

Reconfigurability in Object Database Management Systems: An Aspect-Oriented Approach

Awais Rashid[†] and Ruzanna Chitchyan[‡]

[†]Computing Department, Lancaster University, Lancaster LA1 4YR, UK
E-mail: marash@comp.lancs.ac.uk Fax: +44 1524 593608

[‡]The Open University, UK & Open College of the North West, Storey Institute,
Lancaster LA1 1TH, UK
Email: r.chitchyan@lancaster.ac.uk Fax: +44 1524 388467

ABSTRACT. *Cost-effective reconfiguration in ODBMSs is difficult to achieve due to the trade-off between modularity and efficiency. Existing ODBMS designs offer limited reconfigurability because reconfigurable features are closely woven with the components to improve efficiency. This paper proposes the use of aspects - entities used by Aspect-Oriented Programming to localise cross-cutting concerns - to separate reconfigurable features from the components regardless of their granularity. This provides a cost-effective solution for both static and dynamic reconfiguration. The effectiveness of the approach is demonstrated by discussing dynamically reconfigurable instance adaptation in the SADES evolution system.*

KEY WORDS: *Reconfigurability, Object Database Management Systems, Aspect-Oriented Programming, Aspect-Oriented Databases, Static Reconfiguration, Dynamic Reconfiguration, Evolution*

1. Introduction

Like any other software product reconfigurability is an essential requirement in object database management systems (McCann 2000). An effective reconfiguration mechanism is one which localises the impact of changes and automatically propagates them to the rest of the system without compromising consistency. This reduces maintenance and upgrade costs and makes it possible to customise (with minimal effort) the object database management system to the specific needs of an organisation. An example of the latter is instance adaptation during object database evolution¹. For one organisation it might be sufficient that objects simulate a conversion to a compatible type interface (similar to error handlers employed by ENCORE (Skarra et al. 1986)) while for another organisation it might be essential to actually convert objects between historical type definitions (similar to update/backdate methods employed by CLOSQL (Monk et al. 1993)). Such customisations may even be application specific (this implies dynamic reconfiguration using more than one instance adaptation strategy). An effective reconfiguration mechanism localising changes to the instance adaptation strategy and the associated adaptation routines can provide such customisation in a cost-effective fashion.

Existing approaches e.g. (Guzenda 2000, McCann 2000) employ a component-based architecture for the ODBMS to achieve reconfigurability. However, there exists a trade-off between

¹ Evolution case studies at Open College of the North West where day-to-day activities revolve around the databases.

modularity and performance (Kiczales et al. 1997). As a result reconfigurability is available only at a coarse granularity. For example, the transaction manager component can be exchanged with the ripple effect limited to the glue code for the various components². However, similar functionality is not available at a finer granularity. Relatively minor changes to the individual components are expensive as their modularity is compromised to preserve both modularity and performance of the ODBMS. Code handling any cross-cutting features in these components is spread across them (Kiczales et al. 1997). Consequently changes to these cross-cutting features are not localised making reconfiguration an expensive task.

This paper proposes the use of aspects - entities used by Aspect-Oriented Programming (AOP) (Kiczales et al. 1997) to localise cross-cutting concerns - to separate reconfigurable features at both coarse and fine granularity. This allows cost-effective reconfiguration as changes are localised. The aspects encapsulating reconfigurable features can be modified at compile-time (static reconfiguration) or run-time (dynamic reconfiguration). The performance of the system is not compromised due to the modularity as aspects are merged with the entities cross-cut by them (at compile-time or run-time) to produce efficient code. The next section takes a closer look at the reconfigurability problem in ODBMSs. Section 3 provides an overview of Aspect-Oriented Programming. The aspect-oriented reconfiguration approach is discussed in section 4. Section 5 identifies some open issues while section 6 concludes the paper and discusses future directions.

2. Reconfigurability: The Problem

(Kiczales et al. 1997) demonstrates that a highly modular system is not necessarily the most efficient. Efficient implementations tend to be less modular with the various components closely woven. This is termed as *code tangling* (Kiczales et al. 1997). There is a trade-off between modularity/maintainability and efficiency/performance. This trade-off has a strong bearing on the reconfigurability of an ODBMS as shown in fig. 1.

