", *ECOOP '97 Workshop Reader, LNCS 1357, pp. 483-496*
- [28] 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*
- [29] Monk, S. & Sommerville, I., "Schema Evolution in OODBs Using Class Versioning", *SIGMOD Record, Vol. 22, No. 3, Sept. 1993, pp. 16-22*
- [30] Pazzi, L., "Explicit Aspect Composition by Part-Whole Statecharts", *Proceedings of the AOP Workshop at ECOOP '99, 1999*
- [31] Pulvermueller, E., Klaeren, H., Speck, A., "Aspects in Distributed Environments", *Proceedings of GCSE 1999, Erfurt, Germany (to be published by Springer-Verlag)*
- [32] Rashid, A., Sawyer, P., "Facilitating Virtual Representation of CAD Data through a Learning Based Approach to Conceptual Database Evolution Employing Direct Instance Sharing", *Proceedings of DEXA '98, LNCS 1460, pp. 384-393*
- [33] Rashid, A., "SADES - a Semi-Autonomous Database Evolution System", *Proceedings of PhDOOS '98, ECOOP '98 Workshop Reader, LNCS 1543*
- [34] Rashid, A., Sawyer, P., "Toward 'Database Evolution': a Taxonomy for Object Oriented Databases", *In review at IEEE Transactions on Knowledge and Data Engineering*
- [35] Rashid, A., Sawyer, P., "Evaluation for Evolution: How Well Commercial Systems Do", *Proceedings of the First OODB Workshop, ECOOP '99, pp. 13-24*
- [36] Rashid, A., Sawyer, P., "Dynamic Relationships in Object Oriented Databases: A Uniform Approach", *Proceedings of DEXA '99, LNCS 1677, pp. 26-35*
- [37] Rashid, A., Sawyer, P., "Transparent Dynamic Database Evolution from Java", *Proceedings of OOPSLA 1999 Workshop on Java and Databases: Persistence Options*
- [38] Rashid, A., Sawyer, P., Pulvermueller, E., "A Flexible Approach for Instance Adaptation during Class Versioning", *To Appear in Proceedings of ECOOP 2000 OODB Symposium*
- [39] "SADES Java API Documentation", <http://www.comp.lancs.ac.uk/computing/users/marash/research/sades/index.html>
- [40] 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*
- [41] Suzuki, J., Yamamoto, Y., "Extending UML for Modelling Reflective Software Components", *Proceedings of UML '99*