Using Aspect-Orientation to Add Persistency to Applications

Uwe Hohenstein

CT SE 2
Siemens AG
Otto-Hahn-Ring 6
81730 München
Uwe.Hohenstein@siemens.com

Abstract: This paper presents a comfortable and customizable persistence framework that supports the access of relational databases from Java applications. In order to keep the effort for implementation low, we show how to benefit from aspect-orientation. The framework has advantages over existing mapping tools: Flexibility is higher as the functionality can be freely designed. We demonstrate flexibility by putting emphasis on flexible mapping strategies.

1 Introduction

The classical way to make relational data accessible from a programming language is to embed SQL statements. Java applications can use JDBC and SQLJ to this end. But programmers are then faced with the problem of bridging the semantic gap between object-oriented programming and relational concepts such as tables and SQL. Data exchange between Java and SQL requires special concepts such as host variables and cursors, making application programs difficult to write and hard to read.

From the point of programming, it is desirable to store and retrieve objects of the application in an easy manner. To support this goal, several object/relational (O/R) mapping tools came up, for which Java Data Objects (JDO) defines a common standard interface. Those products start with an object model describing the data to be stored in the database. The object model is then automatically mapped onto relational tables using some predefined strategies. Moreover, an object-oriented persistence layer on top of the relational database system (RDBS) is provided that breaks down objects into tuples accordingly. This layer now provides an interface that allows storing, retrieving and deleting objects of the object model, independently of how they are represented in tables. The high-end of those tools more or less emulate object-oriented DBSs (ODBSs) on top of relational ones. But they can certainly not compete with the performance of real ODBSs, if typical traversals from object to object are performed.

Enterprise JavaBeans (EJB) application servers behave similarly. However, the object model supports only few concepts such as classes (beans) and relationships. They even go one step further allowing one to develop applications without taking notice of persistence, transactions etc. In the mode of Container-Managed Persistence and

Transactions, a container manages persistence and transactions, the configuration of which is purely done outside by means of an XML deployment descriptor.

All these approaches are useful as they make programming relational database applications easier by hiding underlying relational DBS technology. Nevertheless, a user's exertion of influence is sometimes too low, as the layer is mostly a black box. That is, the layer possesses certain functionality, or it does not. Either the performance is sufficient, or is not. One has to live with the layer as it is, there are no possibilities to improve performance, to include new features such as advanced querying or transactions, or the other way round, to remove needless functionality. In these cases, it is inevitable to implement a persistence layer by one's own [KC98]. [He98,SHW98] vote for hand-made persistence layers and discuss important aspects of defining scaleable object-persistence layers in more detail. Such a layer provides better adaptability and flexibility, but requires more effort for implementation. [SHW98] estimates the effort for developing a persistence layer at 30% of the planned resources!

Another problem is accessing existing databases with these tools. The tools operate in a top-down manner: An object model is mapped to tables according to some strategies. Having existing tables means that the internal strategies must be understood and turned up. The choice of the object model then depends on achieving certain tables, and the object model cannot freely be defined to be adequate for the application logic. The same problem occurs if table structures are changed due to performance reasons. Unfortunately, changing the table structure and reorganising the accesses possess the best potential for improving the performance.

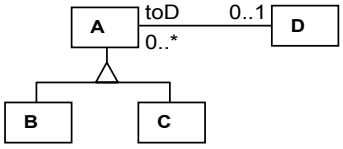
In this paper, we describe how to reduce the effort for developing application-specific persistence layers for relational databases by taking benefit from the new technology of aspect-orientation (AO). AO is promising as it provides systematic means for effective modularization of crosscutting concerns [EFB01]. Such concerns like synchronization and tracing are often described as classical candidates for aspectization. Recent research has shown usefulness: [KG02] uses AO to separate concurrency control and failure handling code in a distributed system. The book of Rashid [Ra04] handles with Aspect-Oriented Databases. It gives an overview about ongoing research and discusses all facets of AO in the context of databases: AO to implement DBSs in a more modularized manner, persistence for aspects, and some ideas on a persistence framework.

