VEJAL: An Aspect Language for Versioned Type Evolution in Object Databases

Awais Rashid, Nicholas Leidenfrost
Computing Department
Infolab21, Lancaster University
Lancaster LA1 4WA
+44-1524-510316
awais@comp.lancs.ac.uk

ABSTRACT

In this paper, we present our aspect language Vejal (Versioned Java Language) which superimposes support for versioned types and AOP onto Java. Vejal has been developed to service the needs of AspOEv[1], an evolution framework that supports dynamic adaptation of evolution strategies in object databases [8, 9]. In this paper, we describe how the framework benefits from the advanced type semantics and the capacity for dynamic manipulation of Java offered by Vejal and its interpreter. We discuss how these capabilities allow programmers to alter types dynamically and undertake reflective analysis and consequent reflective action to preserve structural and behavioural consistency. We also highlight the dynamic AOP capabilities of Vejal and its command-line environment (CLE) which allows programmers to define expressive, single-use evolution primitives.

Keywords

Aspect-oriented software development, aspect-oriented programming, object database evolution, schema evolution, instance adaptation, versioned types, aspect-oriented databases.

1. INTRODUCTION

Determining all requirements for the future use of a software system may be impossible to do during the design and implementation stages of the software life cycle. Therefore, despite modern software engineering practices and attempts to create extensible, reusable, and flexible software, maintenance continues to be the largest phase in the life of any application. As databases serve multiple applications, the extent of evolution in databases, as demonstrated in [10], is beyond that of any single system. Object databases in particular are subject to advanced evolution needs due to the inherent structure of their data. True to the OO paradigm, persistent objects in an object database define their own behaviour, in contrast to a relational entity, which merely provides raw data for external applications to interpret. Therefore evolutionary changes can have complex implications for the behaviour of both existing instances and applications. Furthermore, databases commonly serve as a central storage point for many distributed applications, each of which may evolve independently and have different requirements for the shared data. The use of persistent data in such a heterogeneous environment further compounds the effects of changes. Newer applications may wish to add functionality to a type[2], or may deem functionality which is used by older applications as unnecessary. Compatibility issues arise as applications must operate on persistent data created with diverging definitions of a type.

Furthermore, compound aggregation in object data means that types may be interdependent, and therefore, a change in one type may make another potentially unstable. Altering a type can, therefore, have non-localised effects. Hence evolution in object databases must account for the effect of a change on the entire organisation of types (the database *schema*) and their instances within the database.

The process of evolution in an object database raises a number of major concerns, two of the most prominent being *schema evolution* and *instance adaptation*. Schema evolution pertains to the management of active types within the database environment and their respective evolution, while instance adaptation pertains to the meaningful conversion of existing data from one version of a type to another version.

Object databases usually provide an integrated (fixed) approach to address schema evolution and instance adaptation concerns. This supplied approach, however, may not be the best fit for the application programmer, or the application base that will end up using the database. In addition, it may be impossible to preordain schema evolution and instance adaptation needs when choosing which object database system a project will use.

Our solution to this problem is AspOEv, an aspect-oriented framework which supports dynamic adaptability of schema evolution and instance adaptation strategies in object databases. In such a flexible environment, which permits the simultaneous existence of multiple versions of a type, applications must have the ability to distinguish between versions and allow interaction between instances of different versions. This requires special language features on two levels: first, a language must support the syntactic expression of an explicit version wherever a type is referenced, i.e., allowing clear indication of which version of the type is intended, and second, the type system governing the language must permit, to some extent, the interoperability of instances of different versions of the same type.

Our aspect language, Vejal, and its interpreter have been specifically developed to service the above needs – the framework derives two key benefits from them. Firstly, the advanced type semantics and the capacity for the dynamic manipulation of Java facilitate dynamic evolution of types and adaptation of evolution

[1] The framework can be downloaded from: http://www.comp.lancs.ac.uk/computing/aose/AspOEv.php

[2] In an object database altering the representation of a complex data entity equates to altering the data's *type definition* (henceforth referred to as *type*).

strategies within the framework. Secondly, because Vejal applications are interpreted, the framework is able to capture evolution related events that occur within their execution, including detecting and handling behavioural inconsistencies. Inconsistencies in the execution of Vejal applications arise within the domain of the interpreter, enabling the framework to handle them in a uniform fashion.

