

A Hybrid Approach to Separation of Concerns: The Story of SADES

Awais Rashid

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

Abstract. A number of approaches have been proposed to achieve separation of concerns. Although all these approaches form suitable candidates for separating cross-cutting concerns in a system, one approach can be more suitable for implementing certain types of concerns as compared to the others. This paper proposes a hybrid approach to separation of concerns. The approach is based on using the most suitable approach for implementing each cross-cutting concern in a system. The discussion is based on using three different approaches: composition filters, adaptive programming and aspect-oriented programming to implement cross-cutting concerns in SADES, a customisable and extensible object database evolution system.

1. Introduction

The *separation of concerns* principle proposes encapsulating cross-cutting features [15] into separate entities in order to localise changes to these cross-cutting features and deal with one important issue at a time. A number of approaches have been proposed to achieve separation of concerns. Some of the more prominent of these are meta-object protocols [14], composition filters [1, 3], aspect-oriented programming [15, 30], adaptive programming [17, 19], subject-oriented programming [10] and hyperspaces [11]. All these approaches form suitable candidates for implementing separation of concerns in a system. For example, cross-cutting concerns in collaboration-based designs have been implemented using both adaptive programming [19] and aspect-oriented programming [21]. However, in some scenarios one approach can be more suitable as compared to the others. For example, consider the following three approaches for separation of concerns:

- Composition filters [1, 3]
In this approach separation of concerns is achieved by extending an object with input and output filters. Filters can be of various types e.g. dispatch filters (for delegating messages), wait filters (for buffering messages), error filters (for throwing exceptions), etc.
- Adaptive programming [17, 19]
In this approach the behaviour code is written against a partial specification of a class diagram instead of the complete class diagram itself. The partial specification is referred to as a *traversal strategy* and only specifies the parts of the class diagram essential for a computation.

- Aspect-oriented programming [15, 30]

In this approach special program abstractions known as *aspects* are employed to separate any cross-cutting concerns. The links between aspects and program modules (e.g. classes) cross-cut by them are represented using special reference points known as *join points*. An aspect weaver tool is used to merge the aspects and the program modules (e.g. classes) with respect to the join points at compile-time or run-time.

Composition filters are very effective in implementing concerns involving message interception and execution of actions before and after executing a method [8]. Adaptive programming, on the other hand, is more suitable for separating behaviour from object structure [8]. In contrast to these two approaches based on composition mechanisms, aspect languages and weaving tools (in aspect-oriented programming) are more effective when concerns need to be explicitly represented using linguistic constructs [8]. As a result some concerns in a system can be more suitably implemented using one approach while the others can be more effectively addressed using other approaches. This can be perceived as a “meta separation of concerns”: employing the most suitable approach for implementing a concern or set of concerns. This paper discusses the use of multiple separation of concerns techniques during the implementation of the SADES object database evolution system [23, 24, 25, 26]. SADES employs a class versioning approach to evolve the schema of the object database. In class versioning a new version of a class is created each time it is modified. Applications and objects are bound to individual class versions. Objects can, however, be converted across class versions or made to simulate a conversion. This is termed *instance adaptation*. A detailed description of class versioning is beyond the scope of this paper. Interested readers can refer to [20, 24, 26, 29].

One of the key motivations behind SADES was the development of an extensible and customisable object database evolution system which could be customised to the specific needs of an organisation or application¹. Another motivation was to localise the impact of changes during schema evolution. Separation of cross-cutting concerns in SADES was, therefore, essential in order to localise the impact of changes during customisation and schema evolution. For each concern the most suitable technique was employed to separate it from the rest of the system. As a result three different techniques were used namely composition filters, adaptive programming and aspect-oriented programming. However, specific implementation tools for these three approaches e.g. Sina (for composition filters) [3], Demeter (for adaptive programming) [17] and AspectJ (for aspect-oriented programming) [30] were not used. This was due to the fact that SADES has been implemented on top of the commercially available object database management system Jasmine [12] and makes extensive use of the proprietary language ODQL (Object Data Query Language) and its associated interpreter in order to gain access to some low-level Jasmine features. In addition, most of the concerns cut across persistent entities (entities that live beyond program execution) and, hence, are persistent by nature. To the best of the author’s knowledge existing tools for these three techniques do not take into account persistent

¹ Database evolution case studies at an adult education organization, where day-to-day activities revolve around the database, showed that evolution requirements can vary considerably across organisations and applications [26].

nature of concerns. Therefore, only general concepts and not specific tools (such as Sina, Demeter and AspectJ) from the three separation of concerns approaches have been employed. Specific tools to implement these techniques (e.g. aspect language and weaver for aspect-oriented programming), wherever required, have been developed within the constraints imposed by the SADES requirements and the features of the underlying Jasmine object database management system. From this point onwards, unless otherwise stated, the terms composition filters, adaptive programming and aspect-oriented programming refer to the general concepts proposed by these approaches and not any specific tools.

The next section in this paper describes the SADES architecture. Section 3 discusses the hybrid approach to separation of concerns and the various concerns implemented in SADES using the three separation of concerns techniques mentioned above. Section 4 discusses some related work while section 5 concludes the paper and identifies directions for future work.

2. Overview of SADES Architecture

The separation of concerns principle has been employed during all the stages of development of SADES. Therefore, it also forms the basis of the system architecture. As show in fig. 1 the architecture is based on dividing the database system into a set of *spaces* each hosting entities of a particular type. Three basic types of entities and their encapsulating spaces are shown:

- Objects: reside in the *object space* and are instances of classes.
- Meta-objects: Classes, class properties, class methods, etc. form the *meta-object space*.
- Meta-classes: constitute the *meta-class space* and are used to instantiate meta-objects.

Note that fig. 1 shows only three basic types of entities residing in an object database. Later, while discussing the instance adaptation approach in SADES, it will be demonstrated how the above architecture makes it possible to extend the database with new types of entities (and their encapsulating spaces) with minimal effort.

The links between entities residing in different spaces are maintained through *inter-space relationships*. Fig. 1 shows the bi-directional instantiation inter-space relationships between meta-classes and meta-objects, and meta-objects and objects. *Intra-space* relationships can exist among entities residing within the same space. In the meta-object space, for example, these can be the *derives-from/inherited-by* inheritance relationships among class meta-objects or the *defines/defined-in* aggregation relationships among class meta-objects and attribute and method definition meta-objects. In the object space these can be association and aggregation relationships among the various objects residing in that space.

