Supporting Flexible Object Database Evolution with Aspects

Awais Rashid, Nicholas Leidenfrost

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

Abstract. Object database management systems (ODBMSs) typically offer fixed approaches to evolve the schema of the database and adapt existing instances accordingly. Applications, however, have very specialised evolution requirements that can often not be met by the fixed approach offered by the ODBMS. In this paper, we discuss how aspect-oriented programming (AOP) has been employed in the AspOEv evolution framework, which supports flexible adaptation and introduction of evolution mechanisms – for dynamic evolution of the schema and adaptation of existing instances – governing an object database. We argue that aspects support flexibility in the framework by capturing crosscutting hot spots (customisation points in the framework) and establishing their causality relationships with the custom evolution approaches. Furthermore, aspects help in information hiding by screening the database programmer from the complexity of the hot spots manipulated by custom evolution mechanisms. They also make it possible to preserve architectural constraints and specify custom version polymorphism policies.

1. Introduction

The structure of a database may not remain constant and may vary to a large extent as demonstrated by the measurement of the frequency and extent of such changes [37]. Therefore, it comes as no surprise that the schema, i.e., the class hierarchy and class definitions, governing the objects residing in an object database is often subject to changes over the lifetime of the database. Consequently, a number of models have been proposed to evolve the schema to maintain backward and forward compatibility with applications (in existence before and after the changes respectively). These models can be classified into four categories:

- *Basic schema modification* [3, 16], where the database has only one logical schema to which all changes are applied. No change histories are maintained so the approach only supports forward compatibility with applications.
- *Schema versioning* [22, 27], where a new version of the schema is derived upon evolution hence, ensuring both forward and backward compatibility with applications.
- *Class versioning* [24, 38], where the versioning of schema changes is carried out at a fine, class-level granularity. Like schema versioning, the changes are both forward and backward compatible.

- *Hybrid approaches*, which version partial, subjective views of the schema e.g., [1] or superimpose one of the above three models on another e.g., basic schema modification on class versioning as in [30, 35].

The schema evolution models need to be complemented by appropriate mechanisms to adapt instances to ensure their compatibility with class definitions across schema changes. For example, an object might be accessed by a class definition derived by adding a member variable to the definition used to instantiate the object in the first place hence, resulting in incompatibility between the *expected* and *actual* type of the object. Instance adaptation approaches deal with such incompatibilities and can be classified into simulation-based (e.g., [38]) and physical transformation approaches (e.g., [24]). The former simply simulate compatibility between the expected and actual type of the object while the latter physically convert the object to match the expected type.

Traditionally, an ODBMS offers the database application developer/maintainer one particular schema evolution approach coupled with a specific instance adaptation mechanism. For example, CLOSQL [24] is a *class versioning* system employing *dynamic instance conversion* as the instance adaptation mechanism; ORION [3] employs *basic schema modification* and *transformation functions*; ENCORE [38] uses *class versioning* and *error handlers* to simulate instance conversion.

It has been argued that such "fixed" functionality does not serve application needs effectively [34]. Applications tend to have very specialised evolution requirements. For one application, it might be inefficient to keep track of change histories, hence making *basic schema modification* the ideal evolution approach. For another application, maintenance of change histories and their granularity might be critical. Similarly, in one case it might be sufficient that instance conversion is simulated while in another scenario physical object conversion might be more desirable. The requirements can be specialised to the extent that custom variations of existing approaches might be needed.

Such flexibility is very difficult to achieve in conventional ODBMS designs for several reasons:

1. The schema evolution and instance adaptation concerns are overlapping in nature and are also intertwined with other elements of the ODBMS, e.g., the transaction manager, the object access manager, type consistency checker and so on [29]. Any customisation of the evolution concerns, therefore, results in a non-localised impact posing significant risk to the consistency of the ODBMS and, consequently, the applications it services.
2. Even if it is possible to expose the customisation points, such exposure poses a huge intellectual barrier for the database application programmer/maintainer who needs to understand the intricate workings of the ODBMS and its various components in order to undertake any customisation. Furthermore, vendors are, mostly, unwilling to expose the internal operation of their systems to avoid unwanted interference from programmers and maintainers in order to preserve architectural constraints.
3. Customisation of evolution mechanisms has implications for type consistency checking as different schema evolution approaches might have different perceptions of type equivalence, especially in the presence of different versions of the same type or schema.

The AspOEv evolution framework, that we are developing, supports flexible adaptation and introduction of schema evolution and instance adaptation mechanisms in an ODBMS independently of each other and other concerns in the system. AOP [2, 14] has been employed in the framework to capture crosscutting hot spots (customisation points in a framework [15]) and establish their causality relationships with the custom evolution approaches. The pointcuts expose a new interface to the underlying database environment to facilitate flexible tailoring of the schema evolution and instance adaptation approaches. Furthermore, aspects are used to support information hiding by screening the database programmer/maintainer from the complexity of the hot spots manipulated by custom evolution mechanisms. They also make it possible to preserve architectural constraints and specify custom version polymorphism policies.
Section 2, in this paper, provides an overview of the AspOEv architecture and its implementation. Section 3 discusses three key aspects supporting flexibility in the framework namely, Type Consistency and Version Polymorphism, Evolution Primitive Binding and Exception Handling. Section 4 shows the framework in operation and how the aspects in section 3 facilitate customisation of the evolution mechanisms. Section 5 discusses some related work while section 6 concludes the paper and identifies directions for future work.

2. AspOEv Architecture

The architecture of the AspOEv framework is shown in Fig. 1. The framework has been implemented in Java and AspectJ (v1.0) [2].

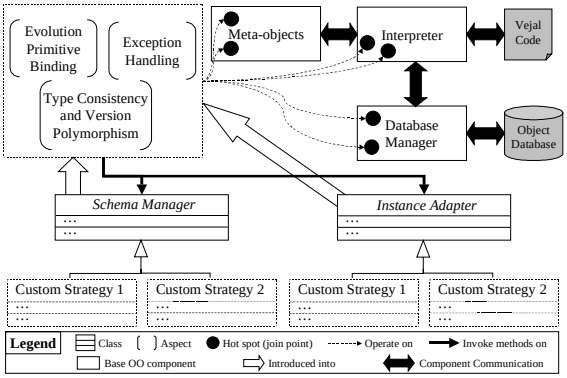

Fig. 1. Architecture of the AspOEv evolution framework

Since most approaches to schema evolution allow several different versions of a class to exist (e.g., individual versions in class versioning or different definitions across schema versions), these versions must be viewed as semantically equivalent and interchangeable. However, most object-oriented (OO) programming languages do not

support type versioning. Therefore, the framework employs its own application programming language, Vejal, an OO language with a versioned type system [19, 31]. Vejal has a two-level versioning identifier system (analogous to that used by the popular free source code control system CVS [11]). `C<1>` indicates class version 1 of class C while `C<s=1>` implies the class version of C that occurs in schema version 1. In Vejal, one version of a class may be present in multiple schema versions. In order to prevent unworkable schema versions being created, a new version of a class can only be present in all the future schema versions in which it is still compatible with the contracts of the other classes in the schema. Readers should note that, in the rest of the paper, we use the term "type" to refer to both classes and their versions in Vejal.