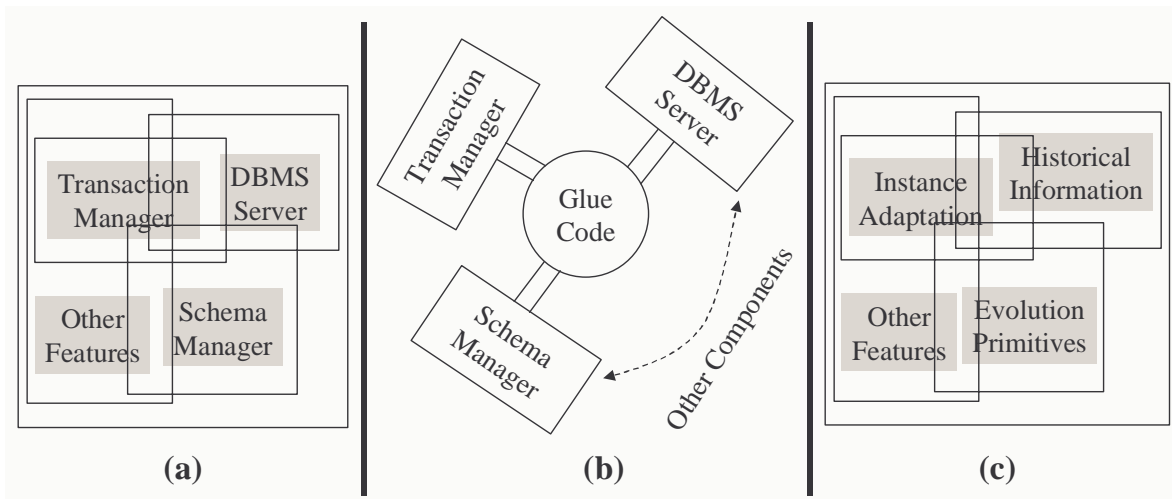


Fig. 1: Reconfigurability issues in (a) a monolithic ODBMS (b) a component-based ODBMS (c) a component (schema manager) in a component-based ODBMS

² Such an exchange might also introduce erroneous behaviour into the system (Brown et al. 1996). However, for the purpose of this discussion it is assumed that consistency is maintained.

Fig. 1(a) shows the structure of a monolithic ODBMS. Only three key components: the DBMS server, the transaction manager and the schema manager are shown. Other components such as the storage manager, etc. have been omitted for simplification. As shown in the figure the various components are tangled in a monolithic system. While such a closely woven implementation provides good performance/efficiency, reconfigurability is an expensive and difficult task as changes to the various components are not localised (due to code tangling). Changing the transaction model employed by the transaction manager, for example, at either compile-time or run-time can have a large ripple effect on the whole system.

Fig. 1(b) shows a modular, component-based implementation of an ODBMS similar to the one proposed by (McCann 2000). Again, only three key components: the DBMS server, the transaction manager and the schema manager are shown for simplification. Such a design provides effective reconfigurability. For example, the transaction manager (or any other component) can be exchanged with the ripple effect mainly limited to the glue code. However, in order to strike the right balance between modularity and efficiency the design of the individual components is not highly modular. As shown in fig. 1(c) the modularity of the individual components is compromised to preserve both modularity and efficiency of the ODBMS. As a result the coarse-grained component, the OODBMS, is reconfigurable. However, reconfigurability at a finer granularity (i.e. the components forming the OODBMS) is expensive as design at this level is largely monolithic. Reconfiguring the instance adaptation strategy in such a system, for example, requires exchanging the whole schema manager component as changes to the instance adaptation strategy are not localised (cf. fig. 1(c)). The reconfigurability problem simply moves to a different granularity. This also limits possibilities for dynamic reconfiguration. Application specific reconfiguration of the instance adaptation strategy, for example, requires dynamically exchanging the whole schema manager component which is an expensive task. Such reconfiguration would be cost-effective if changes at the fine granularity were localised without compromising the system performance obtained through closely woven components i.e. both modularity and efficiency need to be preserved.

3. Aspect-Oriented Programming

Aspect-oriented programming (Kiczales et al. 1997) 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 (Gamma et al. 1995) 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. 2 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*. 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 (Kenens et al. 1998, Kiczales et al. 1997).

Different AOP techniques and research directions can be identified. They all share the common goal of providing an improved separation of concerns. AspectJ (Xerox 2000) 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. (Boellert 1999), for example, describes an aspect language and a weaver for Smalltalk.

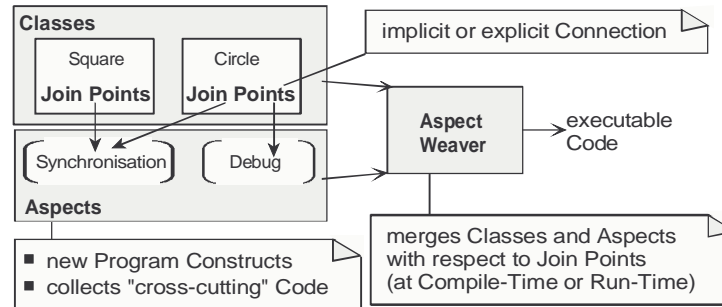


Fig. 2: Aspect-Oriented Programming

Other AOP approaches aiming at achieving a similar separation of concerns include subject-oriented programming (Harrison et al. 1993), composition filters (Aksit et al. 1998), adaptive programming (Lieberherr 2000, Mezini et al. 1998) and Hyperspaces (IBM 2000). 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. Hyperspaces introduce the notion of *multi-dimensional separation of concerns* by permitting clean separation of multiple, potentially overlapping and interacting concerns simultaneously with support for on-demand remodularisation to encapsulate new concerns at any time.