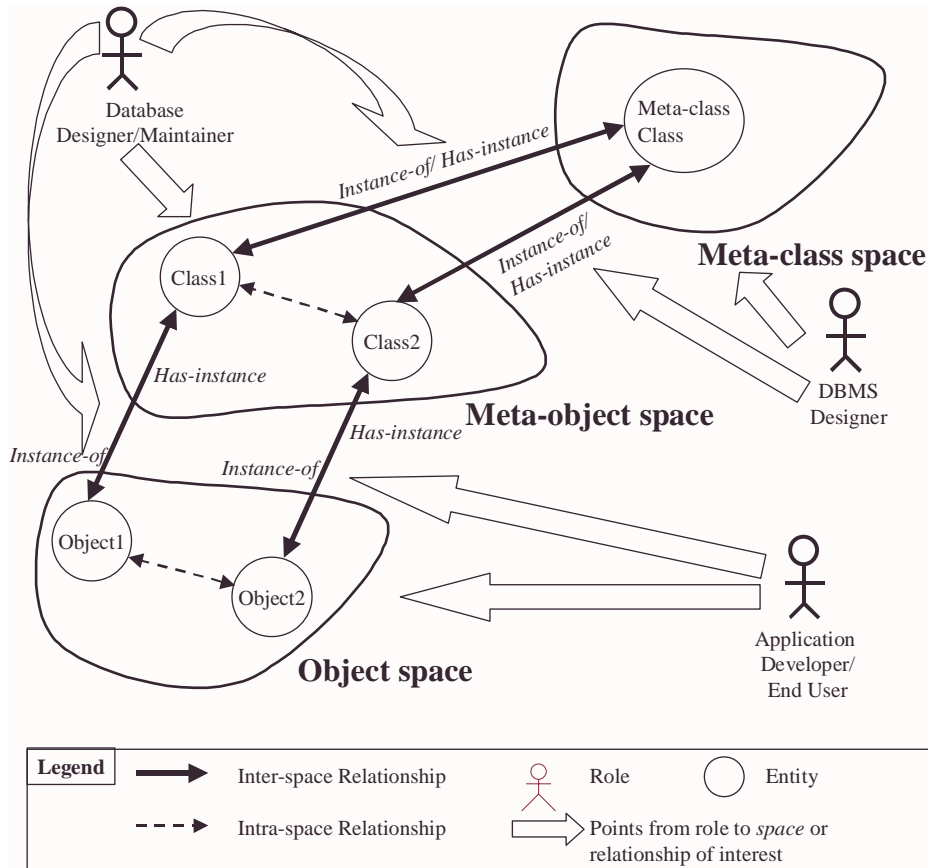


Fig. 1. Spaces, inter-space and intra-space relationships in SADES

As shown in fig. 1 the notion of spaces and inter-space and intra-space relationships has been derived directly from the viewpoints of the various developer/user roles in the database environment:

- The DBMS designer is interested in providing a suitable set of meta-classes that can be instantiated by the database designer/maintainer to create the database schema. S/he is also interested in providing means for introducing new meta-classes so that the system can be extended with new types of entities if required. The new meta-classes might bear some relationships with existing meta-classes. The meta-class space and the notion of intra-space relationships in the meta-class space have been derived from these DBMS designer viewpoints.
- The inter-space relationships between the meta-class space and the meta-object space have been derived from two overlapping viewpoints: that of the DBMS designer and that of the database designer/maintainer. The DBMS designer is interested in providing effective instantiation mechanisms for the meta-classes while the database designer/maintainer is interested in using these mechanisms to create the meta-objects forming the database schema.

- The database designer/maintainer is also interested in creating appropriate meta-objects forming the database schema which can then be instantiated by the application developer/end user to store application data in the database. S/he is also interested in modifying the relationships among meta-objects (e.g. *derives-from/inherited-by* relationships among class meta-objects or *defines/defined-in* relationships among class meta-objects and attribute and method definition meta-objects), introducing new relationships among meta-objects or removing existing ones in order to evolve the schema of the database in line with changes in application requirements. The meta-object space and the notion of intra-space relationships in this space have been derived from these database designer/maintainer viewpoints.
- The overlapping viewpoints of the database designer/maintainer and the application developer/end user roles form the basis of the inter-space relationships between the meta-object space and the object space. The database designer/maintainer is interested in exposing appropriate instantiation mechanisms for the meta-objects while the application developer/end user is interested in using these mechanisms to instantiate the meta-objects and populate the database.
- The application developer/end user is also interested in manipulating the objects in the database and any association or aggregation links among them. These viewpoints have been used to derive the notion of an object space and its intra-space relationships.

The viewpoint based division of the database into spaces and identification of the various inter-space and intra-space relationships makes it possible to address the requirements of the various roles in an incremental fashion. In SADES this has provided a clearer mapping from the extensibility and customisability requirements of the various roles to the system architecture making it possible to extend the system with new types of entities (introduction of new meta-classes), spaces, inter-space and intra-space relationships and customising existing spaces by manipulating their inter-space and intra-space relationships with minimal knock-on effect on entities already residing in the existing spaces.

3. Hybrid Separation of Concerns in SADES

The hybrid approach is based on the observation that one separation of concerns technique can be more suitable for implementing certain types of concerns in comparison with other techniques. Consequently multiple techniques can be employed to implement different types of concerns in a system. Fig. 2 shows how this can be achieved in three different fashions:

- Use one particular approach to implement a set of interrelated and overlapping concerns (cf. fig. 2(a)).
This approach is suitable when all concerns in the set of interrelated and overlapping concerns can be suitably implemented using the particular approach.
- Use multiple approaches to implement different concerns in a set of interrelated and overlapping concerns (cf. fig. 2(b)).

This approach is suitable when different concerns in the set of overlapping and interrelated concerns can be better implemented using different techniques.

- Implement concerns in different system layers using the same or different sets of approaches (cf. fig. 2(c)).

Separation of concerns in the higher layer builds upon the separation of concerns based implementation in the lower layer (cf. fig. 2(c)). Approaches in fig. 2(a) and 2(b) can be used to implement different sets of concerns in each system layer.

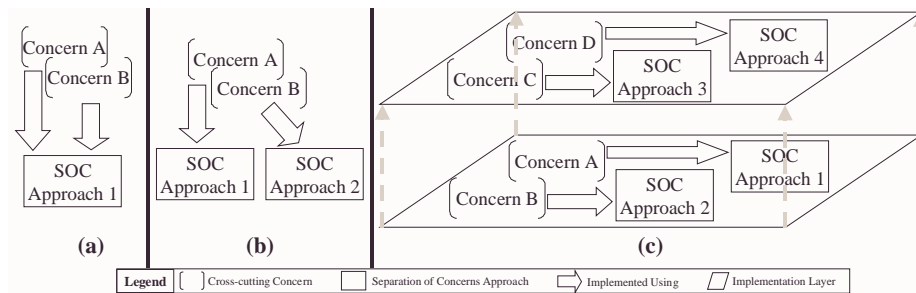


Fig. 2. Hybrid approach to separation of concerns

The following sections describe how the above hybrid approach has been employed to implement SADES. Section 3.1 discusses the implementation of three interrelated and overlapping concerns:

- links among entities within a space or different spaces
- propagation of changes to the links
- maintenance of referential integrity

using two different approaches: composition filters and adaptive programming. Section 3.2 describes the implementation of versioning in SADES using adaptive programming. This versioning implementation is built on top of the system layer implementing links, change propagation and referential integrity. Section 3.3 discusses the use of aspect-oriented programming and composition filters to implement a customisable instance adaptation approach in SADES. This system layer exploits both the versioning implementation and implementation of links, change propagation and referential integrity.

