Transparently Mapping Objects to Relational Databases with AspectJ

Ján SUCHAL∗

Slovak University of Technology
Faculty of Informatics and Information Technologies
Ilkovičova 3, 842 16 Bratislava, Slovakia
suchal@fiit.stuba.sk

Abstract. In this case study we present an aspect oriented prototype implemented in AspectJ which enables to store, update, and retrieve objects from relational database in a fully transparent way. The implemented prototype is based on a simple but robust concept of partially loaded objects which is then extended to support one-to-one and one-to-many relationships between persistent objects and optimized to prevent excessive queries to relational database.

1 Introduction

In object-oriented programming a typical life cycle of an object consists of instantiation, usage and garbage collection, but whenever an application ends all objects are lost. Unfortunately we always need some objects that can be used between application runs or even concurrently in multiple different applications. These objects must be somehow made persistent in external repositories. One option is to use storage capabilities of relational databases, such as MySQL.

Object persistence is a crosscutting concern which is often very hard to refactor out from classes containing business logic with classic programming approaches. Business logic soon starts to be mixed up with persistence mechanisms resulting into code that is hard to read, hard to test a thus hard to maintain. Aspect oriented programming is a novel approach developed to effectively modularize these crosscutting concerns.

In this case study we will show an aspect oriented solution that automatically maps objects to relational database without any need to change source code of these objects.

∗ Supervisor: Ing. Valentino Vranić, PhD., Faculty of Informatics and Information Technologies STU in Bratislava

This greatly simplifies not only code itself, but also testing which can be now done without any dependence on persisting mechanisms.

The rest of the article is organized as follows. In Section 2 we introduce some basic object-relational mapping techniques, in Section 3 we describe the core concept of partially loaded objects, Sections 5, 6 are describing problems with object relationships and identities. Section 7 introduces optimizations used to prevent excessive queries to relational database when object updates happen and loading of related persistent objects is not needed.

## 2 Mapping Classes to Tables

In most cases classes can be directly mapped [2,3] to relational database tables. Consider two classes `Author` and `Article` defined as

```java
public class Article {
    private Long id;
    private Author author;
    private String heading;
    private String body;
    private Date created;
}

public class Author {
    private Long id;
    private String firstName;
    private String lastName;
    private Collection<Article> articles;
}
```

which can be straightforwardly mapped to relational database tables.

```
CREATE TABLE `article` (
`id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
`authorId` INT UNSIGNED NOT NULL,
`created` DATETIME NOT NULL,
`heading` VARCHAR(30) NOT NULL,
`body` TEXT NOT NULL,
PRIMARY KEY (`id`),
FOREIGN KEY (`authorId`) REFERENCES author(id)
) TYPE = innodb;
```

1 We intentionally omit inheritance mapping problems and we will address this more closely in Section 9.
CREATE TABLE 'author' (  
   'id' INT UNSIGNED NOT NULL AUTO_INCREMENT,  
   'firstName' VARCHAR( 30 ) NOT NULL,  
   'lastName' VARCHAR( 50 ) NOT NULL,  
   PRIMARY KEY ( 'id' )  
) TYPE = innodb;

Please notice that relationship between Article and Author is realized using the foreign key authorId.\(^2\)

We have created a ClassMapping object that inspects classes and class fields through reflection API and creates corresponding mappings to database tables and columns. However, this class mapping object needs to distinguish between objects that correspond only to table columns (e.g. Date, String...) or entire rows in other tables (e.g. Author, Article). We decided to annotate persistent classes with @Persistent annotation.\(^3\)

3 Partially Loaded Objects

This object relational mapper stands on a simple but robust idea named partially loaded objects. Every field on a persistent object can be flagged as loaded or unloaded. Before a field is accessed, it is checked that it was already loaded. If it was not, a corresponding SELECT query to database is executed, field data is fetched and field itself is marked as loaded.

In AspectJ this can be done with a simple before advice which captures every access to object fields with following pointcut definition:

```java
pointcut fieldReading(Object entity) : get(* @Persistent .*)  
   && this(entity);
```

Unfortunately in current version of AspectJ get and set pointcuts are not triggered when object fields are accessed through reflection. We defined an additional pointcut to capture such access.

```java
pointcut reflectionFieldReading(Object entity) :
   call(* java.lang.reflect.Field+.get(..)) && args(entity);
```

For each persistent object we also need to store information whether each field has been already loaded or not. This can be done in two ways:

- by creating perthis or pertarget aspect which will hold this information, or
- by defining a common super class for all persistent classes that will hold this information and using declare parents\(^5\) statement to declare this super class for all persistent classes.

