A Flexible Approach for Instance Adaptation during Class Versioning

Awais Rashid[1], Peter Sawyer[1], Elke Pulvermueller[2]
[1]Computing Department, Lancaster University, Lancaster LA1 4YR, UK
{marash | sawyer} @comp.lancs.ac.uk
[2]Wilhelm Schickard Institute for Computer Science, University of Tuebingen, 72076 Tuebingen, Germany
pulvermueller@acm.org

Abstract: One of the consequences of evolution can be the inability to access objects created using the older schema definition under the new definition and vice versa. Instance adaptation is the conversion of objects to a compatible definition or making objects exhibit a compatible interface. Existing evolution approaches are committed to a particular instance adaptation strategy. This is because changes to the instance adaptation strategy or an attempt to adopt an entirely different strategy would be very costly. This paper proposes a flexible instance adaptation approach for systems employing class versioning to manage evolution. Flexibility is achieved by encapsulating the instance adaptation code in *aspects* - abstractions introduced by aspect-oriented programming to localise cross-cutting concerns. This makes it possible to make cost-effective changes to the instance adaptation strategy. The flexibility of the approach is demonstrated by using two instance adaptation strategies: error handlers and update/backdate methods.

1 Introduction

The conceptual structure of an object-oriented database may not remain constant and may vary to a large extent [46]. The need for these variations (evolution) arises due to a variety of reasons e.g. 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. Two types of anomalous behaviour can arise as a consequence of evolution:

- Objects created under the older schema definition might not be accessible under the new definition and vice versa.
- Application programs accessing the database prior to evolution might contain invalid references and method calls.

The former has been termed a *structural consistency* issue and the latter a *behavioural consistency* issue [49].

One of the various evolution strategies[1] employed to address the above issues is class versioning [11, 12, 33, 47, 50]. In this approach a new version of a class is created upon modification. Applications are bound to specific versions of classes or a common interface while objects are bound to the class version used to instantiate them. When an object is accessed using another type version (or a common type interface) it is either converted or made to exhibit a compatible interface. This is termed instance adaptation. This paper proposes a flexible instance adaptation approach for class versioning systems. From this point onwards instance adaptation (and hence structural consistency) will be the focus of the discussion. Behavioural consistency problems will not be discussed. An analysis of such problems in the context of both untyped and strongly typed programming languages can be found in [8].

Existing class versioning approaches are committed to a particular instance adaptation strategy[2]. For example, ENCORE [47] and AVANCE [11] employ error handlers to simulate compatible interfaces while CLOSQL [33] uses update/backdate methods to dynamically convert instances between class versions. Although it is possible to make changes to the instance adaptation strategy or adopt an entirely different 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.

[1] Other evolution strategies include *schema evolution* [5, 14, 15, 26, 36, 48], where the database has one logical schema to which class definition and class hierarchy modifications are applied and *schema versioning* [3, 6, 23, 25, 34, 35, 38], which allows several versions of one logical schema to be created and manipulated independently of each other.

[2] This is also true of schema evolution and schema versioning approaches.

Therefore, existing systems are, to a great extent, bound to the particular instance adaptation strategy being used. Our case studies at an organisation where day-to-day activities revolve around the database have brought to front the need for application or scenario specific instance adaptation. One such case involved the need to simulate a *move attribute* evolution primitive. This could either be achieved by adding an additional attribute to the evolution taxonomy or customising the instance adaptation approach to simulate the primitive. Over the lifetime of the database there might also be a need to move to a better, more efficient instance adaptation strategy. Due to the lack of flexibility in existing instance adaptation approaches such customisation or exchange will be very expensive.

In this paper we propose a flexible approach which localises the effect of any changes to the instance adaptation strategy, hence making such changes possible and cost-effective. This is achieved by encapsulating the instance adaptation code into *aspects*. Aspects are abstractions introduced by aspect-oriented programming (AOP) [21, 22] to localise cross-cutting concerns. A detailed description of the applicability of aspects in OODBs has been provided in [44]. From the above discussion it is clear that the instance adaptation strategy cross-cuts class versions created over the lifetime of the database and, hence, can be separated using aspects. It can be automatically woven into the class versions when required and can also be automatically rewoven if the aspects are modified.

