Object Database Evolution using Separation of Concerns

Awais Rashid, Peter Sawyer
{marash | sawyer}@comp.lancs.ac.uk
Computing Department, Lancaster University, Lancaster LA1 4YR, UK

Abstract

This paper proposes an object database evolution approach based on *separation of concerns*. The lack of customisability and extensibility in existing evolution frameworks is a consequence of using attributes at the meta-object level to implement links among meta-objects and the injection of instance adaptation code directly into the class versions. The proposed approach uses dynamic relationships to separate the connection code from meta-objects and *aspects* - abstractions used by Aspect-Oriented Programming to localise cross-cutting concerns - to separate the instance adaptation code from class versions. The result is a customisable and extensible evolution framework with low maintenance overhead.

Introduction

The conceptual structure of an object-oriented database may be subject to frequent changes due to the need to correct mistakes in the database design, to add new features during incremental design or to reflect changes in the structure of the real world artefacts modelled in the database [18]. A number of approaches have been proposed for evolution in object-oriented databases [1, 3, 9, 10, 19]. One of the shortcomings of these approaches is the use of attributes at the meta-object level to implement links among meta-objects. This results in additional evolution complexity at the meta-object level. The extensibility of the evolution framework is compromised as the code handling the connections among the meta-objects is spread across the meta-object space making changes expensive. Another shortfall of existing evolution approaches is the introduction of instance adaptation code into class versions and commitment to a particular instance adaptation strategy. Although it is possible to make changes to the instance adaptation strategy or adopt an entirely different strategy (due to scenario specific evolution requirements or the availability of a more efficient strategy), such an attempt would be very costly. All the versions of existing classes might need to be modified to reflect the change. In other words, the evolution problem will appear at a different level. Therefore, existing systems are, to a great extent, bound to the particular instance adaptation strategy being used.

This paper proposes an object database evolution approach based on *separation of concerns*. The *separation of concerns* principle makes it possible to provide a customisable and extensible evolution approach with low maintenance overhead. Dynamic relationships are employed to connect the various meta-objects in the database. This provides an implicit separation of concerns as the code handling the connections is removed from the meta-objects and encapsulated in the relationships. The connections can be modified by manipulating the relationships independently of the meta-objects linked by them. Propagation patterns and semantics of relationships can also be modified in an independent fashion. Explicit separation of concerns has been employed for instance adaptation. This is achieved by encapsulating the instance adaptation code in *aspects*: entities used in *Aspect-Oriented Programming* (AOP) [7] to localise cross-cutting concerns. This makes it possible to make cost-effective changes to the instance adaptation strategy.

The following section demonstrates that the code handling links among meta-objects and instance adaptation is by nature cross-cutting. We, then, present our proposed approach (implemented as part of the SADES evolution system [11, 12, 14, 17]) based on separation of concerns. This is followed by an overview of the implementation and evaluation of the proposed concepts.

Shortcomings of Existing Approaches

Links through Meta-object Attributes

ORION [1], ENCORE [19], TSE [10] and other existing OODBMSs use attributes at the meta-object level to implement relationships among meta-objects. Examples of these relationships are inheritance relationships among *class* meta-objects and aggregation relationships between a *class* meta-object and *attribute* and *method* meta-objects. The implementation of inheritance relationships in existing systems is shown in fig. 1(a). Each *class* meta-object maintains collections of references to *class* meta-objects forming its superclasses and subclasses. Such a structure introduces the evolution problem at the meta-object level. Referential integrity has to be managed by the evolution framework. When any inheritance relationships are modified corresponding attributes in the affected meta-objects need to be updated to reflect the change. Modifying the structure of meta-classes and introduction/removal of existing classes is very expensive. This reduces the extensibility and maintainability of the evolution framework.

