

An Aspect-Oriented Framework for Schema Evolution in Object-Oriented Databases¹

Robin Green
Computing Dept.,
Lancaster University
United Kingdom, LA1 4YR
+44 01524 593541
r.d.green@lancaster.ac.uk

Awais Rashid
Computing Dept.,
Lancaster University
United Kingdom, LA1 4YR
+44 01524 592647
marash@comp.lancs.ac.uk

ABSTRACT

Persistent objects in an object database need to be adapted, either by physical conversion or wrapping, when the schema is changed to fix bugs or meet new requirements. Object database schema evolution introduces a number of concerns into the system, such as adaptation rules, the choice between conversion or wrapping, and backward compatibility. Our research aims to allow strategies for addressing such concerns to be dynamically replaced or altered for an existing, running database. An early prototype evolution framework has been developed as an interpreter for a custom object-oriented language, written in AspectJ. This position paper discusses some areas where aspects have been used to separate concerns, and suggests other concerns in the framework which are likely to benefit from an aspect-oriented approach. The concerns discussed include: selective lazy evaluation, contracts, metacrosscutting, and the maintenance of custom version-specific extents.

1. INTRODUCTION

This paper discusses the use of aspect-oriented programming in a prototype framework for schema evolution in object-oriented databases. Since the framework needs to be highly configurable, for reasons outlined below, and since some of the concerns involved are crosscutting, this problem domain is a clear candidate for the use of AOP. The framework has been partially implemented in AspectJ 1.0.3. This paper first introduces the problems of schema evolution in object databases, then discusses aspects currently implemented in our schema evolution framework, and finally concludes with an examination of some other concerns that will be investigated as candidates for AOP as the implementation progresses.

2. BACKGROUND

Just as with relational databases, the schema for an existing, populated object database is subject to modification to fix mistakes or meet new requirements. Two key issues can be identified in schema evolution:

- Existing objects need to be adapted in some way to conform to the new schema, so that they have the expected fields and methods. This can either be performed:
 - by the use of *transparent view wrappers*, which act as if they were instances of the corresponding class from the new schema;

- or by physically converting the object into an instance of the new class [8], which entails dynamic reclassification.

- It may be necessary for old applications to continue to access the database as if it still conformed to an older schema – that is, backward compatibility may be required.

If the schema evolution support in a particular OODBMS is not flexible enough for a desired change, the developer is forced to perform a complete “dump and reload”. This entails copying all the data in the database to an intermediate location, then recreating the database with the new schema, and finally copying all the data back into the new database, making any structural and data modifications as necessary. This is a time-consuming and ad-hoc process for the developer, and could be very wasteful in terms of time and disk space – rendering it unacceptable for some systems.

However, even if the schema evolution facilities of the OODBMS allow the schema to be modified in the desired manner, they may require the database to be taken offline while a full database conversion to the new schema takes place. Alternatively, they may allow the system to stay running, but not allow the new schema to be used until a complete background conversion of the database has taken place. On the other hand, if objects are lazily converted, this could impose an undesirable performance degradation on large database operations. Thus, it is arguable that for some decisions about schema evolution approaches – such as the decision as to whether to use immediate conversion, lazy conversion, a hybrid approach, or simulating conversion with views – no single approach serves the requirements of all database applications in a satisfactory manner.

Another example of such a decision is whether to store multiple versions of a schema in the database – and if so, whether to store only the differences between schema versions (at some granularity) or whether to store each schema version in full.

Moreover, the most suitable evolution approaches to use for a given application may *themselves* change, as and when the application scales up or has to deal with new requirements [10].

Our research therefore involves constructing a schema evolution framework for object-oriented databases which is flexible enough to allow, not only different approaches to schema evolution to be configured, but also the approaches in use to be changed for an existing database (in some cases, at runtime). In order to make the framework easier to understand, configure, and extend, it is desirable to separate out these implementation decisions from each other and from the main bodies of the OODBMS and the runtime environment. This work is grounded in our earlier work

¹ This work is supported by UK Engineering and Physical Sciences Research Council Grant GR/R08612.

on building customisable evolution approaches for object-oriented databases using AOP techniques [10] [11] [12].