The next section provides an overview of aspect-oriented programming. This is followed by a description of the flexible instance adaptation approach. Section 4 uses error handlers and update/backdate methods as examples to demonstrate the flexibility of the approach. Section 5 discusses related work while section 6 concludes the discussion and identifies directions for future work.

2 Aspect-Oriented Programming

Aspect-oriented programming [21, 22] 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. Although patterns [17] 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 (in this case debugging and synchronisation code). The links between classes and aspects are expressed by explicit or implicit *join points*. These links can be categorised[3] as [20]:

- *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 [19, 22].

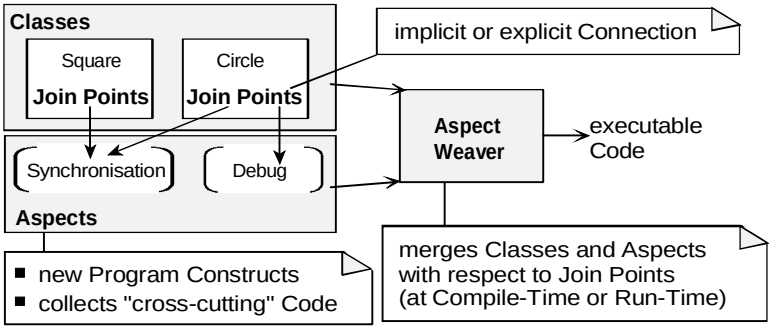

Fig. 1: Aspect-Oriented Programming

[3] The categorisation determines the reusability of classes and aspects.

Different AOP techniques and research directions can be identified. They all share the common goal of providing an improved separation of concerns. AspectJ [4] 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. [10] describes an aspect language and a weaver for Smalltalk. An aspect language for the domain of robust programming can be found in [16].
Other AOP approaches aiming at achieving a similar separation of concerns include subject-oriented programming [18], composition filters [1] and adaptive programming [29, 32]. 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 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.
Experience reports and assessment of AOP can be found in [20, 37].

3 The Flexible Instance Adaptation Approach

Our approach is based on the observation that the instance adaptation code cross-cuts the class version definitions. This is because 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.
Since instance adaptation is a cross-cutting concern we propose separating it from the class versions using aspects (cf. fig. 2(a)). It should be noted that although fig. 2(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. 2(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. 2 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 [47], an update method [33] or a transformation function [14].
Based on the above observation two possible aspect structures are shown in fig. 3. The structure in fig. 3(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. 3(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. 3(a). Any such change will require aspects

containing the adaptation routines to be rewoven. Issues relating to this and a solution based on assertions have been discussed in [24]. If more than one instance adaptation aspects are allowed to exist multiple instance adaptation strategies can coexist (similar to the structure in fig. 3(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 routines.