3.1 Implementation of Links among Entities

In the SADES architecture (cf. fig. 1) the links among entities, whether they reside in the same space or different spaces, are represented by relationships. Relationships are semantic constructs which are natural to manipulate for maintainers and developers of object-oriented system. Therefore, relationships in the SADES architecture have been directly mapped onto implementation. The links among entities are implemented as relationship constructs which are first class objects and encapsulate information about connections among the entities. This results in connection information being

separated from the entities localising changes to these connections. This is in direct contrast with existing object database evolution systems, e.g. ORION [2], ENCORE [29], CLOSQL [20], which embed connection information within the entities (cf. fig. 3 (a)) hence spreading the relationships across them. Fig. 3 (b) shows an example schema evolution scenario for a system embedding inheritance links within class meta-objects. In this scenario a meta-object for the class *Staff* forming a non-leaf node in the class hierarchy graph is introduced into the system. All the references to subclass meta-objects will have to be removed from the *Person* meta-object and all the subclass meta-objects will have to be updated to remove the reference to *Person* in their respective collections of superclass references. A reference to the *Person* meta-object will have to be added to the superclasses collection and references to older subclasses of *Person* will have to be added to the subclasses collection in the *Staff* meta-object. A reference to the *Staff* meta-object will have to be added to the subclasses collection (not shown in fig. 3 (b)) in *Person* and to the superclasses collection in each of its subclasses. The number of entities affected upon modification of a connection is $m+n$ where m and n represent the number of participating entities at each edge of the relationship (in this case super-classes edge and sub-classes edge).

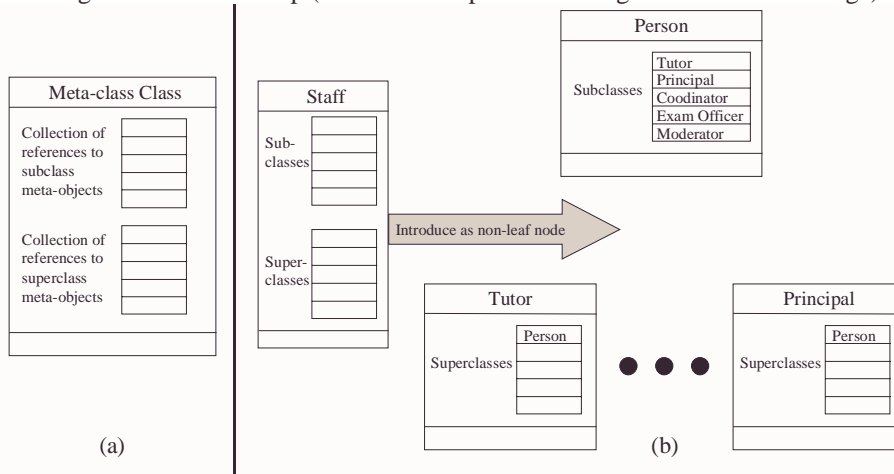


Fig. 3. Implementation of links among entities in existing object database evolution systems

The above example demonstrates that information about connections among entities cuts across them. In SADES this information is separated and encapsulated in the relationship objects. A composition filters mechanism is employed to ensure that any messages manipulating links among entities are received by the relationship objects and not by the entities connected by them. As mentioned earlier composition filters are ideal candidates for situations requiring message interception. In SADES they are implemented as first class objects hence not requiring any extensions to the basic system architecture shown in fig. 1. The method invocation mechanism in SADES has been adapted to keep track of entities with attached filters. It ensures that all messages to and from an entity get routed through its attached input and output filters respectively. As shown in fig. 4 a dispatch filter intercepts all incoming relationship manipulation messages and delegates them to the relationship objects. Since the

relationship information is no longer embedded within the participating entities it can be modified in an independent fashion. It is also possible to introduce new relationships or remove existing ones with localised changes. This results in cost-effective schema modifications (changes to intra-space relationships in the meta-object space). It also improves the extensibility of the system as introduction of new meta-classes only requires introduction of new relationship objects. A detailed description of the structure and semantics of the relationship objects and their effectiveness during schema evolution can be found in [23].

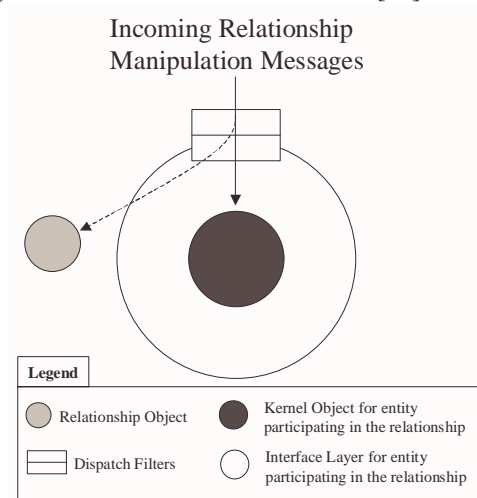


Fig. 4. Implementation of links among entities in SADES using relationship objects and composition filters

Closely related and overlapping with the notion of links or relationships among entities are the concepts of change propagation and referential integrity. Referential integrity means that all entities must always refer to other existing entities. This implies that relationship objects in SADES must always connect entities that exist in the system. If an entity is removed (e.g. a class meta-object during schema evolution) the relationship object should be made aware of this change and must not refer to that entity as a participant in the relationship. Similarly, if an entity no longer participates in a relationship its associated dispatch filter must not attempt to delegate messages to the relationship object.

Change propagation means that any changes to a particular type of relationship or its participating entities should be propagated to the other parts of the system (e.g. other participating entities) in line with the propagation semantics defined for that type of relationship. A number of different types of relationships with different propagation semantics exist in SADES. Inheritance relationships, for example, require that changes to the structure of a class be propagated to its sub-classes. Similarly, aggregation relationships require that operations be propagated from aggregate to part, with deletion of the aggregate resulting in deletion of the parts, but not vice versa. Note that these propagation semantics cannot be fixed as some applications might require changes to them. [31], for example, identifies a number of different

propagation semantics for aggregation relationships. Hence, it must be possible to modify change propagation behaviour with localised changes.