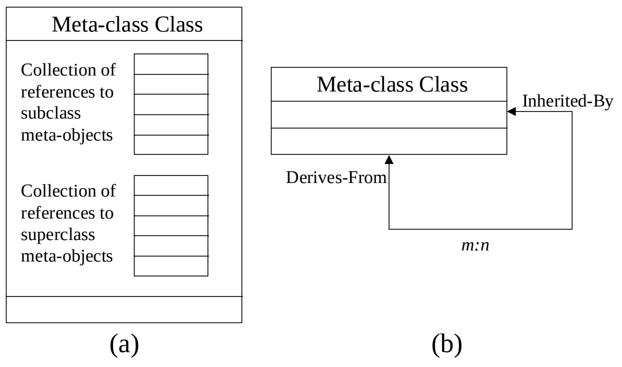
Meta-class Class
Collection of references to subclass meta-objects
Collection of references to superclass meta-objects
Meta-class Class
Inherited-By
Derives-From
m:n
(a)
(b)

Fig. 1: Meta-class structure in **(a)** existing systems **(b)** SADES

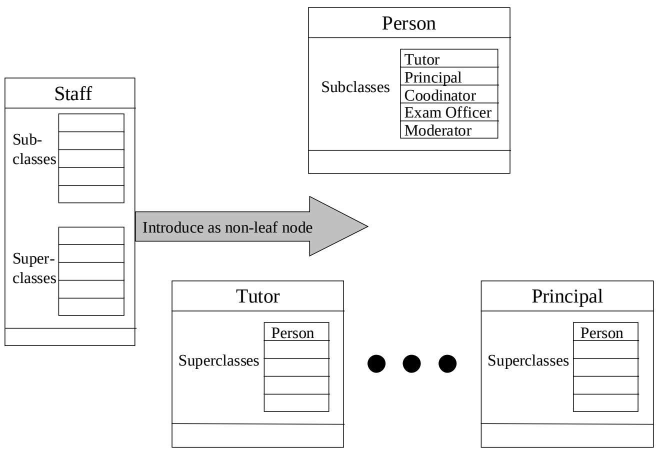

Fig. 2: Introduction of a non-leaf class meta-object in existing systems

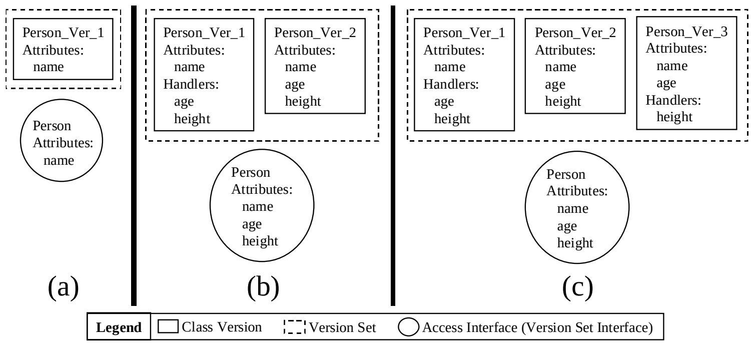

Fig. 3: Instance Adaptation in ENCORE

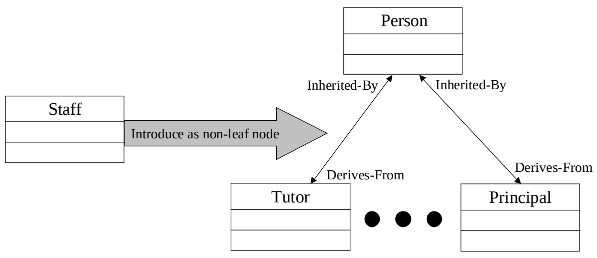

Fig. 4: Introduction of a non-leaf class meta-object in a system using dynamic relationships

Fig. 2 shows the meta-objects in such a system prior to the introduction of a non-leaf class. The scenario is based on a case study carried out at the Open College of the North West, Lancaster, UK. The classes *Person*, *Tutor* and *Principal* already exist in the system while the class *Staff* is the non-leaf class being introduced into the system. Some existing sub-classes of *Person*: *Coordinator*, *Exam Officer* and *Moderator* have been omitted from the fig. for simplification. In this scenario 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 be added to the superclasses collection and references to older subclasses of *Person* will be added to the subclasses collection in the *Staff* meta-object. A reference to the *Staff* meta-object will be added to the subclasses collection (not shown in fig. 2) in *Person* and to the superclasses collection in each of its subclasses. The evolution framework has to maintain referential integrity and address the evolution problems at the meta-object level.

