Multi-Paradigm Implementation of an Object Database Evolution System

Awais Rashid

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

Abstract. This paper provides an overview of the use of multiple paradigms in the implementation of the SADES object database evolution system. The discussion highlights how rules and declarative specification of cross-cutting instance adaptation behaviour have been supported in SADES. Language cross-binding during the implementation is also discussed. It is argued that a multi-paradigm implementation is not only influenced by the system requirements and design but also by the constraints imposed by the implementation environment.

1 Introduction

This paper provides an overview of the use of multiple paradigms in the implementation of the SADES object database evolution system [10] [11] [13]. SADES provides support for three key types of changes in an object database:

- Class hierarchy evolution
- Class structure evolution
- Object evolution

Class hierarchy and class structure evolution is carried out through a mechanism known as *class versioning* [7] [13] [14]. A new version of a class is created each time it is modified. Objects and applications are bound to particular class versions for effective forward and backward compatibility of changes. However, the binding between objects and classes is flexible as objects can either be made to simulate conversion or can be physically converted across versions of the same class. This is termed *instance adaptation*. Changes to the state of an object are managed through *object versioning* [4]. A new version of an object is created each time its state needs to be preserved. This is complemented by suitable workgroup support [4] and long transaction [4] mechanisms.

The various evolution operations in SADES are governed by a set of rules. These rules are production rules [9] represented in a *condition-action* format and must reside in an integrated rulebase so that evolution support may be extended to them in future. Instance adaptation in SADES must be flexible so that it may be customised to the specific needs of an organisation or application. This implies that instance adaptation behaviour is to be specified by the maintainer in a declarative fashion. This problem is compounded by the fact that this behaviour usually cuts across the various versions of

a class [13]. The rest of this discussion highlights how rules and declarative specification of cross-cutting instance adaptation behaviour have been supported in SADES. Language cross-binding during the implementation is also discussed. However, before proceeding on to this discussion it is important to summarise the constraints imposed by the development environment for a better appreciation of the problems.

2 Implementation Constraints

SADES has been implemented as a layer on top of the commercially available Jasmine object database management system [3]. Most evolution operations require low-level access to the underlying Jasmine database and, hence, make extensive use of the proprietary Jasmine language ODQL. ODQL is an object-oriented language, is polymorphic in nature and can be used either through its associated interpreter or by embedding its statements in C or C++. SADES employs a combination of both mechanisms; the implementation embeds ODQL statements in C++ invoking the interpreter dynamically whenever the desired operations cannot be performed through embedded statements (language cross-binding will be discussed in detail in section 5). While both C++ and ODQL provide the usual *if-then* conditional language constructs these are not suitable for representing production rules. Conditional constructs cannot capture the semantic information represented by production rules.
Since the ODQL/C++ implementation is fairly low-level a high-level access to the evolution operations is provided through a Java API. This API is used by both application programmers and maintainers. Due to the cross-cutting nature of instance adaptation behaviour the obvious choice is the use of aspect-oriented programming [6]. However, due to the requirement that this behaviour be specified in a declarative fashion mechanisms such as composition filters [1] and meta-object protocols [5] cannot be employed. Linguistic constructs and, hence, an aspect language is an ideal solution. The use of AspectJ [2] is, however, not possible in this context despite the fact that it has been developed for aspect-oriented programming in Java. This is because the instance adaptation behaviour specified by the maintainer relates to persistent entities (the class versions in the database schema) and, hence, is persistent itself. AspectJ does not provide support for persistent aspects at present[1].

3 Rulebase

The key question for implementing the rulebase in SADES was the choice of an appropriate representation for rules. First-class objects seemed to be the obvious choice due to the OO languages underlying the SADES implementation. However, for performance reasons rules have not been implemented as first-class objects in SADES. Instead, as shown in fig. 1, the *action* parts of all the rules are implemented as static methods of a single class. The *conditions* are specified during the

[1] AspectJ 0.8b2.

implementation of the evolution operation. If the condition is satisfied the evolution operation delegates control to the appropriate rule. While at first glance this might seem to be a poor design, the approach has several advantages:

- One rule can be triggered as a result of different conditions (or their combinations) to be true. By specifying conditions as part of the evolution operation implementation the need to evaluate the conditions against a global database state is avoided. Instead the condition is evaluated against the state defined by the context of the evolution operation in question.
- Specifying conditions as part of the evolution operation makes the rule execution sequences more explicit within the context of the given operation. This highlights the interdependencies among the rules for a particular type of change making future modifications to the behaviour of evolution operations easy and less costly.