---

\(^2\) This relationship is bidirectional just to demonstrate one-to-many relationship behavior in next sections.

\(^3\) Annotating a class can be considered as changing existing source code, thus breaking the claim that our prototype is fully transparent. However, annotations can be also added into existing classes using AspectJ weaving capabilities, therefore this claim holds.
Since `get` and `set` pointcuts do not work well with reflections, we needed to define a pointcut that would match both — basic field access and field access through reflection. Unfortunately, it is not possible to create an aspect that combines these two approaches and simultaneously holds context of currently accessed object. This is the main reason why a common super class approach was preferred.\(^4\)

4 Instantiating Persistent Objects

In context of object persistence there are two types of object instantiation:

**new persistent object is created** Whenever a new persistent object is created it needs to be stored into database with corresponding \texttt{INSERT} statement.

**existing persistent object is loaded** Every time a persistent object is being loaded from database using a \texttt{SELECT} statement, all object fields corresponding to columns in returned result set need to be populated.

Distinguishing between persistent object load and creation is easy because loaded objects are always created from rows of result sets using \texttt{ClassMapping} object. Pointcuts for loading and creation can be defined as

```
pointcut entityLoad() : execution(* ClassMapping.createInstance(..));
pointcut whileLoading() : cflowbelow(entityLoad());
pointcut entityCreation() : execution((@Persistent *).new(..));
pointcut newEntityCreation() : entityCreation() && !whileLoading();
```

Every persistent class needs two constructors — one for creating new persistent objects and one for loading persistent objects. Unfortunately in AspectJ there is currently no way to declare new constructor for a common super class. To solve this problem we introduced a technique called \textit{short circuit with null injection}. This technique uses reflection to look up first constructor of defined class. Parameter types of this constructor are also retrieved and constructor is called with null values for object parameters and default values for other primitive types. All calls within this constructor then need to be avoided because of injection of invalid null data. Easiest way to accomplish this is to short circuit this constructor call with following \texttt{around} advice:

```
Object around() : entityCreation() && whileLoading() {
    return null;
}
```

5 Lazy Loading of Fields

Whenever an object field is not loaded and needs to be accessed a query to database is triggered to retrieved unloaded data. There are two types of \texttt{SELECT} query scenarios we need to handle:

\(^4\) In environments where reflections are not used to access fields of persistent objects a \texttt{perthis} or \texttt{pertarget} approach would be perfectly valid.
Basic field access When a field corresponding to column in table needs to be accessed a simple query based on current entity primary key is created to retrieve this field. For example, consider a Person object with id = 1 and unloaded firstName field. Minimal query\(^5\) to retrieve this field would be SELECT firstName FROM author WHERE id = 1 LIMIT 1.

Persistent object collection field access When a field that holds a collection of other persistent objects needs to be accessed a query to retrieve all these objects is executed. For example, consider a Person object with id = 1 with unloaded articles field. Query to load this field would be SELECT * FROM article WHERE authorId = 1.

There is no need to create another loading strategy for a case where unloaded author field on Article needs to be accessed. This case can be covered by basic field access with a simple trick. Since table article contains column authorId we can create a partially loaded Author object that contains only id field. All other fields in Author will be lazy loaded using basic field access as needed.

6 Transactions as Identity Maps

Consider a scenario of loading persistent objects depending on two different criteria. It is common that these two criteria can overlap on some object, but by loading we will create two different objects that represent the same object in relational database. This identity inconsistency problem is caused by a fact that objects in memory are basically identified by memory address, but objects in database are identified by primary keys.

We need to ensure that all objects with same primary keys will point to only one address in memory. We can do this using an around advice that looks if there already exists a loaded object with identical primary key.

6 Effectiveness issues of these minimalist queries will be addressed in section 7.1.
Relational databases are accessed concurrently by multiple processes and need to ensure consistency of stored and accessed data. Synchronization problems are commonly solved by using transactions, resulting into a statement that we can consider all data accessed within an open transaction as consistent.\(^6\) We can only ensure that loaded objects are consistent within an open transaction.