Because our framework requires a versioned type system at the application programming level for evolution purposes, and since such a type system is not available in mature object-oriented languages such as Java², we chose to implement a new language. This has the benefit of greater flexibility for implementing features such as version conversion. The new language is called *Vejal* and – although it is XML-based for convenience reasons – borrows significantly from Java, AspectJ and Eiffel [7]. Applications – and application-specific aspects to convert between different versions of schemas – are written in *Vejal*. The framework itself, including generic schema evolution strategies, is currently implemented as a *Vejal* interpreter, written in AspectJ (considered here to be a superset of the Java language).

3. SELECTIVE LAZY EVALUATION

Lazy evaluation is a technique from functional programming, in which expressions are only evaluated when their values are required, and compound values such as lists may be evaluated gradually as needed. In a “pure” functional language (i.e. a fully referentially-transparent language) such as Haskell [4], the interpreter or compiler can transparently use lazy evaluation for any expression. However, in an imperative, object-oriented language such as Java, it would clearly be unsafe to lazily evaluate *arbitrary* expressions, since expressions could have undesired side effects if executed out-of-order.

Selective lazy evaluation may be defined as the process of deferring the evaluation of a particular programmer-specified expression until its value is needed. In our current framework, such a deferral is necessary or useful for two key purposes:

- i) *Kind resolution*: *Vejal* types and classes (collectively known as *kinds*) are by default stored persistently in an *unresolved* form, which means that:
 - kind references within them are unversioned, and
 - parameterised kinds are stored as *templates*, rather than reduced to a collection of unparameterised kinds by parameter substitution.

However, for performance reasons, it is essential to resolve every kind at or before the time that it is first used by the interpreter – otherwise the kind would have to be re-resolved every time it was used, which would be disastrous in loops. Moreover, rather than reading in and resolving an entire kind graph at once, it is more efficient to only resolve kinds on demand – similarly to the way in which Java virtual machines typically only load classes as needed. This is a less obvious form of lazy evaluation, which can bring significant benefits in terms of faster restart times for systems in development or systems being upgraded.

Each *Vejal* database application is bound to a specific schema version which specifies precisely which class versions to use for that application. Thus, the behaviour of an application will be unaffected by whether kinds are resolved early or late.

- i) *Vejal object resolution*: For implementation reasons, inside the *Vejal* interpreter, Java representations of *Vejal* objects are
-
- 2 Nor are versioned types - to our knowledge - available in *any* other existing programming language. Explicit versioning of types is distinct from, and more powerful than, versioned assemblies in C#.

typically cloned upon being read from disk. Again, it should not be necessary to read in and perform a deep clone of an entire *Vejal* object graph in order to access just one object.

Design patterns for selective lazy evaluation in imperative OO languages already exist (e.g. [9] and Virtual Proxy in [5]). However, in our evolution framework we have employed an aspect-oriented approach which we call *encapsulated reassignment* [3]. A sufficiently broad wildcarded pointcut designator is used to track, at runtime, all fields that the proxy object (also known as a *thunk*) is assigned to, as shown in this example:

```
aspect SpecificTracker {
    /* Assume the field's type will be SysTypeRef or some
    subtype thereof. */
    after (ReassigningTypeRef ref, Object parent):
    set (SysTypeRef+ Object+.* )
    && args (ref) // Right hand side of assignment
    && target (parent)
        // Object that the field being assigned to belongs to
    && !within (SpecificTracker)
        // exclude code within this aspect
    { ... }
    ...
}
```

The *thunk* implements all the methods that the type of the expression specifies, and forwards all appropriate messages to the actual evaluated object. However, when a message is sent to the *thunk* which requires it to evaluate the corresponding deferred expression, after evaluation all known references to the *thunk* are replaced with references to the evaluated object. This means that future access to the redirected references will be more efficient, since there will be no need for a double dispatch, or a check to see whether the deferred expression has been evaluated yet. Other references, such as local variables (which are not trackable in AspectJ 1.0.3 pointcuts), or fields not addressed by the set-tracking pointcut mentioned above, will still point to the *thunk*, but messages will be forwarded to the evaluated object. A more detailed description of the encapsulated reassignment approach is given in [3].

