Persistence as an Aspect

Awais Rashid, Ruzanna Chitchyan
Computing Department, Lancaster University
Lancaster LA1 4YR
United Kingdom
+44-1524-592647

awais@comp.lancs.ac.uk
r.chitchyan@lancaster.ac.uk

ABSTRACT

Persistence - the storage and retrieval of application data from secondary storage media - is often used as a classical example of a crosscutting concern. It is widely assumed that an application can be developed without taking persistence requirements into consideration and a persistence aspect plugged in at a later stage. However, there are no real world examples showing whether persistence can in fact be aspectised and, if so, can this be done in a manner that promotes reuse and is oblivious[1] to the application. In this paper, we provide an insight into these issues drawing upon our experience with a classical database application: a bibliography system. We argue that it is possible to aspectise persistence in a highly reusable fashion, which can be developed into a general aspect-based persistence framework. Nevertheless, application developers can only be partially oblivious to the persistent nature of the data. This is because persistence has to be accounted for as an architectural decision during the design of data-consumer components. Furthermore, designers of such components also need to consider the declarative nature of retrieval mechanisms supported by most database systems. Similarly, deletion requires explicit attention during application design as mostly applications trigger such an operation.

Keywords

Persistence, aspect reuse, aspect-oriented programming, AspectJ, relational database application

1. INTRODUCTION

Aspect-Oriented Programming (AOP) [14] aims at providing systematic means for effective modularisation of crosscutting concerns. Some concerns such as synchronisation [10, 16] and tracing [11, 17] are often described as classical candidates for aspectisation. Persistence is also one such classical example [21, 31]. It is advocated that these concerns can not only be modularised using AOP techniques, this can be achieved with a high degree of reusability for the aspect code. Furthermore, the rest of the application can be developed oblivious to the fact that a synchronisation, tracing or persistence aspect may be composed at a later stage.

Despite the above claims and widespread use of database management systems in today's businesses, there are no real world examples (involving a significant number of data classes) available that might demonstrate whether:

- persistence can be effectively modularised using AOP techniques;
- persistence aspects can be reused;
- applications can be developed unaware of the persistent nature of the data.

Some existing work on AOP has considered persistence and related concerns. [25, 26], for example, describe an approach, and a prototype PersAJ, to store aspects in an object-oriented database. In order to keep the persistence model independent of a particular AOP approach, an aspect is used to describe the persistent representation of aspects. The focus is on providing a model for aspect persistence and persistence of application data has not been separated. Similarly, [27] presents an approach to store aspects in a relational database. Separation of persistence in relational database applications is not considered. [19], on the other hand, provides an assessment of AOP based on separating concurrency control and failure handling code in a distributed system. However, the aim of the case study is to investigate aspectisation of transactions which are only one facet of persistence. Modularisation of code dealing with storage and retrieval of application data from persistent storage is not dealt with in detail. Furthermore, the transactions considered operate in a pure object-oriented environment. This is seldom the case for database applications as relational databases claim almost 80% of the market share. [30] describes experiences with implementing persistence and distribution aspects with AspectJ. The focus of the work is on refactoring an existing application. It does not explore application development independent of persistence requirements or development of a reusable persistence aspect.

In this paper we present our experiences in separating persistence of application data using AOP techniques. Our general aim is to explore whether persistence can be effectively aspectised in a real world application. More specifically, we wish to determine whether such aspectisation can be reusable with the application and the persistence aspect developed independently of each other.

[1] Obliviousness here means that persistence requirements may be ignored during application development. Filman and Friedman [15] use the term obliviousness to indicate that no special hooks are needed in classes operated upon by an aspect. This is orthogonal to our use of this term.

We have chosen a classical database application: a bibliography system and SQL-92 compliant relational databases (as the underlying persistence mechanism) as the basis for our experiment. The application is written in Java with database interaction, based on JDBC (Java Database Connectivity), aspectised using AspectJ1.06 [1]. Based on this experience we argue that it is possible to aspectise persistence in a highly reusable fashion, which can be developed into a general aspect-based persistence framework. Nevertheless, application developers can only be partially oblivious to the persistent nature of the data. This is because persistence has to be accounted for as an architectural decision during the design of data-consumer components. Furthermore, designers of such components also need to consider the declarative nature of retrieval mechanisms supported by most database systems. Similarly, deletion requires explicit attention during application design as mostly applications trigger such an operation.

Although our experiences are based on AspectJ and relational databases, we also provide some general insight into the suitability of other AOP techniques in this context and discuss how the emerging persistence model may be adapted to suit other database technologies, e.g. object-oriented databases.

In the following, section 2 provides an overview of the bibliography application used as the basis for this discussion. Section 3 describes our approach to modularising persistence using aspects. Section 4 provides a discussion of the lessons learnt from our experience, their possible limitations and generalisation to other persistence scenarios. Section 5 discusses some related work while section 6 concludes the paper and identifies possible future directions.

2. BIBLIOGRAPHY APPLICATION

The data model for our bibliography application has been derived from information stored on the DBLP server [2]. However, it has been simplified as we do not need to maintain links from the table of contents to the articles or aggregate individual conferences in a series (e.g. the AOSD conferences) into a collection. The latter should not be a data model consideration anyway as it is more appropriate to define it as a view. The data model is shown in fig. 1 in UML.

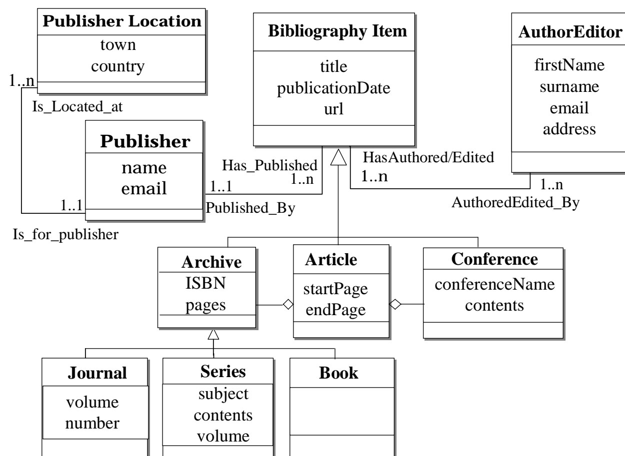

Fig. 1: Data model of the bibliography application

The various association and aggregation relationships in the data model have been implemented as aspects. Similarly, aspects have been employed in the bibliography user interface classes (developed using the Java Swing API) to attach listeners to the various GUI widgets. However, these aspects do not have a bearing on the modularisation of persistence. As discussed in section 3.2, the persistence mechanism employs Java reflection to discover the structure of a persistent object. Hence, it is of no consequence, at least at a conceptual level, whether the relationship edges are specified within a class or encapsulated in an aspect. It should be noted that, unlike the listener aspects, the design of some GUI components does need to take persistent nature of the data into account. We discuss these considerations in the next section.