Traditionally, there has been a natural conflict between language type system constraints and object database evolution approaches, especially those facilitating dynamic evolution and adaptation of types and instances. This is because the constraints of the language type system, which exist to improve safety, act to hinder the required evolution. Consider an example evolution scenario where *A* and *B* are the definitions of a particular class before and after the evolution respectively. After evolution it may be desirable that all values of type *A* now have *B* as their type. However, such an operation is considered potentially dangerous by the type system (programs already bound to these values may rely on the assumption that they are of type *A*) which prevents it. The existence of a language with a versioned type system in our framework makes it possible to ensure that important typing constraints are preserved while at the same time facilitating flexible evolution and adaptation of types and instances.

The framework includes an interpreter for Vejal and, like most dynamic schema evolution approaches, employs a meta-object layer to represent persistent data. The existence of an interpreted language introduces some performance overhead – the interoperability of type versions with potentially large differences requires a thorough analysis based on context and structural equivalence where context includes the evolution approach and the database schema employed by it – but affords us the flexibility of customisation at a fine, program execution level granularity. As discussed in [25], such performance-flexibility trade-offs have to be established during framework design. To decrease the overhead of dynamic type checking, the framework creates tables based on 'rules' defined by the evolution approach. Evolution approaches define a boolean valued method, *equals (Type, Type)*, which evaluates the equivalence of two types. At startup, as well as after the change of evolution strategies or execution of evolution primitives, type equivalencies are computed from the database schema and stored in the schema manager for quick runtime lookup. Each type is compared against other versions of the same type in the schema, as well as the versions of types declared to be substitutable.

The framework also includes a database manager to support persistent storage and manipulation of both user-level and meta-level objects in the underlying ODMG 3.0 compliant [7] object database. Currently the framework is being used to facilitate customisable evolution in the commercially available Jasmine object-oriented database [21]; we have implemented an ODMG 3.0 Java binding wrapper for the Jasmine Java binding.

In order to enable the dynamic restructuring of data (e.g., changes to inheritance relationships or class members), the object structure must be kept in the meta-object layer. When coupled with the needs of an interpreted type-versioning language, we are left with two collections of objects to represent the separate concerns of language execution and object structure. The database manager component dealing with database interaction also contains several areas of interest for schema evolution and instance adaptation mechanisms. Therefore, behavioural and structural concerns pertaining to evolution need to be detected and handled across the three components (the interpreter, the meta-object layer and the database manager), which together provide the base OO separation manipulated by three aspects: Type Consistency and Version Polymorphism, Evolution Primitive Binding and Exception Handling. Note that the three base OO components are completely oblivious of the schema evolution and instance adaptation strategies to be plugged in.

The Type Consistency and Version Polymorphism aspect deals with the interpreter's view of typed versions, i.e., whether two different versions of the same class can be considered to be of the same type. This has implications in instance adaptation, type casting and polymorphism – since the class hierarchy can be manipulated at runtime, one version of a type may be assignable to a base type, while another version may not. Schema evolution approaches might also be interested in providing custom version equivalence and substitutability semantics. The aspect, therefore, facilitates customisation of versioned type equality semantics.

The operations used to modify the schema of the database are often referred to as evolution primitives. These primitives range from modification of the class hierarchy, e.g., introduction or removal of classes and modification of inheritance links, to introduction, removal and modification of individual members of a class [3, 26, 35]. The Evolution Primitive Binding aspect monitors the addition and removal of types and their versions from the database schema as well as modification of inheritance links. Schema evolution strategies must decide at these points how to react to the new or deprecated type or to changes in inheritance relationships. Obviously, the action they take must be propagated to instances of the type – an introduced type may become the new standard for the schema, forcing all existing instances to comply before use, or a deprecated type may need all of its instances transferred to another version. The aspect also traps the occurrence of changes to a class and its members. The execution of operations performing such changes provides suitable hotspots for a schema evolution strategy to predict and circumvent any negative impact they might incur. They also allow an instance adaptation strategy to take a wide range of suitable actions.

The Exception Handling aspect allows the application programmer to attempt to preserve behavioural consistency when runtime exceptions are raised. These exceptions may be the result of missing members or type mismatches and could be rectified by handlers provided in custom strategies that have knowledge of the application's inner workings.

The three aspects direct all advice action to method invocations on the two abstract strategy classes (as in the strategy pattern [18]): Schema Manager and Instance Adapter, which are introduced, in the AspectJ sense, into the aspects (to facilitate callbacks). Implementation of custom approaches requires overriding a subset of the callback methods in these classes. This has a number of advantages: Firstly, the

application programmer/maintainer is shielded from the complexity of the hot spots in the three base OO components hence, facilitating information hiding and avoiding unwanted interference with those components. Secondly, the programmer/maintainer can create custom strategies without knowledge of AOP and thirdly, strategies can be switched without recompiling the framework. Note that the schema evolution and instance adaptation strategies are independent of each other (cf. section 3.1) so the developer/maintainer is free to choose any combination of strategies as long as they are semantically compatible.

3. Aspects Supporting Flexibility

In this section, we discuss the three aspects introduced in section 2 in more detail. We mainly focus on how the aspects support flexibility in the framework. Note that though the aspects themselves do not change during customisation, nevertheless they help modularise a complex, non-trivial set of crosscutting concerns, namely the schema evolution and instance adaptation strategy to be employed by the underlying ODBMS. In this case, the aspects and the strategy pattern jointly facilitate the flexible customisation and adaptation of these strategies. As demonstrated in [33], aspectisation requires that a coherent set of modules including classes and aspects collaborate to modularise a crosscutting concern – such a view of AOP ensures that aspectisation is not forced and in fact leads to a natural separation of concerns. Therefore, the role of the three aspects in AspOEv is not out of sync with the fundamental aims of aspect-oriented modularity. Furthermore, the design of the framework and the causality relationships between the strategy classes and the other components in the framework would have been very complex without the existence of these aspects. They provide us with a clean, modular design of the framework hence making the framework itself more maintainable and evolvable. They also facilitate separation of the evolution model employed by the ODBMS from the actual implementation of the schema in Vejal.