The rest of this paper is structured as follows. Section 2 introduces the requirements that the Vejal type system has to cater for. Section 3 introduces the reflective features of Vejal and their use to maintain structural and behaviour consistency upon evolution. Section 4 discusses how join points and advice are handled in Vejal and describes some of its dynamic AOP capabilities. Section 5 discusses how Vejal and its CLE facilitate specification of specialised, one-off ad hoc evolution primitives by the programmers. Section 6 concludes the paper.

2. VERSIONED TYPE REQUIREMENTS FOR THE VEJAL TYPE SYSTEM

Vejal yields many advantages and features that aid AspOEv's flexibility in supporting adaptable evolution strategies. Applications written in Vejal have the ability to specify an explicit version of a type. The grammar for Vejal attempts to emulate that of Java, however, Vejal allows the additional expression of a ‘<’ / ‘>’ delineated version number where Java would normally only expect a class name:

```
Class Student<1.0> extends Person<1.3> {...}
```

Vejal allows the explicit declaration of a version wherever a type may be referenced, including type definitions, variable declarations, method return types, and method parameter types:

```
Person<1.3> frank = new Person<1.3>(...);
```

Applications also have the option to omit the explicit declaration of a version in type declarations:

```
Person frank = new Person(...);
```

If the application does not specify which version a variable should be treated as, the interpreter binds the variable to the most recent version. Object database schema evolution strategies in AspOEv can uniformly advise various nodes within the interpreter's abstract syntax tree (e.g., variable declaration, constructor invocation) to override this functionality, if desired

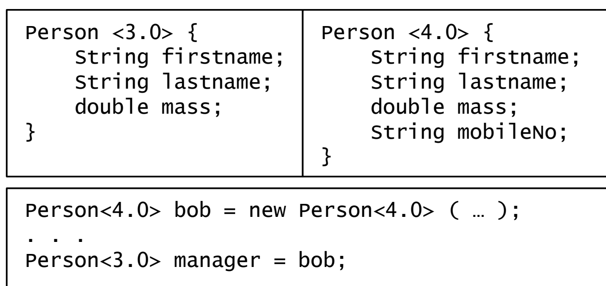

Fig. 1: Assignment of an additive version

In addition to the mere expression of versioned-type semantics, the Vejal type system also allows versions of the same type to be assigned interchangeably, hence providing a database evolution environment that is version polymorphic [9]. Static type safety guarantees are traded for flexibility as most type checking is deferred to runtime, allowing custom strategies to more explicitly define the semantics of type equality. For example, an approach may allow the unconditional assignment of two different versions of the same type if the type of the right-hand value is additive with respect to the type of the left-hand variable. Such an assignment could occur in a class versioning scenario, as shown in Fig. 1, or could happen as the result of a query on the database returning instances of various versions. Regardless of the cause, however, because the properties and operations defined by the right-hand type are a superset of those of the left-hand type, it can be safely assumed that the declared variable will not break behavioural consistency at some later stage.

3. VEJAL META-DATA

Meta data refers generally to objects which represent elements of the executing program itself, allowing the program to take action to alter its behaviour dynamically. Meta objects are the first class representations of program structures in OOP – meta classes model class definitions, meta fields and meta methods model the declarations of properties and operations respectively within meta classes, etc. Meta classes typically maintain inheritance information, as well as any meta fields and meta methods defined by the represented class. Likewise, meta fields typically store the modifiers, type, and name of the field, and meta methods the method signature, including modifiers, return type, name, and parameter types.

Like many schema evolution approaches able to achieve evolution dynamically, e.g., [1, 7], AspOEv relies heavily on a meta-layer to represent the structure of the persistent data. The relationships between versions of a type adds an extra dimension to Vejal meta classes – in addition to storing a type's inheritance graph (which is directed, acyclic), meta classes also maintain links to derived and base versions. This is important as instance adaptation mechanisms often use the derivation path between two versions to determine how conversion should occur upon database evolution. To ensure stability among changing type definitions, the AspOEv framework stores meta-data persistently. The Vejal interpreter loads all type definitions from the database at start-up, preventing duplicate definitions of a type. This ensures that instances cannot have their original representations changed; once a type definition is finalised, it cannot change.

In addition to providing a cohesive view of the types in the schema, Vejal's meta-classes and their relationships allow database programmers to alter types (and hence, the schema) dynamically. Additionally, execution-level meta-data, i.e., a meta representation of procedural (method-body) code, enables the reflective analysis of the impact of a change, and consequent reflective action to preserve consistency.