3. MODULARISING PERSISTENCE

In this section we first describe our approach to modularising database access and the rationale behind the various design decisions. We argue that it is possible to remain oblivious of the need to store or update the data in the database during application development. However, components consuming the data need to account for both the data source and the nature of its retrieval mechanisms. Retrieval can, therefore, only be partially modularised. Similarly, deletion has to be explicitly triggered by the application. We then discuss the design of the SQL translation aspect before moving on to describing the general aspect-based persistence framework emerging from the bibliography application.

3.1 Database Access

There are three important considerations when aspectising database access for an application that has been, at least partly, developed without accounting for persistence:

1. A means to distinguish persistent data from transient data is required.
2. The aspectised database access functionality should have a high degree of reusability.
3. If the database access is reusable, some customisation points should be available to plug-in application requirements such as a specific database management system, location of the database and so on.

In order to distinguish persistent data we have borrowed the concept of a *persistent root class* from object-oriented database systems [9]. These systems often require that all classes whose instances are to be stored in the database extend a common base class. The base class contains some persistence-related functionality and additional functionality is augmented to the persistent classes by a pre or post compilation processor. The *PersistentRoot* class in our approach is shown in fig. 2. It only encapsulates a very basic yet essential feature: marking an object as deleted. This is essential as, due to the automatic garbage collection support in Java, transient objects do not have an explicit notion of deletion. Like retrieval, deletion can, therefore, not be completely ignored during application development. However, it can be simplified by providing this basic functionality within the *PersistentRoot* class with the database access functionality carrying out the actual data removal from the database (this is discussed in more detail later).

An application specific aspect can use AspectJ to declare the *PersistentRoot* class as the superclass of all classes whose instances are to be made persistent[2] (cf. fig. 3).

```
public class PersistentRoot {

    protected boolean isDeleted = false;

    public void delete(){
      this.isDeleted = true;
    }

    public boolean isDeleted() {
      return this.isDeleted;
    }
}
```

Fig. 2: The persistent root class

```
public aspect ApplicationDatabaseAccess {

    declare parents: (BibliographyItem ||
                      AuthorEditor ||
                      Publisher ||
                      PublisherLocation)
            extends PersistentRoot;

    // other code
}
```

Fig. 3: An aspect declaring *PersistentRoot* as the superclass of classes with persistent instances

The *PersistentRoot* class also plays a fundamental role in aspectising database access in a highly reusable fashion. While the *DatabaseAccess* aspect in fig. 4 employs the notion of abstract aspects and pointcuts from AspectJ, the high degree of reusability is derived from the ability to define join points with reference to a common, application independent point: the *PersistentRoot* class. This makes it possible to reuse the *DatabaseAccess* aspect in another application whose data classes have been declared as subclasses of the *PersistentRoot* class (either by means of an aspect or by using the standard Java inheritance mechanisms).

We now describe the various key features of the *DatabaseAccess* aspect labelled in fig. 4 in more detail.

(A) **Connection.** The ability to connect and disconnect from the database is a basic feature for a persistent application. However, as mentioned earlier, reusability requirements dictate that such functionality is generic with the availability of specific customisation points to incorporate application specific requirements such as:

- the location of the database;
- the database management system and/or driver to be used;
- points in the application control flow where a database connection should be established or closed.

In the *DatabaseAccess* aspect the above needs are addressed through the use of two abstract pointcuts and two abstract methods. The two abstract methods are invoked by a *before* advice, operating on the abstract pointcut *establishConnection*, to obtain information to connect to the database (the two static variables are used to hold the connection information). The database URL and driver details are supplied by an application aspect extending the *DatabaseAccess* aspect and concretising the two methods. Such an aspect also concretises the two abstract pointcuts to specify the join points in the application control flow where database connections are to be established or closed. For our implementation we have chosen to use the *ApplicationDatabaseAccess* aspect in fig. 3 for the purpose. Note that at present we do not implement any connection pooling. This can, however, easily be incorporated into the *DatabaseAccess* aspect with localised impact.

```
public abstract aspect DatabaseAccess {

     private static Connection dbConnection;
     private static String dbURL;

     abstract pointcut establishConnection();
(A)  abstract pointcut closeConnection();

     public abstract String getDatabaseURL();
     public abstract String getDriverName();

     pointcut trapInstantiations(): call(PersistentRoot+.new(..));

     pointcut trapUpdates(PersistentRoot obj):
(B)          !cflow(call(public static Vector
                   SQLTranslation.getObjects(ResultSet, String))) &&
             (this(obj) &&
             execution(public void PersistentRoot+.set*(..))
             );

     pointcut trapRetrievals( ):
(C)                call(Vector PersistentData.get*(..));

     public static PersistentData getPersistentData() { … }

     pointcut trapDeletes(PersistentRoot obj): this(obj) &&
                   execution(public void PersistentRoot+.delete());

(D)  pointcut detectDeletedObjects(PersistentRoot obj): this(obj) &&
                   (execution(public * PersistentRoot+.get*(..)) ||
                    execution(public * PersistentRoot+.set*(..))||
                    execution(public String PersistentRoot+.toString())
                    );

     protected static Integer update(String sqlStatement)
                                            throws SQLException { … }

(E)  protected static Vector retrieve(String sqlStatement, String className)
                                            throws SQLException { … }

     protected static Object transactionWrapper(String methodName,
                                                  Object[] params) { … }

(F)  public static aspect MetaDataAccess { … }

     // advice code
}
```

Fig. 4: The key features of the *DatabaseAccess* aspect

Note that there are a number of JDBC drivers available. These range from pure Java drivers to those that act as a bridge to an underlying ODBC (Open Database Connectivity) driver. The various features in the JDBC API, particularly those pertaining to database meta-data access, are not fully supported by all drivers. Consequently, to have a high degree of portability across drivers, we have chosen to base the implementation of the *DatabaseAccess* aspect on the basic Sun Microsystems JDBC-ODBC Bridge Driver which, to the best of our knowledge, offers the lowest common denominator in terms of supported functionality. While this has provided us with the flexibility of choosing a different driver and/or a database management system for our bibliography system in the future, most advanced features of a new driver would not be exploited without modifying the aspect code (though the change will be localised to the aspect). This reflects that, like most other programming approaches, such a trade-off needs to be considered when designing reusable aspects.