3.1 Type Consistency and Version Polymorphism Aspect

This aspect supports flexibility in the following three ways:

- It preserves clean design separation between the schema manager and the instance adapter facilitating flexible, semantically compatible, combinations of custom strategies for schema evolution and instance adaptation.
- It captures version discrepancies between the expected and actual type of an object, i.e., between the version in use by the schema and the one to which the object is bound respectively, and provides this information to the schema manager and instance adapter for rectifying action.
- It facilitates the provision of custom version polymorphism policies, i.e., allowing a custom schema manager to specify which type versions are assignable to each other or are substitutable.

Although schema managers and instance adapters work together to achieve stable schema evolution, they are separate concerns, and should not need to know about each other. The instance adapter has the task of adapting an object from its actual type to the schema manager's expected type. From the viewpoint of the instance adapter, it is simply adapting from one type version to another and, therefore, it does not need to be aware of the evolution strategy being employed. The aspect preserves clean design separation by acting as an intermediary between the two concerns. It queries the schema manager for its expected version and passes the resulting information to the instance adapter (cf. shaded code in Fig. 2). Consequently, the instance adapter remains oblivious of the schema manager, yet gets the information it needs to perform its task, while the schema manager receives a converted object of the version it expects.

The code listing in Fig. 2 also shows a simple example of type version discrepancies captured by the aspect and forwarded to the custom schema manager and instance adapter being used. ODMG compliant object databases support binding specific objects to unique names to act as *root entry points* into the database. These persistent roots are retrieved using the `lookup` method. Since the method only returns a single object, as opposed to other query methods that return collections of matches (and are also monitored by the Type Consistency and Version Polymorphism aspect), it provides a simple example of potential type mismatches in an executing program. The advice in Fig. 2 captures the result of the lookup, queries the schema manager for the type it expects to be returned, and then has the instance adapter perform the proper conversion.

```
pointcut lookup(String name):
   execution(DatabaseManager.lookup(String))
   && args(name);

Object around(String name): lookup(name) {

   MetaObject result = (MetaObject)proceed(name);
   Type resultType = result.getType();
   Type expectedType =
      schemaManager.getActiveType(resultType);
   MetaObject conformed = result;
   if (!resultType.equals(expectedType))
      conformed =
         instanceAdapter.retype(result, expectedType);
   return conformed;

}
```

Fig. 2. Capturing version discrepancies and providing independence of instance adapter and schema manager

```
database.bind(new Foo<1.0>,
              "myFoo");
…
// sometime later
// in execution
Foo<2.0> aFoo =
   database.lookup("myFoo");
```

Fig. 3. Type mismatch at program execution level (numbers in < > indicate version of Foo used)

Fig. 2 shows an example of type version discrepancies that can be captured at the interface with the database. There are other types of discrepancies that occur at the level of program execution. Consider a class versioning approach to schema evolution when multiple versions of a type are allowed to exist in the execution environment. In

this case, the schema manager has no preference to any version[1] and thus the `expectedType` in Fig. 2 would be the same as the `resultType`. The code listing in Fig. 3 shows an example of type mismatch at the program execution level. Since the schema manager does not know or care about the version of object `aFoo`, this can cause serious inconsistencies. Without the use of aspects woven over an interpreted language, this mismatch would be very difficult to catch before an error occurred. However, the type consistency aspect allows instance adapters to monitor assignment in the executing program. Because schema managers will not always work with a particular version of a type only, flexibility must exist to allow other ways of detecting type mismatch.

The type consistency aspect supports flexible type interchangeability by wrapping type comparison, equality and assignability methods, and allowing a custom evolution strategy to specify how different versions of types are to be treated by the interpreter. Since the evolution framework allows such changes as the addition and removal of classes in the class hierarchy, one version of a type may lose the ability to be assignable to other versions or vice versa. Moreover, one version may be more or less equivalent to a second version, but drastically different from a third. With the ability to evaluate type differences at runtime, the flexibility exists to create a versioned polymorphic environment. In the absence of clear version polymorphism semantics for the evolution approach being employed, there can be lots of potentially unnecessary invocations of the instance adapter resulting in performance overhead.

Consider, for example, the scenarios in Fig. 4. Since `Person<2.0>` in Fig. 4(a) is derived because of an additive change to `Person<1.0>`, instances of `Person<2.0>` could be allowed to be substitutable wherever an instance of `Person<1.0>` is expected. If a schema manager chooses to allow such substitutability, implicit instance adaptation can be carried out (through version polymorphism) without invoking the instance adapter to do an explicit transformation. This will reduce performance overhead. However, in some cases, the application needs might dictate the schema manager to disallow this substitutability and force an explicit transformation through the instance adapter.

Another scenario where version polymorphism could be required is shown in Fig. 4(b). A schema manager (in response to application requirements) might allow substitution of instances of `B<2.0>` and `C<2.0>` wherever an instance of `A<1.0>` is required, on the grounds that the predecessors of `B<2.0>` and `C<2.0>` (`B<1.0>` and `C<1.0>` respectively) inherit from `A<1.0>`. The schema manager might need to evaluate the differences between `A<2.0>` and `A<1.0>`before allowing such substitutability.

Equality testing in the Vejal interpreter is similar to that in Java in that it is centred on an `equals` method defined in types. The `typeEquality` pointcut in the Type Consistency and Version Polymorphism aspect (cf. Fig. 5) allows a custom strategy to perform a number of tests, including looking for common base classes, hence supporting version polymorphism. Custom version polymorphism is further facilitated by the `assignableFrom` pointcut[2] (cf. Fig. 5). The default implementation of the

[1] This is in contrast with, for instance, schema versioning where each schema managed by the schema manager specifies a clear expected version of a type.

[2] The !cflow designator is required because the `assignableFrom` method is recursive.

method to which the related advice delegates, simply searches for the second type in the first one's supertypes. A custom strategy can alter assignability semantics by overriding the default implementation and searching for a specific base type or performing a more *deep* comparison of the two types, e.g., by searching the supertypes of their predecessor versions.