We here report on our experience using AspectJ [Ki01,La03], a general-purpose aspect-oriented extension of Java, to implement persistence aspects. We present a simple persistence layer with an object-oriented Java interface in Section 2. Section 3 shows that AspectJ can make the dream come true, to add persistency after having developed the real business logic – without changing existing code. The main advantage of our approach over mapping tools is the higher degree of customization, full control over the code, modularization, and flexibility with regard to mapping strategies and transactions. Section 4 presents the details of handling mapping strategies.

In contrast to other work such as [RC03,SLB02], we provide a deeper elaboration on a useful framework. We put emphasis on flexible mapping strategies to avoid deficiencies of other mapping tools in our paper. As already explained, strong mapping features are very important for accessing existing tables and for tuning databases.

2 Object Model and Sample Application

Our persistence framework follows the principle of JDO and other Java-based object/relational mapping tools, taking Java as a persistent model. The framework then deals with the storage and retrieval of Java objects, underlying table structures and SQL become invisible. The following database schema will form the basis for our discussion:



Class `A` possesses two subclasses `B` and `C` and an n:1 relationship `toD` to another class `D`. For the ease of demonstration, the attributes of theses classes are single-valued. The following is now a sample Java application program based on this model:

```
public class Applic {
    public static void main(String[] args) {
    // (1) DB should open here!
    // (2) transaction should start here
    A objA1 = new A(1,"objA1");    // creation of objects in DB
    B objB2 = new B(2,"objB2");    //           "
    C objC3 = new C(3,"objC3",30); //           "
    D objD10 = new D(10,"objD10"); //           "
    objB2.settoD(objD10); // establish relationship
    objA1.find(2);     // (3) query: find by key 2; sets objA1
    objB2.remove();    // (4) removal of object
    // (5) transaction should commit here
    // (6) DB should close here
}
```

This Java program does not contain persistence – almost: Obviously, queries (3) and removals (4) are present. As already stated by [RC03], the points of retrieval from the database are important for the application and thus cannot be invisible. Designers need to consider the retrieval mechanisms supported by database systems. That is why a method `find` here comes out of the blue *without* being implemented in the Java classes. Later on, we will see that AO is introducing this method.

The same holds for the deletion of persistent data. Due to automatic garbage collection, there is no notion of explicitly deleting an object in Java. Even if there were an explicit `delete` operator as in C++, we could not be sure if the application actually intended to remove the object from disk or merely from the memory. Hence, data has to be deleted from the database upon specific request from the application. Again, a `remove` method must be called although classes do not possess such a method.

Having shown a simple application, we describe how to add database functionality to the application by using AO, i.e., to make objects persistent, to introduce transactions etc..

3 A Persistence Framework based on AspectJ

3.1 Handling Databases and Transactions

AspectJ is an extension of Java offering the new concepts of *aspects* and *pointcuts*. Aspects are special units that crosscut objects. We here use an aspect to enhance applications with statements to open or close a database and to control transactions. The following aspect `DBaccess` defines a pointcut `UseDb` specifying points in the code where we want to add the corresponding functionality:

```
public aspect DBaccess {
  protected static Connection _dbConn;
  pointcut UseDb() : execution (static void Applic.main(..));
  ... }
```

A pointcut essentially specifies a signature for what methods should be trapped. Thus, the pointcut `UseDb` traps the execution of the `main` method in class `Applic`. Wild cards can be used. '`..`' here denotes any number of parameters.

We want to add some behaviour *before* and *after* the execution of `main`. The `before()` and `after()` advices insert adequate JDBC statements to open and close the database:

```
before() : UseDb() { // before join point is performed
  Class.forName(getDriver()).newInstance();
  _dbConn = DriverManager.getConnection
                              (getDbUrl(), getUsr(), GetPw());
  _dbConn.setAutoCommit(false);  }
after() : UseDb() { // after join point is performed
  if (_dbConn!=null) _dbConn.close(); }
```

Any other points could be defined the same way using the full power of specifying pointcuts in AspectJ. A JDBC database connection must be kept somewhere to close the connection and for handling transactions. This can be done as a data member of the aspect `DBaccess`.