(B) **Storage and update.** The two pointcuts, respectively, identify the join points where an object should be stored in the database or its persistent representation updated. An object should be stored in the database as soon as it is instantiated (cf. the

[2] Note that the potential subclasses must inherit from *Object* as *PersistentRoot* does.

trapInstantiations pointcut). However, two factors need to be considered when aspectising this functionality:

1. Once an object is stored in the database all objects reachable from it should also be made persistent. This is in line with well-known *persistence by reachability* requirement for object persistence [13] and ensures that the object and all its references can be appropriately re-established upon retrieval. Due to the underlying relational model, the objects are written to the database through translation to SQL *insert* statements. The enforcement of reachability semantics is, therefore, left to the *SQLTranslation* aspect (cf. section 3.2).
2. The object can only be stored in the database after its constructor has been executed. Naturally, an *after returning* advice is employed. However, in case of transaction rollback, the transient instantiation of the object is not automatically aborted. So, once the underlying *transactionWrapper* (discussed later) signals a rollback, the *after* advice must ensure that either an exception is raised or the transient copy is marked deleted (by invoking the *delete( )* method on the object) so that it may be detected as unusable (cf. pointcut in block Ⓓ). However, any exception has to be wrapped as an AspectJ *SoftException* because a Java *throws* clause is not currently supported for advices (with the exception of the *around* advice). In our opinion, it is essential to treat advices as first class entities in order to clarify the signature of the behaviour specified within an aspect. Since one can already supply arguments to advices in AspectJ in the same fashion as Java methods, it is only natural that features such as declaration of exceptions thrown from the advice code should be incorporated and more reflective access supported. As discussed later, such reflective access is fundamental in the development of reusable aspects.

The update mechanism relies on trapping all invocations of setter methods for persistent objects. However, if such invocations happen within the control flow of the *getObjects* method in the *SQLTranslation* aspect they are ignored. This method is used to rebuild the objects from their relational representation (which might span multiple tables due to the normalisation constraints in the relational model). Setter method calls in its control flow, therefore, are used to populate an empty copy of the object and, hence, do not have update semantics from a persistence perspective. A *before* advice is employed to ensure that the database state is updated prior to the transient object being modified. This makes it possible to abort the transient update (in a fashion similar to that described for instantiation) if a transaction is rolled back. Also, note that we have made the intentional decision to rely on strict encapsulation for access to member variables of persistent objects i.e. only setter and getter methods can be used and no direct public access otherwise is allowed. We are of the view that such good practice should be enforced for all persistent applications as it ensures that the interface of the class is not often modified due to changes to internal representation of member variables. However, if required, only the *trapUpdates* pointcut definition needs to be modified to trap direct updates to member variables.

The *trapInstantiations* and *trapUpdates* pointcuts do not require any special preparation on part of the application code instantiating the classes in fig. 1 or calling the setter methods on their instances. The developer can, therefore, remain oblivious to the fact the advices referring to these pointcuts will store the objects in the database or update their persistent representations.

Ⓒ**Retrieval.** Unlike storage and update, it is virtually impossible to remain oblivious of the persistent nature of the data during retrieval. The term “retrieval” means “to get and bring back; especially**:** to recover (as information) from storage” [5]. The application, therefore, cannot ignore the fact that the persistent objects (in this case instances of the classes in fig. 1) or the references to these are obtained from an external source. This is further compounded by the declarative nature of retrieval mechanisms in database systems which retrieve data based on predicates or selection conditions. Query languages remain the dominant retrieval mechanism. Although in object-oriented databases retrieval is possible through traversal, the Object Query Language (OQL) forms part of the ODMG standard [9] and either its implementation or a proprietary query language is supported by most commercial systems e.g. [4, 6, 7, 8]. Similarly, the Java Data Objects (JDO) specification [29] also supports a query language.

Despite the above observation, aspects can play an important role in modularising parts of the retrieval related code. In case of our application this is achieved through a special interface called *PersistentData* which offers a number of methods to be implemented by a class. The methods expose functionality such as retrieving the extent i.e. the set of all objects of a class or specific objects of a class based on a selection condition. All these methods return a Vector containing the objects retrieved. The *getPersistentData( )* method in the *DatabaseAccess* aspect provides a reference to an instance of a class implementing this interface. An application can obtain this reference and use it as the basis of any retrieval-related code.

The class implementing the *PersistentData* interface is used to provide hooks that are used by the *trapRetrievals* pointcut to identify the points at which the application attempts to retrieve the data. Note that this class is not application-specific and is a reusable, support mechanism for the *DatabaseAccess* aspect. An *around* advice for the *trapRetrievals* pointcut employs the AspectJ reflection API to obtain the various selection conditions passed as arguments to the hook methods. With the help of the *SQLTranslation* aspect it retrieves the objects and returns the resulting Vector to the application.

The modularisation approach described above provides a high degree of reusability for the retrieval code as it remains application independent. Although the application cannot remain unconcerned with retrieval, we consider this to be a positive conclusion. This is because retrieval is often an important architectural consideration in the design of data consumer components. The amount of data that will be available as a result of retrieval can, for instance, be a significant factor in the design of user interface components. Several of these user interface design considerations were encountered in our bibliography application. For instance, we support the user to relate an author already existing in the database to a new item being added. This is done by providing a list of existing authors when entering the new item details. Since the bibliography database contains thousands of authors, it is a very expensive operation to retrieve all the authors particularly when the user might not choose an existing author. Even if the user were to choose an existing author the operation remains expensive as only a few authors will be

selected. It, therefore, makes sense to provide the user with some mechanism to navigate through a set of lists containing subsets of authors (possibly based on alphabetical ordering) and retrieving (on demand) only the subsets of authors the user might be interested in. Retrieval considerations in this case make it possible to not only optimise the database operations but also provide a user interface presenting a large amount of data to the user in a manageable fashion.