Fixed Instance Adaptation Strategy
Existing systems introduce the adaptation code 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. There is a high probability that a number of adaptation routines in a class version will never be invoked as only newer applications will attempt to access properties and methods unavailable for objects associated with the particular class version. The adaptation strategy is fixed and adoption of a new strategy might trigger the need for changes to all or a large number of versions of existing classes.
To demonstrate the above shortcomings we consider instance adaptation in ENCORE [19]. As shown in figure 3, 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. 3(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. 3(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 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. The instance adaptation strategy is closely integrated with the evolution framework. Changes to the instance adaptation strategy or moving to a more efficient strategy will be very expensive.

Proposed Approach

Dynamic Relationships

[4] characterises relationships among classes as static since these are fixed at compile-time. Relationships among instances are comparatively dynamic in nature and can be changed at run-time. Our approach differs from the viewpoint presented by [4]. Relationships among classes do not need to be fixed at compile-time. These can be dynamic in nature and can be changed at run-time. Therefore, the schema of an object-oriented database can be dynamically modified if the various meta-objects (classes, etc.) that form the schema are interconnected through dynamic relationships. The use of dynamic relationships to achieve dynamic evolution serves a two-fold purpose. First, a coherent view of the conceptual structure of the database is provided making maintenance easier. Second, the information about the connections among the various meta-objects is separated and encapsulated in the relationship constructs.

As shown in fig. 1(b) the links among meta-objects are realised using bi-directional semantic relationships which are first-class objects. As a result modification of these links is natural. Change propagation and referential integrity is provided by the semantics of the underlying dynamic relationships architecture [13]. As a result the evolution framework is relieved of this responsibility. Dynamic schema changes can be made by dynamically modifying the various relationships in which the meta-objects participate. These can be the *derives-from*/*inherited-by* relationships between classes or the *defines/defined-in* relationships between classes and their members. Relationships among meta-objects and objects can be used to propagate schema changes to the affected objects. Since relationships can be dynamically introduced, removed or modified the evolution framework is more extensible and maintainable. The introduction, removal or modification of meta-classes only requires introduction, removal or modification of the various semantic relationships with referential integrity managed by the system. Propagation patterns and semantics of relationships can also be modified in an independent fashion without a significant impact on the evolution framework.

Fig. 4 shows the meta-objects connected through dynamic relationships prior to the introduction of the non-leaf class *Staff* in the evolution scenario from fig. 2. When the meta-object *Person* is removed from the participants list of the *Derives-From/Inherited-By* relationship with its subclass meta-objects the referential integrity mechanism automatically propagates the change to the subclass meta-objects. A similar argument applies to the introduction of *Staff* as a participant in these relationships. It should also be noted that in contrast with existing systems the class hierarchy is *truly connected*. Maintenance and extensions to the evolution framework are less expensive as the maintainer manipulates semantic relationships which are first class objects; these concepts are natural to maintainers and developers of object-oriented systems.

Our case study at the Open College of the North West, UK brought to front the need for a *move* primitive in order to move attributes across classes without loss of information i.e. moving attributes common to *Staff* objects from *Person* to a new class *Staff*. Existing systems will need to update the collections containing references to attribute meta-objects in order to implement the move primitive. Using dynamic relationships only the *class* meta-object to which the attributes are being moved will need to be added as a participant in the *Defines/Defined-in* relationship of these attributes. Since it is a one-to-many relationship the existing *class* meta-object defining the attributes will automatically be removed and the referential integrity mechanism will propagate the changes to the attributes and their new defining scope (the class).

Instance Adaptation using Aspects

Aspects are abstractions introduced by Aspect-Oriented Programming (AOP) [7] in order 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 [5, 7].

Since instance adaptation is a cross-cutting concern we propose separating it from the class versions using aspects (cf. fig. 5(a)). It should be noted that although fig. 5(a) 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. 5(b) 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 code is dynamically woven into the particular class version (version 1 in this case). This code is then invoked to return the results to the application.