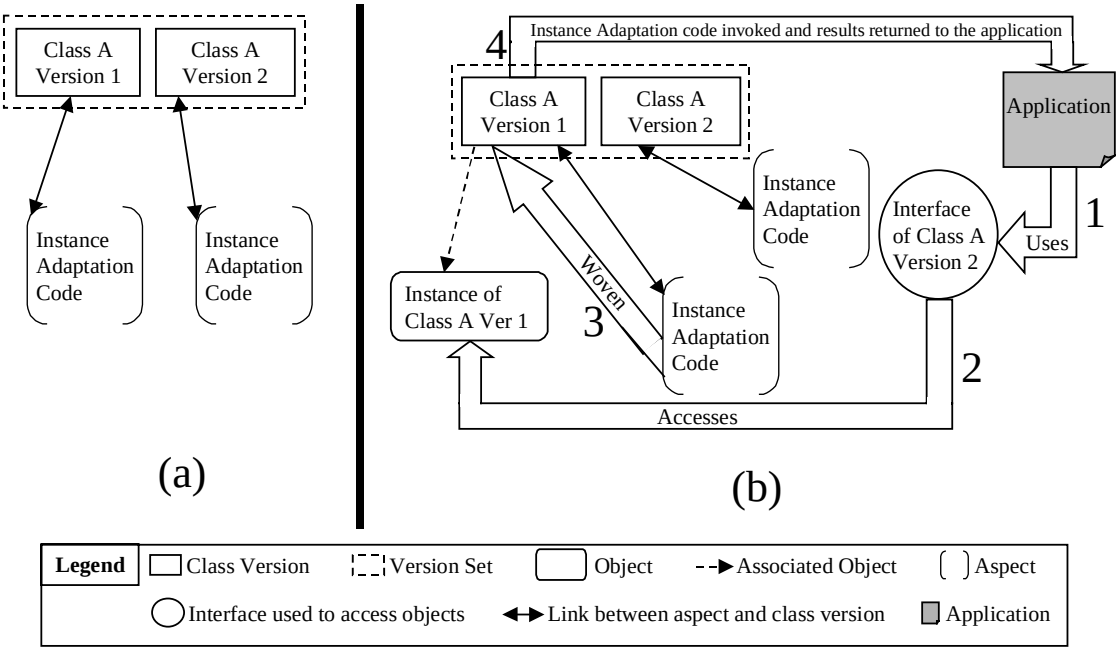

Fig. 2: Instance Adaptation using Aspects

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 can easily be identified by a simple run-time check based on timestamps.

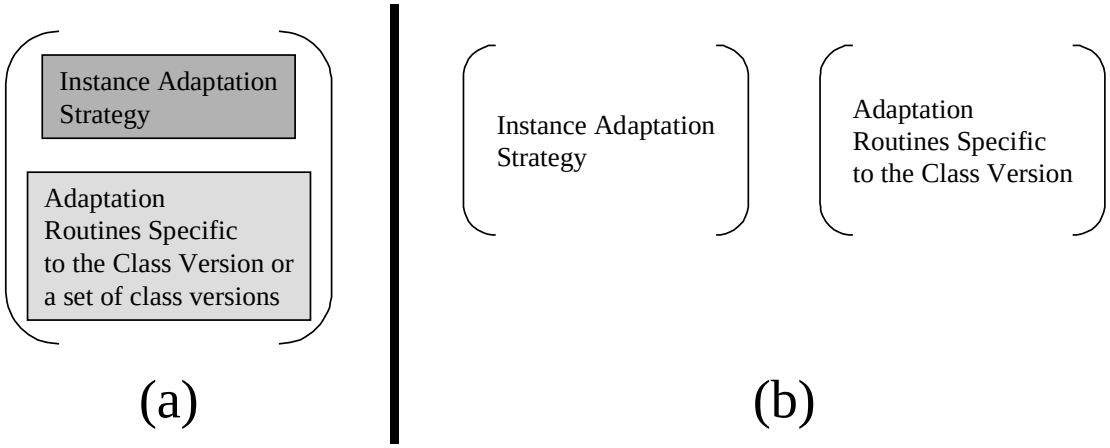

Fig. 3: Two Possible Aspect Structures

The approach has been implemented as part of the SADES evolution system [39, 40, 41, 43, 45] which has been built as a layer on top of the commercially available Jasmine[4] object database management system. Applications can be bound to particular class versions or a common interface for the class. An example of switching between two different instance adaptation strategies in the SADES system is discussed in the following section.

4 Example Instance Adaptation Strategies

In this section we discuss the use of two different instance adaptation strategies in SADES: error handlers from ENCORE [47] and update/backdate methods from CLOSQL [33]. The example aims to demonstrate the flexibility of the 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. 3(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.

4.1 Error Handlers

We first consider the instance adaptation strategy of ENCORE [47]. As shown in figure 4, 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. 4(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. 4(c)).