Ⓓ **Deletion.** As mentioned earlier, like retrieval, deletion of persistent data has to be explicitly considered during application development and cannot be fully aspectised. This is because data has to be deleted from the data source upon specific request from the application. In addition, there are implementation specific factors. Due to the automatic garbage collection, there is no notion of explicitly deleting an object in Java. Consequently, there is no reference point available for the *DatabaseAccess* aspect to remove the persistent representation of an object from the database. In a language such as C++, the invocation of the *delete* operator can be trapped (using an aspect language) and the object removed from the database. Still one cannot be sure if the application actually intended to remove the object from disk or merely from the memory. It is, therefore, good practice to explicitly delete persistent objects hence providing a non-fuzzy point of reference on which the aspect can operate.

In case of our application this reference point is provided by the *delete( )* method in the *PersistentRoot* class (cf. fig. 2). The application invokes this method for the persistent instances (their classes are declared as sub-classes of the *PersistentRoot* class as shown in fig. 3). The *trapDeletes( )* pointcut captures these invocations and a *before* advice, for reasons similar to update, translates the request to SQL using the *SQLTranslation* aspect and removes the persistent representation of the object. It also marks the object for early collection by using the garbage collector interaction features in the java.lang.ref package. The *detectDeletedObjects* pointcut and its associated *before* advice complements the above functionality by throwing an exception (wrapped as an AspectJ *SoftException*) whenever a piece of code attempts to access the transient representation of a deleted persistent object that has not yet been collected by the garbage collector.

The use of the *delete( )* method in the *PersistentRoot* class as a reference point makes it possible to keep the deletion functionality reusable and application independent. However, the application programmer should be aware of the existence of the *PersistentRoot* class, its public interface and that it will eventually be declared as a super-class of all classes whose instances are to be stored in the database. This is essential as otherwise the application programmer will be calling a method that to him/her is unspecified and such a practice can lead to inconsistencies in the code. The application programmer does not need to be aware of the existence of the deletion functionality in the *DatabaseAccess* aspect or the *SQLTranslation* aspect.

Ⓔ **Transactions.** The three methods: *update*, *retrieve* and *transactionWrapper* together encapsulate the transaction functionality. This is because, although JDBC has an explicit (optional) notion of transaction commit and rollback, transactions are always implicitly started. The *update* and *retrieve* methods encapsulate the code that results in the start of read-write and read-only transactions respectively. Naturally the *update* method caters for operations that change the state of the database i.e. SQL *insert, update* and *delete* statements while the *retrieve* method supports querying operations on the database. The *sqlStatement* argument in both methods is obtained by the appropriate advice code through the *SQLTranslation* aspect. The *className* argument in the *retrieve* method is obtained reflectively by the advice operating on the *trapRetrievals* pointcut. Class name is an argument for all the methods in the *PersistentData* interface as it is required to establish the mapping between the object structure and the underlying relational schema (cf. section 3.2).

```
protected static Object transactionWrapper(String methodName,
                                           Object[] params) {

    try {

        boolean commitable = true;
        Object obj = null;

        try {
          Class thisClass = Class.forName("DatabaseAccess");
          Method[] methods = thisClass.getDeclaredMethods();
          Method theMethod = null;
          for (int i=0; i<methods.length; i++) {
              if (methods[i].getName().equals(methodName))
                theMethod = methods[i];
          }
          obj = theMethod.invoke(null, params);
        }

        catch (Exception e) {
          System.out.println(e.toString());
          dbConnection.rollback();
          commitable = false;
        }

        finally {
          if (commitable)
              dbConnection.commit();
          return obj;
        }
    }

    catch(SQLException e) {
        System.out.println("Error in committing or
                            rolling back: " + e.toString());
        return null;
    }
  }
```

Fig. 5: The transaction wrapper method

The various advices within the *DatabaseAccess* aspect do not directly invoke the *update* or the *retrieve* method. Instead they pass the name of the method to be invoked together with an array of arguments to the *transactionWrapper* method (cf. fig. 5). This helps us modularise the nested try-catch blocks as otherwise these have to be repeated in individual advice code. The outer try-catch block is responsible for catching any SQL exceptions (thrown by JDBC) during the invocation of the *commit* and *rollback* methods. The inner try-catch-finally block in the *transactionWrapper* method reflectively invokes the required method. It uses a mechanism similar to that presented in [19] i.e. a boolean variable to decide whether to commit the transaction or rollback. Note that we choose to abort a transaction when any exception is thrown regardless of whether it arises from reflective access or the database operation. We have taken this safer option intentionally as reflective operations play a fundamental role during translation to/from SQL. Consequently, it is highly likely that any exception directly or indirectly relates to database access. Furthermore, all database access functionality, though at times not oblivious, is aspectised. Therefore, the application does not need to signal exceptions to abort transactions as these are signalled by the aspecitsation infrastructure: JDBC, the Java Reflection API or the

SQLTranslation aspect. A null value returned to the invoking advice implies an unsuccessful transaction prompting it to execute transient rollback and signal an exception (wrapped as an AspectJ *SoftException*). Unlike [19] where an *around* advice is employed for transaction wrapping, we have chosen to explicitly invoke the *transactionWrapper* from the advice code dealing with storage, update, removal and retrieval of persistent objects. As a result, the *transactionWrapper* is triggered strictly for database operations and no unnecessary wrapping overheads exist for transient operations. In this case the fact that the transactions do not operate in a pure OO environment benefits our aspect design. For database operations reflection, of course, adds some overhead to the transaction. However, some locking optimisation is provided by the *update* and *retrieve* methods which establish the appropriate read-write and read-only locks respectively.

Ⓕ **Meta-data Access.** This static inner aspect encapsulates helper functionality to access the database meta-data such as the column names in a relational table or its foreign key links. This functionality is required by the *SQLTranslation* aspect. The *MetaDataAccess* aspect, therefore, serves two purposes.