4. VERSIONING MODES AND DYNAMIC ASPECTS

One of the dimensions of configuration supported by our schema evolution framework is the versioning mode axis, which currently consists of a one-version mode, an N-version mode, and a mode to transition between them. The one-version mode is predicated on the assumption that only one schema version exists in the database, which allows a number of optimisations to be enabled. However, for any schema evolution to take place, in the current prototype the N-version mode must be entered, because schema evolution requires the existence of an old schema version and a new schema version. Hybrid modes are also planned.

The versioning modes are implemented as an aspect hierarchy, inheriting from the abstract aspect *VersioningMode* which

contains some shared functionality. However, the bulk of the functionality in all of the versioning mode aspects is currently located in ordinary methods, rather than advice. This is because the methods involved, such as `createClassRef` and `typeCheckAll`, are invoked by callers for which their functionality is central, rather than a peripheral concern. It would be unnecessarily complex and would serve no real purpose to create artificial join points to allow the direction of invocation of these methods to be reversed by AspectJ with advices. Adopters of AOP should carefully consider whether a configurable concern really benefits from being implemented with advices rather than methods.

However, there are a few advices which are part of versioning modes, such as a “postLookup” advice which ensures that *persistent root* objects read from the database are adapted as necessary to the current schema version in use (other objects are handled by the lazy object cloning mechanism). “postLookup” is a good example of an advice which is not *by itself* crosscutting, since it only advises one method, but which still usefully separates a peripheral and *configuration-specific* concern from the core functionality of – in this case – a lookup method. However, the “postLookup” advice forms part of an aspect addressing a crosscutting concern, so it certainly qualifies as aspect-oriented programming.

Versioning-mode-specific advice always begins with a check that the versioning mode aspect to which the advice belongs is in fact enabled. This is in effect a *metacrosscutting concern* – a concern which crosscuts all the advices in an aspect. Basic metacrosscutting facilities are provided in AspectJ 1.0.3 with clauses such as **perthis** and **percflow** which can be applied to entire aspects, and which are implicitly ANDed to the pointcut designators of every advice in that aspect. However, none of these clauses strongly facilitate programmatic disabling and re-enabling of aspects – which is a crucial concern for implementing “dynamic aspects”. The current alternatives are either to scatter redundant **if** statements through the advice, or to turn each advice into a stub “trampoline” into an individual aspect method, and then advise all such aspect methods using a wildcarded pointcut.

A more convenient way to enable and disable aspects would be to have an optional **when** clause in the aspect header, specifying a boolean condition that has to hold for the advice to be activated – similar to the **if** PCD, but applying to the whole aspect. (Advice that should run irrespective of whether the **when** condition is satisfied, such as system initialisation advice, could simply be moved into a static inner aspect or a separate privileged aspect.)

This would deal with one particular class of metacrosscutting concerns – enabling and disabling dynamic aspects. Other metacrosscutting concerns – such as synchronizing every advice in an aspect – might be dealt with by introducing a new primitive PCD for advice execution. It would be strictly speaking unnecessary to have a hierarchy of aspects, meta-aspects, meta-meta-aspects etc., because aspects can already operate on themselves. However, it might nevertheless be a better separation of concerns to separate base advice from meta-advice in this way.

5. CONTRACTS AS ASPECTS

In a complex software system, such as a highly configurable schema evolution framework, it is helpful to make the *intent* of code clear by abstracting away unnecessary details, and this is one of the key goals of AOP. Clarifying the intent of code and division of responsibilities in a system also supports reliability – which is very important for a piece of core system infrastructure such as a

database evolution framework. A complementary approach to the same age-old intent problem is Design by Contract (DbC) [7], in which the behaviour of a class is semi-formally specified with preconditions and postconditions for methods and constructors, and a class invariant. [6] uses aspects to separate out runtime checks for preconditions, postconditions and invariants from a class, so that they can be selectively or fully disabled for performance reasons. However, there are other reasons for using aspects here. Firstly, there are simplicity and safety advantages compared to e.g. using **try...catch...finally** to implement reliable postconditions. The second reason, strict substitutability, points towards more rigorous guidelines for using AspectJ to check contracts at runtime.

