

An Aspect-Oriented Implementation of the EJB3.0 Persistence Concept

Uwe Hohenstein
Siemens AG, CT SE 2
Otto-Hahn-Ring 6
D-81730 München
+49 89 636 44011

Uwe.Hohenstein@siemens.com

Regine Meunier
Siemens AG, CT SE 2
Otto-Hahn-Ring 6
D-81730 München
+49 89 636 53530

Regine.Meunier@siemens.com

Christa Schwanninger
Siemens AG, CT SE 2
Otto-Hahn-Ring 6
D-81730 München
+49 89 636 49477

Christa.Schwanninger@siemens.com

ABSTRACT

This paper demonstrates the power of aspect-orientation by implementing the EJB3.0 persistence framework. Our approach has advantages over existing mapping tools: Flexibility is higher as the functionality can be freely implemented and extended to user's needs.

Categories and Subject Descriptors

D2.3 [Software Architectures]: Coding Tools and Techniques.

D.3.3 [Programming Languages]: Language Constructs and Features – *frameworks, modules*.

General Terms

Algorithms, Design, Standardization, Languages, Experimentation

Keywords

AOP, Enterprise JavaBeans, EJB3, Persistence

1. INTRODUCTION

Aspect-orientation (AO) is a recent technology that helps to develop software in a modular manner. AO is a promising solution for a better code structure. It aims at providing systematic means for effective modularization of crosscutting concerns by avoiding scattering of code [4]. Recent research has shown usefulness, e.g., [15] uses AO to separate concurrency control and failure handling code in a distributed system. [1] reports on a tool for monitoring, and [6] implemented a tool for performance management using AO techniques. This is exciting work that benefits from AO a lot, but often not enough to prove a broad acceptance of the technology. What is still missing are some *larger* AO implemented systems. The lack of larger systems keeps one of the obstinate myths living: “*AO is good only for logging/tracing*” [18].

Having larger AO-based implementations, we could and should show the benefits of AO in a second step, e.g., a smarter

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop ACP4IS '07, March 12-13, 2007 Vancouver, British Columbia, Canada

Copyright 2007 ACM 1-59593-657-8/07/03... \$5.00"

implementation, a smaller amount of implementation work, a better flexibility, adaptability, customization, configurability, and so on. Possible candidates for AO implementations are commonly accepted and widely used tools such as EJB containers, Object Request Brokers, Object/Relational (O/R) Mapping Tools, or database systems, which are implemented in a conventional manner so far. If such tools are implemented with AO, it becomes easier to convince industry. The book of Rashid [20] already 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 database systems in a more modularized manner, persistence for aspects, and ideas on a persistence framework.

In this paper, we focus on persistence. In fact, database applications have a lot of persistence code spread over several classes. The idea of separating persistence of application code using AO techniques is not new. [19] argue that it is possible to aspectize persistence in a highly reusable fashion. They show how to add persistence to existing Java applications: There are pointcuts for operations that require database actions such as new (to create a persistent object) and setters (to perform changes in the database). The handling of mapping strategies, which affect the database operations to be performed, is only described roughly. However, this point is solved in [11] by using an aspect for describing mapping strategies.

Although [19] show that persistence is a crosscutting concern, manageable by AO, programmers cannot completely be oblivious to the persistent nature of the data. Obviously, the programmer must consider the declarative query capabilities supported by database systems. And the handling of queries and result sets must be part of the application logic. This requires some hints in the program where querying should take place and how. Similarly, the deletion of persistent data has to be explicitly considered during application development in Java, since there is no notion of deleting an object. Consequently, there is no reference point available to remove the persistent representation of an object from the database. 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 memory. Hence, data has to be deleted from the database upon specific request from the application.

As a consequence, there is no really satisfactory solution to add persistence to *existing* applications. This seems to be disappointing at a first glance. Nonetheless, we are showing in

this paper that AO is well-suited to implement persistence frameworks such as JDO, Hibernate, or EJB3.0, thus achieving several advantages.

At first, an AO implementation does not require direct byte-code manipulations, which is indeed hard to understand and handle. This simplifies the implementation.