1. It avoids unnecessary duplication of JDBC meta-data calls during SQL translation. For example, in our case, one of the features encapsulated by the aspect is obtaining a JDBC *ResultSet* object containing the column names for a table and returning them as an *Enumeration* for ease of traversal during SQL translation.
2. If a desired meta-data access feature is not supported by the underlying database driver, it can be built on top of more primitive features available. An update can then be carried out without affecting the SQL translation functionality once a newer version of the driver or a better driver becomes available.

Note that meta-data access functionality should be viewed as a subset of the overall database access functionality. Its modularisation as an inner aspect of the *DatabaseAccess* aspect, therefore, provides a more natural separation of concerns than it being encapsulated in a sub-aspect. This argument is further strengthened by the observation that, in our *MetaDataAccess* aspect, we have not needed to concretise or override any features of the *DatabaseAccess* aspect.

3.2 SQL Translation

SQL translation must be considered as a separate concern when aspecitising persistence of OO data using relational databases. This is because database access (and in general persistent storage access) is a concern for any application involving persistent data. However, it is not necessary that any translation to the underlying data model will be required e.g. if an object-oriented database is being used. When an OO application employs a relational database as a persistent store, there is a need to flatten the object structure to a relational model due to the lack of support for complex data types in the latter. Fig. 6 shows part of the relational database structure for our bibliography application. The *Article* objects are mapped to two tables, one capturing the attributes defined in the superclass *BibliographyItem* while the other containing those defined within the *Article* class itself. The inheritance relationship is captured by a simple one-to-one relationship (for each *Article* object there can be only one row in each table). The many-to-many relationship between bibliography items and authors/editors is captured in a separate relational table (this is a defacto mechanism for capturing many-to-many relationships).

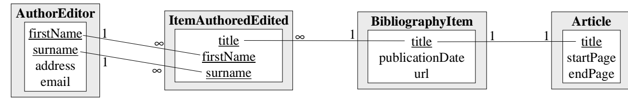

Fig. 6: Part of the OO data model mapped to the relational model

There is a need for an intermediate mechanism to provide the object-to-relational mapping and SQL translation provides a standard-based approach for the purpose. One might argue that, for update and deletion purposes, JDBC *ResultSet* objects may be employed to modify the database instead. However, not all JDBC drivers support use of bi-directional cursors on result sets. This is an essential feature to search for records pertaining to an object within a *ResultSet*. Also, this requires retrieving the object into a *ResultSet* and applying the update which results in unnecessary disk access. The *SQLData* interface in JDBC, on the other hand, only supports mapping to/from user-defined SQL types in an object-relational model and, hence, cannot be employed for pure relational databases.

If an SQL translation mechanism is to be reusable, it must be highly independent of any application-specific mapping. Such mapping can then be specified when the aspectised persistence mechanism is composed with the application. In our approach a singleton lookup table is used to establish the mapping. We minimise the use of the lookup table by only maintaining mapping at a coarse-granularity i.e. the tables to which objects of a class and many-to-many relationships map. Mapping of individual object attributes onto relational table columns is not maintained in the lookup table and is instead achieved through the use of identical names. However, if different naming schemes are being used, the mapping can be contained within the lookup table. The mapping in the lookup table is specified through the *EstablishMapping* aspect (cf. fig. 7) which sets up the mapping *before* the connection with the database is established Ⓑ. An *AuthorEditor* object maps onto the *AuthorEditor* table Ⓒ, an *Article* object maps onto the *BibliographyItem* and *Article* tables Ⓓ while the many-to-many *authorsOrEditors* relationship maps onto the *ItemAuthoredOrEdited* table Ⓔ. Note that the *EstablishMapping* aspect must dominate (have higher execution priority than) the *DatabaseAccess* aspect Ⓐ to ensure that the mapping is established before connecting to the database.

The main features of the *SQLTranslation* aspect are shown in fig. 8. The *sqlExecution* pointcut captures the fact that an object might map to multiple tables and hence result in translation to multiple SQL statements. An *around* advice tests if a single SQL statement is being executed through the JDBC *Statement* object in which case the normal execution in the *DatabaseAccess* aspect proceeds. On the other hand, if multiple SQL statements are found, execution is carried out in batch mode (JDBC has specific support for the purpose). Note that mapping to multiple SQL statements is an SQL translation concern and, hence, the pointcut dealing with this must form part of the corresponding aspect. Although the *sqlExecution* pointcut captures *Statement.executeUpdate(String)* calls from a single method (the *update* method) in the *DatabaseAccess* aspect, it makes it possible to separate an essential piece of SQL translation functionality and incorporate it

within the *SQLTranslation* aspect. Its use is, therefore, not out of step with good aspect-oriented programming practices.

The various *getXXXSQL* methods and the *getObjects* method employ Java Reflection and the mapping information in the lookup table to map the objects, their updates and deletion to the database and recreate the objects upon retrieval. Since strict encapsulation is imposed, we recursively identify the object attributes corresponding to the relational table columns by obtaining the declared members and not just the public ones. If propagation of updates for linked tables is supported in the underlying database design, this feature is exploited otherwise the linked tables are individually updated, but within a single transaction boundary to ensure consistency. Note that the use of reflection for object-to-relational mapping results in some additional overhead during database interaction. As pointed out in [22] such trade-offs have to be made when designing highly flexible components such as the *SQLTranslation* aspect.

```
(A) public aspect EstablishMapping dominates DatabaseAccess {

(B)   pointcut setupMapping():
              ApplicationDatabaseAccess.establishConnection();

      before(): setupMapping() {

        LookupTable mappingTable = LookupTable.getLookupTable();

(C)     mappingTable.createClassToTableMapping("AuthorEditor",
                                               "AuthorEditor");

(D)     mappingTable.createClassToTableMapping("Article",
                                               "BibliographyItem");
        mappingTable.createClassToTableMapping("Article",
                                               "Article");

(E)     mappingTable.createRelationshipToTableMapping(
                                             "authorsOrEditors",
                                             "ItemAuthoredEdited");

        ...

      }
  }
```

Fig. 7: Aspect used to specify the object-to-relational mapping

```
public aspect SQLTranslation {

  pointcut sqlExecution(Statement statement,
                        String sqlStatement):
                        target(statement) &&
                        call(public int
                           Statement.executeUpdate(String)) &&
                        args(sqlStatement);

  // around advice for sqlExecution pointcut

  public static String getInsertionSQL(PersistentRoot obj);

  public static String getUpdateSQL(PersistentRoot obj,
                                    String methodName,
                                    Object arg);

  public static String getDeleteSQL(PersistentRoot obj);

  public static String getQuerySQL(String className,
                                   String selectionCondition);

  public static Vector getObjects(ResultSet rs,
                                  String className);

  // helper methods

}
```

Fig. 8: The main features of the *SQLTranslation* aspect

3.3 The Emerging Persistence Framework

Based on the discussion in section 3.1 and 3.2 we can observe a general aspect-based persistence framework emerging. This framework is shown in UML in fig. 9. Members have been omitted from the classes, aspects and the interface for simplicity.