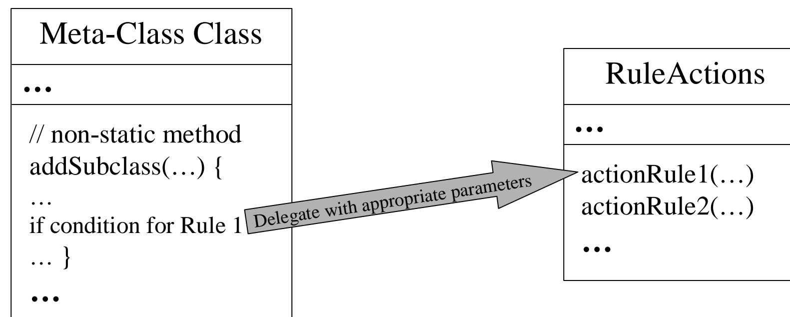

Fig. 1. Condition, action separation during rule implementation in SADES

4 Instance Adaptation

As discussed in [13] the instance adaptation behaviour in an object database system is cross-cutting in nature. This is because traditionally 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. Adaptation routines for a particular class version often reference the structure of other class versions hence resulting in code tangling across various versions of a class. Due to the cross-cutting nature of code handling instance adaptation behaviour aspect-oriented programming techniques have been employed in SADES for implementing this behaviour. The instance adaptation mechanism has been implemented as a combination of two aspect-orientation mechanisms: composition filters and an aspect language (and its associated weaver).

The aspect language is declarative in nature and has been modelled on AspectJ [2]. It provides three simple constructs facilitating:

- identification of join points between the aspects and class versions
- introduction of new methods into the class versions
- redefinition of existing methods in the class versions

The maintainer specifies the instance adaptation aspects as declarative statements passed as strings to methods in the Java API. The aspect specification is parsed to generate the persistent aspects which are in turn associated with the class versions. The weaver supporting the aspect language has been developed in Java. It provides support for persistent aspects and exposes its functionality to the rest of the system through a *weaver interface* object.

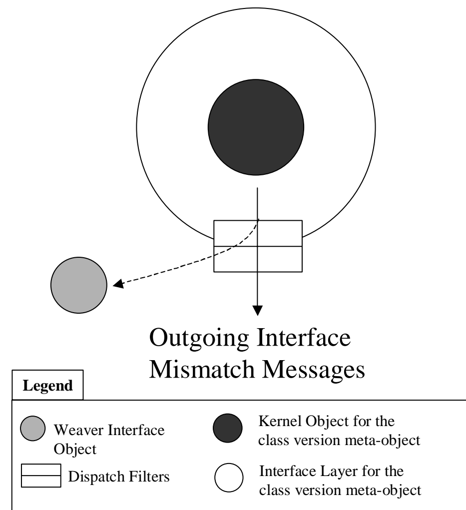

Fig. 2. Interception of interface mismatch messages and their delegation to the weaver using composition filters

In order to trap interface mismatch messages arising from incompatibility between objects and class version definitions used to access them, and delegate these messages to the weaver to weave (or reweave), composition filters are employed (cf. fig 2). Composition filters are very effective in message interception and hence are an ideal choice for this purpose. An output dispatch filter intercepts any interface mismatch messages and delegates them to the weaver which then dynamically weaves (or reweaves) the required instance adaptation aspect based on a timestamp check. The appropriate instance adaptation routine is then invoked to return the results to the application. It should be noted that the composition filters mechanism is used by the system internally and, hence, does not violate the system requirement for declarative specification of instance adaptation behaviour.

5 Language Cross-Binding

As mentioned earlier SADES has been implemented using a combination of three object-oriented programming languages: Java, C++ and ODQL. The system comprises of two layers:

- The ODQL-C++ layer which provides most of the low-level functionality.