Moreover, an AO implementation is more modular: Such a system allows us to configure and extend the functionality easily. For example, we can add new mapping strategies. This is important if we have an object model and want to access existing tables. If we cannot map our object model to those tables with existing mapping strategies, we have to change the model in order to be mappable, i.e., the object model is no longer stable: The choice of the object model depends on achieving certain tables. Offering new mapping strategies helps [9].

Furthermore, we obtain a higher degree of customization, and we can achieve better solutions. For example, we used a mapping tool together with database products that have a bad query optimizer. The queries produced by the O/R tool run with a bad performance. In order to achieve better query executions, we had to force the database optimizer to use certain strategies by means of SQL optimizer hints. But these hints usually cannot be added to the O/R tool; we must use JDBC and loose the O/R capabilities. The lazy fetching concept of EJB3.0 is also often criticized because its functionality is very limited. If the more extensive features of Hibernate are requested, those could be added easily in our approach. Another drawback of existing implementations is that they have an infinite cache. This causes problems if mass inserts are done. All the loaded objects remain in the cache (in spite of not being needed), and an `OutOfMemoryException` will presumably occur. Bulk deletes are a problem, too. The objects to be deleted can be qualified by a query, and the objects are fetched and then deleted one by one. This results in a bad performance. Those strange effects could be avoided if we are the masters of implementation.

This is in contrast to most O/R mapping tools. They have a major drawback: A user's exertion of influence is too low, as the tool is mostly a black box. That is, the tool possesses certain functionality, or it does not. Either the performance is sufficient, or is not. One has to live with the tool as it is; there are no possibilities to improve performance, to include new features such as advanced querying, or the other way round, to remove needless functionality or expensive query analysis if only primitive querying is required. Thus, several papers such as [13,7,22] vote for hand-made persistence layers.

The purpose of this paper is to describe how to take benefit from the technology of aspect-orientation (AO) in order to provide better adaptability and flexibility. We report on our experience using AspectJ [14], a general-purpose aspect-oriented extension of Java, to implement the EJB3.0 persistence concept; however, the ideas can be adapted to other persistence approaches, too. In Section 2, we present the EJB3.0 persistence framework. Section 3 shows that AspectJ is appropriate to implement the EJB3.0 persistence framework with modest effort, offering a higher degree of customization, full control over the code and flexibility.

2. EJB3.0 PERSISTENCE CONCEPT

From the point of programming, it is desirable to store and retrieve objects of the application in an easy manner. Several object/relational mapping tools aim at providing comfort by managing objects. These approaches are useful as they make programming database applications easier by hiding underlying relational DBS technology. Those products start with an object model modeling 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 the RDBS is provided that breaks down objects into database records 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 EJB3.0 persistence framework JSR220 [5] follows the best practices of JDO and other Java-based object/relational mapping tools such as Hibernate. It takes Java as a persistent model: An entity in EJB 3.0 is a lightweight persistent object, a POJO (plain old Java object). The framework then deals with the storage and retrieval of Java objects, underlying table structures and SQL become invisible. A persistence provider cares about mapping object operations to SQL and tables.

The EJB3.0 persistence model is quite intuitive: Persistent classes are ordinary Java classes the attributes of which can be of any JDK type. That is, primitives such as `int`, `char`, `Integer`, `BigDecimal`, `String`, `Date` can be used as well as all the interfaces from the Java collection framework such as `Map`, `Set`, `SortedSet`, `List`, `Collection`, and arrays.

Making a class persistence-capable mainly consists of annotating a Java class. The EJB persistence model defines several annotations that specify persistent classes as well as optional mappings to database tables and behavioral properties such as lazy fetching. The following example defines a persistent class `Customer` including its relationship to orders.

```
@Entity @Table(name="C")
public class Customer {
    private String id;
    private String name;
    public Customer() { }
    @Id @GeneratedValue(strategy=
        GenerationType.SEQUENCE)
    @Column(name="cid")
    public int getId() { return id; }
    public void setId(int id) {
        this.id = id; }
    @Column(nullable=false,length=100)
    public String getName() {
        return name; }
    public void setName(String name){
        this.name = name; }
    @OneToMany(targetEntity=Order.class,
        fetch=FetchType.LAZY)
    @JoinColumn(name="custid")
    public Set<Order> getOrders() {
        return orders; }
    public void setOrders(Set<Order> set) {
        this.orders = set; }
}
```