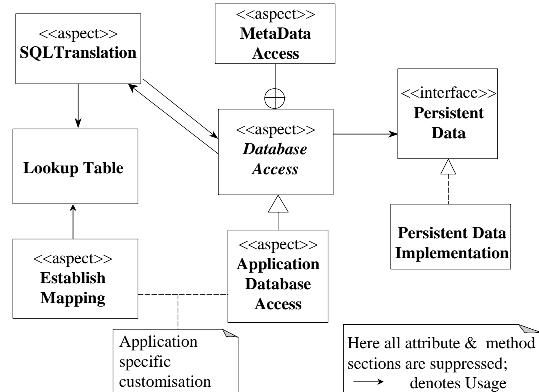

Fig. 9: The persistence framework emerging from the application

The above framework challenges one of the widely misunderstood promises of AOP. It is often assumed that an aspect implies a large piece of code modularised by a single AspectJ-like *aspect* construct. This is not true but for the simplest of cases. As shown in fig. 9, aspectisation requires that a coherent set of modules including classes and *aspects* collaborate to modularise a crosscutting concern. Such a view of AOP also ensures that aspectisation is not forced and in fact leads to a natural separation of concern e.g. the separation of the *DatabaseAccess* and *SQLTranslation* aspects in our persistence framework. Furthermore, it makes it possible to draw upon established best practices and guidelines from the frameworks community, as has been the case for flexibility trade-offs in our aspectisation.

4. DISCUSSION

4.1 Using other persistence mechanisms

The persistence framework shown in fig. 9 has emerged from a classical relational database application. It can, therefore, be reused in any other OO application employing an SQL-92 compliant relational database. For object-relational databases employing SQL-3, the framework implementation should still be reusable. However, the SQL translation mechanism will need to be enhanced to cater for the user-defined types in SQL-3. One option is to exploit the *SQLData* interface in JDBC which provides support for such mapping. As far as object-oriented databases are concerned the framework will need to be re-implemented. However, the persistence model used by the framework can still be exploited. The various pointcuts in the *DatabaseAccess* aspect will be required as these are the points in the application control flow where persistence features are composed regardless of the nature of the persistence mechanism. Similarly, a transaction wrapper will be needed and a *PersistentData* interface to support declarative access from the application. Of course, the SQL translation infrastructure (the *SQLTranslation* aspect, lookup table and *EstablishMapping* aspect) will not be required as there will be no data model mismatch between the OO application and the database. The *MetaDataAccess* aspect will not be needed either as it is only

needed to support SQL translation. The *PersistentRoot* class will be required to act as a surrogate inheriting from the object database system's *root* class which cannot be modified due to proprietary restrictions. This approach has worked successfully when designing aspect persistence mechanisms in the past [25, 26].

4.2 Reflection and other AOP techniques

In our persistence framework, reflection has played a fundamental role in the design of a reusable transaction wrapper and, more importantly, SQL translation mechanism. We have exploited not only the Java reflection API but also the AspectJ reflection API for the purpose. AspectJ pointcuts are mainly used to register points of interest in the application or persistence framework control flow. Genericity is provided by the use of reflection whenever application specific code would be required otherwise. This strengthens the argument for a hybrid approach to separation of crosscutting concerns [24]. It also implies that if the resulting persistence framework were to be implemented in another language environment, both the base language and the aspect language would need to support reflection.

While the use of reflection during aspectisation has led to the emergence of a generic, reusable framework it has certain drawbacks as well. Most of these relate to the SQL translation. For instance, the SQL translation mechanism makes the well-defined assumption that strict encapsulation is enforced and only getter and setter methods will be used to provide public access to an object's state. However, programmers might choose to ignore this assumption or simply might forget to include appropriate getter/setter methods. This will result in the translation mechanism becoming inoperative as methods are discovered dynamically and static checks cannot be applied. This risk can be reduced by providing support for generating getter and setter methods. Alternatively, the AspectJ *declare error* feature can be used to force programmers to define these methods.

Reflective invocation, of course, has its performance penalties. These can be counter-balanced by the use of a cache. In fact, a cache becomes an essential concern as the size of the database grows. The various pointcuts and, the *update* and *retrieve* methods within the *DatabaseAccess* aspect provide excellent reference points for plugging a cache into the persistence framework. However, the introduction of a cache should only be considered for applications with a large database as it is likely to result in unnecessary overheads otherwise. The nature of the application must also be taken into account e.g. whether it is necessary to optimise retrieval (due to frequent querying and infrequent updates as is the case for the bibliography application) or updates or both.

The AOP model in AspectJ is well suited for aspectising persistence. This is because persistence is a general concern regardless of the individual state of an object. Therefore, the extent-oriented nature of AspectJ pointcuts and advices is very useful in this context. Relationships, on the other hand, are an entirely different matter. In our application these have been implemented as aspects mainly relying on AspectJ *introductions*. As mentioned earlier conceptually this should be of no consequence. This is, unfortunately, not the case and the use of introductions has introduced additional overhead during SQL translation as the reflectively obtained attributes of an object have to be tested to check whether they are collections. If so the information in the lookup table is used to establish if a collection represents an edge of the relationship. This is further complicated by the fact that the AspectJ weaver renames the introduced edges to avoid conflicts. When reflectively accessed the name of the relationship is different from what has been specified in the mapping. The translation mechanism, therefore, requires knowledge of the renaming scheme making it susceptible to breakdown as the language and its weaver evolve. Based on experience with relationships in the bibliography application, we are of the view that introductions must be used with great caution especially if their use results in loss of semantic information. In case of relationships in the bibliography application a well-defined relationship model such as the one proposed in [28] with relationship aspects attached on a per-instance basis using composition filters as in [24] would have been more suitable. This, in turn, indicates the need for environments that allow multiple AOP techniques and platforms to co-exist hence allowing the use of the most appropriate technique for modularising a particular crosscutting concern.

