Applying AspectJ to Solve Problems with Persistence Frameworks

Uwe Hohenstein and Michael C. Jaeger

Siemens AG, CT T DE IT 1, Otto-Hahn-Ring 6, D-81730, Munich, Germany
{Uwe.Hohenstein,Michael.C.Jaeger}@siemens.com

Abstract. This work reports on problems we had with persistence frameworks in an industrial project. Most problems occurred when replacing the persistence framework Hibernate with OpenJPA. Such a substitution basically means exchanging API calls and dealing with functional differences. But the replacement involved challenging problems since some important Hibernate functionality was missing in OpenJPA and could not be emulated, and other functionality did not work appropriately in OpenJPA. Conventional techniques such as wrapping code are not sufficient to tackle those points. However, we found powerful mechanisms in the aspect-oriented programming language AspectJ to solve problems fast, easily, and in a straightforward manner. All the problems are well-motivated and the aspect-oriented solutions are explained in detail.

1 Introduction

Whenever Java and relational database systems (DBS) are used, object-relational (O/R) persistence frameworks or tools such as Hibernate, Java Data Objects (JDO) or Java Persistence API (JPA) come into play: Application programmers can store and retrieve Java objects in relational tables without knowing about the underlying table structure and/or how to formulate SQL queries. Programming can be done at an object-oriented level, i.e., by storing and retrieving Java objects. The O/R framework translates those object-oriented operations into SQL.

We were involved in an industrial project with Siemens Enterprise Communications (SEN), where the Hibernate persistence framework was used. The project develops a Java-based service-oriented telecommunication middleware which serves as an open service platform for the deployment and provision of communication services [1]. Examples for such services are the capturing of user presence, the management of calling domains, administration functionality for the underlying switch technology, and so forth. The technical basis is OSGi.

Hibernate was used for managing persistent data in a relational DBS. Hibernate is a widely used and popular O/R framework. It is open-source software and provides only a thin layer upon the Java Database Connectivity API (JDBC), offering developers much control on performance-relevant settings. Hibernate was used for two reasons: First, to be independent of various DBSs to be supported in the product, namely solidDB, MySQL, and PostgreSQL. And second, to benefit from the higher, object-oriented level of database programming.

L.A. Maciaszek and P. Loucopoulos (Eds.): ENASE 2010, CCIS 230, pp. 45–60, 2011.

Some time ago, the owner of Hibernate was accused of violating a patent on O/R frameworks in the United States. This patent infringement claim seemed to be a problem of Hibernate at a first glance. However, every software product that is shipped to the United States with Hibernate inside is affected as well; any redistribution of Hibernate implies the role of a supplier. To avoid the risk of a patent infringement, the project management decided to replace Hibernate with another O/R framework. An additional business issue was the GNU Lesser General Public License (LGPL) used by Hibernate. LGPL was not fully compatible with agreements that SEN has with its business partners. As a consequence, the project management decided to replace such LGPL software in general.

The Hibernate replacement started with a first brief evaluation, where several substitute candidates were roughly assessed: Proprietary frameworks such as iBATIS and tools conforming to the JDO or JPA standards. As a quick result, the OpenJPA framework was chosen because it is open-source and implements the JPA specification. The JPA standard seems to be appropriate because it is part of the EJB 3.0 specification and is more recent than JDO. Thus, OpenJPA could easily be replaced with other JPA-conforming tools if OpenJPA would also be gripped by the patent. Moreover, OpenJPA is provided with the more convenient Apache software license.

Migrating from Hibernate to OpenJPA is merely straightforward at a first glance: It is possible to wrap OpenJPA by still offering Hibernate interfaces; changes are thus minimal. However, during the replacement effort, severe problems raised that were difficult to detect in an OpenJPA evaluation. Those issues occurred lately and endangered the success of replacement. In order to cope with them, we found and applied solutions using Aspect-Orientation (AO).