As any change to a type will likely affect all of its subtypes, (with the exception of the alteration of a member which the subtype overrides) the derivation of a new version of a type must also implicitly create new versions of every type which inherits, directly or indirectly, from the type. Bidirectional inheritance relationships between a type and its super-type enable Vejal meta-classes to automatically create these implicit versions.

Meta-classes can determine type dependencies by analysing the types of instance and local variable declarations (including method parameters) and the return types of method invocations. Moreover, meta-classes can analyse the use of variables, i.e., the fields referenced and methods invoked on variables, to determine compatibility with versions of other types.

The representation of Vejal programs enables the framework to query the parsed code for program structures which are directly affected by evolution. For example, an evolutionary change which removes an instance variable from a type invalidates all references, both internal and external, to that instance variable. The meta-representation of Vejal operations allows querying the code for relevant references. Evolution primitives which operate on Vejal meta-data perform such consistency searches, and draw attention to circumstances resulting from the enacted change. This enables evolution approaches to take appropriate action. An evolution approach could implement a set of generative handlers to take autonomous action, or could simply notify the database programmer that further evolution need occur. Take, for example, the case of renaming an instance variable, as shown in Fig. 2.

Person <1.0> {	Person <2.0> {
String firstname;	String firstname;
String surname;	String lastname;
double weight;	double weight;
}	}

Fig. 2: A renamed field

Any operations defined by the type, or by dependent types, with references to the previous field handle *surname* must also evolve to use the new field name, *lastname*. As this evolutionary change does not have any subtle semantic implications, it is a perfect example of behavioural inconsistencies which can be detected and handled automatically.

Execution-level meta-data in the evolved type (Person<2.0>) are searched for field references to the deprecated field name and altered to refer instead to the new field name. External references to the altered field are detected by searching dependent types and applications. Matching field references are first queried by field name. Subsequently, the types of field references are verified by using the execution-level meta-data in a manner similar to a compiler to determine the static type of their target variables.

Note that although such an action presents a significant overhead, it only occurs once when a type is evolved, and is considerably less work than manually revising code.

3.1 Consistency using Reflective Handlers

To demonstrate the consistency management features of Vejal, we use the restructuring scenario from [2] and [4] which affects the direction of an aggregate relationship between two types, Supplier and Part. Initially, a Part contains a set of Suppliers. The proposed reorganisation of the data, however, reverses the nature of the relationship between the two entities – a part loses the knowledge of which suppliers carry it and a supplier gains a set of parts.

The nature of this evolution removes an instance variable, *suppliers* from a type, *Part*. This removal creates inconsistencies in any code which references the removed entity, which [2] proposes to handle via reflectively altering dependent code. With Vejal's ability to detect invalidated references in loaded types and applications, inconsistencies can be handled by reflectively introducing and initialising a local variable with the appropriate value prior to use.

Fig. 3(a) shows a simple method, defined within the type *Part,* which uses the removed field *suppliers*. Note that the removal of *suppliers* leaves the method (as well as any other dependent entities) in an inconsistent state as it contains a reference to a removed entity. Fig. 3(b) illustrates how such an inconsistency could be uniformly handled by using reflective generators [2]. The handler defined in Fig. 3(b) responds to a Removed Member Referenced Exception, which occurs as the result of consistency checking within evolution primitives. The exception carries information relevant to the inconsistency, e.g., the Vejal method in which it occurs, enabling handlers to take necessary action. The handler shown in Fig. 3(b) updates the inconsistent method by adding code which declares and initialises a local variable, *suppliers*, to the appropriate value. Line 8 of Fig. 3(b) retrieves the inconsistent Vejal method from the exception. Subsequently, lines 14, 15, and 16 declare the necessary Vejal handling code, parse it, and merge it with the inconsistent method, respectively. The Vejal method resulting from the application of the handler in Fig. 3(b) to the Vejal method in Fig. 3(a) is listed in Fig. 3(c).

```
void listSuppliers () {
  Iterator iter = suppliers.iterator();
  while (iter.hasNext())
    print(iter.next());
}
```