Applied consistently, Design by Contract implies that a class should always be *strictly* substitutable wherever it is type-substitutable at all – in other words, if a `Person` variable can hold either an `Employee` or a `Customer` object, then both the `Employee` class and the `Customer` class should conform to the `Person` contract, as well as their own contracts. (It should be noted that this strict substitutability view of inheritance can cause problems with other uses of inheritance which are arguably still quite valid [13]; however, these problems are beyond the scope of this paper, and are touched on to some extent in [3].)

We first assume that contracts, apart from their invariants, apply to methods irrespective of whether they are called from the same class or not. (In practice, contract-checking would sometimes have to be excluded in cases where a method was called from the contract-checking aspect, in order to avoid indefinite recursion, but we ignore this here for the sake of simplicity.) Strict substitutability then implies that, for a method `m` on a type `T` with argument types `{A1, A2, ...}` and return type `R`:

- i) The postcondition check should normally be implemented as an advice approximately equivalent to the following (context-yielding pointcut designators such as **this** and **args** may of course be added):

```
after () returning: execution (R T+.m (A1, A2, ...))
{ ... }
```

returning must be used because postconditions are not required to hold when a method exits abnormally – and it would be extremely misleading, not to mention incorrect, to ignore exceptions thrown by the method and throw a “postcondition check failed” instead!

`T+`, indicating “`T` and all its subtypes”, is used because the contract of a method on a type should apply to all its subtypes. The use of `T+` ensures that erroneous code will be caught if and when it breaks the strict substitutability principle at runtime (assuming that the postconditions being checked are sufficiently detailed). Additionally, consistent use of this idiom allows postcondition checking to be implemented incrementally, because all the supertype postconditions, if any, will always be checked before a method returns control to its caller. As postconditions in Design by Contract should always be side-effect-free (though AspectJ cannot guarantee this), the order of checking should be irrelevant.

However, it is generally important for postcondition-checking advice *not* to assume the corresponding precondition. This is because strict substitutability allows preconditions to be strictly weakened in subtypes. So, for example, if a method with argument `x` has a precondition `x>=0 && x<array.length`, then strictly speaking an unsafe postcondition check such as

`array[x]!=null` should be replaced with the safe equivalent `x>=0 && x<array.length && array[x]!=null`. (In some cases, however, adhering to this rule would be too pedantic because of the very low likelihood of the precondition being weakened by a subclass.)

Arguably, it would be incorrect to simply substitute **call** for **execution** in the above advice, without any added restrictions. Suppose that `T` has a supertype `S` which declares a method with the same signature as `m`, but with a strictly weaker postcondition. Then the advice above with **call** substituted for **execution** would *not* be activated for code such as:

```
S var = new T ();
var.m (...);
```

since `S`, the declared type of `var`, is not a subtype of `T`.

It could be argued that this is not strictly speaking a failure to check a postcondition, but is rather a type error in the client code. If the client code wanted to guarantee that the postcondition of `T.m` would be fulfilled, it should have declared **var** to be of type `T`, or cast it to type `T`. However, this is not the case, for two reasons:

- Perhaps client code should not in general assume that a non-null value of an expression statically-typed to `S` will necessarily adhere to the contract of `T`; perhaps instead it should make that assumption explicit with a cast. However, the developer is entitled to rely upon a subtly different assumption at *all* times: namely, the universal conditional that *if* an object is of type `T`, then it will adhere to the contract of `T`.
 - Similarly, if the class `T` fails to adhere to its contract at any time, that is unequivocally a bug, and should be detected by a postcondition check if such checking is enabled – regardless of in what manner the method was invoked. In particular, in AspectJ 1.0.3, the **execution** PCD (pointcut designator) matches method executions even when they are invoked by code outside the compilation unit, including `java.lang.reflect.Method.invoke`, unlike the **call** PCD.
- ii) The precondition check for `m` should normally be implemented as something similar to:

```
before (): call (R (T || T1 || T2 ||...) .m (A1, A2, ...)) { ... }
```

where `{T1, T2, ...}` are optional and are all those types, if any, which have *identical* preconditions for that method signature. It is *not* in general appropriate to use the unrestricted form `T+`. This is because in general subclasses should be allowed to make preconditions strictly weaker for methods which override or implement other methods, and such an unrestricted advice in effect states that subclasses will not do so. Furthermore, for a similar reason, it is *essential* not to use `T+` here if third parties without access to the source code might subclass `T` in future, since it is difficult to override a call advice in AspectJ 1.0.3 without also overriding the destination of the call.