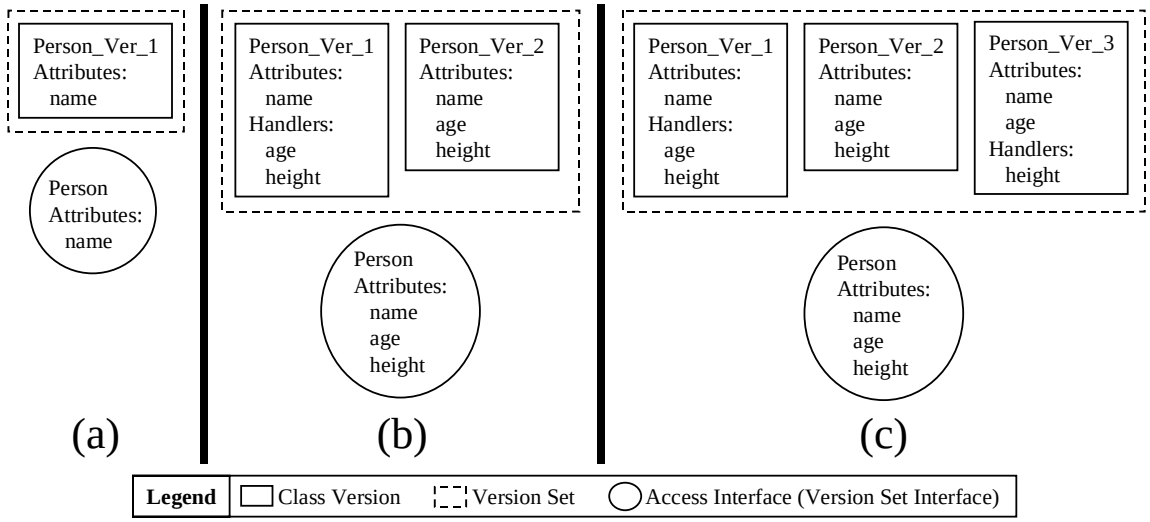

Fig. 4: Error Handlers in ENCORE

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 exist prior to the creation of a new one. If the behaviour of some handlers needs to be changed maintenance has to be performed on all the class versions in which the handlers were introduced. To demonstrate our approach we have chosen the scenario

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

in fig. 4(b). Similar solutions can be employed for other cases. As shown in fig. 5(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* [20] 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. 5(b) depicts the case when an application attempts to access the *age* and *height* attributes in an object associated with version 1 of *class Person*. The aspect containing the handlers is woven into the particular class version. The handlers then simulate (to the application) the presence of the missing attributes in the associated object.