Experience reports and assessment of AOP can be found in (Kersten et al. 1999, Pulvermueller et al. 1999).

4. Reconfigurability: The Aspect-Oriented Solution

Since reconfigurability is a cross-cutting concern (as discussed in section 2), we propose separating reconfigurable features from components using aspects. This applies to both coarse and fine-grained components. As a result the approach can be employed for cost-effective reconfiguration in monolithic ODBMSs (cf. fig. 1(a)), individual components in component-based ODBMSs (cf. fig. 1(b) & (c)) or components at finer granularities.

As shown in fig. 3 aspects are used to separate reconfigurable features from the core functionality and other non-configurable features of a component. Changes to the features encapsulated by aspects are localised making it possible to achieve a high degree of reconfigurability. Both minor and major reconfigurations can be carried out in a cost-effective fashion. It also makes it possible to exchange fine-grained reconfigurable features without a ripple effect on the rest of the component. Changes are automatically propagated during weaving (at compile-time or run-time) or reweaving (at run-time). It might be argued that the proposed approach simply shifts the complexity of the reconfiguration process to the aspect weaver. This might be true if for each reconfigurable feature a specifically designed aspect language and weaver were being used. Since most ODBMSs are developed using general-purpose programming languages, a general-purpose

aspect language similar to AspectJ (Xerox 2000) and its associated weaver will be sufficient to provide a woven solution which is both performance optimised and correct. Understandably the correctness of the code produced by the weaver and efficiency of the weaver (especially during dynamic weaving) will be crucial to the process. In case one or a few reconfigurable features require specific aspect languages and weavers the effort to develop these will pay-off in terms of reduced reconfiguration and maintenance costs.

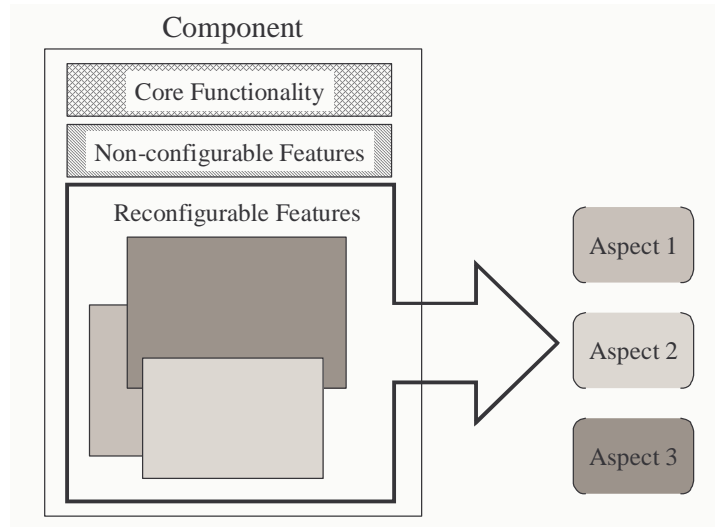


Fig. 3: Using aspects to separate reconfigurable features from a component (whether coarse-grained or fine-grained)

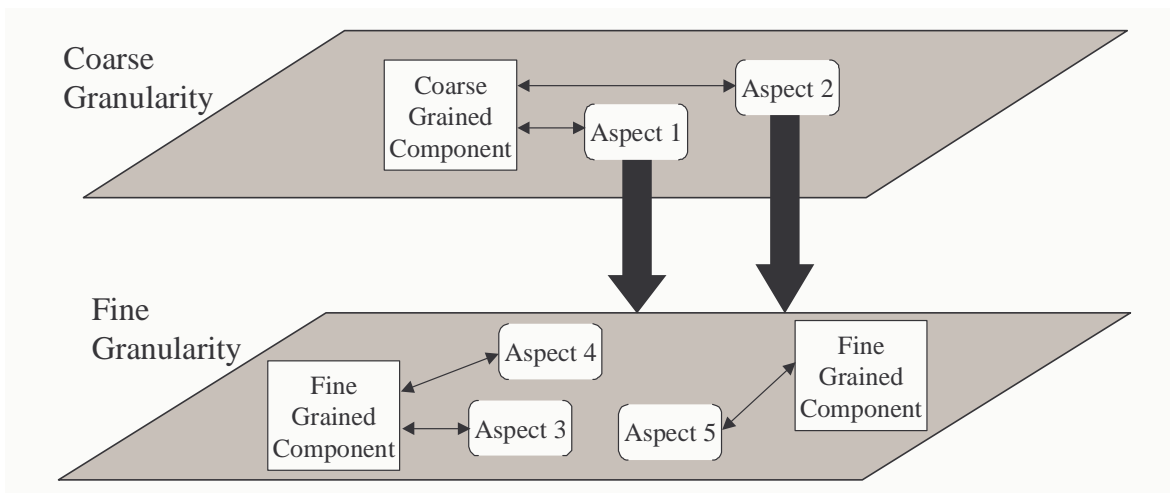


Fig. 4: Propagating reconfigurability aspects at coarse granularity to fine granularity