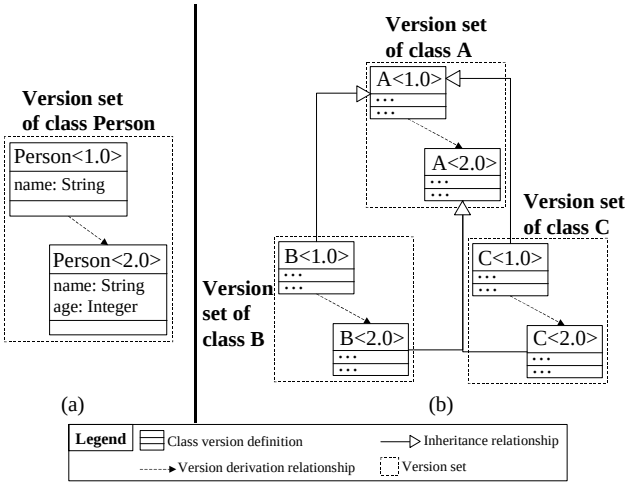

Fig. 4. Potential scenarios involving version polymorphism

Note that in class versioning approaches to schema evolution, the execution environment can, potentially, have instances of several versions of a type to manage. Therefore, the schema manager does not necessarily have a default expected type that an instance adapter can convert to when the instance is returned from a database query. Therefore, different type versions could potentially be used interchangeably, without instance adaptation (because of version polymorphism), until a conversion is forced by an incompatible combination or assignment of versions.

```
pointcut typeEquality(Type one, Type two):
      execution(Type.equals(Object))
      && args(two)
      && this(one);

pointcut assignableFrom(Type one, Type two):
      execution(Type.assignableFrom(Type))
      && args(two)
      && this(one)
      && !cflow(Type.assignableFrom(..));
```

Fig. 5. Pointcuts facilitating custom type version polymorphism

3.2 Evolution Primitive Binding Aspect

The execution of evolution primitives, which carry out the schema changes, is of key interest to custom schema managers. Different schema evolution strategies respond to

execution of the primitives differently. For instance, schema versioning results in creation of a new schema version upon each change. Class versioning, on the other hand, leads to newer versions of the modified classes. Context versioning [1] only creates newer versions of the partial subjective views in which the modified class participates. Other approaches, e.g., [6], group primitive evolution operations into high-level primitives.

At the same time, each evolution strategy responds to individual evolution primitives differently. For instance, in the class versioning approach proposed in [35], addition, removal or modification of a class member results in creation of a new version of the class. However, if the class version being modified forms a non-leaf node in the hierarchy graph, new versions of all the sub-classes are transitively derived. If a class or class version forming a non-leaf node is removed or repositioned then stubs are introduced to maintain consistency of sub-classes of the removed or repositioned class.

Execution of evolution primitives also has implications for instance adaptation which would almost inevitably be required. Each particular instance adaptation approach will respond to evolution in its own fashion. This variability in the nature of evolution and adaptation approaches, and even in the handling of individual primitives by a particular strategy, makes it essential to allow custom strategies to react to the execution of the primitives in their specialised fashion. The Evolution Primitive Binding aspect facilitates this flexibility.

As an example of evolution primitive binding captured (and passed on to custom strategies) by the aspect, let us consider the case of removing a type from the class hierarchy in an object database. This would raise the need to reclassify (or remove altogether) any existing instances of the type. If the removed type has subtypes, there are further implications in terms of structural and behavioural consistency of their definitions and instances. Furthermore, different evolution strategies would handle type removal differently. For instance, class versioning approaches might only remove a single version of the class while schema versioning and schema modification strategies would, most probably, remove the class altogether from the resulting schema definition. Fig. 6 shows the pointcut, trapping the execution of the type removal primitive, and its associated advice informing the schema manager and instance adapter of this change (note that the two strategies are decoupled and can independently pursue appropriate action in response). By providing a schema evolution strategy with pertinent information at the time of type removal, it can reclassify instances of the removed type appropriately with respect to the schema to avoid information loss. Depending on the evolution strategy being employed, a schema manager could retype existing instances of the removed type to an appropriate base type or retype them to the most similar type version. In either case, the instance adaptation strategy can use the provided information and loaded meta-classes in the interpreter to forward the deprecated type to its replacement.

Note that addition of a new type into the class hierarchy can have similar implications depending on the evolution strategy being employed, i.e., whether subtypes and their existing instances automatically gain functionality provided by the new superclass or new versions of each subtype are required. Similarly, any conflicts between inherited and locally defined members and inheritance paths (Vejal supports multiple inheritance) would need to be resolved in line with preferences of the custom

evolution strategy. The aspect captures execution of type addition and other type hierarchy manipulation primitives in a fashion similar to type removal.
For another example of evolution primitive binding, let us consider the modification of class members. The simple evolution scenario shown in Fig. 7 involves *renaming* one of the class members: `surname` to `lastname`. This will result in a behavioural consistency problem as existing references to the renamed member will become invalid. By capturing the state of the member before and after the change, the aspect makes it possible for instance adapters to take corrective action, e.g., by forwarding references. Other class modification primitives are captured and handled in a similar fashion.

```
pointcut removeType(Type removed):
     execution(DatabaseManager+.removeType(Type))
          && args(removed);

before(Type removed): removeType(removed) {
     schemaManager.removeType(removed);
     Type reclassify =
          schemaManager.getReassignedType(removed);
     instanceAdapter.reclassifyType(removed,
                                    reclassify);
}
```

Fig. 6. Pointcut and advice pertaining to type removal

```
class Person<1.0> {

   String firstname,
          surname;

}
```

(a)

```
class Person<2.0> {

   String firstname,
          lastname;

}
```

(b)

Fig. 7. Evolution scenario: renaming a member
(a) before evolution
(b) after evolution

```
pointcut fieldChanged(Metaclass versionOne,
                      Metaclass versionTwo,
                      MetaField oldField,
                      MetaField newField):
                 execution(MetaClass+.changeField(
                                 MetaClass,
                                 MetaField,
                                 MetaField))
                              && args(versionOne,
                                      oldField,
                                      newField)
                              && target(versionTwo);

Object around(Metaclass versionOne,
              Metaclass versionTwo,
              MetaField oldField,
              MetaField newField):
    fieldChanged(versionOne,
                 versionTwo,
                 oldField,
                 newField) {

    /* Should create bridge if non-existent,
    otherwise retrieve from transient memory. */
    VersionBridge bridge =
       instanceAdapter.getVersionBridge(versionOne,
                                        versionTwo);
    bridge.mapFields(oldField, newField);
}
```

Fig. 8. Pointcut and advice pertaining to version bridges

The aspect also facilitates optimisation of repeated adaptations required in response to an evolution primitive execution. Once again, consider the simple scenario of renaming a member from Fig. 7. While this seems like a simple enough change, references to this member by the handle `surname` could exist in a number of different places – more so than one could hope to catch and change at the moment of primitive execution. By capturing hot spots in the meta-object layer, the aspect can create a *version bridge* that is aware that the reference to the member `surname` in `Person<1.0>` should map to `lastname` in `Person<2.0>`. The pointcut and associated around advice corresponding to version bridges is shown in Fig. 8.