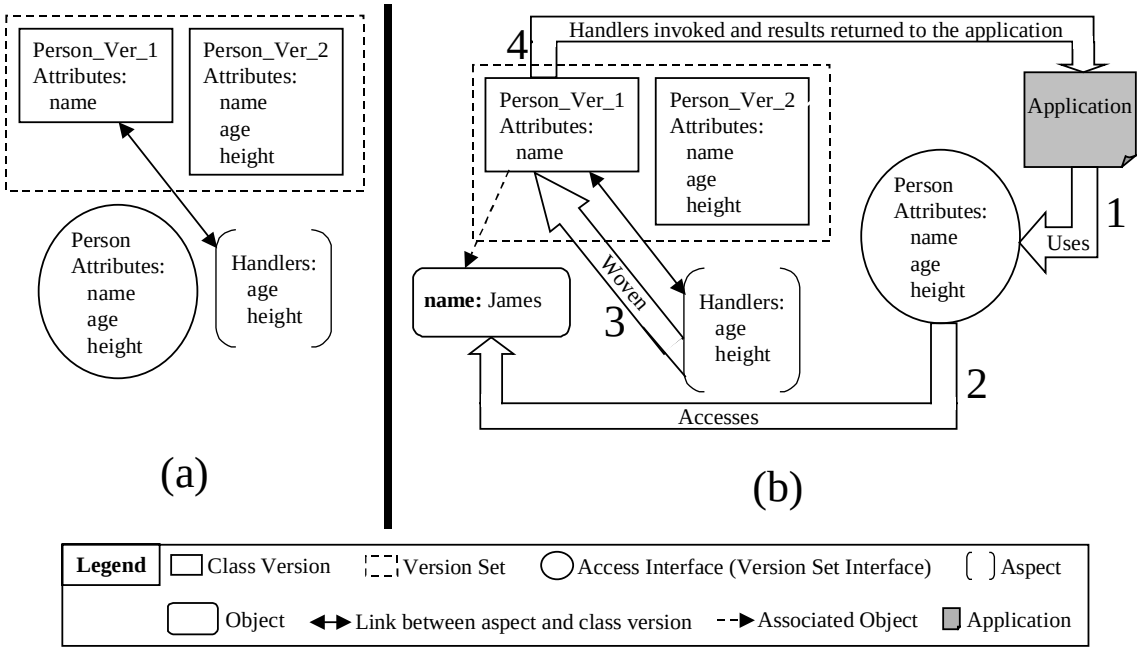

Fig. 5: Error Handlers in SADES using the Flexible Approach

4.2 Update/Backdate Methods

We now discuss implementation of the instance adaptation strategy of CLOSQL [33] 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 6 shows the implementation of the update/backdate methods strategy using our approach. As shown in fig. 6(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. 6(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. 6(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.

4.3 Changing the instance adaptation strategy

This section describes how the flexible approach allows seamless transformation from one instance adaptation strategy to another. As discussed earlier such a need can 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. 5) and discuss how this strategy can be replaced by the update/backdate methods strategy (as shown in fig. 6). In order to adopt this new strategy in SADES the aspect containing the instance adaptation strategy in fig. 3(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. 6 corresponds to the one in fig. 5 after the instance adaptation strategy has been changed. As shown in fig. 6(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.

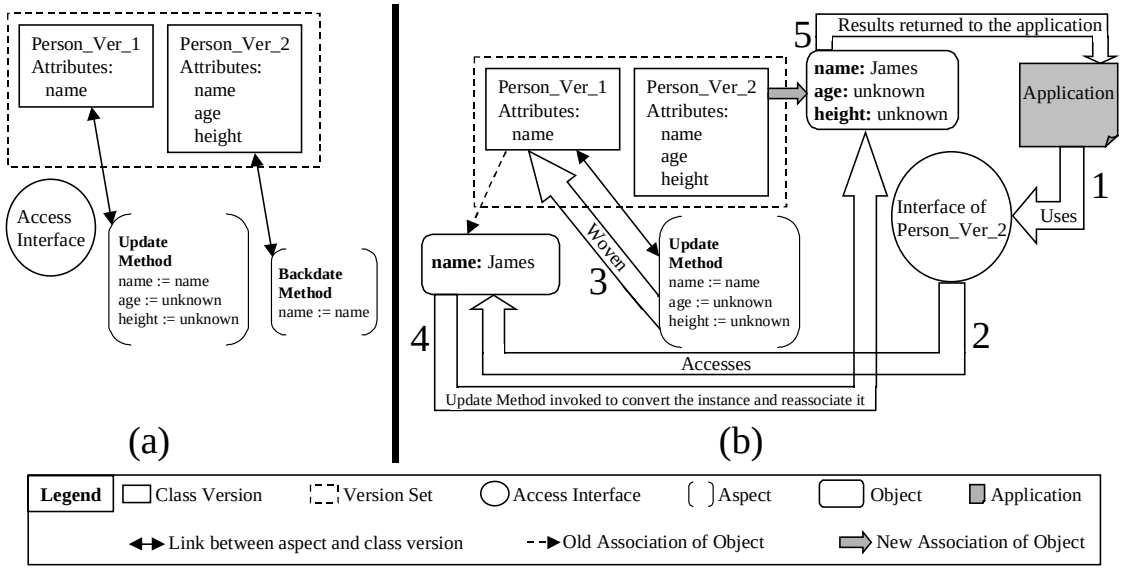

Fig. 6: Update/Backdate Methods in SADES using the Flexible Approach

5 Related Work

A number of approaches have been employed to ensure structural consistency. [7, 9] provide theoretical foundations for the purpose. [13] proposes the use of a special consistency checking tool. [14] employs transformation functions and object migration. [5] suggests *screening* deleted or modified information from application programs instead of deleting or modifying it physically. Only conversion functions have access to the screened information. The screening information and transformation functions can be encapsulated in aspects. [25] exploits the difference between old and new definitions through a program generator used to produce transformation programs and tables. A similar approach can be employed to generate required aspects upon creation of a new class version.