It should be noted that although fig. 3 shows reconfigurability aspects associated with one particular component, one such aspect can serve more than one component if the same reconfigurable feature exists in all of them. These semantics make it possible to separate common reconfigurable features at a coarse granularity and transparently propagate them to finer granularities as shown in fig. 4. One example of such reconfigurable aspects is error-handling. Although the specific error-handling routines for various components considerably vary, the ODBMS normally has a uniform error-handling strategy. The error-handling strategy can exist at the coarse granularity (the ODBMS level) as a reconfigurable aspect while the error-handling

routines specific to the finer-grained components can exist at their particular granularities. The reconfigurable error-handling strategy is automatically propagated to the finer-grained components when it is woven into the ODBMS.

The aspect-oriented solution also provides support for both static and dynamic reconfiguration as discussed in the following sections.

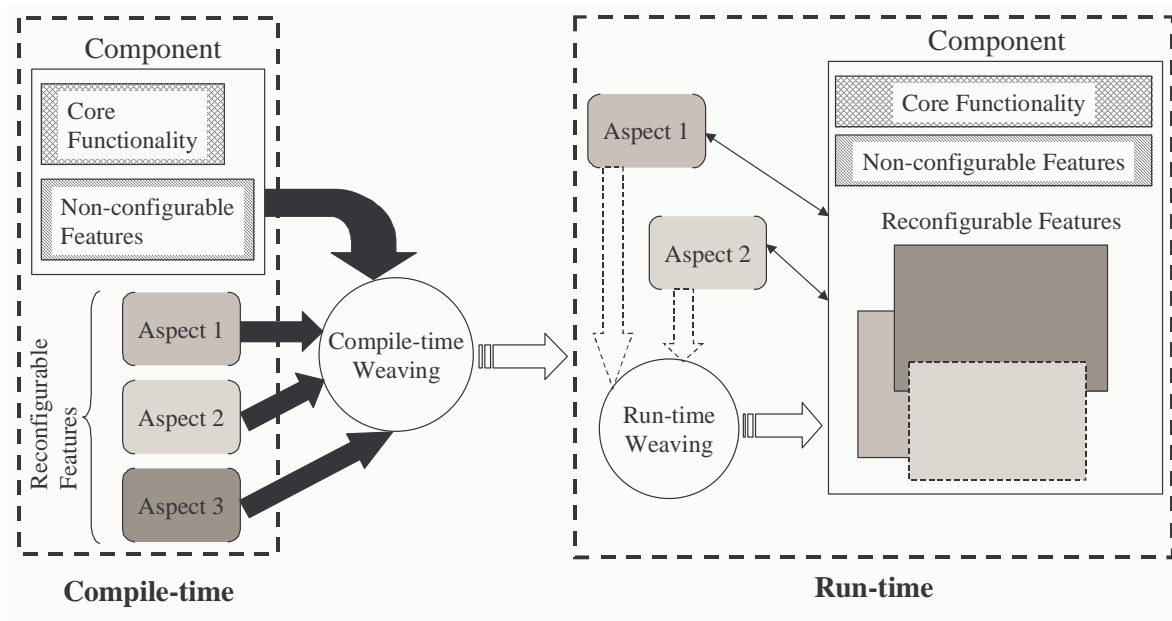


Fig. 5: Static and dynamic reconfiguration using the aspect-oriented approach

4.1 Static Reconfiguration

The aspect-oriented approach makes static reconfiguration possible by allowing reconfiguration or exchange of features encapsulated by aspects before the weaving tool performs the weaving at compile-time. As shown in fig. 5 *Aspect 1*, *Aspect 2* and *Aspect 3* can be reconfigured or exchanged with localised changes before the compile-time weaving process weaves them with the rest of the component. Aspects encapsulating features that need reconfiguration at compile-time only and not at run-time are merged with the rest of the component at compile-time and do not exist at run-time. This reduces the overhead of managing a large number of aspects and their weaving/reweaving at run-time. *Aspect 3* in fig. 5 is an example of such an aspect.