Every persistent class must be annotated with `@Entity` to mark it as persistent-capable. Thus, class `Customer` becomes persistence-capable. It is furthermore mapped to a database table `C` due to `@Table(name="C")`. Without a `@Table` annotation, the class name would be taken as table name. Table `C` gets the properties of that class; renaming columns is possible with `@Column`. Non-persistent properties must be annotated with `@Transient`. Each class must possess a unique identification, a key, which is annotated with `@Id`. `@GeneratedValue` determines how a key is provided, here by using a sequence generator (`GenerationType.SEQUENCE`).

`Customer` possesses a 1-many relationship to `Order`, which is described by `@OneToMany`. The `@JoinColumn` annotation defines which foreign key attribute `custId` in `Order`'s table is used to express the relationship. Assuming a similar specification for class `Order`, this results in tables

```
C (cid int, name varchar(100) NOT NULL)
O (oid int, ..., custId int)
```

The following example sketches the usage of persistence in standalone mode, i.e., without running an EJB container.

```
EntityManagerFactory emf = Persistence
    .createEntityManagerFactory("default");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
Customer c = new Customer(1, "Dr.AO");
em.persist(c); // Create a new Entity
// lookup an entity by id,
// returns null if not found
Customer cc =
    (Customer)em.find(Customer.class, 1);
// lookup by EJB-QL query
List<Customer> clist =
    em.createQuery("SELECT c FROM "
        + "CUSTOMER c JOIN "
        + "c.orders o "
        + "WHERE o.oid = :id")
        .setParameter("id", 4711)
        .getResultList();
for (Customer c : clist) {
    ... List l = c.getOrders() ...
}
em.remove(c); // delete entity
tx.commit();
```

There are three basic interfaces `EntityManagerFactory`, `EntityManager`, and `EntityTransaction`. `EntityManagerFactory` manages a database and maintains the mapping strategies that are specific for the database. `EntityManager` represents a database connection and offers the real database functionality:

```
public interface EntityManager {
    public void persist(Object entity);
    public void remove(Object entity);
    public <T> T find(Class<T> entityClass,
        Object primaryKey);
    public void flush();
    public Query createQuery(String ejbql);
    public void close();
    public boolean isOpen();
    public EntityTransaction getTransaction();
}
```

```
...
}
```

Method `persist(obj)` makes an instance managed by the Entity Manager, i.e., persistent, while `remove(obj)` deletes the entity instance in the database. Both operations throw a `TransactionRequiredException` if there is no current transaction.

`flush()` synchronizes the persistence context to the underlying database, i.e., the SQL operations are executed; so far, they have only been collected.

`getTransaction()` returns the transaction object, an `EntityTransaction` instance. The `EntityTransaction` instance may be used serially to begin and commit transactions. Class `EntityTransaction` offers functionality to handle transactions, essentially `commit()` and `rollback()`.

3. A PERSISTENCE FRAMEWORK BASED ON ASPECTJ

Our AO-based persistence framework has to implement the predefined interfaces `EntityManagerFactory`, `EntityManager` and `EntityTransaction` that care for establishing database connections and transaction management, as it is described above. There is no real need for aspects here, although we benefit from AO to control the configuration, e.g., handling the various database systems or offering various solutions.

Two further tasks have to be performed and benefit from aspect-orientation:

1. Using the annotations, the mapping strategies must be extracted; they affect the implementation of database operations. This is done by pointcuts and advices of a specific aspect `Mapping`.
2. The database operations must be performed, mostly implementing EJB operations such as `persist()`, `remove()` etc., depending on the mapping strategies.

Both points are discussed below.