Separation of concerns in object-oriented databases has been explicitly considered in [44] which identifies some of the cross-cutting concerns in object databases and proposes an aspect-oriented extension to capture these explicitly. Some existing work has also addressed separation of concerns implicitly. The concept of object version derivation graphs [31] separates version management from the objects. A similar approach is proposed by [39, 40, 41, 43, 45] 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. [30] employs propagation patterns [27, 28] to exploit polymorphic reuse mechanisms 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. [42] 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.

6 Conclusions

We have proposed a flexible approach for instance adaptation during class versioning. The approach has been implemented as part of the SADES system. Changes to the instance adaptation strategies in existing systems are expensive due to the cross-cutting nature of the instance adaptation code. The adaptation routines are spread across various class versions making maintenance difficult. We have employed aspects to separate the instance adaptation code from the class versions. This localises the effect of any changes to the instance adaptation code. As a result the behaviour of adaptation routines can be modified in a cost-effective manner. It is also possible to seamlessly move to an entirely different instance adaptation strategy. This has been demonstrated by using error handlers and update/backdate methods as examples. Our approach also allows multiple instance adaptation strategies to coexist in the system. This makes it possible to use different strategies for different parts of the system or to use one strategy when data is accessed using one application and a different one when using another application.

One of the essential requirements for the approach to be effective is the need for a highly efficient weaver. Reflecting on experiences with our initial prototype we are building a sophisticated weaver for the purpose. Our future work will explore the applicability of the approach in systems employing schema evolution and schema versioning as evolution strategies. We are also interested in developing an aspect-oriented evolution framework for object-oriented databases.

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] Andany, J., Leonard, M., Palisser, C., "Management of Schema Evolution in Databases", *Proceedings of the 17th International Conference on Very Large Databases, Morgan Kaufmann 1991, pp. 161-170*

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

[5] Banerjee, J., Kim, W., Kim, H., Korth, H. F., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", *Proceedings of ACM SIGMOD Conference, SIGMOD Record, Vol. 16, No. 3, Dec. 1987, pp. 311-322*

[6] Benatallah, B., Tari, Z., "Dealing with Version Pertinence to Design an Efficient Schema Evolution Framework", *Proceedings of the International Database Engineering and Applications Symposium, Cardiff, Wales, U.K., IEEE Computer Society 1998, pp. 24-33*

[7] Bergstein, P. L., "Object-Preserving Class Transformations", *Proceedings of OOPSLA 1991, ACM SIGPLAN Notices, Vol. 26, No. 11, pp. 299-313*

[8] Bergstein, P. L., Huersch, W. L., "Maintaining Behavioral Consistency during Schema Evolution", *Proceedings of the International Symposium on Object Technologies for Advanced Software, Springer-Verlag 1993, pp. 176-193*