4.3 Aspect Interaction

The *dominates* construct in AspectJ has been sufficient to resolve the simple interaction between the *EstablishMapping* and *DatabaseAccess* aspects. However, if a cache is plugged in we expect the interactions to become more complex as the advices in the caching aspect will operate with reference to the same pointcuts as in the *DatabaseAccess* aspect. Furthermore, several aspects are often composed only for development purposes and can introduce more interactions. For example, during the development of our bibliography application we employed a *Tracing* aspect for debugging purposes. Another aspect used was the *Extent* aspect which maintained transient extents for persistent classes to allow testing of the storage and update features without having to design the retrieval mechanism. The two testing aspects interacted with the advices in the *DatabaseAccess* aspect. However, the domination relationship among the three aspects was highly dynamic. For example, at times it was desirable to start displaying tracing information before the advice performing a database operation while at other times it was required to begin tracing output afterwards. Similarly, sometimes the *Extent* aspect was required to be compiled-in and vice versa. This required a lot of changes to the *dominates* relationship among these aspects. It became quite clear that, even for a system with few aspects and classes, such an interaction resolution model could be error-prone and cumbersome. Since interactions cut across aspects in a system, it is essential that AOP techniques in general (and not just AspectJ) offer significant support for the detection, modularisation and resolution of interactions. This support will play a fundamental role in the testing and verification of aspect-oriented applications and hence act as a critical factor in large-scale adoption of aspect-orientation.

5. RELATED WORK

Kienzle and Guerraoui [19] present an analysis of using AspectJ for separating transactions based on the OPTIMA framework [18]. They argue that using an AspectJ *around* advice to wrap transactions around transactional methods is inefficient due to lack of locking optimisations. They also discuss the dangers of using exceptions to signal transaction abortion in a multi-

threaded, distributed environment. They conclude that transactions should be part of the phenomenon simulated by objects. During our aspectisation of persistence we have also considered transactions. However, the JDBC transaction model is much simpler than OPTIMA and fewer factors need to be considered for providing an optimal transaction mechanism. Firstly, due to the implicit start of transactions in JDBC, the *transactionWrapper* is explicitly invoked from advices manipulating persistent data. Consequently, although transactions are not part of the application phenomenon, the code dealing with persistent objects (in the *DatabaseAccess* aspect) is not unaware of the existence of transaction boundaries. This also means that transactions are only wrapped around operations that result in database access hence avoiding unnecessary overheads. Moreover, the existence of the *update* and *retrieve* methods reflectively invoked by the transaction wrapper provides some degree of locking optimisation. The advices in our *DatabaseAccess* aspect also use exceptions (wrapped as an AspectJ *SoftException*) to indicate aborted transactions. We are of the view that handling of such exceptions is a concern for the integration stage within the application development process and can be dealt with by exception handling aspects as in [30].

Soares et al. [30] describe their experiences with AspectJ as a means for refactoring distribution and persistence concerns in a layered web-based information system. The data model for this application is much simpler than our bibliography application. Furthermore, most of their persistence aspects are application specific and highly reliant on the layered architecture. The persistence framework emerging from our application is not bound to a particular architecture and can be reused directly in any relational database application. Soares et al. widely employ interfaces to limit the dependence of aspects on the signatures of the methods (implemented in specific classes) that the aspects advise. While this may be a convenient way of aspect – class decoupling, it has also led to code duplication within interfaces as is the case for the hierarchy of transactional interfaces provided for locking optimisation. Moreover, these transactional interfaces are application specific and also result in duplication of code within the transaction aspect. In our persistence framework only a single interface with application-independent methods is employed in order to expose retrieval functionality to the application. Our locking optimisations are generic and, due to the reflective transaction wrapper, new optimisations can be introduced without duplication of transactional code.

A generic persistence aspect has also been implemented within JAC [23]: a framework for dynamic aspect-oriented programming in Java. Like [30] the architecture used for this aspectisation is also based on the existence of an additional layer between the persistent storage and JAC. No large applications (comparable to the bibliography application in this paper) using the JAC persistence aspect are yet available.

Neither JAC nor Soares et al. have considered issues of data normalisation while mapping objects to relational databases. As discussed earlier in this paper, normalisation brings a new level of complexity to persistence modelling with additional considerations such as mapping and retrieving an object from multiple tables. Furthermore, both of the above approaches have not identified SQL generation as a crosscutting concern. This has resulted in SQL statements being spread throughout components supporting the persistence aspect implementation. In [30] application-specific SQL statements have been hard-coded into the persistence code. Any changes to the database structure will, therefore, result in a ripple effect on a large portion of the persistence code.

[20] describes a simple database application where aspects are employed for authentication, exception handling, caching, pooling and so on. Storage and retrieval of application data has not been aspectised and SQL statements are hard-coded.

The work presented in this paper also bears a relationship with the notion of aspect-oriented frameworks e.g. [12]. However, unlike [12] which describes a general AOP framework, the framework emerging from our application is specific to the persistence domain.

6. CONCLUSION

This paper has presented our experience in aspectising persistence in a classical database application: a bibliography system. Our general aim was to explore whether AOP techniques offer an effective means to modularise persistence in a real world application scenario. The discussion in the paper demonstrates that the answer is indeed “yes”. However, like all other pieces of software, the designers of aspects also need to consider a number of software engineering factors. Firstly, trade-offs between genericity and performance need to be made. In our aspectisation, we could have hard-coded the application-specific SQL statements in the *SQLTranslation* aspect instead of using reflection. However, this would have seriously compromised the genericity and reusability of the SQL translation mechanism and, hence, the aspectised persistence mechanism. Secondly, a well-engineered aspect requires one to evaluate the suitability of the available techniques for implementing the various concerns within the aspect. For instance, we have employed AspectJ constructs to identify points where persistence-related behaviour has to be composed while reflection has been used to keep the SQL translation generic and avoid duplication of transaction code during database access. However, our experience also shows that the choice of suitable techniques is also constrained by the available set of tools and their interoperability. Ideally, we would have liked to implement our relationships using the composition filters approach. However, given the available tools and their interoperability constraints, we had to employ AspectJ introductions for the purpose.