We also employ the assumption that when a message is sent to the value of an expression statically-typed to `T`, the relevant precondition in `T` should always be adhered to, irrespective of the runtime type of the value. The rationale for this assumption that precondition selection should depend on the static type of an expression is almost a mirror-image of the

argument above that postcondition selection should depend on the *runtime* type of an object. In both cases, the conclusion is that the strictest relevant condition should be checked. For the precondition, that suggests using a **call** PCD, in most cases. Exceptions to this principle would be cases where a **call** PCD would not capture all calls of interest – either for implementation reasons, or because it is desired to check **super** calls, which **call** does not match in AspectJ.

6. FUTURE WORK

6.1 Dynamic Reassignment

The encapsulated reassignment approach can be seen as a special case of *dynamic reassignment* – tracking all references to an object and then switching them all to point to a different object at the same time. This is not a new idea, since it is supported by the **become** primitive in Smalltalk. However, aspect-orientation now allows adding this feature (or at least an approximation of it) to a language with no native “become” primitive or similar.

Dynamic reassignment could be useful for purposes other than lazy evaluation, such as simulating dynamic reclassification in languages which do not directly support it. (Again, this is one of the uses of the **become** primitive in Smalltalk – it can be used to extend an object with a new instance variable.) Dynamic reclassification can be (crudely) simulated with explicit proxy objects, but in some cases it might be more efficient to dispense with proxies and point directly to “real” objects, while using the dynamic reassignment approach to reclassify objects. For this to work, however, it would be essential – not merely useful as in encapsulated reassignment – for the aspect language involved to support pointcut designators referring to local variables and parameters.

However, this approach to dynamic reclassification would still be vulnerable to some of the criticisms levelled at the proxy approach, such as the well-known object identity problem: reclassification produces not the same object, as desired, but a different one – which is detectable with methods such as `java.lang.System.identityHashCode()`.

6.2 Version-specific Extents

Extents are simply collections of all the persistent instances of a given class in a database. They make it easy to run SQL-like queries such as “Select * from Employees”. However, the object data standard ODMG 3.0 [1] does not fully define, nor require, extents. Also, some OODBMSs (e.g. Ozone) do not have any explicit support for extents.

For the purpose of physically converting all persistent objects that currently belong to an older schema into instances of a corresponding class in a new schema, it would be useful to have extents specific to particular class versions to speed up the process of finding the objects that still need to be converted. However, this performance gain needs to be balanced against the time and space costs of maintaining version-specific extents for the rest of the time.

Our preliminary investigations suggest that implementing version-specific extents in our current framework would involve a high degree of crosscutting code which could usefully be localised using aspects. As well as standard extent maintenance tasks such as deleting an object from its extent when the `Database.delete` method is called on it, and tracking which objects have been

added to and removed from³ the database at the end of each transaction, there is also the need to move objects between extents when they are dynamically reclassified. Typically this would be converting between class versions, but this could possibly be extended to arbitrary reclassification.

6.3 Version Conversion Aspects

In Vejal we are planning to allow the programmer to specify arbitrarily complex transformations between class versions, in the form of *version conversion aspects*. These are essentially transparent view wrappers, written by the application programmer to present e.g. a Person[1] as a Person[2], which are invoked by the runtime environment automatically whenever an object needs to be adapted to a different class version. Crucially, they work by transforming data at the field level, and do not attempt to emulate methods (and nor do they require the application-specific evolution code to emulate methods) – the “real” methods are always used from the Vejal class version required by the application. Although this means a version conversion aspect breaks the encapsulation of the destination class version, this is arguably a good trade-off, because the alternative of emulating method behaviour leaves more room for error, and converting an object between class versions often requires knowledge of implementation details. There are no language restrictions on changes that can be made between one class version and the next – in particular, methods can be added, deleted and rewritten.