Please note JDBC statements can throw an `SQLException`. We can put a `try-catch` block around the execution again and again. But it is easier to add a globally usable exception handling for any class and method in package `java.sql` by:

```
after() throwing(SQLException x) : call(* java.sql..*(..))
  { throw new SoftException("ERROR in DB " + x.getMessage()); }
```

Another pointcut `ToCommit` can specify points where to commit transactions, while the logic for committing is done by one advice: `before () : ToCommit()`. Furthermore, a special advice `after throwing(Exception x)` can be used to trap thrown exceptions. Then the advice can decide to initiate a database rollback.

It is also possible to run specific methods in a transaction of their own. [SLB02] shows how to specify this. [RC03] run each operation in a transaction. This is similar to an auto-commit mode, which however is not sensible in real-world applications.

3.2 Adding Persistency

The overall aim of our framework is not to touch every persistent class. Instead, we want to define a generic handling. The basic idea is not new and known from JDO and the ODMG standard [Ca00]: A new superclass `PersistenceCapable` is added that offers common persistence functionality. The new root adds some "invisible" functionality such as `remove` and `find` (as explained above).

```
public abstract class PersistenceCapable {
  protected boolean isDeleted = false;
  public void find(int key)  {                   }
  public void remove()       { isDeleted = true; }
  public boolean isDeleted() { return isDeleted; }    }
```

The only place to specify persistent-capable classes is `DBaccess` where we put the `PersistenceCapable` class on top of persistent classes `A` `B`, `C`, and `D` by means of

```
declare parents : (A || B || C || D) extends PersistenceCapable;
```

The original classes are in general implemented without any notice of persistence.

Please note `find` is only a stub. It is essentially used to mark join points. The semantics is to retrieve one object by means of a key. Certainly, specific query methods could be defined, too. Their real implementation will be done in the aspect `DBaccess`.

Now, we have to intercept all those methods that cause database operations by corresponding pointcuts in order to perform JDBC operations:

```
public pointcut InsertObj() :
    call (PersistenceCapable+.new(..)) && !within(_Class);
public pointcut UpdateObj(PersistenceCapable obj):
       this(obj) && !within(_Class)
      && execution(public void PersistenceCapable+.set*(..))
public pointcut RemoveObj(PersistenceCapable obj) : this(obj)
      && execution(public void PersistenceCapable+.remove());
public pointcut FindObj() :
      call(public void PersistenceCapable+.find(..));
```

These pointcuts trap invocations of `new`, of methods that start with `set`, `remove` and `find` to insert, update, remove or find objects in the database. These pointcuts work for class `PersistenceCapable` and also for all subclasses due to the '`+`'; the latter are in fact the classes we want to handle. To get information about the object `obj` on which a method is invoked (e.g., the object to be removed), `obj` is passed on as a parameter. In case of constructor calls (cf. advice for `InsertObj`), the newly created object can be accessed via a `returning` clause. Advices can then access the object's information.

The clause `!within(_Class)` is important to exclude all methods from the helper class `_Class` (see 3.3 later) from being trapped: `_Class` also uses setters to set object properties after fetching objects from the database, but these must not get the new behavior.

Advices delegate database operations to specific static `DB...` methods, for example:

```
after() returning(PersistenceCapable obj) : NewObject()
  { DBinsert(obj); }
before(PersistenceCapable obj) : RemoveObj(obj)
  { DBdelete(obj); }
void around() : FindObj() {
  Object obj = thisJoinPoint.getTarget();
  int key = ((Integer) thisJoinPoint.getArgs()[0]).intValue();
  PersistenceCapable res = null;
  res = DBfind(obj.getClass().getName(), key);
  obj = res;                                          }
before(PersistenceCapable obj) : AccessRemovedObj(obj) {
  if (obj.isDeleted()) throw new SoftException("ERROR"); } }
```

Special attention has to be paid to the removal. From a syntactical point of view, it is possible to call `obj.remove()` and afterwards still access attributes of `obj`. The last advice is used to forbid this: The `remove` method marks the object by setting `isDeleted=true`. Any successive access, which is caught by a pointcut `AccessRemovedObject`, will raise a `SoftException` if `isDeleted` is true.