AO has been proposed for developing software to eliminate crosscutting concerns, i.e., functionalities that are typically spread over several classes. Those lead to code tangling and scattering [2] in conventional programming [3]. Research has shown its usefulness: Hannemann and Kiczales [4] identify several crosscutting concerns in the GoF patterns [5] and extract them into aspects. [3,6] use aspects for designing and building flexible middleware. Rashid [7] discusses several facets of AO in the context of databases, in particular implementing DBSs in a more modular manner and an AO-based persistence framework [8]. Others use AO to maintain database statistics [9] or to implement ACID properties [10]. It turns out in all these studies that aspect-orientation increases programming productivity, quality and traceability, degree of code reuse, software modularity, and is better supporting evolution [11].

In this paper, we discuss another application of AO, to apply aspects to existing 3rd party software libraries in order to add missing functionality or to change internal behavior. Our intent is to show that AO provides a straightforward solution being suitable for software migrations in enterprise settings. The essential and novel value of our AO approach is a method to address the challenges of integrating 3rd party software, keeping the original software untouched and being able to manage the concerns of replacement in a maintainable manner.

In Section 2, we summarize the general strategy for replacing Hibernate and outline a selection of replacement issues that we solved by applying conventional methods. Section 3 presents some critical problems that occurred during the replacement, for which conventional solutions are hard to find and apply. After

having introduced the fundamentals of AO and AspectJ [12], Section 4 explains our solutions using AO. Our lessons learned during the replacement are summarized in Section 5. The paper ends with Section 6 that gives a summary on our experiences and our conclusions.

2 Replacement Strategy

In order to perform the Hibernate replacement, a master plan was established in the beginning. This plan consists of the following steps:

1. The goal was to start a practical replacement as early as possible. A selection and brief assessment of potential Hibernate substitutes leads to an early decision for the JPA standard with OpenJPA as implementation, because obvious similarities exist between OpenJPA and Hibernate.
2. A checklist was established for those Hibernate concepts that were seen specific or critical. A short evaluation of the checklist let appear OpenJPA appropriate.
3. We transformed the project's central persistence infrastructure to OpenJPA, particularly its configuration and deployment.
4. As a proof of concept, the most complicated software project was migrated first in a sandbox environment. By this step, we expected to identify as many problems as early as possible.
5. The real replacement on the affected software projects was scheduled and planned.
6. Finally, we performed the replacement in coordination with the affected development teams. Training and coaching was also necessary.

The short theoretic evaluation of Step 2 was successful, and no major problems have been detected at that time. Of course, several differences between Hibernate and OpenJPA APIs exist. For instance, we have to use an `EntityManager` instead of a `Session`. `EntityManager.persist()` instead of `Session.save()`, etc. But since most concepts of Hibernate seemed to have an equivalent counterpart in OpenJPA, we got an optimistic impression about the replacement. This first impression was also confirmed by [13] who state that it is no problem to migrate from Hibernate to OpenJPA.

It became clear that obvious differences are easy to cope with a wrapper approach. Implementing the Hibernate interface on top of OpenJPA has the advantage that the old Hibernate interface in use can still be retained. Only import statements have to be changed. Even the change of import packages is not really mandatory, but useful since Hibernate and OpenJPA could thus run in parallel in an OSGi container during the replacement phase. This allows for a step by step replacement of services. Ongoing development work on the middleware is not really affected.

Despite several conceptual similarities, the practical evaluation of Step 4 brought up some differences which we would like to mention briefly (see also [13] for further topics).

One problem is that JPQL delete-by-queries do not work correctly because OpenJPA generates an SQL query with a self-reference which cannot be executed by most DBSs:

```
DELETE FROM Tab
WHERE key IN (SELECT key FROM Tab WHERE <condition>)
```

A solution is to omit delete-by-queries by implementing the functionality manually, i.e., by querying the objects to be deleted first and then deleting each object one by one. This poses a performance problem due to lots of DELETE operations. A sustainable solution is to correct the query generation by avoiding the unnecessary subquery. The relevant translation is part of so-called `Dictionary` classes. Hence, the change can simply be done by defining a dictionary class `MyMySQLDictionary` that extends the predefined `MySQLDictionary` in such a way.

Furthermore, the life cycle of the persistent objects is different. For example, it is possible in Hibernate to overwrite an existing persistent object in the database by creating a new object having the same key values; saving that object overwrites the existing one. However, OpenJPA treats the (temporary) object as a new one, which let the database system complain about duplicates.