Vejal version conversion aspects are intended to support either views or conversions with exactly the same aspect. Thus, the real adaptation approach in use is abstracted out. If the system is configured to use views for a particular class, the version conversion aspect will just be used as-is; if not, the runtime environment will “scan through” the aspect to physically convert the object to the new class. In either case, “hidden fields” will be used if required, to store data from previous schemas that is invisible now but may become visible upon another adaptation [8]. Thus no data is lost due to destructive conversions – unless a previous schema is itself deleted.

In this way, version conversion aspects can be specified once for each pair of source and destination class versions, independently of whether a view technique or a physical conversion technique is being used to adapt objects.

Version conversion aspects arguably meet both criteria set out in [2] for a technique to be aspect-oriented: quantification and obliviousness. However, this is not the only candidate definition of AOP – and there exist systems such as metaobject protocols which effectively offer quantification and obliviousness, but are not necessarily aspect-oriented. Also, the planned join point model for version conversion aspects is currently much simpler than that of AspectJ's: simply matching on any Vejal objects read from the database which need adapting for the current schema, and belong to particular specified class versions.

However, one way in which more powerful join point models might be useful for version conversion aspects is to select different conversions depending on the aggregation context. For example, in a schema evolution operation on an engineering

database, one might wish to specify that a Pipe object should be converted to an ActivePipe object if it represents a pipe that is currently part of a physical structure, or a StockPipe if it is just a spare part. A context PCD for version conversion aspects would offer an alternative to scattering if statements around the conversion aspects for the relevant parent classes. The interpreter, compiler and/or runtime environment would be responsible for validating the type-safety of conversions.

REFERENCES

- [1] Cattell, R.G.G., ed. *The Object Data Standard: ODMG 3.0*. Morgan Kaufman, 1999.
- [2] Filman, R.E. and Friedman, D.P. “Aspect-Oriented Programming is Quantification and Obliviousness”. *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, Minneapolis, 2000.
- [3] Green, R.D. and Rashid, A. “Aspect-Oriented Selective Lazy Evaluation.” Submitted to *IFIP Working Conference on Generic Programming*; under review.
- [4] Jones, S.P. and Hughes, J., eds. *Haskell 98: A Non-strict, Purely Functional Language*. <http://haskell.org/definition/>
- [5] Larman, C. *Applying UML and Patterns*. PrenticeHall, 1998.
- [6] Lippert, M. and Lopez, C.V. “A Study on Exception Detection and Handling Using Aspect-Oriented Programming”. Xerox PARC Technical Report P9910229CSL-99-1, Dec. 1999.
- [7] Meyer, B. *Object-Oriented Software Construction, 2nd ed.* PrenticeHall, 1997.
- [8] Monk, S. *A Model for Schema Evolution in OO Database Systems*. PhD, Lancaster University, 1993.
- [9] Nguyen, D. and Wong, S. Design Patterns for Lazy Evaluation. *Proceedings of the 31st Technical Symposium on Computer Science Education (SIGCSE '00)*, ACM, pp.21-25. 2000.
- [10] Rashid, A., Sawyer, P. and Pulvermueller, E. “A Flexible Approach for Instance Adaptation during Class Versioning”. *ECOOP 2000 Symposium on Objects and Databases*, pp.101-113, Springer-Verlag LNCS 1944, 2000.
- [11] Rashid, A. and Sawyer, P. “Aspect-Oriented and Database Systems: An Effective Customisation Approach”. *IEE Proceedings - Software*, **148**(5), pp.156-164, IEE 2001.
- [12] Rashid, A. “A Hybrid Approach to Separation of Concerns: The Story of SADES.” *3rd International Conference on Meta-Level Architectures and Separation of Concerns*, pp.231-249, Springer-Verlag LNCS 2192, 2001.
- [13] Taivalsaari, A. “On the Notion of Inheritance”. *ACM Computing Surveys* **28**(3), 1996.

3 If an extent uses ordinary references, it is impossible for an object in that extent to become no longer reachable (except by an explicit delete invocation). However, extents may instead use weak references, which do not prevent the garbage collection of the objects they point to.