It should be noted that the instance adaptation code in fig. 5 has two parts:

- Adaptation routines
- Instance adaptation strategy

An 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. The instance adaptation strategy is the code which detects the interface mismatch and invokes the appropriate adaptation routine e.g. an error handler [19], an update method [9] or a transformation function [3].

Based on the above observation two possible aspect structures are shown in fig. 6. The structure in fig. 6(a) encapsulates the instance adaptation strategy and the adaptation routines for a class version (or a set of class versions) in one aspect. This has the advantage of having the instance adaptation code for a class version (or a set of class versions) in one place but unnecessary weaving of the instance adaptation strategy (which could previously be woven) needs to be carried out. This can be taken care of by *selective weaving* i.e. only weaving the modified parts of an aspect if it has previously been woven into the particular class version. Such a structure also has the advantage of allowing multiple instance adaptation strategies to coexist. One set of class versions could use one strategy while another could use a different one. This choice can also be application dependent. A disadvantage of this structure is the need to modify a large number of aspects if the instance adaptation strategy is changed. This is addressed by the structure in fig. 6(b) which encapsulates the instance adaptation strategy in an aspect separate from the ones containing the specific adaptation routines. In this case only one instance adaptation aspect exists in the system providing better localisation of changes as compared to the structure in fig. 6(a). Any such change will require aspects containing the adaptation routines to be rewoven. If more than one instance adaptation aspects are allowed to exist multiple instance adaptation strategies can coexist (similar to the structure in fig. 6(a)). If only one instance adaptation strategy is being used and the system is single-rooted (this implies that the system has a root class which has only one class version) the instance adaptation aspect can be woven into the root class version and rewoven only if the instance adaptation strategy is modified. Versions of subclasses will need a small amount of code to be woven into them in order to access the functionality woven in the root class. This code can be separated in an aspect if a generic calling mechanism is being used. Otherwise, it can reside in aspects containing the adaptation

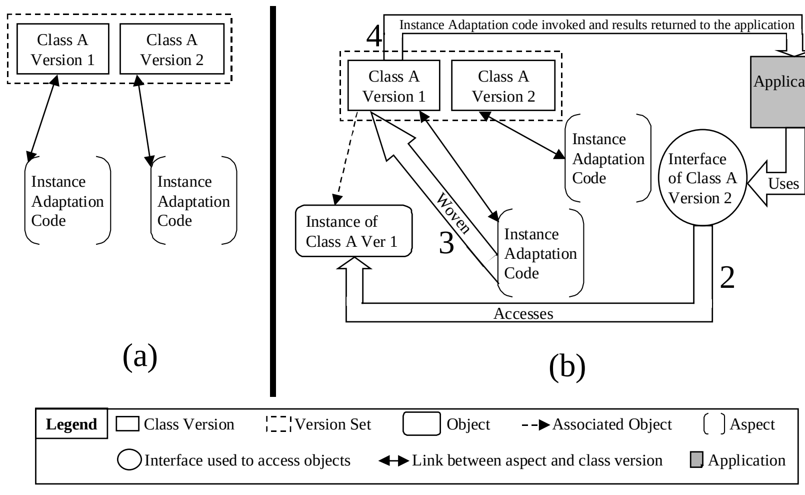

Fig. 5: Instance Adaptation using Aspects

routines.

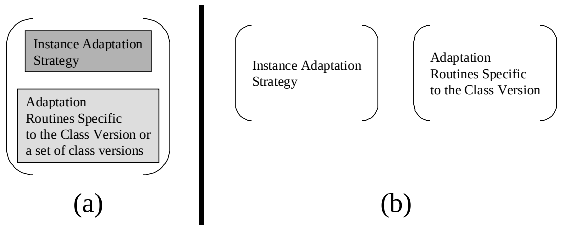
Instance Adaptation Strategy
Adaptation Routines Specific to the Class Version or a set of class versions
(a)
Instance Adaptation Strategy
Adaptation Routines Specific to the Class Version
(b)

Fig. 6: Two Possible Aspect Structures

The above discussion shows that aspects help in separating the instance adaptation strategy from the class versions. Behaviour of the adaptation routines can be modified within the aspects instead of modifying them within each class version. If a different instance adaptation strategy needs to be employed 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 (and the new adaptation routines) when the aspects are rewoven. The need to reweave is easily identified by a simple run-time check based on timestamps.