[9] Bergstein, P. L., "Maintenance of Object-Oriented Systems during Structural Evolution", *TAPOS - Theory and Practice of Object Systems, Vol. 3, No. 3, pp. 185-212*
[10] Boellert, K., "On Weaving Aspects", Proc. of the AOP Workshop at ECOOP '99
[11] 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*
[12] Clamen, S. M., "Type Evolution and Instance Adaptation", *School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-92-133R, June 1992*
[13] Delcourt, C., Zicari, R., "The Design of an Integrity Consistency Checker (ICC) for an Object Oriented Database System", *Proceedings of ECOOP 1991, Lecture Notes in Computer Science 512, pp. 97-117*
[14] 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*
[15] Fishman, D. H. *et al.*, "Iris: An Object Oriented Database Management System", *ACM Transactions on Office Information Systems, Vol. 5, No. 1, 1987, pp. 48-69*
[16] Fradet, P., Suedholt, M., "An Aspect Language for Robust Programming", *Proceedings of the AOP Workshop at ECOOP '99, 1999*
[17] Gamma, E. *et al.*, "Design Patterns - Elements of Reusable Object-Oriented Software", *Addison Wesley, c1995*
[18] 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*
[19] Kenens, P., *et al.*, "An AOP Case with Static and Dynamic Aspects", *Proceedings of the AOP Workshop at ECOOP '98, 1998*
[20] Kersten, M. A., Murphy, G. C., "Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming", *Proceedings of OOPSLA 1999, ACM SIGPLAN Notices, Vol. 34, No. 10, Oct. 1999, pp. 340-352*
[21] 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*
[22] 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*
[23] 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*
[24] Klaeren, H., Pulvermueller, E., Rashid, A., Speck, A., "Supporting Composition using Assertions", *Cooperative Systems Engineering Group, Computing Department, Lancaster University, Technical Report No: CSEG/4/00*
[25] Lerner, B. S., Habermann, A. N., "Beyond Schema Evolution to Database Reorganisation", *Proceedings of ECOOP/OOPSLA 1990, ACM SIGPLAN Notices, Vol. 25, No. 10, Oct. 1990, pp. 67-76*
[26] Li, Q., McLeod, D., "Conceptual Database Evolution through Learning in Object Databases", *IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 2, April 1994, pp. 205-224*
[27] Lieberherr, K. J., Huersch, W., Silva-Lepe, I., Xiao, C., "Experience with a Graph-Based Propagation Pattern Programming Tool", *Proceedings of the International CASE Workshop, IEEE Computer Society 1992, pp. 114-119*
[28] 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*
[29] Lieberherr, K. J., "Demeter", http://www.ccs.neu.edu/research/demeter/index.html
[30] 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*
[31] Loomis, M. E. S., "Object Versioning", *JOOP, Jan. 1992, pp. 40-43*
[32] 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*
[33] Monk, S., Sommerville, I., "Schema Evolution in OODBs Using Class Versioning", *SIGMOD Record, Vol. 22, No. 3, Sept. 1993, pp. 16-22*
[34] Odberg, E., "A Framework for Managing Schema Versioning in Object Oriented Databases", Proceedings of the International Conference on Database and Expert Systems Applications (DEXA) 1992, pp. 115-120

[35] Odberg, E., "A Global Perspective of Schema Modification Management for Object-Oriented Databases", *Proceedings of the 6th International Workshop on Persistent Object Systems (POS) 1994, pp. 479-502*
[36] Peters, R. J., Ozsu, M. T., "An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems", *ACM Transactions on Database Systems, Vol. 22, No. 1, March 1997, pp. 75-114*
[37] Pulvermueller, E., Klaeren, H., Speck, A., "Aspects in Distributed Environments", *Proceedings of GCSE 1999, Erfurt, Germany (to be published by Springer-Verlag)*
[38] 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*
[39] 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*
[40] Rashid, A., "SADES - a Semi-Autonomous Database Evolution System", *Proceedings of PhDOOS '98, ECOOP '98 Workshop Reader, LNCS 1543*
[41] Rashid, A., Sawyer, P., "Toward 'Database Evolution': a Taxonomy for Object Oriented Databases", *In review at IEEE Transactions on Knowledge and Data Engineering*
[42] Rashid, A., Sawyer, P., "Dynamic Relationships in Object Oriented Databases: A Uniform Approach", *Proceedings DEXA '99, LNCS 1677, pp. 26-35*
[43] Rashid, A., Sawyer, P., "Transparent Dynamic Database Evolution from Java", *Proceedings of OOPSLA 1999 Workshop on Java and Databases: Persistence Options*
[44] Rashid, A., Pulvermueller, E., "From Object-Oriented to Aspect-Oriented Databases", *In review at DEXA 2000*
[45] "SADES Java API Documentation",
http://www.comp.lancs.ac.uk/computing/users/marash/research/sades/index.html
[46] Sjoberg, D., "Quantifying Schema Evolution", *Information and Software Technology, Vol. 35, No. 1, pp. 35-44, Jan. 1993*
[47] 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*
[48] Tamzalit, D., Oussalah, C., " Instances Evolution vs Classes Evolution" , *Proceedings of DEXA 1999, LNCS 1677, pp. 16-25*
[49] Zicari, R., "A Framework for Schema Updates in an Object-Oriented Database System", *Proceedings of the International Conference on Data Engineering 1991, IEEE Computer Society Press, pp. 2-13*
[50] Zdonik, B., "Maintaining Consistency in a Database with Changing Types", *ACM SIGPLAN Notices, Vol. 21, No. 10, Oct. 1986*