The implementation of all these static `DB` methods should be done according to mapping strategies that determine how to represent relationships or class hierarchies.

3.3 Mapping Strategies

We here focus on mapping subclass hierarchies. Several relational representations exist to handle subclass hierarchies. Each one has pros and cons in regard to fast access, redundancy, and easy update. An extensive discussion of mapping other concepts onto tables can be found in [Ho96a,Ho96b].

In order to describe the mappings, we do not use a textual specification language or an XML deployment descriptor as in EJB or JDO. Instead, mapping strategies are defined in a `Mapping` aspect by using special (meta-)classes. This aspect `Mapping` is responsible for specifying mapping strategies:

```
public aspect Mapping {
  before() : DBaccess.UseDb() {
    _Class a     = new _Class("A", "tabA", "a1");
    _Class d     = new _Class("D", "tabD", "d1");
    _FKRelship toD = new _FKRelship(a,"toD","toD");
    _VSubclass b = new _VSubclass("B","tabB",a);
    _HSubclass c = new _HSubclass("A","tabA",a);    } }
```

This setup is executed before the database has been opened. The advice describes the mapping in a quite simple way. There is a constructor `_Class` that maps a class (first parameter) to a table (second parameter), thereby determining key attributes (third parameter). Thus, classes `A` and `D` are mapped to tables `tabA(a1,a2,toD)` and `tabD(d1,d2)`, respectively. For the ease of demonstration, we assume that single-valued attributes are implicitly mapped to columns without any further specification.

Similarly, a relationship between A and D is introduced, represented by a foreign key "toD" in table tabA. Other mapping strategies, e.g., for handling set-valued attributes and relationships by linking tables, can be expressed the same way.

B and C are subclasses of A, which are represented by tables tabB and tabC. We here use mixed strategies, a vertical strategy for B (_VSubclass) and a horizontal one for C (_HSubclass). The last parameter denotes the meta-superclass instance. Due to the vertical strategy, table tabB(a1,b2) possesses the specific attribute b2 of the related class to reflect its properties. The tables tabB and tabA share a common key a1. There are inclusions between the keys of superclass and subclass tables. In order to access attributes of superclasses, the tables must be joined over the key attributes.

Using a horizontal strategy for C, the structure of tabC(a1,a2,toD,c2) comprises the specific information of its corresponding class and the attributes of its superclasses, too. Hence, the attributes inherited from A are directly available without any join. But tabA contains only the instances of the class itself. The key values of tabC and tabA are here disjoint. The deep extent of A has to be computed by a union of tabA and tabC.

Certainly, other strategies such as representing a whole hierarchy in one single table with all the attributes of all subclasses can be used. A flag can determine the specific subclass.

The advantage of our approach is obvious: The configuration of mapping strategies is described in an easy form that let the specification stay in a homogeneous language environment without any need for deployment files. We can exchange strategies by simply changing the Mapping aspect, the application code is not affected. Moreover, the mapping strategies are customizable to our needs. Strong mapping capabilities are indispensable if existing tables that do not originate from the classical mappings have to be accessed. The less strategies are supported, the less optimal the object model will be, since the object model cannot be freely defined; there must be a mapping of the model onto the existing tables. Most mapping tools usually support only a subset of strategies.

The meta-information is internally stored in the root class PersistenceCapable. The class maintains a static HashMap(name, _Class) of all classes "registered" that way. Entries are class names with a reference to the _Class instance. Moreover, each object can access its meta-information because PersistenceCapable has a reference to its _Class meta-information which can be obtained by get_Class().

_Class maintains the table name, the list of columns, the key column, strategy information etc.. It also keeps a list of SQL statements for INSERT, DELETE, UPDATE, SELECT, which certainly depend on mappings. For instance, deleting a B object implies a DELETE on tabB and also tabA because the inherited attributes are stored in tabA.