Referring to the need for a *move* primitive identified in our case study it should be noted that the aspect-oriented instance adaptation approach allows customisation of the instance adaptation strategy in order to simulate the primitive. There is no maintenance overhead for existing class versions as changes are local to the instance adaptation aspects and are propagated to the class versions through dynamic weaving. Attempting to simulate a new evolution primitive using instance adaptation strategies in existing systems (e.g. error handlers [19] or update/backdate methods [9]) will be very expensive as extensive changes to the instance adaptation strategy and adaptation routines will be required with a *spill-over* effect on the rest of the system [15].

Implementation and Evaluation

The approach has been implemented as part of the SADES evolution system [11, 12, 13, 14, 17] which has been built as a layer on top of the commercially available Jasmine[1] object database management system. Applications can be bound to particular class versions or a common interface for the class. The implementation of the SADES conceptual schema using dynamic relationships and flexibility of switching between two different instance adaptation strategies using the aspect-oriented approach in SADES are discussed in this section.

SADES Conceptual Schema

The SADES conceptual schema [11] is a fully connected directed acyclic graph (DAG) depicting the class hierarchy in the system. SADES schema DAG uses *Version derivation graphs* [8]. Each node in the DAG is a *class version derivation graph*. Each node in the class version derivation graph (CVDG) keeps: *reference(s) to predecessor(s)*, *reference(s) to successor(s)*, *reference to the versioned class object*, *descriptive information about the class version such as creation time, creator's identification, etc.*, *reference(s) to super-class version(s)*, *reference(s) to sub-class version(s)*, and *a set of reference(s) to object version derivation graph(s).*

Each node of a CVDG keeps a set of reference(s) to some object version derivation graph(s) (OVDG). An OVDG is similar to a CVDG and keeps information about various versions of an instance rather than a class. Each OVDG node [8] keeps: *reference(s) to predecessor(s)*, *reference(s) to successor(s)*, *reference to the versioned instance*, and *descriptive information about the instance version.*

Since an OVDG is generated for each instance associated with a class version, a set of OVDGs results when a class version has more than one instance associated with it. As a result a CVDG node keeps a set of references to all these OVDGs.

Fig. 7 identifies the various meta-classes in the system and the relationships in which their instances, the meta-objects, participate. The dashed block arrows indicate that all meta-classes inherit from the class *Root*. The line arrows, both solid and dashed, represent the various relationships meta-objects participate in. A solid arrow pointing from a meta-class back to itself indicates a recursive relationship. The dashed line arrows representing relationships among certain meta-classes and the class *Root* indicate that the relationship exists among instance(s) of the particular meta-class and instance(s) of a sub-class of the class *Root*. An instance of the class *OVDG Node*, for example, manages an object which is an instance of a sub-class of the class *Root* (since all classes inherit from the class *Root*). Similarly, a class *Property* or a method *Parameter* will normally have a sub-class of the class *Root* as its type.

The various relationships among instances of meta-classes shown in fig. 7 can be dynamically modified to achieve dynamic schema modifications. Schema changes are propagated to the instances using the *has-OVDG*/*is-head-of-OVDG-for* relationship that exists between a class version and its associated OVDGs.

The use of dynamic relationships to implement the SADES conceptual schema has greatly simplified the implementation of the evolution framework and improved its maintainability and extensibility. It is possible to introduce new relationships and even new meta-classes if desirable. Change propagation and referential integrity are managed by the dynamic

[1] http://www.cai.com/

relationships architecture. Let us consider an extensibility example. It might be desirable for an experienced application developer trying to extend the database model with active features [2] to introduce a meta-class *Rule* which can then be used to define classes (meta-objects) of rules which in turn will be used to create the various rule objects. This can be easily achieved by introducing a new meta-class and defining the relationships its instances will participate in. The dynamic relationships architecture will propagate changes to instances of existing meta-classes and manage referential integrity. The feature has been exploited to implement a meta-class *Aspect* for the aspect-oriented extension to SADES [16] used to implement the instance adaptation approach.