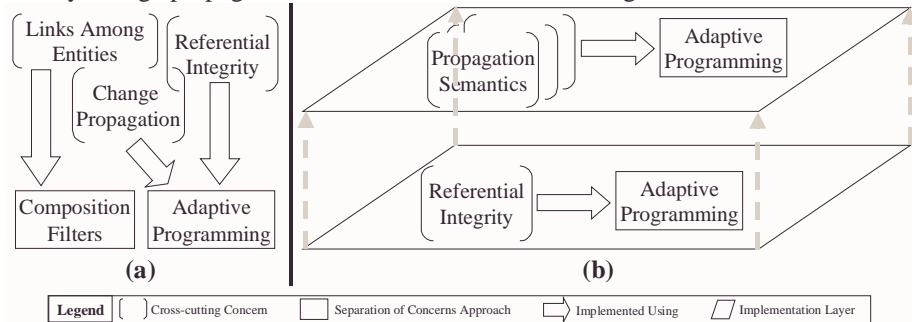


Fig. 5. Implementation of change propagation and referential integrity using adaptive programming

The change propagation and referential integrity behaviour refers to both relationship objects and entities connected by them. It, however, requires very little information about the structure of individual relationship objects or entities. Instead, it requires more generic information such as the type of relationship or the type of evolution operation performed on an entity. Earlier on it was pointed out that adaptive programming is highly suitable for separating behaviour from structure. As shown in fig. 5(a), it has, therefore, been employed to implement change propagation and referential integrity in SADES. A traversal strategy specifies the various operations on relationships and entities requiring referential integrity enforcement. The referential integrity behaviour is described on the basis of this traversal strategy. Similarly, a set of traversal strategies specifies different propagation semantics for different types of relationships in the system. The change propagation behaviour is specified on the basis of these traversal strategies. In SADES traversal strategies are implemented as directed graphs with each node in the graph representing a traversal node in the traversal strategy. Similar to the composition filters implementation this does not require any extensions to the basic system architecture shown in fig. 1. Fig. 6(a) shows a traversal strategy (in pseudo code) for operation propagation in aggregation relationships in SADES. It represents the propagation of operations from aggregate to parts as mentioned earlier. Fig. 6(b) shows the traversal graph dynamically generated when an operation needs to be propagated. The black node represents the aggregate where an operation is initially performed while the first-level gray nodes represent parts of the aggregate to which the operation is propagated. Note that the propagation strategy in fig. 6(a) defines recursive propagation. Therefore (as shown in fig. 6(b)), the operation is propagated to the nodes to which the first-level nodes might be an aggregate.

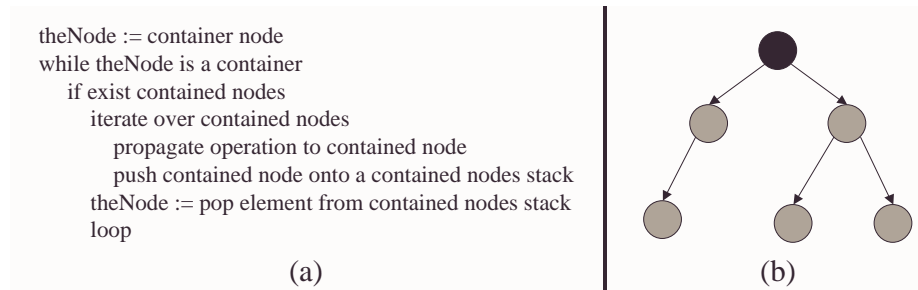


Fig. 6. (a) Traversal strategy for operation (change) propagation in aggregation relationships
 (b) Run-time traversal graph for the traversal strategy in (a)

Fig. 5(b) shows that the change propagation behaviour is built on top of the system layer implementing referential integrity. This is because change propagation often triggers the need to ensure referential integrity e.g. dropping of a class meta-object forming a non-leaf node in the database schema requires that this change be propagated to the sub-classes in line with the propagation semantics for inheritance relationships. However, this also requires that the relationship object for the inheritance relationship no longer refers to the removed meta-object as a participant.

3.2 Versioning

As mentioned earlier SADES employs class versioning to evolve the object database schema. In addition to this support for class versioning SADES also provides support for keeping track of changes to the objects in the object space through object versioning. In object versioning a new version of an object is created each time its state needs preservation. This functionality is essential for supporting applications such as computer-aided design (CAD) and computer-aided software engineering (CASE) where there is a need to keep track of changes to design, documentation and code artefacts. In SADES versioning support for entities is customisable. This is because versioning requirements can vary from one application (or organisation) to another. Mostly the workgroup support features (e.g. checking data in and out of the shared database to private or group workspaces) [13] are strongly dependent on the individual work practices of each organisation. Some applications might want to bind objects into *configurations* [13] and use these configurations as units of versioning. Some applications might also want to take class versioning to a higher granularity i.e. version schemas instead of classes [16, 22]. Some applications might only require support for *linear versioning* [18] (where a version is derived from only one existing version) while others might require support for *branch versioning* [18] (where a version may be derived from multiple existing versions). Therefore, it is essential to separate versioning behaviour from the versioned entities in order to localise changes during customisation of the versioning approach.