Note that a version bridge in the framework is a piece of meta-data that mimics more than one meta-class at once. A type instance can be safely reclassified to reflect its ability to fulfill the requirements of another version, while still maintaining its own type identity. Version bridges know about the meta-classes for both versions, and can thereby easily perform tasks such as forwarding. Obviously, with several existing types, version bridges could quickly become tedious and time consuming for the application programmer/maintainer to generate manually. Therefore, the use of the Evolution Primitive Binding aspect to dynamically create and retype instances in accordance with a general strategy saves a non-trivial amount of work.

3.3 Exception Handling Aspect

Since exceptions are a means of attracting attention to abnormal or incorrect circumstances in program execution, they offer a natural mechanism for a framework designed to preserve consistency during object database evolution in a flexible fashion. In fact, some approaches to instance adaptation, e.g., [36, 38], rely entirely on handling exceptions raised in response to type mismatches to take rectifying action – rigidity affords no other opportunities to capture mismatches. Such approaches, that completely rely on exception handling for instance adaptation, therefore, need to know about a variety of events such as type mismatch, name mismatch and missing, extraneous or incorrect data members and parameters, etc.

While exception handling is by no means a new territory for AOP and AspectJ, the ability to handle exceptions thrown from an interpreter, over which custom evolution strategies have some control, provides additional flexibility. There could arise situations under which even the most well adapted strategy would fail without exception handling. Furthermore, if approaches for schema evolution or instance adaptation change, the new approaches may not have had the opportunity to handle changes that occurred before the switch. The new schema managers and instance adapters cannot rely on their respective previous strategies to inform them of all the changes that might have occurred. By carrying information pertinent to the context of the mismatch, specialised exceptions give instance adaptation strategies a final chance to preserve behavioural consistency. Two examples of such exceptions and the information they carry are shown in Fig. 9. Note that, as discussed in sections 3.1 and 3.2, the AspOEv framework offers other opportunities to detect and handle situations requiring instance adaptation.

Fig. 10 shows a pointcut and its associated around advice from the Exception Handling aspect intercepting such a type mismatch exception and deferring it to the instance adapter and schema manager for rectifying action.

```
throw new
  TypeMismatchException(
         <Type> expected,
         <Type> actual);

throw new
  NoSuchFieldException(
     <FieldReference> field,
     <Type> owner);
```

Fig. 9. Examples of information provided by exceptions advice

```
pointcut typeMismatch(MetaObject found,
                      Type required):
   call(TypeMismatchException.new(MetaObject,
                                  Type))
   && args (found, required);

Object around(MetaObject found, Type required):
   typeMismatch (found, required) {
     try {
          proceed(found, required);
     }
     catch (TypeMismatchException tme) {
        instanceAdapter.retype(found, required);
        schemaManager.typeMismatch(tme);
     }
}
```

Fig. 10. An exception handling pointcut and associated with callbacks to instance adapter and schema manager

Since the exceptions are raised by consistency issues at runtime, they are unexpected by the program. Therefore, no handling code, aside from that of the custom strategy on which the Exception Handling aspect calls back, is in place. As these exceptions are raised in an interpreted environment, adapters have the opportunity to make changes and retry the failed operation. Execution could effectively be paused, or even taken back a step. Furthermore, due to the nature of the meta-data provided to exception handlers, it is possible to determine the types of changes that have resulted in the inconsistency in question. Members such as meta-fields and meta-methods all have unique identifiers, so it is possible to determine that a member has been renamed, as opposed to it seeming as if a removal and subsequent addition occurred.

4. The Framework in Operation

In this section we discuss how two significantly different evolution strategies namely, class versioning and basic schema modification can be implemented using the framework. Before providing an insight into the implementation of the two strategies, it is important to highlight the callback methods available in the abstract Schema Manager class as both evolution strategies would extend this class and override the callbacks of relevance to them. The Schema Manager class is shown in Fig. 11. The callback methods are categorised into *general callbacks*, *evolution primitive binding callbacks*, *version polymorphism callbacks* and *exception handling callbacks*. Here we briefly summarise the role of the general callbacks as the roles of the other three sets are self-explanatory:

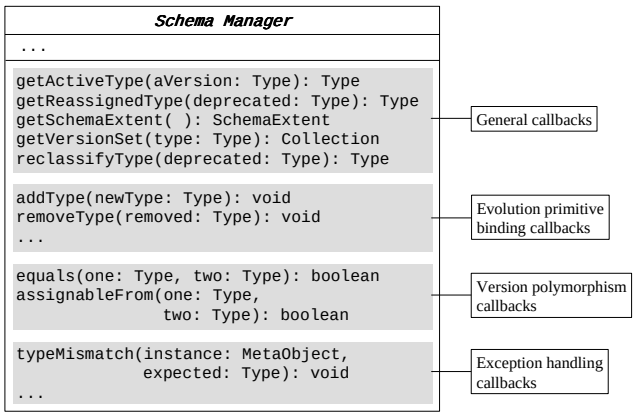

Fig. 11. Callbacks in the abstract Schema Manager strategy class

- `getActiveType`: returns the active type version (expected type), corresponding to the argument type version, in use by the schema.
- `getReassignedType`: a simple accessor method to obtain, from the type map in the schema manager, a reference to the type to which instances of a deprecated type have been reassigned.
- `getSchemaExtent`: to get the set of all classes in a schema or schema version.
- `getVersionSet`: to get the set of all versions of a class. Note that if a class versioning approach is being used, the `getSchemaExtent` method will use this method to obtain and return a set of all the version sets of all the classes in the schema.
- `reclassifyType`: encapsulates the algorithm to identify and specify the *reassigned* type for instances of a deprecated type.