(a)

```
1  public class RemoveHandler extends ExceptionHandler {
2    public RemoveHandler (InterpreterException exception) {
3      super(exception);
4    }
5
6    public void handleException () {
7      RemovedMemberReferencedException rmre =
       (RemovedMemberReferencedException)exception;
8      MetaMethod method = rmre.getReferer();
9      MetaMember removed = rmre.getRemovedMember();
10     Type declaredIn = method.getDeclarer();
11     Type part = new Type("Part");
12     if (declaredIn.equals(part)) { // Handling for Part
13       if (removed.getName().equals("suppliers")) {
14         String handlerCode = "QueryEnumeration result =
Database.where([Supplier], \"hasPart(\" + partNo + \")\");  Set suppliers
= new HashSet(result.toCollection());";
15         Statement parsedHandlerCode =
              VejalClassLoader.parseStatement(handlerCode);
16           Block methodBody = Method.getMethodBody();
17           MethodBody.prependStatement(parsedHandlerCode);
18       }
19     }
20   }
21 }
```

(b)

```
void listSuppliers () {
  QueryEnumeration result= Database.where([Supplier], "hasPart(" +
           partNo +")");
  Set suppliers = new HashSet(result.toCollection());
  Iterator iter = suppliers.iterator();
  while (iter.hasNext())
    print(iter.next());
}
```

(c)

Fig. 3. (a) Code invalidated by the removal of instance variable *suppliers* **(b)** Handler declaration **(c)** Inconsistency correctly handled by introducing and initializing a local variable

Unfortunately, at present there is no way to declare handlers in an ad hoc manner, and all handlers as seen in Fig. 3(b) must be written and compiled in Java, and then associated with the schema evolution strategy prior to the correlated evolution. Notice the use of bracket ('[', ']') delineated type names in Figs. 3(b) and 3(c). This is a provision of the Vejal language which allows referencing types as first class entities. Note also that Figs. 3(a) and 3(c) are code listings in Vejal, whereas Fig. 3(b) is Java.

4. THE META-JOINPOINT/ADVICE MODEL

AspOEv utilises integrated aspects to achieve modularisation of evolution concerns: both join points and advice are first class objects within the framework. Points of interest are captured with meta join points (similar to AspectWerkz), known as *Bindings*, that are woven into the core OO portion of the framework using

AspectJ. At the application level, a Binding is simply an object representation of a join point and its associated advice. In some respects, a Binding is similar to an Event-Condition-Action (ECA) rule within active databases [3]. It must be observed, however, that Bindings and their associated advice exist outside of the database and are not rule-driven.

Fig. 4 illustrates how a Binding operates over a join point. Program control passes to the Binding when the join point is met, the Binding then executes *before* and *pre-proceed around* advice, then the join point itself (if not circumvented), followed by its *post-proceed* and *after* advice, and finally returns control and execution resumes. At runtime, a Binding can be viewed simply as a collection of *before*, *after*, and *around* advice surrounding a join point.

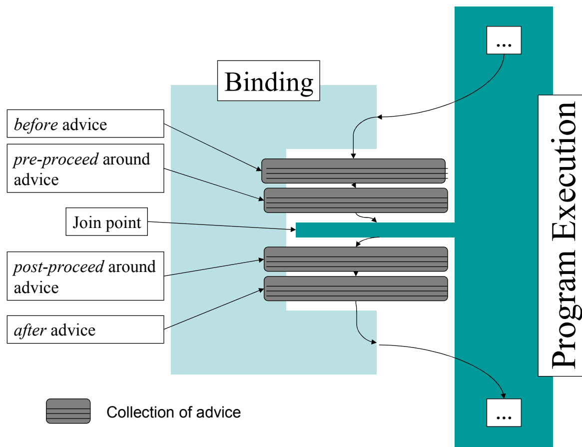

Fig. 4: Activation of a binding

When AspOEv is built, the framework uses declared Bindings to generate AspectJ code and create a seamless interface between AspectJ advice and the executing Vejal environment. Declaration of Bindings in AspOEv is nearly identical to the specification of activation of join points in [6]. Bindings are declared and added to a 'Binding Manager.' The Binding Manager creates an AspectJ aspect, to which each binding adds:

- a runtime declaration of the Binding itself;
- a declaration of the Binding's arguments;
- an AspectJ pointcut targeting the Binding's specified join point;
- an AspectJ *around* advice creating an interface with the executing Vejal environment.