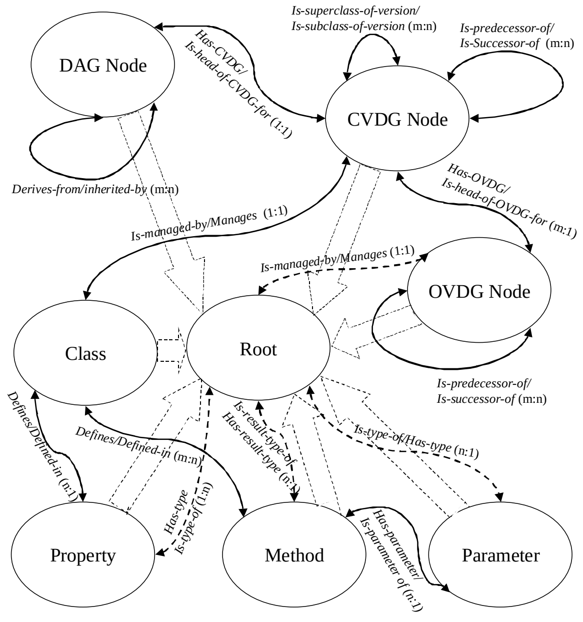

Fig. 7: Meta-classes in SADES and the relationships their instances participate in

Instance Adaptation in SADES

The instance adaptation approach has been implemented through an aspect-oriented extension to SADES [16]. We now discuss the use of two different instance adaptation strategies in SADES: error handlers from ENCORE [19] and update/backdate methods from CLOSQL [9]. The example aims to demonstrate the flexibility of the aspect-oriented approach. We first discuss how to implement the error handlers strategy using our approach. We then present the implementation of the update/backdate methods strategy. This is followed by a description of seamless transformation from one instance adaptation strategy to another.

We have employed the aspect structure in fig. 6(b) in SADES as the class hierarchy is single-rooted with strictly one version for the root class. Versions of all classes directly or indirectly inherit from this root class version. Although not shown in the following example, it should be assumed that appropriate code has been woven into the class versions to access the adaptation strategy woven into the root class version.

In order to demonstrate the implementation of error handlers using the aspect-oriented instance adaptation approach we have chosen the scenario in fig. 3(b). Similar solutions can be employed for other cases. As shown in fig. 8(a), instead of introducing the handlers into the former class versions they are encapsulated in an aspect. In this case one aspect serves all the class versions existing prior to the creation of the new one. Links between aspects and class versions are *open* [6] 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. 8(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.

We now discuss implementation of the instance adaptation strategy of CLOSQL [9] using our approach. In CLOSQL, unlike ENCORE, instances are converted between class versions instead of emulating the conversion. The conversion is reversible, hence allowing instances to be freely converted between various versions of their particular class. When a new class version is created the maintainer specifies update functions for all the attributes in the class version. An update function specifies what is to happen to the particular attribute when it is converted to a newer class version. Backdate functions provide similar functionality for the conversion to older class versions. All the update and backdate functions for the class version are grouped into an *update method* and a *backdate method* respectively. Applications are bound to the class versions. Therefore, the access interface is that of the particular class version being used to access an object. When an instance is converted some of the attribute values can be lost (if the class version used for conversion does not define some of the attributes whose values exist in the instance prior to conversion). This is addressed by storing removed attribute values separately.

Figure 9 shows the implementation of the update/backdate methods strategy using our approach. As shown in fig. 9(a) an aspect containing an update method is associated with version 1 of *class Person* in order to convert instances associated with version 1 to version 2. An aspect containing a backdate method to convert instances associated with version 2 to version 1 is also introduced. Although not shown in fig. 9(a) when an instance is converted from *class Person* version 2 to version 1 a storage aspect will be associated with the converted instance in order to store removed information. Fig. 9(b) depicts the case when an application attempts to access an object associated with version 1 of *class Person* using the class definition in version 2. The aspect containing the update method is woven into version 1. The method then converts the accessed object to the definition in version 2. The converted object is now associated with the new class version definition (i.e. version 2) and returns information to the application.