4.1 Implementation of Custom Evolution Approaches

Let us first consider the implementation of class versioning using the framework. We will refer to this as Class Versioning Schema Manager (CVSM).
Class versioning approaches are unique in their ability to permit several versions of a type to exist in a schema and execution environment hence, facilitating active type consistency. For example, looking at the advice code in Fig. 2 in the context of a class versioning approach, the CVSM will need to override `getActiveType` to simply return the argument type. This is because a class versioning schema manager has to manage multiple versions of a class and hence, does not specify a single expected version of the class. The CVSM will also need to override the other general callbacks especially the `getVersionSet` method as it will be employed by the `getSchemaExtent` method to obtain the set of all types in the schema.
The CVSM will most certainly have to override the evolution primitive binding callbacks to provide its own custom response to execution of the various primitives. This is facilitated by the Evolution Primitive Binding aspect. A detailed discussion of

evolution primitive implementation during class versioning is beyond the scope of this paper. A number of such implementations exist. Interested readers are referred to [5, 24, 35, 38].

A class versioning implementation also has the responsibility of managing the assignment and accessibility of many versions of a type at the level of the executing program. To ensure consistency at these points, the corresponding instance adapter would be invoked (cf. section 3.1). However, with the flexibility provided by the Type Consistency and Version Polymorphism aspect, the CVSM has the option of overriding the version polymorphism callbacks to stipulate semantics for type equality, assignability and version polymorphism within the executing interpreter, thus preventing unnecessary instance adaptation.

The CVSM may also override the exception handling callbacks (afforded by the Exception Handling aspect) to capture any inconsistencies in type version equivalence or instance adaptation not captured by the version polymorphism callbacks or the instance adapter.

Let us now consider the implementation of the basic schema modification strategy. We will refer to this as Basic Schema Modification Manager (BSMM). The BSMM must first define a working schema, listing all types that will be used (usually the most recent versions of a class), and must override the `getActiveType` callback to return the correct type defined by the schema. Though this might seem trivial at first glance, this is a vital callback for the BSMM as it facilitates maintenance of consistency of objects by comparing them with the expected types of the schema before they enter execution.

The BSMM must also define how it will deal with the various evolution primitives through the evolution primitive binding callbacks. Interested readers are referred to [3, 16] for a discussion of potential behaviour of basic schema modification primitives that could be implemented by the BSMM, as a detailed discussion is beyond the scope of this paper.

Like the CVSM, the BSMM could specify handlers for any mismatches not captured by other callbacks. However, it will, most likely, not utilise the version polymorphism callbacks as these mainly facilitate an informed evaluation of the need for instance adaptation in an environment with multiple versions. Since the BSMM exercises strict control in the form of having the most recent version of a type in the schema, these callbacks would be of little interest to it.

Note that so far we have not discussed customisation of the instance adapter to suit the CVSM and the BSMM. Though one might think that instance adapters for these two approaches would not be compatible, this is not the case. If an instance adapter is designed to respond to instance adaptation needs at both the program execution level and at the interface with the database, it will certainly service the needs of both. It will serve the BSMM effectively as most of the instance adaptation for this would occur at the interface with the database. This is because a BSMM dictates that the persistent instance retrieved from the database is compliant with the expected type in the schema before it is loaded into the execution environment. Therefore, the instance adapter will adapt the instance to comply with the expected type upon retrieval. Conversely, the instance adapter would not be invoked for the CVSM when query results are received from the database as the need for adaptation is postponed till an actual collision occurs (recall that the CVSM has no default expected type version for

a class). Here the ability to monitor join points at program execution level comes into play. Note that this orthogonality of instance adaptation and schema evolution concerns is afforded by the Type Consistency and Version Polymorphism aspect (cf. Section 3.1).

4.2 Handling Structural Consistency Issues

Let us consider the problems arising if an evolution approach prior to customisation may have remapped instances of a type (such as a removed type) to one of its base classes. If the new approach employs the (previously removed) type in its schema, it would want to return the remapped instances to their original type. Also, in the event that data from two different schemas are merged, several types from one schema may have been remapped to their equivalent type in the second schema. The new schema manager will need to know about these type equivalencies in order to continue to serve applications employing the merged data. The framework facilitates the handling of these situations by providing communication between the current and former schema managers when a switch occurs (cf. Fig. 12). The SchemaExtent object is a collection of types utilised by a schema manager (its schema) as well as relationships between types, such as remapped (deprecated) and substitutable (semantically or structurally equivalent) types. The new approach can either immediately convert these types, or convert them lazily. AspOEv facilitates lazy conversion in this case by providing functionality for automatically "enriching" type specific queries with remapped and substitutable types. Objects that have been remapped by another schema, however, must be differentiated from any existing instances of the type they are remapped to. Therefore, in AspOEv, each object has two stored types, the *current* type and the *declaration* type. While schemas may classify an object as many different types during its lifetime, the object's declaration type never changes. Remapped objects can be identified as such by their declaration types.

4.3 Handling Behavioural Consistency Issues

Database evolution primitives can break behavioural consistency by altering types or members that are referenced elsewhere in the program space. The persistent storage of Vejal programs as AbstractSyntaxTree elements in AspOEv facilitates querying for instances of programming components, such as field references, variable declarations, or method invocations in dependent types. This enables the framework to present the programmer with detailed information about where problems might occur. Moreover, this level of granularity in Vejal meta-data allows the framework to create new versions of dependent classes to automatically comply with some changes. If, for example, a member is renamed, the framework can detect and change any field references or method invocations in dependent classes (cf. Fig. 13).
Of course, changes such as renaming are seldom completely free of semantic implications, and any generated versions must be verified by the programmer before being put into effect.

```
SchemaExtent oldSchema =
      oldSchemaManager.getSchema();
schemaManager.transitionFrom(oldSchema);
```

Fig. 12. Communication between old and new schema managers

```
// Start a transaction
QueryCursor ageAccesses =
   FieldReference.where("fieldName == \"surname\"");
while (ageAccesses.moreElement( )) {
   FieldReference reference =
      (FieldReference)ageAccesses.nextElement( );
   reference.setFieldName("lastName"); }
// Commit transaction
```

Fig. 13. Querying and altering program elements

5. Related Work

The SADES object database evolution system [30, 35, 36] employs a declarative aspect language to support customisation of the instance adaptation mechanism. Two instance adaptation strategies, error handlers (a simulation-based approach) [38] and update/backdate methods (a physical transformation approach) [24], have been successfully implemented using the aspect language [36]. Though the work on customisable evolution in AspOEv has been inspired by SADES, there are some marked differences between the two systems. SADES uses aspects to directly plug-in the instance adaptation code into the system. Consequently, the complexity of the instance adaptation hot spots has to be exposed to the developer/maintainer and, at the same time, there is a risk of unwanted interference with the ODBMS from the aspect code. This is in contrast with the framework discussed in this paper as the developer/maintainer is shielded from the complexity of the hot spots. Furthermore, the inner workings of the ODBMS do not have to be exposed hence, facilitating preservation of architectural constraints by avoiding unwanted interference. SADES supports customisation of the instance adaptation approach only – the schema evolution strategy is fixed. It does not support version polymorphism either.