Hibernate's `Criteria` interface for queries is not supported in OpenJPA release 1.1.0. Thus, `Criteria` queries must be re-formulated in the JPQL language.

Smaller differences exist between the query languages HQL and JPQL, e.g., an explicit alias `t` has to be used at any place, as in `SELECT t FROM Type t WHERE t.attr=1` instead of Hibernate's short form `FROM Type WHERE attr=1`. This affects conditions that could be composed as `attr=1` in the GUI and now need to be extended with an alias `t`.

Hibernate has a special delete-orphan cascade option: While the ordinary delete-cascade removes with a father object all depending son objects, delete-orphan removes son objects in addition when the association with the father object is destroyed; a son object cannot exist without a father. Despite being not supported by the JPA standard, OpenJPA provides such a feature by means of an extended mapping annotation. If one stays with XML mapping files, those cascades must be resolved and implemented manually.

OpenJPA comes with an easy integration of the Apache DBCP connection pool, while we used Hibernate with the C3P0 pool. DBCP behaves differently and performance tests brought up different connection pool settings for DBCP.

Although those issues represent a very individual effort, such a correction did not pose any problems to the progress of the replacement.

3 Harder Problems

The differences between Hibernate and OpenJPA explained in the previous section are easy to solve. However, some problems – being detected in later phases of the replacement unfortunately – endangered the success of the overall replacement and were hard to solve with conventional programming techniques. This section discusses those problems in detail. Corresponding AO solutions are presented in Section 4.

3.1 Lack of Key Generation

An O/R framework requires mapping information on how to map classes onto database tables, attributes to table columns, associations to foreign keys etc. This can either be done by means of XML mapping files or by Java-5 annotations in the entity classes. Our project used XML mapping files. The following Hibernate mapping example relates a class `MyClass` (`<class>`) to a table `MyTable` (`table=…`), fields `id` and `p2` to table columns `pk` and `c2`, respectively.

```
<class name="MyClass" table="MyTab">
  <id name="id" column="pk">
    <generator class="sequence"/> </id>
  <property name="p2" column="c2"/> ...
</class>
```

Thereby, `<id>` defines a key field that uniquely identifies objects in a class; the corresponding column `pk` is used as a database primary key.

Indeed, the mapping specification in OpenJPA is different; a file `orm.xml` specifies mappings with a different syntax. The transformation of Hibernate mapping files into OpenJPA syntax is straightforward and can be achieved by an XSLT script for most differences. However, some differences are fundamental. For example, there are various alternatives for providing `<id>` values in Hibernate, e.g., to let the application be responsible for providing the key values and ensuring their uniqueness (`<generator class= "assigned"/>`), to let Hibernate generate an id by means of creating a globally unique identifier, or to use mechanisms that DBSs offer such as `sequence` generators (in solidDB) or auto-increment columns (in MySQL). These strategies are supported by OpenJPA, too. But Hibernate also offers a more abstract `native` key generation: Depending on what the underlying DBS supports, either `sequence` or `identity` (for auto-increment columns) is used. Since the project must support several DBSs, especially solidDB, MySQL, and PostgreSQL, and since the type of DBS should be invisible, such an abstract strategy is required.

OpenJPA has a similar `auto` strategy that lets OpenJPA decide what to do, but it uses a table for maintaining highest values instead of taking auto-increment columns or sequences. This is not appropriate as database installations already exist at customers, containing keys generated by either sequences or auto-increment columns. For these, the probability is high that `auto` generates already existing values. Hence, value clashes are most likely when upgrading to an OpenJPA-based implementation.

One solution is certainly to maintain three XML mapping files, one for each DBS with the supported strategy. A simple model-driven approach that generates DBS-specific variants with `sequence` or `identity`, respectively, could help here. This was regarded as an inappropriate solution as it causes a problem for deployment. OpenJPA expects the mapping file in a JAR. The overall project strategy is to have one unchangeable deployment JAR: All parameters that might vary from one installation to another, such as the database URL, its port, user and password, must be placed outside the deployed JAR file. This is because only parts of the JDK are installed on target machines and unzip/zipping of JAR files is not available to exchange parts such as mapping files. Hence, the resulting installation procedure would now need to handle several JAR files for deployment, one for each DBS.