4.2 Dynamic Reconfiguration

Aspects encapsulating features which require reconfiguration at run-time have a lifetime extended beyond compile-time. They exist at run-time and may even outlive the program execution (Rashid 2000a). *Aspect 1* and *Aspect 2* in fig. 5 are examples of such aspects. As shown in fig. 5 such aspects fall into two categories:

- Aspects encapsulating features less frequently reconfigured at run-time
- Aspects encapsulating features frequently reconfigured at run-time

Aspect 1 in fig. 5 is an example of the former while *Aspect 2* is an example of the latter. In the example *Aspect 1* is woven at compile-time and rewoven at run-time only when it is reconfigured

(rarely). The weaving of *Aspect 2* on the other hand is left to run-time. As shown by the dotted line around the woven feature (encapsulated by *Aspect 2*) *Aspect 2* is woven and rewoven at run-time frequently due to extensive dynamic reconfiguration. It should be noted that the scenario in fig. 5 is just an example. *Aspect 2* could have been woven at compile-time and still frequently reconfigured at run-time. The example simply demonstrates that it is not necessary to weave all the dynamically reconfigurable features at compile-time and vice versa.

4.3 Implementation

The aspect-oriented approach has been employed to provide a dynamically reconfigurable instance adaptation strategy in the SADES evolution system (Rashid et al. 1998, Rashid et al. 1999a, Rashid et al. 1999b, Rashid 2000b). It has also been applied to achieve reconfigurable versioning, clustering and inheritance in object-oriented databases (Rashid et al. 2000b). Due to space limitations the following discussion focuses on reconfigurable instance adaptation in SADES only. Interested readers are referred to (Rashid et al. 2000b) for a description of other reconfigurable features mentioned above.

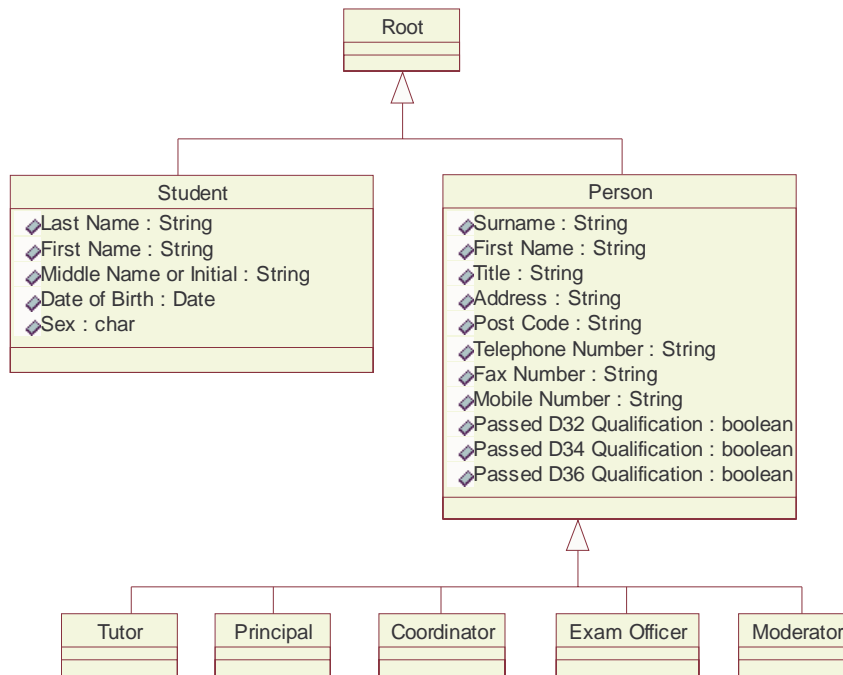


Fig. 6: Database Structure before Evolution

In order to demonstrate the reconfigurability of the instance adaptation strategy in SADES we have employed one of the evolution scenarios from our case studies at Open College of the North West. Fig. 6 shows the database structure prior to evolution (it is assumed that the class hierarchy is single-rooted). The class hierarchy does not conform to good OO design principles as the class *Student* is not a subclass of the class *Person*. Fig. 7 shows the structure to be adopted to conform to good practice. In this structure the class *Student* becomes a subclass of the class *Person*. The class *Person* defines attributes common to both student and staff objects. A non-leaf class *Staff* is introduced to capture attributes specific to staff objects. In order to simplify the description, the modification of the class *Person* in the evolution scenario is used for describing the SADES

instance adaptation strategy. For further simplification attributes defined by subclasses are not considered in the objects associated with *Person*. This simplification is syntactically and semantically correct as *Person* is not an abstract class and can be instantiated directly.