We now discuss how the aspect-oriented approach allows seamless transformation from one instance adaptation strategy to another. As discussed earlier such a need might arise due to application/scenario specific adaptation requirements or the availability of a more efficient strategy. We assume that the system initially employs the error handlers strategy (as shown in fig. 8) and discuss how this strategy can be replaced by the update/backdate methods strategy (as shown in fig. 9). In order to adopt this new strategy in SADES the aspect containing the instance adaptation strategy in fig. 6(b) (in this case error handlers) is replaced by an aspect encapsulating the new strategy (in this case update/backdate methods). The instance adaptation strategy aspect is rewoven into the root class version. There is no need to reweave access from subclass versions as the code for this is independent of the instance adaptation strategy in SADES. The aspects containing the handlers are replaced by those containing update/backdate methods. Let us assume that the scenario in fig. 9 corresponds to the one in fig. 8 after the instance adaptation strategy has been changed. As shown in fig. 9(a) the aspect containing error handlers for version 1 of *class Person* will be replaced with an aspect containing an update method to convert instances associated with version 1 to version 2. An aspect containing a backdate method to convert instances associated with version 2 to version 1 will also be introduced. Since the approach is based on dynamic weaving the new aspects will be woven into the particular class version when required. It should be noted that it is also possible to automatically generate the aspects encapsulating update/backdate methods from those containing error handlers and vice versa. This eases the programmer's task who can edit the generated aspects if needed.

Conclusions and Future Work

This paper has proposed an object database evolution approach based on separation of concerns. Dynamic relationships have been employed to separate the code handling links among meta-objects. This has provided improved customisability and extensibility demonstrated by the development of an aspect-oriented extension to the system. The aspect-oriented extension has been used to implement a flexible instance adaptation approach. The flexibility of the approach has been demonstrated through implementation of two different instance adaptation strategies and seamless transformation between them. Our future work will explore the applicability of the aspects to separate persistence related code from applications. This will localise the impact of schema changes to the aspects encapsulating the persistence related code providing a more efficient approach for maintaining behavioural consistency during evolution.

Acknowledgement: The authors would like to acknowledge the support of Open College of the North West, Lancaster, UK for providing opportunities for database evolution case studies.

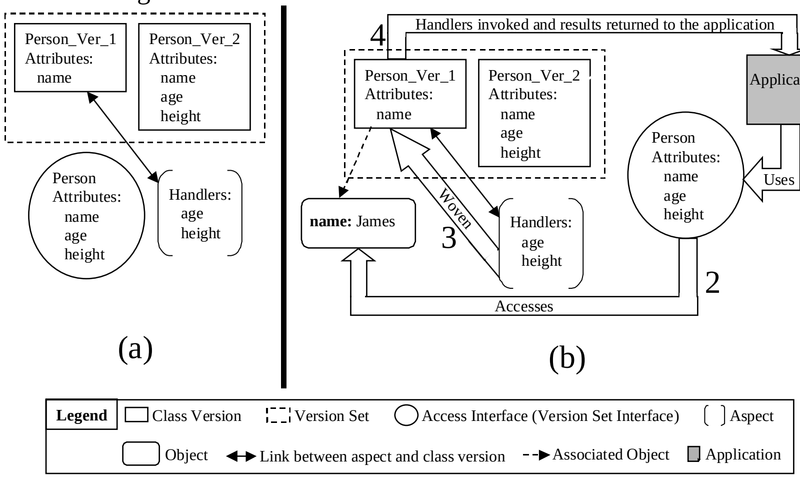

Fig. 8: Error Handlers in SADES using the Aspect-Oriented Instance Adaptation Approach