- The Java layer which employs the functionality exposed by the ODQL-C++ layer to offer a client API.

While all three languages used in the implementation support the OO paradigm, they differ considerably in their nature and semantics. For example, both ODQL and C++ support parametric polymorphism. However, in ODQL users cannot define new parameterised types. Both ODQL and Java are single-rooted while C++ is not. Sometimes the differences arise from the way the three languages are supported by Jasmine. For example, ODQL is interpreted while C++ is compiled. ODQL statements can be embedded within C++ but not Java. Such differences make interaction and cross-binding among various system parts (implemented using different languages) a challenging problem.

5.1 ODQL-C++ Layer

In order to achieve a balance between flexibility and performance, the static, computation intensive behaviour has been implemented using C++ (which is compiled to native code) while any dynamic behaviour (e.g. dynamic introduction of new classes, etc.) has been implemented using ODQL (which is interpreted) (cf. fig. 3). This approach offers an optimal point on the compiled-interpreted continuum [8] (with fully interpreted systems at one end and fully compiled systems on the other). ODQL statements embedded within the compiled C++ code and Jasmine-provided macros used within the ODQL code bind the C++ and ODQL code together (cf. fig. 3).

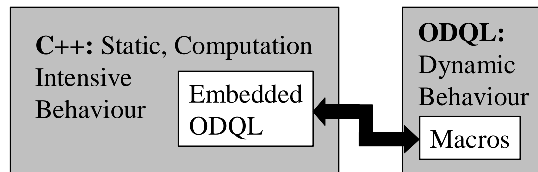

Fig. 3. Cross-binding C++ and ODQL in SADES

The interaction between the static and dynamic behaviour can be observed in various parts of the system. One example is dynamic type casting. SADES employs semantic relationships to connect the various entities within a system. Implementation of the relationship mechanism has been kept generic so that it is possible to connect a variety of system entities e.g. objects, classes, meta-classes. This is dictated by the requirement for the system to be extensible. The relationships can be defined, removed and modified dynamically. The generic nature of the relationship mechanism means that most relationship manipulation methods (written in ODQL due to the dynamic nature of relationships) are not always aware (and do not need to be aware) of the real type of the object being operated upon. As a result most return values and parameters are cast to the class type at the root of the ODQL class hierarchy. Some methods, however, do require that an object (which was passed to the method as an object of root type) be cast to its actual type (the lowest type in the class hierarchy to which the object can belong). The problem is compounded by the fact that this type

cannot be known at the time of writing the method code due to the evolving nature of classes and bindings between objects and classes in the system. The type needs to be discovered dynamically and the object needs to be cast to the correct type. The mechanism to achieve this is shown in fig. 4 and demonstrates the interaction between compiled and interpreted code.