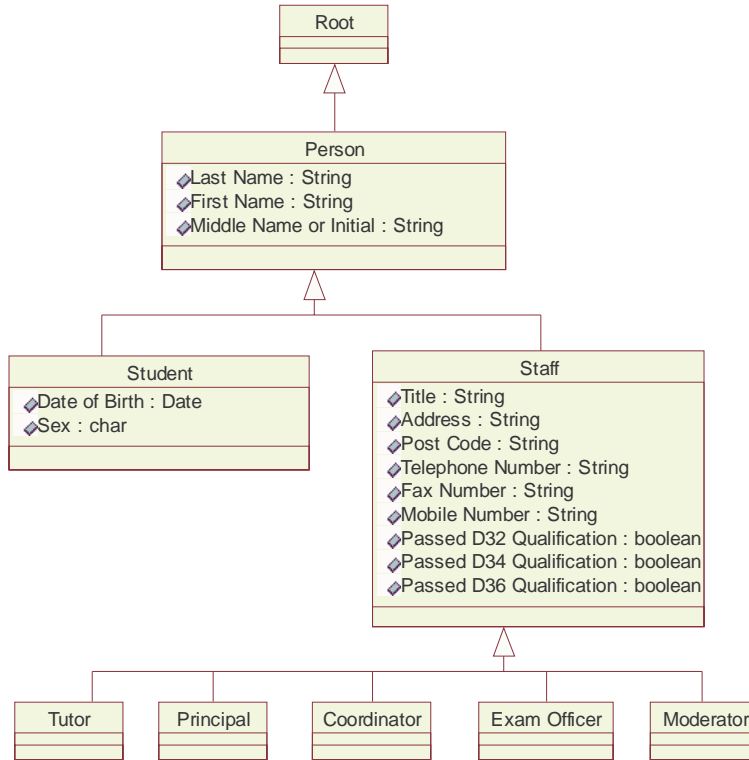


Fig. 7: Database Structure after Evolution

As shown in fig. 8 the instance adaptation strategy and adaptation routines in SADES are separated from the schema manager and class versions respectively using aspects (the aspects are specified using a general-purpose, declarative aspect language modelled on AspectJ (Xerox 2000)). The instance adaptation strategy is reconfigurable and can be dynamically woven into the schema manager using a weaver providing support for compile-time, run-time and persistent aspects. The selection of aspects containing the adaptation routines is dependent on the instance adaptation strategy chosen. For example, from fig. 8 aspects containing the handlers will be woven only when the error handlers strategy (Skarra et al. 1986) is woven into the SADES schema manager. If the aspect containing the error-handlers strategy is exchanged with the one containing the update/backdate methods strategy (Monk et al. 1993), the aspects containing the handlers will be exchanged with those containing the update/backdate methods. Note that the choice of instance adaptation strategies is not limited to error-handlers and update/backdate methods. Other instance adaptation strategies can be employed as shown in fig. 8.

Separating the instance adaptation code into aspects allows reconfiguration of the instance adaptation strategy and adaptation routines without posing maintenance problems for the schema manager or existing class versions. The changes are local to the aspect and are propagated to the class versions through dynamic weaving. This makes it possible to reconfigure the instance adaptation strategy for specific evolution scenarios such as performing or simulating an *information preserving* move of attributes across classes or class versions. For example, in fig. 8, the backdate method for objects associated with *Person_V2* can test whether the object being

associated with *Person_V1* is a *Staff* object. If this is the case the attributes *Title*, *Address*, etc. can take on the values of the attributes in the *Staff* object. This is a customisation of the update/backdate methods strategy employed by CLOSQL and makes it possible to simulate the *move attribute primitive* not available in the SADES evolution taxonomy.