The issue with providing different mappings becomes even worse, since we were forced to use mapping annotations in some cases. Some OpenJPA features are only available as annotations, but not in XML mappings, e.g., a “delete-orphan” cascade (cf. Section 2): This is a special option that removes son objects when their association with the father object is destroyed. On the one hand, using the delete-orphan option with annotations means that also several code variants have to be maintained, since the mapping is part of the source code. On the other hand, implementing delete-orphan behavior manually, i.e., deleting objects explicitly whenever they become parentless can be very cumbersome since cascades go over several levels in the object model.

Any of both proposals would require massive changes in the implementation and deployment infrastructure.

3.2 Failover Problem

The main DBS to be supported in our project is solidDB. solidDB is not as popular as other DBSs. However, it is often used in telecommunication projects. One reason is its hot-standby failover concept: It is possible to install two DBSs, one primary and one secondary, the databases of both being synchronized. If the primary solidDB server crashes, the secondary becomes the new primary and silently takes over the work immediately. To apply failover, applications have to use a specific *dual-node* URL of the form `jdbc:solid://h1:1315,h2:1315/usr/pw`. This URL specifies two database servers on host h1 and host h2.

The failover concept is important for our project and certainly one of the first priority requirements. We knew that Hibernate and the solidDB JDBC driver can handle the dual-node URL. Since, the O/R framework is supposed to pass this URL through to the JDBC driver, no particular problems were expected. But since the setup and accomplishment of failover test scenarios involves many steps, the final check has been postponed in the first assessment of OpenJPA.

When it came to test deployments, the failover feature of the solidDB DBS did not work for OpenJPA; connections to the database could not be established at all with the given URL. The first problem occurred: How can we find out why no connections are possible? Debugging was very tedious as the problem occurred in the depth of OpenJPA and the JDBC driver. As we are describing later, AO helped us to detect the cause for the problem.

It turned out that the dual-node URL was damaged by OpenJPA: Only the first part `jdbc:solid://h1:1315` arrived at the solidDB server. The reason is that a string is used to set several facets of connection properties in one `openjpa.ConnectionProperties`, the URL, the driver class name etc.:

```
String str = "Url=jdbc:solid://h1:1315,h2:1315/usr/pw,
              DriverClassName=solid.jdbc.SolidDriver,
              ...";
props.setProperty("openjpa.ConnectionProperties",str);
EntityManagerFactory emf
         = persProvider.createEntityManagerFactory("mydb",props);
```

A deeper investigation brought up that OpenJPA takes the comma as a separator during the analysis of `openjpa.ConnectionProperties` and thus derives the following units from the properties:

```
Url=jdbc:solid://h1:1315
h2:1315/usr/pw
DriverClassName=solid.jdbc.SolidDriver
…
```

That is, `h2:1315/usr/pw` is taken as a unit of its own, and since it does not satisfy the form `property=value`, it is simply ignored; and the URL degrades to `jdbc:solid://h1:1315`.

To solve the problem and to leave the dual-node URL intact, we obviously have to change the internal behavior of OpenJPA.

3.3 Missing Connection Property

Unfortunately, the previous solution solves only half of the failover problem: It allows establishing connections to solidDB, but no failover occurs. Indeed, the solidDB JDBC driver requires a special failover property `solid_tf_level` to be set for any database connection. OpenJPA allows passing additional properties, but only OpenJPA properties starting with "`openjpa.`", are analyzed and passed to the JDBC driver; others are ignored.

A solution must somehow change the behavior of the solidDB driver, the source code of which is unavailable.

3.4 Possible Solutions

What are possible solutions to solve the above problems? There is no easy workaround such as wrapping OpenJPA or JDBC methods because we have to intervene in the internal behavior.