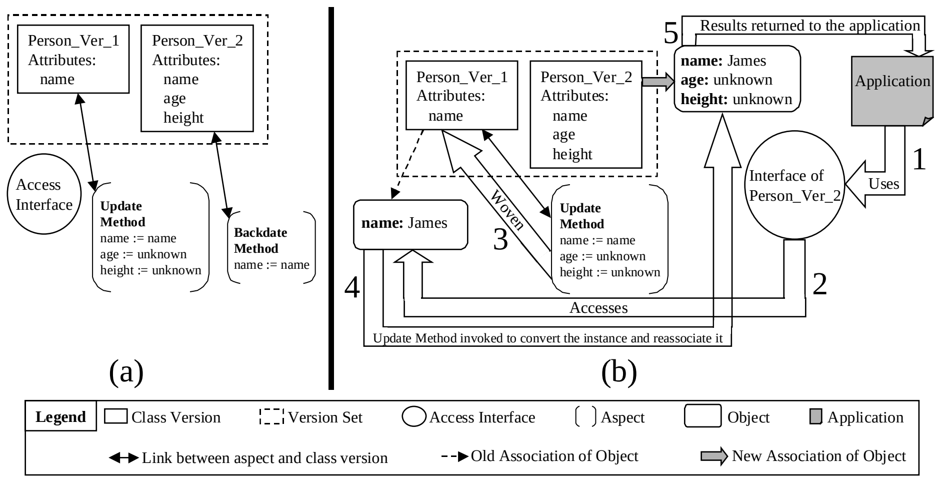

Fig. 9: Update/backdate Methods in SADES using the Aspect-Oriented Instance Adaptation Approach

References

[1] Banerjee, J. *et al.*, "Data Model Issues for Object-Oriented Applications", *ACM Trans. Office Information Systems, 5(1), Jan. 1987, pp. 3-26*

[2] Dittrich, K. R. *et al.*, "The Active Database Management System Manifesto: A Rulebase of ADBMS Features", *Proc. 2nd Workshop on Rules in Databases, Sept. 1995, LNCS. 985, pp. 3-20*

[3] Ferrandina, F., Meyer, T., Zicari, R., Ferran, G., "Schema and Database Evolution in the O2 Object Database System", *Proc. 21st VLDB Conference, Morgan Kaufmann 1995, pp. 170-181*

[4] Gamma, E. *et al.*, "Design Patterns - Elements of Reusable Object-Oriented Software", *Addison Wesley, c1995*

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

[6] Kersten, M. A., Murphy, G. C., "Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming", *Proc. OOPSLA 1999, ACM SIGPLAN Notices, 34(10), Oct. 1999, pp. 340-352*

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

[8] Loomis, M. E. S., "Object Versioning", *JOOP, Jan. 1992, pp. 40-43*

[9] Monk, S. & Sommerville, I., "Schema Evolution in OODBs Using Class Versioning", *SIGMOD Record, 22(3), Sept. 1993, pp. 16-22*

[10] Ra., Y.-G., Rundensteiner, E. A., "A Transparent Schema-Evolution System Based on Object-Oriented View Technology", *IEEE Trans. Knowledge & Data Engg., 9(4), 1997, pp. 600-624*

[11] Rashid, A., Sawyer, P., "Facilitating Virtual Representation of CAD Data through a Learning Based Approach to Conceptual Database Evolution Employing Direct Instance Sharing", *Proc. DEXA '98, LNCS 1460, pp. 384-393*

[12] Rashid, A., "SADES - a Semi-Autonomous Database Evolution System", *Proc. ECOOP'98 PhDOOS Workshop, LNCS 1543*

[13] Rashid, A., Sawyer, P., "Dynamic Relationships in Object Oriented Databases: A Uniform Approach", *Proc. DEXA '99, LNCS 1677, pp. 26-35*

[14] Rashid, A., Sawyer, P., "Transparent Dynamic Database Evolution from Java", *OOPSLA '99 WS on Java and Databases: Persistence Options*

[15] Rashid, A., Sawyer, P., Pulvermueller, E., "A Flexible Approach for Instance Adaptation during Class Versioning", *Proc. ECOOP 2000 OODB Symposium (Springer-Verlag LNCS)*

[16] Rashid, A., Pulvermueller, E., "From Object-Oriented to Aspect-Oriented Databases", *To Appear in Proc. DEXA 2000 (Springer-Verlag LNCS)*

[17] "SADES Java API Documentation", *http://www.comp.lancs.ac.uk/computing/users/marash/research/sades/index.html*

[18] Sjoberg, D., "Quantifying Schema Evolution", *Information and Software Technology, 35(1), pp. 35-44, Jan. 1993*

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