After the Mapping setup, we use an aspect Preparation to derive adequate SQL statements according to mapping strategies. This aspect can also execute corresponding CREATE TABLE statements in JDBC provided the tables do not already exist in the database.

```
public aspect Preparation {
  declare precedence : Mapping, DBaccess, Preparation;
```

```
  before() : DBaccess.UseDb() { // build SQL statement:
    Iterator itr = _Class.getListIterator();
    while (itr.hasNext()) { // for each persistent class:
      _Class theClass = (_Class)itr.next();
      // use mapping info in _Class to build SQL statements:
      PreparedStatement stmt = _dbConn.prepareStatement
        ("DELETE FROM " + theClass.tabName + " WHERE " + ... );
      theClass.delStmts.add(stmt); ...
} } }
```

The purpose of this advice is to compute SQL statements only once for each class in a single step. `delStmts` will contain a list of statements because several tables can be affected. The composed SQL strings are then executed in the static `DB...` methods. The following code describes the implementation of `DBdelete` (which is called by the advice `before() : RemoveObj()`):

```
public static void DBdelete(PersistenceCapable obj) {
  Iterator itr  = obj.get_Class().delStmts.iterator();
  while (itr.hasNext()) {  // for each stmt produced by Mapping
    PreparedStatement stmt = (PreparedStatement) itr.next();
    obj.get_Class().setKey(obj, stmt);
    stmt.executeUpdate(); } }
```

This code is now quite generic and does not depend on mapping information. The mapping is purely contained in `delStmts`, which are executed here. Unfortunately, some parts cannot be prepared that way and require interpretative work. The following JDBC code must be built at runtime to set key values after `prepareStatement`:

```
stmt.setInt(0, obj.geta1()); // a1 should be key attribute here
```

The key value of an object is obtained by `get<Attr>`; but the names of key attributes differ from class to class. Furthermore, the right `stmt.set<Type>` must be called according to the data type of the key. Hence, `DBdelete` invokes a method `setKey` declared for the meta class `_Class`. This method knows the key information about a class, and invokes the right `get/set<Attr>` to get the key values.

```
public void setKey
       (PersistenceCapable pobj, PreparedStatement stmt) {
  _Class c = pobj.get_Class();
  String keyCol = c._key; // get key column (not composite here)
  Method methods[] = pobj.getClass().getMethods();
  for (int j=0; j<methods.length; j++) { // find get for keyCol
    Method method = methods[j]; Object res = null;
    if (method.getName().equals("get" + keyCol)) {
      res = method.invoke(pobj,null); // call pobj.get<Key>
      if (method.getReturnType().getName() == "int")
      { Integer i = (Integer)res; stmt.setInt(1, i.intValue());}
      else if (method.getReturnType().getName() == " String")
      { stmt.setString(1, (String)res); }
      ...  } } }
```

4. Additional Features

4.1 Check for Homogeneous Subclass Hierarchies

In principle, the various strategies for mapping subclass hierarchies onto tables can be used in a mixed manner. The SQL queries described by [Ho96a] to fetch objects from the tables are then quite complex; a restriction to homogeneous hierarchies may be desired, i.e., applying one overall strategy for the whole hierarchy.

Such a check can simply be achieved, in addition, by a new aspect `CheckHierarchy`. This aspect adds to class `_Class` a new member by means of `public char _Class.strategy`. Any time a new subclass is specified, i.e., a `_H/_VSubclass` constructor is invoked, it is checked whether the strategy of the superclass is followed. Otherwise, a `SoftException` is raised. It is important to note that such an aspect can be activated and deactivated any time.

4.2 Forbidding the Change of Key Attributes

Since our persistence framework – at least as presented here – relies on key attributes for uniquely identifying objects, it could be dangerous to change the values of key attributes. We either have to check the uniqueness of the new value in this case, or we can disallow those changes. The latter can be easily introduced by catching `set<Attr>` calls and checking whether it is a `set<Key>` (which should be forbidden).

4.3 Caching Object Modifications

One problem of some EJB containers is that every invocation of a set-method on a persistent object performs an `UPDATE` statement in the database. This causes overhead, as an object might be updated several times in a transaction: a database operation is performed for each new setting again and again, although only the object needs to be stored once before the transaction commits. The pointcut `UpdateObj` can be implemented in such a way that it only collects all the modified objects, e.g., by gathering their unique keys. Another pointcut traps the commit and performs `UPDATE`s on all these objects, writing back all properties in one step. This can perfectly be combined with a cache.