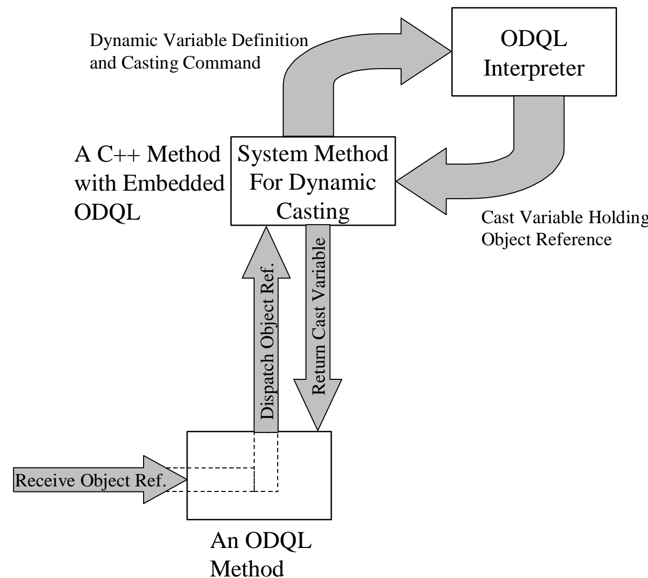

Fig. 4. Dynamic type casting in the SADES ODQL-C++ layer

When an ODQL method needs to dynamically type cast an object to its actual type it dispatches the object reference to a system method written in C++ with embedded ODQL. The compiled code in this method, with the aid of the embedded ODQL statements, discovers the type of the method (through ODQL's reflective capabilities) and sends a command to the ODQL interpreter to define a variable of the correct type and assign the object reference to it with the correct type cast. ODQL interpreter is used because this is the only means for dynamically defining variables in Jasmine. Note that this results in the variable being defined for the ODQL interpreter instance associated with the current process. As a result the C++ code keeps track of the variable names used within the process so that duplicate names are avoided and recycles the variable names once the process and its associated ODQL interpreter instance are terminated. The correctly type cast variable holding the object reference is returned to the ODQL method.

5.2 Java Layer

The Java layer provides the SADES server and the client API. As shown in fig. 5 the SADES server is an RMI server which interacts with the ODQL-C++ layer through an implementation of the Java Naming and Directory Interface (JNDI) in Jasmine J-

API, one of the several Jasmine Java APIs. The client API provides wrappers around remote method calls to simplify the programming interface for developers not familiar with RMI [12].

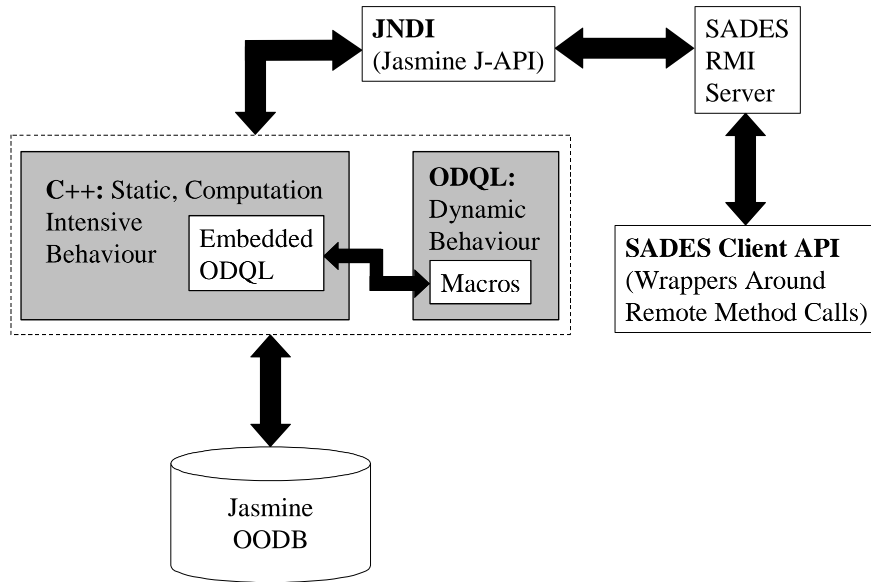

Fig. 5. SADES Java layer linked with the C++, ODQL layer

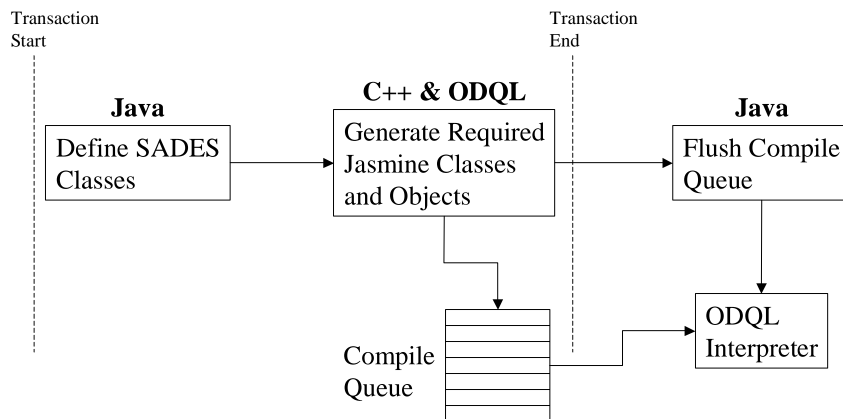

Fig. 6. Dynamic compilation in SADES

The interaction between the Java layer and ODQL-C++ layer can be observed in various parts of the system. One example is dynamic compilation. Since classes and their methods can be dynamically introduced, removed or modified they also need to be dynamically compiled or recompiled. The ODQL interpreter (which also invokes the C++ compiler for native code compilation) cannot be invoked from within a