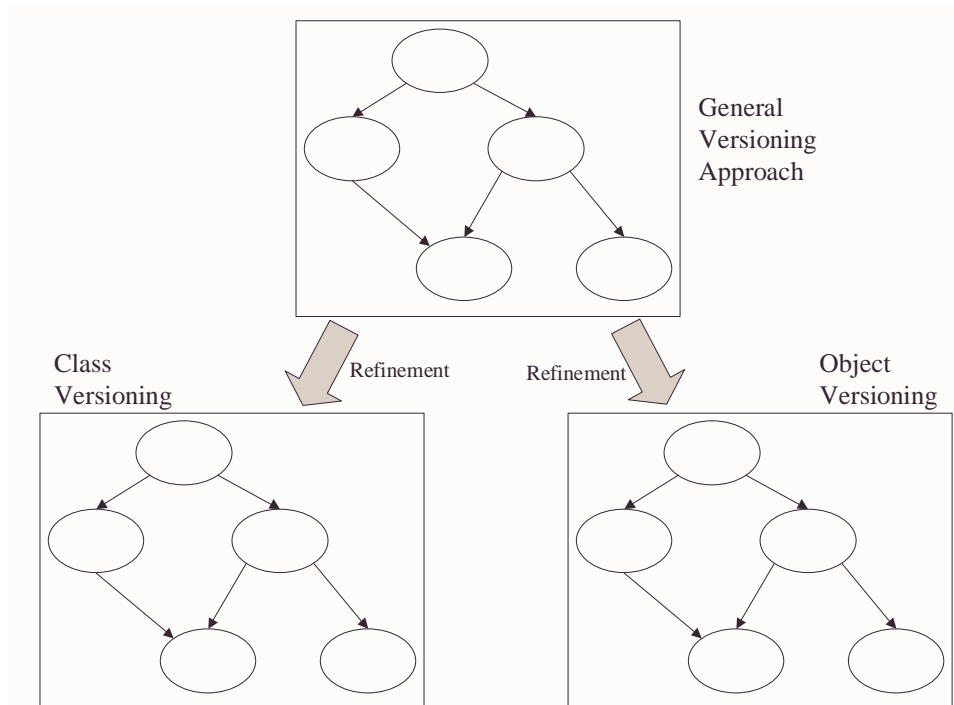


Fig. 7. Traversal strategies based on version derivation graphs

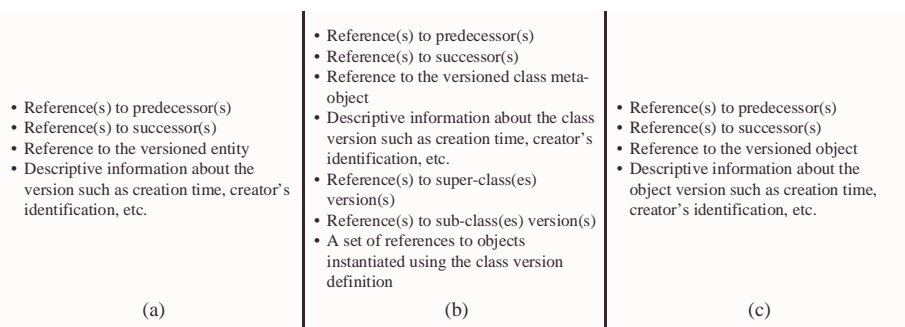


Fig. 8. Structure of: (a) general version derivation graph nodes (b) version derivation graph nodes refined for class versioning (c) version derivation graph nodes refined for object versioning

As mentioned earlier adaptive programming is highly effective in separating behaviour from structure and hence, has been employed to separate versioning behaviour from the versioned entities. As shown in fig. 7 traversal strategies for this separation are specified as version derivation graphs [18]. A general versioning graph exists for any versioned entity. Each node in the general versioning graph has the structure shown in fig. 8(a). Special refinements of the general versioning graph can

be defined for specific types of entities (e.g. for classes and objects as shown in fig. 7). The structures of nodes in class version derivation graphs and object version derivation graphs are shown in fig. 8(b) and 8(c) respectively. The versioning behaviour involving any workgroup support features, special semantics for versions, support for linear or branch versioning, etc. is written against the general structure of the various version derivation graphs.

Fig. 9 shows that the versioning functionality has been implemented in a system layer which exploits the separation of concerns in the system layer implementing links among entities, change propagation and referential integrity. The *predecessor/successor* links among the nodes of version derivation graphs are implemented using relationship objects. New propagation semantics are defined for the *predecessor/successor* relationships to ensure change propagation and referential integrity when these relationships are modified by introduction of new versions or removal of existing ones. Further details of versioning in SADES can be found in [26].

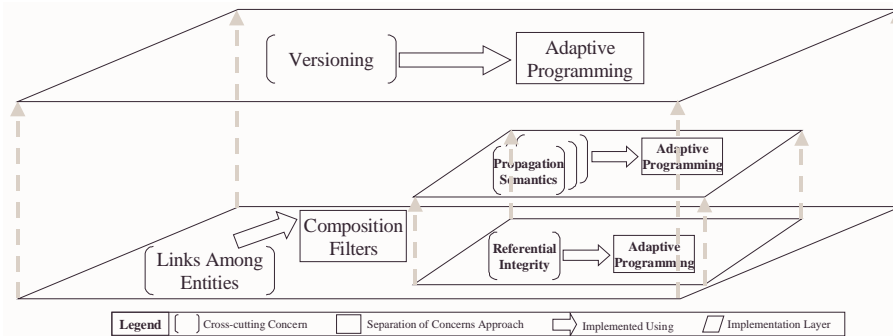


Fig. 9. Implementation of the versioning layer using separation of concerns in the layer implementing links, change propagation and referential integrity

3.3 Instance Adaptation

As mentioned earlier SADES provides support for customising the instance adaptation approach to the specific needs of an organisation or application. For the customisation to be cost effective it is essential to localise the impact of changes. Before discussing the cross-cutting nature of instance adaptation code it is essential to distinguish the *instance adaptation approach* from *instance adaptation routines*. An instance adaptation routine is the code specific to a class version or a set of class versions. This code handles the interface mismatch between a class version and the accessed object. It might be an error handler [29], an update method [20] or a transformation function [9]. The instance adaptation approach is the code which is part of the schema manager and, upon detection of an interface mismatch, invokes the appropriate instance adaptation routine with the correct set of parameters.

Customisation of instance adaptation in existing object database evolution systems is expensive because they introduce the instance adaptation routines directly into the class versions upon evolution. Often 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. Consider the example of ENCORE [29] which employs error handlers to trap interface mismatches and simulate the presence of missing attributes or methods for an object. As shown in fig. 10 error handlers for missing information are introduced into all the former versions of a class upon each additive change: change which modifies the version set interface defined as a union of all the properties and methods of all the versions of a class. If the behaviour of the *address* handler in fig. 10 (c) needs to be customised maintenance has to be carried out on all the former class versions.

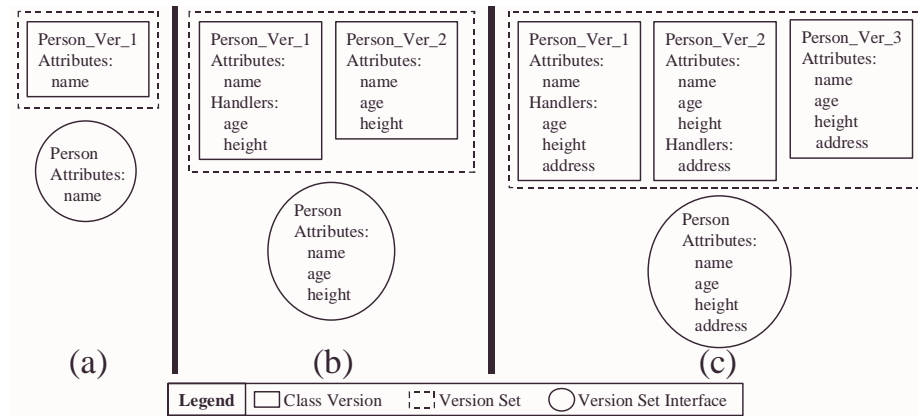


Fig. 10. Instance adaptation routines in ENCORE (a) before evolution (b) upon an additive change (c) upon another additive change

The instance adaptation approach in existing systems is “fixed” as code describing its behaviour is spread across the schema manager. Customisation of the instance adaptation approach or adoption of a new approach is very expensive as it has a large ripple effect on the schema manager and might trigger the need for changes to all, or a large number of existing classes or class versions [24].