We also aimed to answer two specific questions with the help of our experiment. Firstly, we wished to explore whether a persistence aspect can be developed that exhibits a high degree of reusability. The persistence framework emerging from our application demonstrates that this is indeed the case. Furthermore, this framework does not rely on the existence of an additional layer masking the relational database features. The framework is very simple to adapt and reuse i.e. concretise the *DatabaseAccess* aspect, specify the *EstablishMapping* aspect and use the *PersistentData* interface for retrieval purposes. However, for effective reuse such a framework (and aspects in general) should be complemented by a reuse specification. Such a reuse specification should clearly define the interface of an aspect’s behaviour e.g. the exceptions the various advices might throw. This is essential as the integration phase in the development process needs to specify behaviour to respond to any exceptions

raised by the advices hence improving the soundness of the composition.

Our second specific aim was to investigate whether an application and a persistence aspect could be developed independently of each other. As far as the application is concerned this can only be partially achieved. Storage and update of persistent data does not need to be accounted for but retrieval and deletion must be explicitly considered. However, this does not compromise the independent development or reusability of the aspect. While we took into account the need to expose retrieval and deletion functionality to the application during the course of developing our persistence aspect, we did not consider any specific implementation details of the application. Consequently, we had to design the persistence mechanism to be generic resulting in a highly reusable persistence framework. It is also interesting to point out that we did not set out to design a persistence framework. We followed the natural separation of concerns while developing the persistence infrastructure keeping the reusability and application independence requirements in mind and the framework naturally emerged.

Our future work will focus on putting the reusability of our aspectisation to test in other application contexts. Performance comparison with non-AO techniques and AO implementations such as the persistence aspect in JAC are also planned. We also aim to explore the effectiveness of Hyper/J [3] to aspectise persistence. This will be an interesting direction as the AOP model of Hyper/J differs considerably from that of AspectJ. The implementation of persistence in a real world application with the two techniques will, therefore, provide exciting opportunities for a thorough comparison.

7. REFERENCES

[1] Xerox PARC, USA, "AspectJ Home Page", http://aspectj.org/, 2002.

[2] Ley, M., "DBLP: Digital Bibliography and Library Project", http://dblp.uni-trier.de/, 2002.

[3] IBM Research, "Hyperspaces", http://www.research.ibm.com/hyperspace/, 2002.

[4] *The Jasmine Documentation*, 1996-1998 ed: Computer Associates International, Inc. & Fujitsu Limited, 1996.

[5] Merriam-Webster, "Merriam-Webster Online Dictionary", http://www.m-w.com/, 2002.

[6] *The O2 System - Release 5.0 Documentation*: Ardent Software, 1998.

[7] *Object Store C++ Release 4.02 Documentation*: Object Design Inc., 1996.

[8] *POET 5.0 Documentation Set*: POET Software, 1997.

[9] 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.

[10] S. Clarke, "Designing Reusable Patterns of Cross-Cutting Behaviour with Composition Patterns", OOPSLA Workshop on Advanced Separation of Concerns, 2000.

[11] S. Clarke and R. J. Walker, "Composition Patterns: An Approach to Designing Reusable Aspects", ICSE, 2001.

[12] 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.

[13] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems (3rd ed.)*: Addison-Wesley, 2000.

[14] T. Elrad, R. Filman, and A. Bader (eds.), "Theme Section on Aspect-Oriented Programming", *CACM*, 44(10), 2001.

[15] R. Filman and D. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", OOPSLA Workshop on Advanced Separation of Concerns, 2000.

[16] D. Holmes, J. Noble, and J. Potter, "Towards Reusable Synchronisation for Object-Oriented Languages", ECOOP Workshop on Aspect-Oriented Programming, 1998.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. A. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ", ECOOP, 2001, Springer-Verlag, LNCS 2072, pp. 327-353.

[18] J. Kienzle, "Open Multi-threaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming", PhD Thesis, Swiss Federal Institute of Technology, 2001.

[19] J. Kienzle and R. Guerraoui, "AOP: Does It Make Sense? The Case of Concurrency and Failures", ECOOP, 2002, Springer-Verlag, LNCS 2374, pp. 37-61.

[20] I. Kiselev, *Aspect-Oriented Programming with AspectJ*: SAMS, 2002.

[21] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales, "Aspect-Oriented Programming Workshop Report", ECOOP Workshop Reader, 1997, Springer-Verlag, LNCS 1357.

[22] D. Parsons, A. Rashid, A. Speck, and A. Telea, "A 'Framework' for Object Oriented Frameworks Design", TOOLS Europe, 1999, IEEE CS Press, pp. 141-151.

[23] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin, "JAC: A Flexible Solution for Aspect-Oriented Programming in Java", Reflection Conf., 2001, Springer-Verlag, LNCS 2192, pp. 1-24.

[24] A. Rashid, "A Hybrid Approach to Separation of Concerns: The Story of SADES", Reflection conf., 2001, Springer-Verlag, LNCS 2192, pp. 231-249.

[25] A. Rashid, "On to Aspect Persistence", GCSE Symp., 2000, Springer-Verlag, LNCS 2177, pp. 26-36.

[26] A. Rashid, "Weaving Aspects in a Persistent Environment", *ACM SIGPLAN Notices*, Feb. 2002.

[27] A. Rashid and N. Loughran, "Relational Database Support for Aspect-Oriented Programming", Proceedings of NetObjectDays, 2002 (to appear in Springer-Verlag LNCS).

[28] A. Rashid and P. Sawyer, "Dynamic Relationships in Object Oriented Databases: A Uniform Approach", DEXA, 1999, Springer-Verlag, LNCS 1677, pp. 26-35.

[29] R. Roos, *Java Data Objects*: Addison Wesley, 2002.

[30] S. Soares, E. Laureano, and P. Borba, "Implementing distribution and persistence aspects with AspectJ", OOPSLA, 2002, ACM Press, pp. 174-190.

[31] J. Suzuki and Y. Yamamoto, "Extending UML for Modelling Reflective Software Components", International Conference on the Unified Modelling Language (UML), 1999.