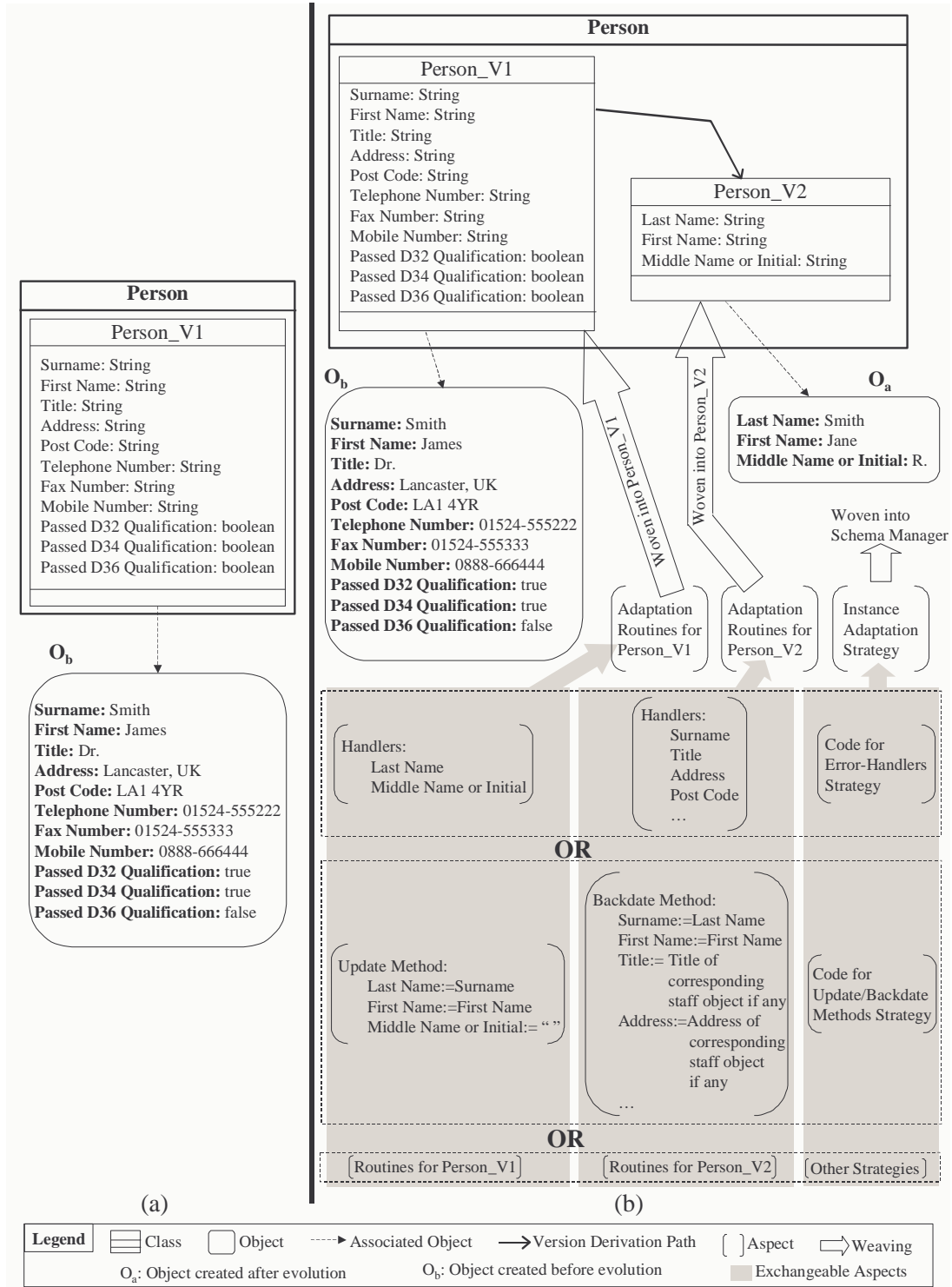


Fig. 8: Instance adaptation in SADES (a) Before evolution (b) After evolution

The high degree of reconfigurability in the SADES instance adaptation strategy is in direct contrast with existing evolution systems such as ENCORE (Skarra et al. 1986) and CLOSQL (Monk et al. 1993). These 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 spread across the system and adoption of a new strategy has a ripple effect on the rest of the system and triggers the need for changes to all or a large number of versions of existing classes. A more detailed description of the dynamically reconfigurable instance adaptation strategy can be found in (Rashid et al. 2000a).

5. Open Issues

The reconfigurability approach proposed in this paper heavily relies on weaver efficiency. One of the key research issues is the development of correct and efficient weaving mechanisms. Efficiency is particularly critical for dynamic weaving as it introduces additional overhead at run-time and can be feasible only with efficient weavers. AOP research, to date, has mainly focussed on the development of AOP techniques, their correctness and design issues for aspect languages. Different lifetimes of aspects have also been explored (Kenens et al. 1998, Kiczales et al. 1997, Rashid et al. 2000b, Rashid 2000a). However, the development of efficient weavers has not been considered. We are of the view that efficiency and correctness of weavers will be the determining factors for commercial feasibility of AOP and hence need extensive research. One interesting solution worth exploring is *selective weaving* (Rashid et al. 2000a): weaving only the modified or previously unwoven parts of an aspect instead of the whole aspect itself.

Another issue identified during the course of our work is the need for parameterised aspects. At present this is not supported by aspect languages. If aspect parameterisation is available the aspects encapsulating reconfigurable features can be more generic and maintainable. An aspect can be parameterised by classes in which it is to be woven, hence, making the join points and crosscuts generic. Special *weave parameters* can be used to provide a generic reconfiguration mechanism during dynamic weaving.

6. Conclusions and Future Work

This paper has proposed the use of aspects to achieve a high degree of reconfigurability in object database management systems. The novelty of the work lies in the cost-effective reconfiguration mechanism localising changes to reconfigurable features and automatically propagating them during weaving. It was argued that similar reconfigurations are expensive in existing systems because reconfigurable features are closely woven with non-configurable features and each other. This code tangling is a direct effect of the trade-off between modularity and efficiency. The use of aspects to separate reconfigurable features provides a mechanism which preserves both modularity and efficiency. The features encapsulated by aspects can be reconfigured at both compile-time and run-time with localised changes and closely woven with other features using weaving tools providing support for different aspect lifetimes. The effectiveness of the approach has been demonstrated through its application to achieve dynamically reconfigurable, cost-effective instance adaptation in the SADES evolution system. Our work in the future will focus on the open issues identified in section 5. We will explore the development of efficient weaving

mechanisms in order to reduce the overhead associated with such a highly reconfigurable architecture especially at run-time. We will also investigate the syntax and semantics of aspect parameterisation and its potential to make the reconfiguration mechanism more generic.