An earlier version of AspOEv [28, 29, 31] also facilitated customisation of both schema evolution and instance adaptation mechanisms. However, like SADES, aspects were employed to specify and directly plug-in the strategies into the system. Also, version polymorphism was not supported.

The composition of aspects in a persistent environment is treated in [32], which discusses various issues pertaining to weaving of aspects which are persistent in nature. Though the three AspOEv aspects discussed here are also operating in a persistent environment, they are not persistent in nature. They, however, do have to account for the fact that the pointcuts they employ span both transient and persistent spaces in the underlying database environment.

The GOODS object-oriented database system [23] employs a meta-object protocol to support customisation of its transaction model and related concerns such as locking and caching. Custom approaches can be implemented by refining existing meta-objects through inheritance. Customisation of the schema evolution and instance adaptation strategies is not supported. Furthermore, though customisation of the transaction model, locking and cache management is supported, the three concerns are not untangled. Consequently, customisation of one of the three concerns has an impact on the others.

Component database systems, e.g., [13, 20] offer special customisation points, similar to hot spots in OO frameworks, to allow custom components to be incorporated into the database system. However, customisation in these systems is limited to introduction of user-defined types, functions, triggers, constraints, indexing mechanisms and predicates, etc. In contrast our framework facilitates customisation or complete swapping of fundamental ODBMS components encapsulating the schema evolution and instance adaptation mechanisms. Our work, therefore, bears a closer relationship with DBMS componentisation approaches such as [4, 8]. However, unlike these works which focus on architectures to build component database systems, the focus of our framework is to provide customisable evolution in existing object database systems without rearchitecting them.

The use of aspects in our framework also bears a relationship with the work on active database management systems [12]. These systems employ event-condition-action (ECA) rules to support active monitoring and enforcement of constraints (and execution of any associated behaviour). If a simple or composite event that a rule responds to is fired, the specified condition is evaluated and subsequently the action is executed provided the condition evaluates to true. Aspects in our framework are similar in that they monitor the hot spots in the three base OO components and, upon detection of an event of interest, invoke the callback methods in the strategy classes. In some instances, the aspects evaluate some conditions before invoking the callbacks, e.g., comparing the expected and actual type (cf. Fig. 2) before invoking the instance adapter. This similarity between the use of aspects in the framework and active databases does not come as a surprise as already work has been undertaken to demonstrate the relationship between the two concepts [9].

[17] discusses the need for application-specific database systems and proposes the use of domain-specific programming languages with embedded, domain-specific database systems for the purpose. Our approach is similar in that it supports application-specific customisation of evolution concerns and employs its own database programming language, Vejal, with a versioned type system to facilitate capturing of behavioural concerns at program execution level. However, the customisation is carried out through a general-purpose OO programming language, i.e., Java, and is facilitated by a general-purpose AOP language, i.e., AspectJ. Furthermore, our framework is motivated by the need to modularise crosscutting evolution concerns.

The work presented in this paper also bears a relationship with the notion of aspect-oriented frameworks, e.g., [10]. However, unlike [10], which describes a general AOP framework, AspOEv is specific to the domain of object database evolution.

6. Conclusion and Future Work

This paper has discussed the use of aspects to support flexibility in an object database evolution framework. Aspects are employed to capture points of interest in three base OO components and to forward information from these points to custom schema evolution and instance adaptation strategies by invoking callback methods. This way aspects establish the causality relationships between the points in the base OO components and the evolution concerns interested in manipulating the behaviour at those points. However, these relationships are established without exposing the

complexity of the customisation points hence, promoting information hiding. The benefits of this are twofold. Firstly, the programmer/maintainer does not need to understand the details of the ODBMS design to implement a custom evolution concern. Secondly, the ODBMS is shielded from unwanted interference from the custom concerns hence, ensuring that architectural constraints are preserved and integrity is not compromised. This is significant as quite often AOP attracts criticism because aspects break encapsulation. In this instance, however, aspects are being employed to preserve encapsulation and information hiding.

Another key feature of aspects in the framework is the provision of support for version polymorphism. To the best of our knowledge, substitutability and assignability in a type-versioning environment has not been explored to date let alone application-specific customisation of such semantics. This flexibility would not have been possible without the use of aspects to establish the required causality relationships.

The use of aspects coupled with the strategy pattern makes it possible to swap schema evolution and instance adaptation strategies in the framework without recompilation. This implies that over the lifetime of an application one can choose to employ a different strategy in response to changing requirements. In our future work, we aim to focus on facilitating communication between old and new strategies to ensure that the new strategy can utilise information from changes by the previous one if needed. We are also developing a mechanism whereby the possible set of pointcuts to be captured by the three aspects is provided as a set of *bindings* and the aspects generated from these bindings. This ensures that if some pointcuts are needed for a specific customisation and are not already included in the aspects, they can be introduced without significant effort. Another important area of future interest is exploring the introduction of new, application-specific, evolution primitives by exploiting the flexibility afforded by the three aspects and the bindings. We also aim to carry out further case studies to test drive the customisability of the framework.

Acknowledgements

This work is supported by UK Engineering and Physical Science Research Council (EPSRC) Grant: AspOEv (GR/R08612), 2000-2004. The Jasmine system is provided by Computer Associates under an academic partnership agreement with Lancaster University. The authors wish to thank Robin Green for implementation of the previous version of AspOEv.

References

[1] J. Andany, M. Leonard, and C. Palisser, "Management of Schema Evolution in Databases", Proc. VLDB Conf., 1991, Morgan Kaufmann, pp. 161-170.

[2] AspectJ Team, "AspectJ Project", http://www.eclipse.org/aspectj/, 2004.