3.1 Accessing and Maintaining Mapping Strategies

An aspect `Mapping` is responsible for getting the mapping strategies that are set in classes. The principle is nearly always the same:

```
public pointcut annotatedTable(Table attr)
    : within(@Table *) && @annotation(attr)
    && staticinitialization(@Table *);
```

`within(@Table *)` restricts pointcuts to all classes annotated with `@Table`. `@annotation(attr)` allows determining the values, e.g., by `attr.name()` to get the name value of `@Table(name="X")`. `staticinitialization` is useful to let a joinpoint only occur once at construction time of the class. This means the following advice is only executed once:

```
before(Table attr) : annotatedTable(attr) {
    Class c = thisJoinPointStaticPart
        .getSignature().getDeclaringType();
    String className = c.getName();
```

```

        // name of current class
String tabName = attr.name();
        // table name
handleFields(c);
...
}

```

This advice catches any class that possesses a `@Table` annotation. Using `thisJoinPointStaticPart`, we can access the class, and `attr` gives us access to annotation properties, here the specified table name. `thisJoinPointStaticPart` provides better performance since it does not expose runtime information (method arguments values etc.). A similar pointcut can be used to trap any classes that are annotated with `@Entity`, but do not possess a `@Table` annotation. In this case, the table name will be the class name by default.

The method `handleFields` accesses all annotations for fields in a class. For instance, it is important to know the `@Id` (because this field determines the key attribute) and the various forms of relationship annotations:

```

private void handleFields(Class c) {
    Field[] fields = c.getDeclaredFields();
    for (int i=0; i<fields.length; i++) {
        Annotation a =
            fields[i].getAnnotation(Id.class);
        if (a!=null)
            String id = fields[i].getName();
            a = fields[i].getAnnotation
                (OneToMany.class);
        if (a!=null)
            String oneToN = fields[i].getName();
        ...
    }
}

```

Other annotations such as `@Inheritance` (defining the subclass mapping strategy) are handled similar, always keeping in mind that there are defaults.

Having caught an annotation, the information is placed into some kind of internal meta-definition given by specific “_” classes `_Class`, `_Relshp` etc. For example, if a pointcut detects a class `A` annotated with `@Entity` and `@Table(name="tabA")` with an `@Id id`, the advice would execute the following instructions:

```

_Key keyA = new _Key("id", "id");
_Class classA = new _Class("A", "tabA", keyA);
_Attr a2 = new _Attr("a2", "a2", classA);
...

```

`_Class` has a constructor that maps a class (first parameter) to a table (second parameter), thereby determining key attributes (third parameter).

The main purpose of these meta-classes is to make the mapping information available in Java, particularly, for deriving the implementation of Java operations.

3.2 Life Cycle Management

AO is also used to add a new root class `P_Root` to those classes that are annotated with `@Entity`. Class `P_Root` keeps general persistence functionality such as life cycle management (as explained below) and a dirty flag that indicates object

modifications. The idea is not new and known from the JDO specification and the ODMG standard [2]. `P_Root` is defined as:

```

public abstract class P_Root {
    public enum Status {
        NEW, MANAGED, REMOVED, NEWREMOVED,
        NEWMANAGED, DETACHED
    }
    protected Status status;
    protected boolean dirty;
    ...
}

```

An aspect `Persistency` puts the `P_Root` class on top of all persistent classes, i.e., classes that are marked with `@Entity`. This is simply done by declare parents:

```

public aspect Persistency {
    declare parents : @Entity * extends P_Root;
    public pointcut newObj()
        : call(P_Root+.new(..));
    public pointcut persistObj(P_Root obj)
        : call(void EntityManager.persist(Object)
            && args(obj));
    public pointcut removeObj(P_Root obj)
        : call(void EntityManager.remove(Object)
            && args(obj));
    public pointcut modifyObj(P_Root obj)
        : execution(public void P_Root+.set*(*))
            && !execution(public void P_Root.set*(*))
            && this(obj);
    after() returning(P_Root obj) : newObj(){
        obj.setStatus(P_Root.Status.NEW);
    }
    after(P_Root obj) : persistObj(obj) {
        if (obj.getStatus() == P_Root.Status.NEW)
            obj.setStatus
                (P_Root.Status.NEWMANAGED);
        else if (obj.getStatus()
            == P_Root.Status.REMOVED)
            obj.setStatus(P_Root.Status.MANAGED);
        else if (obj.getStatus()
            == P_Root.Status.NEWREMOVED)
            obj.setStatus(P_Root.Status.NEW);
        else if (obj.getStatus()
            == P_Root.Status.DETACHED)
            throw new IllegalArgumentException();
    }
    after(P_Root obj) : removeObj(obj) {
        if (obj.getStatus() == P_Root.Status.NEW)
            obj.setStatus
                (P_Root.Status.NEWREMOVED);
        else if (obj.getStatus()
            == P_Root.Status.MANAGED)
            obj.setStatus(P_Root.Status.REMOVED);
        else if (obj.getStatus()
            == P_Root.Status.NEWMANAGED)
            obj.setStatus
                (P_Root.Status.NEWREMOVED);
        else if (obj.getStatus()
            == P_Root.Status.DETACHED)
            throw new
                IllegalArgumentException();
    }
    ...
}