Acknowledgements. The work presented in this paper is supported by EPSRC grant GR/R08612: AspOEv - An Aspect-Oriented Evolution Framework for Object-Oriented Databases. The authors would also like to thank the Open College of the North West, UK for providing opportunities for database evolution case studies.

References

- (Aksit et al. 1998): M. Aksit, B. Tekinerdogan, *Aspect-Oriented Programming using Composition Filters*, Proceedings of the AOP Workshop at ECOOP '98, 1998
- (Boellert 1999): K. Boellert, *On Weaving Aspects*, Proceedings of the AOP Workshop at ECOOP '99, 1999
- (Brown et al. 1996): A. W. Brown, K. C. Wallnau, *Engineering of Component-Based Systems*, Component-Based Software Engineering, IEEE Computer Society Press, 1996, pp. 7-15
- (Gamma et al. 1995): E. Gamma, et al., *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley, c1995
- (Guzenda 2000): L. Guzenda, *Objectivity/DB - A High Performance Object Database Architecture*, Invited Talk, Workshop on High Performance Object Databases, Cardiff, UK, July 2000
- (Harrison et al. 1993): W. Harrison, H. Ossher, *Subject-Oriented Programming (A Critique of Pure Objects)*, Proceedings of OOPSLA 1993, ACM SIGPLAN Notices, Vol. 28, No. 10, Oct. 1993, pp. 411-428
- (IBM 2000): IBM Research, USA, *Multi-dimensional Separation of Concerns using Hyperspaces*, <http://www.research.ibm.com/hyperspace/>
- (Kenens et al. 1998): P. Kenens, et al., *An AOP Case with Static and Dynamic Aspects*, Proceedings of the AOP Workshop at ECOOP '98, 1998
- (Kersten et al. 1999): M. A. Kersten, G. C. Murphy, *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
- (Kiczales et al. 1997): G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, *Aspect-Oriented Programming*, Proceedings of ECOOP '97, LNCS 1241, pp. 220-242
- (Lieberherr 2000): K. J. Lieberherr, *Demeter*, <http://www.ccs.neu.edu/research/demeter/index.html>
- (McCann 2000): J. McCann, *Component-based Operating Systems and their Implications for Database Architectures*, Invited Talk, Workshop on High Performance Object Databases, Cardiff, UK, July 2000
- (Mezini et al. 1998): M. Mezini, K. J. Lieberherr, *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
- (Monk et al. 1993): S. Monk, I. Sommerville, *Schema Evolution in OODBs Using Class Versioning*, SIGMOD Record, Vol. 22, No. 3, Sept. 1993, pp. 16-22
- (Pulvermueller et al. 1999): E. Pulvermueller, H. Klaeren, A. Speck, *Aspects in Distributed Environments*, Proceedings of GCSE 1999, Erfurt, Germany (to be published by Springer-Verlag)
- (Rashid et al. 1998): A. Rashid, P. Sawyer, *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
- (Rashid et al. 1999a): A. Rashid, P. Sawyer, *Dynamic Relationships in Object Oriented Databases: A Uniform Approach*, Proceedings of DEXA '99, LNCS 1677, pp. 26-35
- (Rashid et al. 1999b): A. Rashid, P. Sawyer, *Transparent Dynamic Database Evolution from Java*, Proceedings of OOPSLA 1999 Workshop on Java and Databases: Persistence Options (extended version to appear in L' Object Journal, Vol. 6, No. 3, November 2000)
- (Rashid et al. 2000a): A. Rashid, P. Sawyer, E. Pulvermueller, *A Flexible Approach for Instance Adaptation during Class Versioning*, Proceedings of ECOOP 2000 OODB Symposium (in print as an LNCS volume by Springer-Verlag)
- (Rashid et al. 2000b): A. Rashid, E. Pulvermueller, *From Object-Oriented to Aspect-Oriented Databases*, Proceedings of DEXA 2000, LNCS1873, pp. 125-134
- (Rashid 2000a): A. Rashid, *On to Aspect Persistence*, To Appear in Proceedings of Net.ObjectDays 2000 Symposium on Generative and Component-Based Software Engineering (GCSE 2000)
- (Rashid 2000b): A. Rashid, *SADES Java API Documentation 1999-2000*, <http://www.comp.lancs.ac.uk/computing/users/marash/research/sades/index.html>
- (Skarra et al. 1986): A. H. Skarra, S. B. Zdonik, *The Management of Changing Types in an Object-Oriented Database*, Proceedings of the 1st OOPSLA Conference, Sept. 1986, pp. 483-495
- (Xerox 2000): Xerox PARC, USA, *AspectJ Home Page*, <http://aspectj.org/>