transaction. Classes and methods, on the other hand, must be introduced, removed or modified within transaction boundaries as they are persistent entities; classes and objects in SADES are mapped on to Jasmine classes and objects for storage purposes. The solution employed in SADES is shown in fig. 6 and shows the interaction between the Java layer and the ODQL-C++ layer. The classes are defined in the Java layer and passed on to the ODQL-C++ layer which generates the required Jasmine classes and objects. Any affected classes (new or old) are placed on a compilation queue before the transaction ends. Once the transaction is complete the Java layer invokes the ODQL interpreter with the contents of the compilation queue as parameters.

6 Conclusions and Future Work

This paper has summarised the implementation of a rulebase and a hybrid aspect-orientation mechanism involving a declarative aspect language and a weaver supporting persistent aspects within a highly object-oriented environment. Cross-binding and interaction between three different OO languages have also been discussed. The discussion has demonstrated that a multi-paradigm implementation is not only influenced by the system requirements and design but also by the constraints imposed by the implementation environment. In case of SADES the performance requirements for the system dictated that rules should not be implemented as first-class objects. On the other hand, a specific aspect language and weaver had to be developed as existing implementation tools were either inadequate or unsuitable. The ODQL-C++ layer employed a combination of compiled-interpreted behaviour to strike a balance between the performance and flexibility requirements. The Java layer, however, needed a specific dynamic compilation mechanism due to the transaction constraints imposed by Jasmine.
The work in the future will draw upon these experiences to develop guidelines for multi-paradigm implementation and language cross-binding based on both system requirements and implementation constraints.

References

1. Aksit, M. and Tekinerdogan, B. *Aspect-Oriented Programming using Composition Filters*. *ECOOP '98 AOP Workshop*, 1998:
2. Xerox PARC, USA, *AspectJ Home Page*, http://aspectj.org/
3. *The Jasmine Documentation*. 1996-1998 ed. 1996: Computer Associates International, Inc. & Fujitsu Limited.
4. Katz, R.H., *Toward a Unified Framework for Version Modeling in Engineering Databases.* ACM Computing Surveys, 1990, **22**(4): p. 375-408.
5. Kiczales, G., et al., *The Art of the Metaobject Protocol*. 1991: MIT Press.
6. Kiczales, G., et al. *Aspect-Oriented Programming*. *ECOOP*, 1997: Springer-Verlag, Lecture Notes in Computer Science 1241:

7. Monk, S. and Sommerville, I., *Schema Evolution in OODBs Using Class Versioning.* ACM SIGMOD Record, 1993, **22**(3): p. 16-22.
8. Parsons, D., et al. *A 'Framework' for Object Oriented Frameworks Design*. *Technology of Object Oriented Languages and Systems (TOOLS Europe)*, 1999: IEEE Computer Society Press: 141-151
9. Paton, N.W., *Supporting Production Rules Using ECA Rules in an Object-Oriented Context.* Information and Software Technology, 1995, **37**(12): p. 691-699.
10. Rashid, A., *A Database Evolution Approach for Object-Oriented Databases*. PhD Thesis, Computing Department, Lancaster University, UK, 2000
11. Rashid, A. and Sawyer, P., *Object Database Evolution using Separation of Concerns.* ACM SIGMOD Record, 2000, **29**(4): p. 26-33.
12. Rashid, A. and Sawyer, P., *Transparent Dynamic Database Evolution from Java.* L' Object, 2000, **6**(3): p. 373-386.
13. Rashid, A., Sawyer, P., and Pulvermueller, E. *A Flexible Approach for Instance Adaptation during Class Versioning*. *ECOOP 2000 Symposium on Objects and Databases*, 2000: Springer-Verlag, Lecture Notes in Computer Science 1944: 101-113
14. Skarra, A.H. and Zdonik, S.B. *The Management of Changing Types in an Object-Oriented Database*. *1st OOPSLA Conference*, 1986: 483-495