The biggest deciding factor in the choice of an appropriate separation of concerns approach for separating the instance adaptation approach from the schema manager and the instance adaptation routines from the class versions was the need to specify them (and any customisations to them) at the application programming level. This would make it possible for the application programmer to customise the instance adaptation mechanism to the specific needs of an organisation or application with minimal effort. This dictated the need to specify these concerns in a declarative fashion. As pointed out earlier, aspect-oriented programming is an effective mechanism to represent concerns in a linguistic fashion. It was, therefore, chosen as a basis for implementing instance adaptation.

The choice of aspect-oriented programming required support for new persistent abstractions, the aspects, in SADES. The extensible architecture of SADES described in section 2 made it possible to extend the system with the notion of an *aspect space* in a seamless fashion. As shown in fig. 11 aspects reside in the aspect space and are instances of the meta-class *Aspect*. For simplicity only one aspect has been shown.

Also note that the concept of inter-space and intra-space relationships (and their implementation through relationship objects and composition filters) seamlessly extends to the aspect space. Aspects in the aspect space bear *aspect-of/has-aspect* relationships with entities residing in other virtual spaces. Although not shown in fig. 11, intra-space relationships can exist within an aspect space.

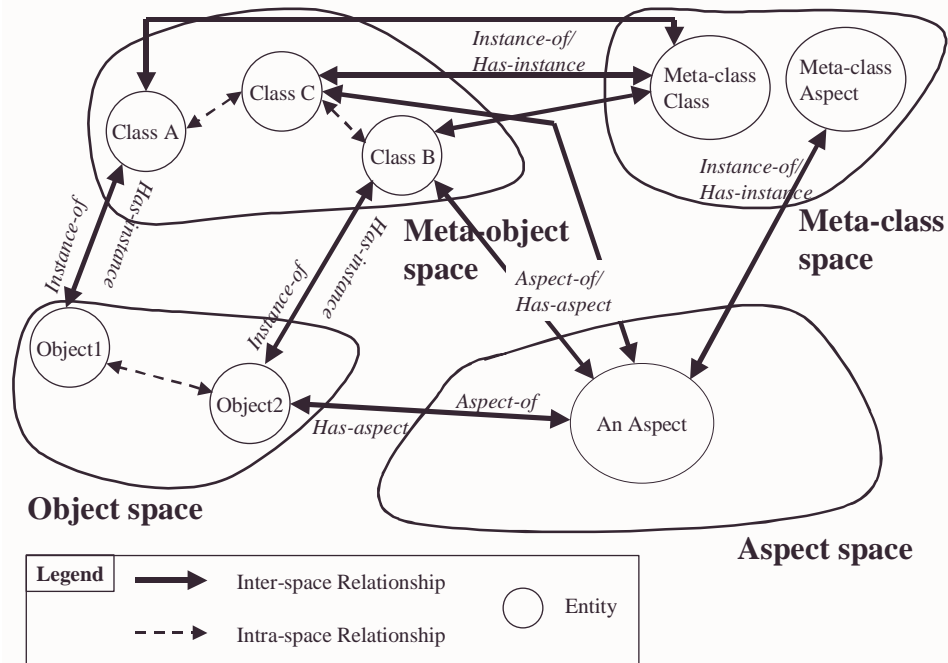


Fig. 11. Extension of SADES with an aspect space

Using the aspect-oriented extension as a basis the instance adaptation approach is separated from the schema manager and encapsulated in an aspect (cf. fig. 12(a)). As shown by the shaded boxes in fig. 12(a) the instance adaptation approach can then be customised or completely exchanged with changes localised to the aspect. The customisation or exchange may be carried out statically or dynamically as the instance adaptation aspect can be woven (or rewoven) into the schema manager at both compile-time and run-time (as shown by the dotted arrow).

In a similar fashion the instance adaptation routines are separated from the class versions using aspects (cf. fig. 12(b)). It should be noted that although fig. 12(b) shows one instance adaptation aspect per class version, one such aspect can serve a number of class versions. Similarly, a particular class version can have more than one instance adaptation aspect. Fig. 12(c) depicts the case when an application attempts to access an object associated with version 1 of *class A* using the interface offered by version 2 of the same class. The aspect containing the instance adaptation routines is dynamically woven into the particular class version. This code is then invoked to return the results to the application.

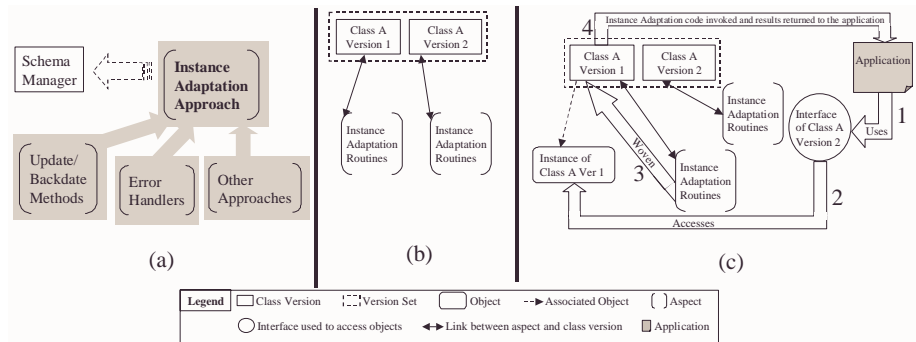


Fig. 12. (a) Customising the instance adaptation approach using aspects (b) Separating instance adaptation routines from class versions using aspects (c) Invocation process for instance adaptation routines

The instance adaptation aspects in fig. 12 are specified using a declarative aspect language modelled on AspectJ [30]. 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 aspect specification provided by the aspect language is parsed to generate the persistent aspects and associate these with the class versions using relationship objects. When an interface mismatch is detected between an object and the class definition used to access it, the weaver dynamically weaves (or reweaves) the required aspect based on a timestamp check. A detailed description of the weaver design and implementation will form the subject of a forthcoming paper.