[3] J. Banerjee, H.-T. Chou, J. F. Garza, W. Kim, D. Woelk, and N. Ballou, "Data Model Issues for Object-Oriented Applications", *ACM Transactions on Office Inf. Systems*, 5(1), pp. 3-26, 1987.

[4] D. S. Batory, "Concepts for a Database System Compiler", Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1988, ACM, pp. 184-192.

[5] A. Bjornerstedt and C. Hulten, "Version Control in an Object-Oriented Architecture", in *Object-Oriented Concepts, Databases, and Applications*, W. Kim, Lochovsky, F. H., Ed., 1989, pp. 451-485.

[6] P. Breche, F. Ferrandina, and M. Kuklok, "Simulation of Schema Change using Views", Proc. DEXA Conf., 1995, Springer-Verlag, LNCS 978, pp. 247-258.

[7] R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russel, O. Schadow, T. Stenienda, and F. Velez, *The Object Data Standard: ODMG 3.0*: Morgan Kaufmann, 2000.

[8] S. Chaudhuri and G. Weikum, "Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System", Proc. VLDB Conf., 2000, Morgan Kaufmann, pp. 1-10.
[9] M. Cilia, M. Haupt, M. Mezini, and A. Buchmann, "The Convergence of AOP and Active Databases: Towards Reactive Middleware", Proc. GPCE, 2003, Springer-Verlag, LNCS 2830, pp.169-188.
[10] C. Constantinides, A. Bader, T. Elrad, M. Fayad, and P. Netinant, "Designing an Aspect-Oriented Framework in an Object-Oriented Environment", *ACM Computing Surveys*, 32(1), 2000.
[11] CVS, "Concurrent Versions System", http://www.cvshome.org/, 2003.
[12] K. R. Dittrich, S. Gatziu, and A. Geppert, "The Active Database Management System Manifesto", 2nd Workshop on Rules in Databases, 1995, Springer-Verlag, LNCS 985, pp. 3-20.
[13] K. R. Dittrich and A. Geppert, *Component Database Systems*: Morgan Kaufmann, 2000.
[14] T. Elrad, R. Filman, and A. Bader (eds.), "Theme Section on Aspect-Oriented Programming", *Communications of the ACM*, 44(10), 2001.
[15] M. E. Fayad and D. C. Schmidt, "Object-Oriented Application Frameworks", *Communications of the ACM*, 40(10), pp. 32-38, 1997.
[16] F. Ferrandina, T. Meyer, R. Zicari, and G. Ferran, "Schema and Database Evolution in the O2 Object Database System", Proc. VLDB Conf., 1995, Morgan Kaufmann, pp. 170-181.
[17] K. Fisher, C. Goodall, K. Högstedt, and A. Rogers, "An Application-Specific Database", 8th Int'l Workshop on Database Programming Languages, 2002, Springer-Verlag, LNCS 2397, pp. 213-227.
[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*: Addison Wesley, 1995.
[19] R. Green and A. Rashid, "An Aspect-Oriented Framework for Schema Evolution in Object-Oriented Databases", AOSD 2002 Workshop on Aspects, Components & Patterns for Infrastructure Software.
[20] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, "Extensible Query Processing in Starburst", Proc. SIGMOD Conf., 1989, ACM, pp. 377-388.
[21] Jasmine, *The Jasmine Documentation*, 1996-1998 ed: Computer Associates International, Inc. & Fujitsu Limited, 1996.
[22] W. Kim and H.-T. Chou, "Versions of Schema for Object-Oriented Databases", Proc. VLDB Conf., 1988, Morgan Kaufmann, pp. 148-159.
[23] Knizhnik, K.A., "Generic Object-Oriented Database System", http://www.ispras.ru/~knizhnik/goods/readme.htm, 2003.
[24] S. Monk and I. Sommerville, "Schema Evolution in OODBs Using Class Versioning", *ACM SIGMOD Record*, 22(3), pp. 16-22, 1993.
[25] D. Parsons, A. Rashid, A. Speck, and A. Telea, "A 'Framework' for Object Oriented Frameworks Design", Proc. TOOLS Europe, 1999, IEEE Computer Society Press, pp. 141-151.
[26] R. J. Peters and M. T. Ozsu, "An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems", *ACM Transactions on Database Systems*, 22(1), pp. 75-114, 1997.
[27] Y.-G. Ra and E. A. Rundensteiner, "A Transparent Schema-Evolution System Based on Object-Oriented View Technology", *IEEE Trans. Knowledge and Data Engg.*, 9(4), pp. 600-624, 1997.
[28] A. Rashid, *Aspect-Oriented Database Systems*: Springer-Verlag, 2003.
[29] A. Rashid, "Aspect-Oriented Programming for Database Systems", in *Aspect-Oriented Software Development (To Appear)*, M. Aksit, S. Clarke, T. Elrad, and R. Filman, Eds.: Addison-Wesley, 2004.
[30] A. Rashid, "A Database Evolution Approach for Object-Oriented Databases", IEEE International Conference on Software Maintenance (ICSM), 2001, IEEE Computer Society Press, pp. 561-564.
[31] A. Rashid, "A Framework for Customisable Schema Evolution in Object-Oriented Databases", International Data Engineering and Applications Symposium (IDEAS), 2003, IEEE, pp. 342-346.
[32] A. Rashid, "Weaving Aspects in a Persistent Environment", *ACM SIGPLAN Notices*, 37(2), pp. 36-44, 2002.
[33] A. Rashid and R. Chitchyan, "Persistence as an Aspect", 2nd International Conference on Aspect-Oriented Software Development, 2003, ACM, pp. 120-129.
[34] A. Rashid and P. Sawyer, "Aspect-Orientation and Database Systems: An Effective Customisation Approach", *IEE Proceedings - Software*, 148(5), pp. 156-164, 2001.
[35] A. Rashid and P. Sawyer, "A Database Evolution Taxonomy for Object-Oriented Databases", *Journal of Software Maintenance - Practice and Experience (To Appear)*, 2004.
[36] A. Rashid, P. Sawyer, and E. Pulvermueller, "A Flexible Approach for Instance Adaptation during Class Versioning", ECOOP 2000 Symp. Objects and Databases, 2000, LNCS 1944, pp. 101-113.
[37] D. Sjoberg, "Quantifying Schema Evolution", *Information and Software Technology*, 35(1), pp. 35-44, 1993.
[38] A. H. Skarra and S. B. Zdonik, "The Management of Changing Types in an Object-Oriented Database", Proc. OOPSLA Conf., 1986, ACM, pp. 483-495.