Transactions emerged as natural places for identity maps [3,6] holding loaded objects dependent on primary keys. In our prototype we use transactions by annotating methods with @Transactional annotation. Since transactions can be nested, but most relational databases do not support them, we need to store identity maps at top level transaction. In AspectJ this can be done using a percflow aspect instantiation [4].

```java
aspect TransactionManagement percflow(topLevelTransaction()) {
    pointcut transactedOperation() : execution(@Transactional *.*(..));
    pointcut inTransaction() : cflowbelow(transactedOperation());
    pointcut topLevelTransaction() : transactedOperation() && !inTransaction();
}
```

7 Avoiding Excessive Queries

Every query to database is a potential performance bottleneck, it is better to avoid them in advance.

7.1 Loading Entire Rows

Loading each object field independently with one query is ineffective in real world. In most common cases a method also needs data from other fields to make an operation. Loading all fields of an object with one query solves this problem. For example instead of executing

```
SELECT firstName FROM author WHERE id = 1
```

we execute

```
SELECT * FROM author WHERE id = 1
```

and populate all fields of this object at once.

7.2 Accumulating Updates

From time to time object fields change value. Executing an UPDATE query every time a persistent object changes a field would in most cases lead to an UPDATE query explosion. We solved this problem by tracking changes to object fields and postponing these UPDATE queries to transaction end. After a top level transaction commit, we iterate through all persistent entities created or loaded within this transaction and create UPDATE queries that contain all field changes for objects.

For example these two queries that would be needed without update accumulation

```
UPDATE author SET firstName = 'Jan' WHERE id = 1
UPDATE author SET lastName = 'Suchal' WHERE id = 1
```

are merged into one query

\(^6\) Consistency of data depends on a good choice of transaction isolation level.
UPDATE author SET firstName = 'Jan', lastName = 'Suchal' WHERE id = 1

7.3 Adding Items to Collections Without Loads

Consider a scenario that a loaded Author adds an Article. This is done by adding a new Article object to articles field.

```java
class Author {
    // ...
    public Article createArticle(String heading, String body) {
        Article article = new Article(this, heading, body);
        articles.add(article);
        return article;
    }
    // ...
}
```

When `articles.add(article)` is called and `articles` field is not loaded a query is executed. Unfortunately, this query is completely unnecessary because we only need to add a new Article, we do not need a collection of all Articles an Author has ever written. Retrieving such a collection can result into a very expensive query.

This problem can be solved by returning a LazyCollection that executes a SELECT query only if it is needed. In our prototype a method that returns collection iterator executes this query. We have implemented this behavior using an around advice which returns a LazyCollection when a collection field is not loaded.

8 Related Work

From mapping definition point of view, our solution is mainly inspired by EJB3 Persistence API [1], which uses annotations for additional custom metadata information about mapping. Other approaches (TopLink7, Hibernate8) mostly use mapping definitions stored in XML files.

A popular object-relational mapper Hibernate uses concept of partially loaded objects together with a set of proxy classes which are trying to simulate pointcut and advice behavior to a certain degree.

9 Conclusions

In this work we have showed that object persistence can be implemented transparently using AspectJ language. Our prototype based on concept of partially loaded objects can create, retrieve, and update objects stored in relational database and handle common one-to-one and one-to-many object relationships.

We have done some optimizations to prevent excessive queries by loading entire rows at once, accumulating updates until transaction commit and postponing collection field loading.

---

8 http://www.hibernate.org/
Interestingly, some well known drawbacks of a popular object-relational mapper Hibernate do not emerge in our prototype. Problems such as explicit need for getters and setters or excessive proxy class substitutions are naturally solved using AspectJ capabilities which makes our aspect-oriented approach a more considerable alternative.

Furthermore a small gap in AspectJ syntax has been identified and solved using a novel simple technique named short circuit with null injection.

Our prototype completely omits inheritance of persistent objects. However, we think that core concept of partially loaded objects can be used without any modifications on objects stored in multiple tables. Further performance optimizations, such as usage of JOIN queries, are subject to additional research.

Acknowledgement: This work was partially supported by the Scientific Grant Agency of Slovak Republic, grant No. VG1/3102/06.

References