The cache avoids loading those objects from the database, which have already been accessed previously. Hence, performance can be dramatically improved if object traversals are done with a cache. To this end, any object is put into a cache with a fixed size. If objects are retrieved again, they are found in the cache without any new database access. A new Caching aspect can take care about this.

Again, the behavior of the existing framework can be easily modified without touching the application code.

5. Conclusions

In this paper, we accommodated ourselves to the significance of accessing relational DBSs from Java. We proposed a flexible and homogeneous coupling of both worlds. Programs are given the ability to store and retrieve Java objects without taking notice of tables and SQL. We demonstrated how to provide an object-oriented persistence framework with modest effort, taking benefit from the paradigm of aspect-orientation. While other approaches such as [Ho96a,Ho03] use generator techniques, we do not.

In contrast to O/R mapping tools, we are highly customisable: the persistence layer‘s functionality can be tailored to the real needs of an application. Particularly, we can implement our set of mapping strategies. Flexible strategies play an important role for accessing existing tables with sometimes strange structures. Otherwise, an application would be forced to adjust its object model. The approach achieves its full flexibility by offering a high degree of configuration: Transactions and concrete mappings can be configured and applied to the code after having implemented the real business logic.

Future work is dedicated to enhance the framework, e.g., adding persistence by reachability as in JDO. It should generally be no problem to implement a JDO interface.

References

[Ca00] Cattell, R.; Barry, D.; Berler, M.; Eastman, J.; Jordan, D; et al.: The Object Data Standard: ODMG3.0. Morgan-Kaufmann Publishers, San Mateo (CA) 2000

[EFB01] Elrad, T.; Filman, R.; Bader, A. (eds.): Theme Section on Aspect-Oriented Programming. CACM 44(10), 2001

[He98] Heinckiens, P.: Building Scaleable Database Applications. Addison-Wesley 1998

[Ho96a] Hohenstein, U.: Using Semantic Enrichment to Provide Interoperability between Relational and ODMG Databases. In J. Fong, B. Siu (eds.): Int. Conf. on Multimedia, Knowledge-Bases and Object-Oriented Databases, Hong Kong 1996

[Ho96b] Hohenstein, U.: Bridging the Gap between C++ and Relational Databases. 10th European Conference on Object-Oriented Programming (ECOOP'96), Linz 1996

[Ho03] Hohenstein, U.: A UML-based Approach for Generating Object-Oriented Database Access Layers. Practitioner’s report on ECOOP 2003, Darmstadt 2003

[KC98] Keller, W.; Coldewey, J.: Accessing Relational Databases. In: R. Martin, D. Riehle, F. Buschmann (eds.): Pattern Languages of Program Design 3. Addison-Wesley 1998

[Ki01] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W.: An Overview of AspectJ. ECOOP 2001, Springer LNCS 2072

[KG02] Kienzle, J.; Guerraoui, R.: AOP: Does it Make Sense? The Case of Concurrency and Failures. ECOOP 2002, Springer LNCS 2374

[La03] Laddad, R.: AspectJ in Action. Manning Publications Greenwich 2003

[Pa99] Pawlak, R.; Seinturier, L.; Duchien, L.; Florin, G.: JAC: A Flexible Solution for Aspect-Oriented Programming in Java. Reflection Conf. 2001, Springer LNCS 2192

[RC03] Rashid, A.; Chitchyan, R.: Persistence as an Aspect. In M. Aksit (ed.): 2nd Int. Conf. Aspect-Oriented Software Development, Boston, ACM 2003

[Ra04] Rashid, A.: Aspect-Oriented Database Systems. Springer Berlin Heidelberg 2004

[SLB02] Soares, S.; Laureano, E.; Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ. OOPSLA 2002, ACM Press

[SHW98] Salo, T.; Hill, J.; Williams, K.: Scalable Object-Persistence Frameworks. Journal of Object-Oriented Programming, Nov/Dec. 1998