AspectJ *around* advice generated for a binding is summarised in Fig. 5. Via manipulation of Vejal scopes, the AspectJ advice prepares the Vejal environment in such a way as to be both supportive of Vejal advice and safe for the executing Vejal code to which the Vejal advice applies. The AspectJ advice can then execute any Vejal advice associated with the binding.

Our *binding model* yields a number of advantages. Firstly, Bindings are accessible dynamically, allowing management of applied advice at runtime. Moreover, because all advice is written in Vejal, and Vejal is interpreted, advice can be added and removed dynamically, as well as making advice subject to the flexibility and safety of the interpreter.

The creation of a seamless interface between AspectJ advice and the Vejal environment allows the framework to apply interpreted advice to aspects, permitting the dynamic addition and removal of advice from specified binding points. The framework is thereby able to take a unified approach to evolution, viewing all evolution events – from changes incited by primitives, to widespread, long lasting evolution strategies – as aspects applied to Vejal meta-data and instances.

```
Object around ( arguments ) : bindingPointcut( arguments ) {
  - Push Vejal environment scope
  - Declare arguments as variables in Vejal environment
  - execute binding before advice
  - execute binding pre-proceed advice
  if ('proceed ( )' has been called in pre-proceed advice) {
     - pop and save Vejal environment scope
     Object result = proceed( arguments );
     - restore Vejal environment scope
     - store result of proceed call in Vejal 'proceed' variable
  }
  - execute binding post-proceed advice
  - execute binding after advice
  - Pop Vejal Environment Scope
  - return appropriate value
}
```

Fig. 5: Pseudo code of Binding-generated AspectJ advice

4.1 Dynamic AOP in Vejal

The ability to dynamically add and remove Vejal advice from bindings provides dynamic AOP capability to Vejal. However, currently, advice defined within types can only utilise variables which are declared statically within the execution environment, or associated with the Binding to which they subscribe – instance variables within the type are inaccessible as the advice is essentially static, and therefore, not associated with any individual instance of the type. Fig. 6(a) shows the declaration of a piece of advice within a class declaration.

```
class BindMonitor {
  before (bind) {
    bindCount = bindCount + 1;
  }

  before (unbind) {
    bindCount = bindCount - 1;
  }
}
```
(a)

```
after (someBinding) {
  // Some functionality
  if (someCondition) {
    thisBinding.removeRule(thisAdvice);
  }
}
```
(b)

Fig. 6. (a) Simple AO functionality within Vejal **(b)** Removal of advice from a binding

The Vejal class shown in Fig. 6(a), *BindMonitor*, merely keeps track of the number of objects bound within the database. Bindings *bind* and *unbind* refer to methods on the database manager of the same name. Note that the variable referenced within the advices to *bind* and *unbind*, *bindCount*, must be statically declared (not shown in Fig. 6(a)) before it can be used. While meta advice declared within a type does store a reference to the type in which it was declared, currently there is no mechanism for resolving which instance of the type advice code refers to – i.e. as in AspectJ's deployment model via *percflow*, *perinstance*, etc. – thus only statically declared variables are available for manipulation within advice.

Fig. 6(b) shows dynamic AOP functionality in Vejal by demonstrating how advice can dynamically unsubscribe itself from a Binding. The variable *thisBinding* refers to the currently executing Binding, while *thisAdvice* to the currently executing piece of advice.

5. THE EVOLUTION PRIMITIVE MODEL

Vejal allows AspOEv to provide database programmers the option of using expressive, single-use primitives. Many object database

systems supply the database programmer with a special language for executing predefined evolution primitives, e.g., Odberg's *Change Specification Language* [5]. These languages are focused, however, only on accepting input concerning the evolution primitive desired, as well as any parameters required by the primitive to carry out its particular change. This is problematic for a number of reasons:

- The programmer must learn another language.
- Often these languages are created for the sole purpose of expressing primitives which are predefined by the object database system, and thereby lack the capability to express custom evolution semantics.
- Extending the language (if possible) could potentially require modifying the language's parser. Due to the simplicity of most primitive expression languages, the parser is apt to be ad hoc.

As opposed to introducing yet another language into the framework, we allow database programmers to manipulate the schema through Vejal's Command Line Environment (CLE). The CLE allows schema changes to be expressed and interpreted in a manner consistent with evolution rules defined within the AspOEv framework. Vejal primitives can also utilise predefined framework primitives, and apply to multiple types, or in fact, the entire schema. As opposed to creating and managing potentially large collections of Java objects that represent obscure primitives that might only be used once, ad-hoc use of Vejal code provides the immediacy, convenience, and expressive ability to create powerful and diverse primitives.