```

Furthermore, the `Persistency` aspect defines pointcuts for `EntityManager` operations. There are pointcuts `newObj`, `persistObj`, `removeObj`, and `modifyObj`, for trapping `new`, `persist()`, `remove()` as well as methods that start with `set`, respectively. The pointcuts work for class `P_Root` and subclasses. Trapping methods of `P_Root` itself is not that useful, but due to the '+' in the pointcut specification, derived classes are taken into account, too; the latter are in fact the classes we want to handle. Using `P_Root` restricts the number of classes to be monitored.

Advices define the basic action to be performed at the corresponding join points. This is mainly the management of life cycle, i.e., no real JDBC database operations are performed right now; these are performed when a `commit()` or a `flush()` is executed (see 3.3).

When an object is newly created, it possesses state `NEW`. Whenever `persist()` is invoked on this object, its state turns to `MANAGED`. This means that the object is now persistence capable: When a transaction commits, a managed object will be stored in the database. Invoking `remove()` makes an object unmanaged and marks it for deletion. However, it can become managed again, if `persist()` is invoked. After a `commit`, all objects are detached. This means they are no longer synchronized with the database.

The EJB3 specification defines four states: `NEW`; `MANAGED`, `REMOVED` and `DETACHED`. In addition to these, we define two new states `NEWMANAGED` (object was newly created, but then persisted) and `NEWREMOVED` (object was new, but then removed). This enables us to avoid unnecessary operations at commit time. For instance, objects with state `NEWREMOVED` will be ignored since they are not stored in the database.

Whenever an object becomes persistent, it is stored in a cache. A cache is one of the most important features of O/R wrappers. 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. All operations are performed in the cache first. The cache is also responsible for managing references to objects, i.e., if one object is fetched several times from the database, the references refer to the same object.

“Implementing” the `EntityManager` interface by means of aspects, we can offer several implementations, e.g., a highly efficient but restricted and a full-fledged one. That is, users can select a desired implementation easily by using the corresponding aspect.

3.3 Performing Database Operations

The code of `Persistency` advices essentially maintains the life cycle of objects. The real database operations are performed when a `commit()` or `flush()` occurs; `commit` executes all the collected database operations and commits the transaction, while `flush` only performs SQL operations, but does not commit the transaction. The implementation is done by advices that trap the method calls:

```
before()
: call(void EntityManager.commit()) {
    Caching.write();
```

```
    Caching.clean();
}
before()
: call(void EntityManager.flush()) {
    Caching.write();
    Caching.reset();
}
```

`Caching` is another aspect that cares about caching. This aspect allows one to use different cache implementations. `Caching.write()` iterates over all objects collected so far in the cache and then decides for each object `obj` what action to perform, `DBinsert(obj)`, `DBupdate(obj)` or `DBremove(obj)`, depending on the object’s life cycle state.

The implementation of these static `DB...` methods must be done according to mapping strategies that determine how to represent classes, relationships, or class hierarchies in tables. To this end, we let another aspect `Preparation` access the mapping information, which has been collected by the `Mapping` aspect and is represented in the meta classes `_Class` etc., in order to derive adequate SQL operations.

The meta-information is internally stored in the root class `P_Root`. The class maintains a static `HashMap` (`className, _Class`) of all classes “registered” that way. Entries are class names with a reference to a `_Class` instance. Furthermore, `P_Root` has a static `find`-method to retrieve the mapping information for a certain class name. `P_Root` has also a reference to its `_Class` meta-information which can be obtained by a non-static `get_Class()`. Consequently, each object can access its meta-information.