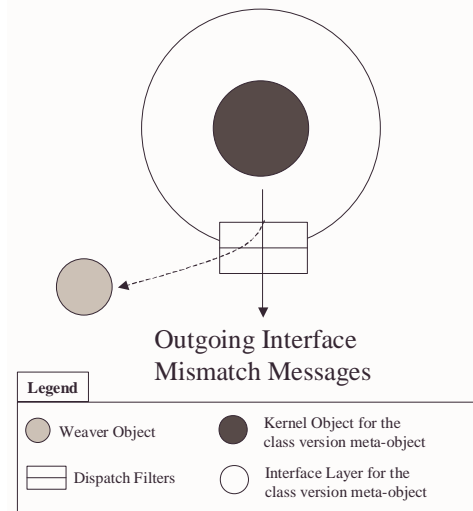


Fig. 13. Interception of interface mismatch messages and their delegation to the weaver using composition filters

In order to trap interface mismatch messages and delegate them to the weaver to weave (or reweave), composition filters are employed (cf. fig 13). Composition filters are very effective in message interception and hence are an ideal choice for this purpose. An output dispatch filter intercepts any interface mismatch messages and delegates them to the weaver which then weaves (or reweaves) the appropriate instance adaptation routines which are in turn invoked to return results to the application.

The aspect-oriented approach makes it possible to modify the behaviour of the instance adaptation routines within the aspects instead of modifying them within each class version. If a different instance adaptation approach needs to be employed it can be woven into the schema manager with the ripple effect limited to the aspects encapsulating the instance adaptation routines. This avoids the problem of updating the various class versions. These are automatically updated to use the new strategy (and the new adaptation routines) when the aspects are rewoven. As mentioned earlier the need to reweave is identified by a simple run-time check based on timestamps. [24] demonstrates the use of two different instance adaptation mechanisms: error handlers (which simulate object conversion across class versions) [29] and update/backdate methods (which physically convert objects across class versions) [20] in SADES. [26] demonstrates the effectiveness of the aspect-oriented approach in achieving cost-effective customisation in contrast with existing systems.

Fig. 14 shows that the instance adaptation layer in SADES exploits the functionality exposed by the versioning layer and the layer implementing links, referential integrity and change propagation. The links among class versions and aspects are implemented using relationship objects. The instance adaptation mechanism also makes frequent calls to the class versioning mechanism during simulated or physical conversion across class versions (multiple class versions need to be referred to hence requiring traversal through the version derivation graphs with the aid of the versioning mechanism).

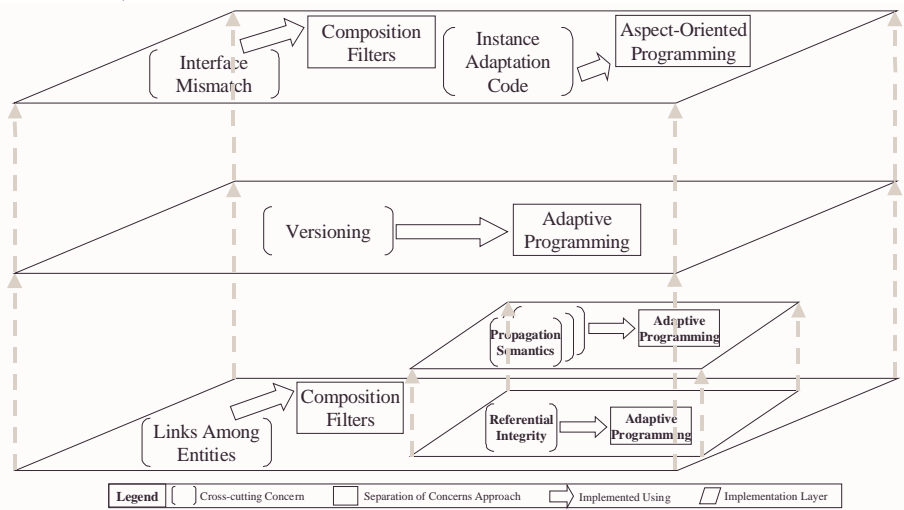


Fig. 14. Implementation of the instance adaptation layer on top of the versioning layer and layer implementing links, change propagation and referential integrity

4. Related Work

The hybrid approach proposed in this paper bears a close relationship with multi-paradigm approaches [5, 6]: using the most suitable mechanism to implement a particular feature or concern. The integration and interaction of features implemented using different separation of concerns techniques poses challenges similar to integration and interaction of features implemented using multiple paradigms. Language cross-binding issues similar to multi-paradigm implementations arise when integrating an aspect language based mechanism with separation of concerns mechanisms realised using an object-oriented language.

The proposed approach can also serve as an implementation mechanism for multi-dimensional separation of concerns [11] as the concerns encountered in SADES are mostly interconnected, overlapping and reside in different system layers. The SADES architecture also bears a close relationship with the meta-object facility for distributed systems described in [7]. However, in [7] the spaces are layered on top of each other and extensions tend to be linear. This is in contrast with both linear and non-linear extensions possible in SADES. Other related work includes meta-space architectures in reflection e.g. [4]. In [4] a space encapsulates a number of closely related but distinct meta-models (e.g. encapsulation, composition, environment, resource, etc.) while in SADES a space encapsulates a particular type of entities (objects, meta-objects, aspects, etc.). The work presented in this paper is also closely related to the author's earlier work on exploring cross-cutting concerns in persistent environments [27] and developing persistence mechanisms for such concerns (which live beyond program execution) [28].

5. Conclusions and Future Work

This paper has proposed a hybrid approach to achieve separation of concerns in a system. The approach is based on the observation that although different techniques can form suitable candidates for implementing a set of cross-cutting concerns, some of them can be more suitable for the purpose as compared to the others. The hybrid approach has been used to achieve separation of concerns in the customisable and extensible object database evolution system, SADES. Three different separation of concerns techniques: composition filters, adaptive programming and aspect-oriented programming have been employed. The discussion has demonstrated that multiple separation of concerns approaches can co-exist, interact and effectively implement concerns most suitably addressed by them. While some of the approaches can use existing programming constructs in a system (e.g. objects, meta-objects, etc.), others might require specific extensions (e.g. aspects). A system, therefore, should have an extensible architecture in order to effectively exploit the hybrid approach.