Fig. 7 shows a primitive which adds the type *Object<1.0>* as a base type to any type which does not have any base types, cf. lines 7 and 8. Additionally, in lines 3 – 6, the primitive circumvents any collisions, which might occur as a result of the introduced base type, by renaming the field *uniqueID* in any type which defines it.

```
class Object<1.0> {
  double uniqueID;
}
```

```
1  edit (*) {
2    List supertypes = Type.getSuperTypes();
3    if (Type.declaresField("uniqueID")) {
4      String newName =
         Type.getTypeName() + "_uniqueID";
5      RenameFieldCommand(Type, "uniqueID", newName);
6    }
7    if (supertypes.cardinality() == 0) {
8      Type.addSuperType([Object<1.0>]);
9    }
10 }
```

Fig. 7: Specialised, single-use, ad hoc primitive

Note that the primitive in Fig. 7 is very specialised and would likely only need to be used once. Therefore, ad hoc declaration of the primitive in Vejal is preferable to creating a dedicated Java object – complex and specialised primitives written in Java would have to be written in a separate text editor, compiled, reflectively fetched and loaded, and then executed. Vejal code, on the other hand, can be more readily integrated, as the CLE provides AspOEv with a means of accepting and executing valid Vejal code.

By clarifying the semantics of a change, detailed evolution primitives can lessen the burden on the database programmer to specify how that change affects existing data during instance conversion. Moreover, the ability to apply primitives to multiple types facilitates the evolution of the schema as a whole.

6. CONCLUSION

In this paper, we have presented, Vejal, an aspect language with a versioned type system. The language underpins the AspOEv evolution framework supporting adaptation of object database evolution strategies. A major advantage of object databases lies in transparent persistence, i.e. to supply OO developers with the ability to express persistence concerns through techniques that they already know and understand, without the need for specialised syntax. Vejal extends this idea to the AO level by integrating aspect capabilities within its versioned type system, hence providing a means of expressing AO concerns in user-defined persistent types that can transparently act upon persistent data. In addition, it facilitates aspects to operate uniformly on multiple levels of execution - at both the Java and Vejal level. As aspects bind to elements within the Vejal Abstract Syntax Tree (AST), their causality relationships with evolution strategies in fact originate from the Vejal execution the AST node expresses.

Vejal also facilitates specification of both static (i.e. on types) and dynamic (i.e. on instances) action to associate with a change. An evolution primitive can, therefore, be viewed as an aspect to be applied to a type and its instances. The aspect-primitive, written in Vejal, gives the database programmer greater expressive ability than a single-purpose primitive specification language. Furthermore, structures provided for declaring primitive execution permit schema-wide changes by allowing conditional evaluation and application of the primitive to multiple types.

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

7. REFERENCES

[1] J. Banerjee, et al., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", ACM SIGMOD Conference, 1987, ACM, SIGMOD Record, 16(3), pp. 311-322.

[2] R. C. H. Conner, et al., "Using Persistence Technology to Control Schema Evolution", ACM SIGAP Conference, 1994.

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

[4] G. N. C. Kirby, et al., "Using Reflection to Support Type-Safe Evolution in Persistent Systems", University of St. Andrews, UK, Technical Report No. CS/96/10 1996.

[5] E. Odberg, "MultiPerspectives: The Classification Dimension of Schema Modification Management for Object-Oriented Databases", Proc. TOOLS-USA, 1994, IEEE.

[6] A. Popovici, et al., "Just-In-Time Aspects: Efficient Dynamic Weaving for Java", Proc. AOSD 2003, ACM, pp. 100-109.

[7] A. Rashid, "A Database Evolution Approach for Object-Oriented Databases": PhD Thesis, Computing Department, Lancaster University, UK, 2000.

[8] A. Rashid, "Aspect-Oriented Programming for Database Systems", in *Aspect-Oriented Software Development*, Addison-Wesley, 2004, pp. 657-680.

[9] A. Rashid, N. Leidenfrost, "Supporting Flexible Object Database Evolution with Aspects", Proc. GPCE 2004, LNCS 3286, pp. 75-94.

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