Each `_Class` instance keeps a list of SQL statements for `INSERT`, `DELETE`, `UPDATE`, `SELECT`, which certainly depends on mappings.

The aspect `Preparation` computes SQL statements only once for each class during a single preparation step after having collected mapping information with `annotatedEntity`.

```
public aspect Preparation {
    after() : Mapping.annotatedEntity() {
        // build SQL statement:
        Class c = thisJoinPointStaticPart
            .getSignature().getDeclaringType();
        _Class theClass =
            P_Root.get_Class(c.getName());
        theClass.insStmts = new ArrayList();
        theClass.selStmts = new ArrayList();
        theClass.delStmts = new ArrayList();
        // now use mapping info in _Class to
        // build SQL statements for class
        // and store them:
        PreparedStatement stmt = getConnection()
            .prepareStatement("DELETE"
                + " FROM " + theClass.getTabName()
                + " WHERE " + ...);
        theClass.delStmts.add(stmt);
        ...
    }
}
```

These are lists of strings, e.g., `delStmts`, because several tables can be affected. The composed SQL strings are then executed in the implementation of the static `DB...` methods.

Another aspect could execute corresponding CREATE TABLE statements in JDBC, provided the tables do not already exist.

The following code describes the implementation of DBremove (which is called by Caching.write()):

```
public static void DBremove(P_Root 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 the delStmts, which are executed here.

Unfortunately, some parts cannot be prepared that way and require interpretative work. Prepared statements contain '?' to denote parameters, e.g., to substitute the key of an object. These parameters must be set by:

```
stmt.setInt(0, obj.getal());
```

here assuming that al is the key attribute. Obviously, the code for setting the keys must be done at runtime because 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, DBremove invokes a method setKey declared for the meta class _Class. This method uses the key information about a class, and then invokes the right get<Attr> to get the key values.

3.4 Design Overview

This section briefly summarizes the major design of our AO framework. Aspect EntityManagerAspect (in general) and subspect EntityManagerAspect4MySQL (in particular for a certain database system) care about calls of createEntityManagerFactory and createEntityManager by managing JDBC database connections. The subspect EntityManagerAspect4MySQL contains an implementation of the EntityManagerFactory, EntityManager and EntityManagerTransaction interfaces and injects corresponding objects.

The aspect Mapping collects all EJB3 persistence annotations and stores them in meta-classes _Class, _Relship etc. Each persistence-capable class has a related _Class instance that contains mapping information, especially a list of SQL statements to be performed for each EJB3 operation. These classes represent the mapping of classes to tables and give access to this information.

Taking this information another aspect Preparation produces adequate SQL statements for each class. In fact, this must be done according to the mapping specifications. The resulting JDBC PreparedStatement objects are stored in the _Class instance.

Aspect Persistency puts a P_Root class on top of all persistent classes, i.e., classes that are marked with @Entity. P_Root itself keeps the life cycle of objects. Furthermore, it holds a reference to the _Class instance, i.e., each persistent object has a reference to its meta-information.

Finally, an aspect Caching manages a cache that contains all persistent objects.

4. CONCLUSIONS

In this paper, we accommodated ourselves to the significance of accessing relational DBSs from the programming language Java. We demonstrated how to provide an EJB3.0 conforming persistence layer for relational database systems with modest effort. Our approach is implemented in AspectJ [14], thus taking benefit from the paradigm of aspect-orientation. Even advanced features such as an intelligent caching of objects and synchronizing the states of objects [BW96] can be implemented by using aspect-orientation. In contrast to other approaches such as [8, 10], we can do without generator techniques.

Object/relational mapping tools and EJB3.0 containers also provide object-oriented persistence layers. Our approach has the advantage that it is highly customizable: The persistence layer can be tailored to the real needs of an application in several respects. We obtain the full control over the implementation, and we can extend or reduce the functionality of the EJB3.0 persistence layer by our own. Hence, the implementation can be optimized in performance critical cases. Similarly, we can implement our set of mapping strategies in order to re-engineer existing, often strange table structures in an adequate manner.

This paper is supposed to be a feasibility study. Our future activities will show the high potential for modularization and customization. Performance studies should compare our AO solution with O/R mappers. Further work is dedicated to implementing the complete EJB3.0 stack using aspect-orientation. This has been done for EJB2.1 by [3]. However, we rely on EJB3.0 and take into account features such as transactional annotations, dependency injection, failover etc.

5. REFERENCES

- [1] Bodkin, R.: AOP@Work: Performance monitoring with AspectJ. <http://www-128.ibm.com/developerworks/java/library/j-aopwork10/index.html>
- [2] Cattell, R.; Barry, D.; Berler, M.; Eastman, J.; Jordan, D.; Russell, C.; Schadow, O.; Stanienda, T.; Velez, F. (eds.): The Object Data Standard: ODMG3.0. Morgan-Kaufmann Publishers, San Mateo (CA) 2000
- [3] Choi, J.P.: Aspect-oriented programming with Enterprise JavaBeans. In: 4th Int. Enterprise Distributed Object Computing Conference (EDOC 2000), IEEE Computer Society (2000)
- [4] Elrad, T.; Filman, R.; Bader, A. (eds.): Theme Section on Aspect-Oriented Programming. CACM 44(10), 2001
- [5] JSR 220 specification: Proposed Final Draft 19.12.2005 (EJB Simplified API, Java Persistence API, EJB Core Contracts and Requirements). <http://jcp.org/aboutJava/communityprocess/pr/jsr220/index.html>

- [6] Govindraj, K., Narayanan, S. et al.: On Using AOP for Application Performance Management. In Chapman, M., Vasseur, A., Kniesel G. (eds.): Proc. Of Industry Track 3rd Conf. on Aspect-Oriented Software Development, AOSD 2006, Bonn, ACM Press
- [7] Heinckens, P.: Building Scaleable Database Applications. Addison-Wesley 1998
- [8] 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
- [9] Hohenstein, U.: Bridging the Gap between C++ and Relational Databases. 10. European Conference on Object-Oriented Programming (ECOOP'96), Linz 1996
- [10] Hohenstein, U.: A UML-based Approach for Generating Object-Oriented Database Access Layers. ECOOP 2003, Darmstadt 2003
- [11] Hohenstein, U.: Using Aspect-Oriented to Add Persistency to Applications. Proc. of Datenbanksysteme in Business, Technologie und Web (BTW), Karlsruhe 2005
- [12] Keller, W.; Coldewey, J.: Relational Database Access Layers - A Pattern Language. In Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences, Washington University, Department of Computer Science, Technical Report WUCS 97-07, Feb. 1997
- [13] Keller, W.; Coldewey, J.: Accessing Relational Databases. In: R. Martin, D. Riehle, F. Buschmann (eds.): Pattern Languages of Program Design 3. Addison-Wesley 1998
- [14] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W.: An Overview of AspectJ. ECOOP 2001, Springer LNCS 2072
- [15] Kienzle, J.; Guerraoui, R.: AOP: Does it Make Sense? The Case of Concurrency and Failures. ECOOP 2002, Springer LNCS 2374
- [16] Laddad, R.: AspectJ in Action. Manning Publications Greenwich 2003
- [17] Laddad, R.: AOP@Work: AOP and Metadata: A Perfect Match. <http://www-128.ibm.com/developerworks/java/library/j-aopwork3>
- [18] Laddad, R.: AOP@Work: Myths about AOP. <http://www-128.ibm.com/developerworks/java/library/j-aopwork15>
- [19] Rashid, A.; Chitchyan, R.: Persistence as an Aspect. In M. Aksit (ed.): 2nd Int. Conf. Aspect-Oriented Software Development Boston, ACM 2003
- [20] Rashid, A.: Aspect-Oriented Database Systems. Springer Berlin Heidelberg 2004
- [21] Soares, S.; Laureano, E.; Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ. OOPSLA 2002, ACM Press
- [22] Salo, T.; Hill, J.; Williams, K.: Scalable Object-Persistence Frameworks. Journal of Object-Oriented Programming, Nov/Dec. 1998