The experiences with SADES make it possible to abstract general guidelines for employing a hybrid approach for separation of cross-cutting concerns in a system. These guidelines relate to the suitability of the three techniques (employed in SADES) to express different types of concerns and integration of concerns implemented using these different techniques. The guidelines are as follows:

- Adaptive programming is highly suitable for expressing concerns which involve separation of behaviour from object structure. It is very effective in situations where the behaviour may be written against a partial specification of a class diagram or a generic specification of the structure of objects (not relating to information specific to individual instances). Examples of such generic specifications were seen during the implementation of traversal strategies for change propagation and referential integrity. These strategies needed to know the types of relationships or evolution operations but did not need to be aware of the information specific to individual relationship instances.
- Composition filters are very effective in implementing concerns involving message interception and execution of actions before and after executing a method. Their key characteristic is the ability to operate at instance granularity and attachment on a per-instance basis. Examples of these characteristics were observed during implementation of relationships among entities. A dispatch filter for handling a relationship was attached with an instance when the relationship was established. If message interception were not required on a per-instance basis then mechanisms similar to *advices* in aspect languages (e.g. AspectJ [30]), which operate on a per-class basis, would have been more suitable than composition filters.
- Aspect languages and weavers are most suitable in implementing concerns that need to be explicitly represented using linguistic constructs. Expression of such concerns can be simplified if the aspect language is declarative in nature. An example of such a declarative aspect language was seen in the specification of instance adaptation behaviour in SADES.
- The choice of a technique to implement a certain concern may not only be dictated by the chosen design but also by the system requirements. For example, in SADES, the design decision to separate information about links among entities into relationship objects dictated that these relationship objects be attached to their participating entities on a per-instance basis and, hence, led to the choice of composition filters. On the other hand, the system requirement that instance adaptation behaviour be declaratively specified by the maintainer made an aspect language and a weaver the most suitable choice.
- The integration of concerns implemented using different techniques can be simplified by identifying concerns that build upon the functionality offered by other concerns. The latter can then be implemented in lower system layers exposing their functionality to the former which reside in higher system layers.
- One separation of concerns mechanism can be employed to integrate another into the system. For instance, composition filters were employed to integrate the declarative aspect language and its associated weaver into SADES.

The future direction for this work will be the development of tool support for hybrid separation of concerns. Another area of interest is the development of visualisation mechanisms for observing interactions among concerns implemented using different techniques especially those residing in different system layers.

References

- [1] Aksit, M., Tekinerdogan, B., “**Aspect-Oriented Programming using Composition Filters**”, *Proceedings of the AOP Workshop at ECOOP '98, 1998*
- [2] 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*
- [3] Bergmans, L., “**Composing Concurrent Objects – Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs**”, *PhD Thesis, Department of Computer Science, University of Twente, The Netherlands, 1994*
- [4] Blair, G., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran, H., Parlavantzas, N., Saikoski, K., “**A Principled Approach to Supporting Adaptation in Distributed Mobile Environments**”, *Proceedings of the 5th International Symposium on Software Engineering for Parallel and Distributed Systems, IEEE Computer Society Press, 2000, pp. 3-12*
- [5] Budd, T. A., “**Multiparadigm Programming in Leda**”, *Addison-Wesley, 1995*
- [6] Coplien, J. O., “**Multi-Paradigm Design for C++**”, *Addison-Wesley, 1998*
- [7] Crawley, S., Davis, S., Indulska, J., McBride, S., Raymond, K., “**Meta-Meta is Better-Better!**”, *Workshop on Distributed Applications and Interoperable Systems (DAIS) Cottbus, Germany, 1997*
- [8] Czarnecki, K., Eisenecker, U., “**Generative Programming: Methods, Tools and Applications**”, *Addison-Wesley 2000, ISBN 0-201-30977-7*
- [9] Ferrandina, F., Meyer, T., Zicari, R., Ferran, G., “**Schema and Database Evolution in the O2 Object Database System**”, *Proceedings of the 21st Conference on Very Large Databases, Morgan Kaufmann 1995, pp. 170-181*
- [10] 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*
- [11] IBM, USA, “**Multi-dimensional Separation of Concerns using Hyperspaces**”, <http://www.research.ibm.com/hyperspace/>
- [12] “**The Jasmine Documentation**”, Computer Associates International, Inc., Fujitsu Limited, c1996-98
- [13] 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*
- [14] Kiczales, G., et al. “**The Art of the Metaobject Protocol**”, *MIT Press 1991*
- [15] 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*
- [16] Kim, W., Chou, H.-T., “**Versions of Schema for Object-Oriented Databases**”, *Proceedings of 14th International Conference on Very Large Databases, Morgan Kaufmann 1988, pp. 148-159*
- [17] Lieberherr, K. J., “**Demeter**”, <http://www.ccs.neu.edu/research/demeter/index.html>
- [18] Loomis, M. E. S., “**Object Versioning**”, *Journal of Object Oriented Programming, Jan. 1992, pp. 40-43*
- [19] 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*
- [20] Monk, S., Sommerville, I., “**Schema Evolution in OODBs Using Class Versioning**”, *SIGMOD Record, Vol. 22, No. 3, Sept. 1993, pp. 16-22*
- [21] Pulvermueller, E., Speck, A., Rashid, A., “**Implementing Collaboration-based Designs using Aspect-Oriented Programming**”, *Proc. TOOLS USA 2000, IEEE Computer Society Press, pp. 95-104*

- [22] Ra., Y.-G., Rundensteiner, E. A., “**A Transparent Schema-Evolution System Based on Object-Oriented View Technology**”, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 4, July/Aug. 1997, pp. 600-624
- [23] Rashid, A., Sawyer, P., “**Dynamic Relationships in Object Oriented Databases: a Uniform Approach**”, *Proc. of DEXA '99, Springer-Verlag LNCS 1677*, pp. 26-35
- [24] Rashid, A., Sawyer, P., Pulvermueller, E., “**A Flexible Approach for Instance Adaptation during Class Versioning**”, *Proc. of ECOOP 2000 Symposium on Objects and Databases, Springer-Verlag LNCS 1944*, pp. 101-113
- [25] Rashid, A., Sawyer, P., “**Object Database Evolution using Separation of Concerns**”, *ACM SIGMOD Record*, Vol. 29, No. 4, December 2000, pp. 26-33
- [26] Rashid, A., “**A Database Evolution Approach for Object-Oriented Databases**”, *PhD Thesis, Computing Department, Lancaster University, UK, 2000*
- [27] Rashid, A., Pulvermueller, E., “**From Object-Oriented to Aspect-Oriented Databases**”, *Proceedings of the 11th International Conference on Database and Expert Systems Applications DEXA 2000, Lecture Notes in Computer Science 1873*, pp. 125-134
- [28] Rashid, A., “**On to Aspect Persistence**”, *Proceedings of 2nd International Symposium on Generative and Component-based Software Engineering (GCSE 2000 part of proceedings of NetObjectDays 2000)*, pp. 453-463
- [29] 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
- [30] Xerox PARC, USA, “**AspectJ Home Page**”, <http://aspectj.org/>
- [31] Zhang, N., Haerder, T., Thomas, J., “**Enriching Object-Relational Databases with Relationship Semantics**”, *Proc. of the 3rd Int. Workshop on Next Generation Information Technologies and Systems (NGITS), Israel, 1997*