We can certainly ask the vendor of solidDB to change its JDBC driver. This is in general expensive and must be done again and again when a new version is launched. For patches of OpenJPA, the open source community could provide solutions. However, the problem affects the interplay between OpenJPA and the rather specific solidDB DBS. We require solidDB-specific patches to the OpenJPA source code, but solidDB is not officially supported by OpenJPA. We reported those solidDB specific issues to the OpenJPA project, but we could not wait for a solution because this would have caused a significant delay.

Patching source code is possible, if the code is available. This is not always the case, e.g., the sources of the solidDB JDBC driver are unavailable. In case of OpenJPA, a deeper understanding of the complete source code is necessary because several logical parts are involved: The XML parser for mapping files, the handling of annotations, storing and using meta-data, interpreting the meta-data to perform database operations etc. One technical difficulty is then to patch the code in such a way that changes apply only for solidDB, but not for other DBSs. OpenJPA knows the JDBC driver and can derive the used DBS. However, this information is needed in

a different class. Hence, we have to let unrelated classes exchange this kind of information, which means the change cannot be done locally.

Moreover, the build process must be understood in order to produce a new OpenJPA JAR file. This could also cause trouble with integrating two different build approaches such as Ant and Maven.

Aspect-orientation provides simpler solutions.

4 AspectJ Solutions

Aspect-orientation is a solution for our problems, especially if 3rd party tools behave in a wrong manner and if no source code is available. We applied AO to change the internal behavior of OpenJPA and JDBC drivers in order to achieve in OpenJPA some missing Hibernate functionality.

The most popular AO language is certainly AspectJ [14]. Special extensions to Java enable separating the definition of crosscutting concerns. Programming with AspectJ is essentially done by Java and by new *aspects*. The main purpose of aspects is to change the program flow. An aspect can intercept certain points of the program flow, called *join points*. Examples of join points are method calls or executions, and attribute accesses.

Join points are syntactically specified by means of *pointcuts*. Pointcuts identify join points in the program flow by means of a signature expression. For example, a specification can determine exactly one method. Or it can use wildcards to select several methods of several classes by `* MyClass*.get*(..,String)`. A star "`*`" in names denotes any character sequence, hence, `get*` means any method that starts with "get". A type "`*`" denotes any type. Parameter types can be fixed or left open (`..`). Interception of methods can be done at the caller or callee side. An `execution(...)` pointcut intercepts at the callee side, i.e., any caller is affected. In contrast, `call(...)` intercepts at the caller side.

Once join points are captured, *advices* specify weaving rules involving those joint points, such as taking a certain action before or after the join points. Pointcuts can be specified in such a way that they expose the context at the matched join point, i.e., the object on which the intercepted method is invoked. Parameter values can be accessed in advices as well.

The AspectJ language requires a compiler of its own. Usually, the AJDT plug-in will be installed in Eclipse. However, a new compiler requires changes in the build process, which is often not desired, so for us. Then, using Java-5 annotations such as `@Aspect` is an alternative: Aspects can be written in pure Java. This was important for us, because we could rely on standard Eclipse with an ordinary Java compiler, without AJDT. In order to use annotations, the AspectJ runtime JAR is required in the classpath. To make the aspect active, we also have to start the JVM (e.g., in Eclipse) with a `-javaagent` argument referring to the AspectJ weaver. Annotations are then evaluated and become really active, because load-time weaving takes place: Aspects are woven whenever a matching class is loaded.

We now show AspectJ examples that solve our problems.

4.1 Solving the Lack of Key Generation

The basic idea to remedy the lack of key generation is to accept both strategies `sequence` and `identity`, but to change the internal OpenJPA behavior in such a way that it uses the strategy available in the DBS. Hence, if `identity` has been chosen, but if the DBS does not supply auto-increment columns, then let OpenJPA internally switch to the `sequence` strategy. This is much easier than adding a new `native` strategy for mapping specifications and/or annotations, which requires a corresponding modification of the XML parser, the analysis of annotations, the use of this kind of meta-data to derive SQL operations adequately etc.

Changing the OpenJPA behavior to handle `identity` appropriately according to the type of DBS can easily be done by the following aspect.

```
@Aspect
public class KeyGenerationAspect {
  private String db = null;
  @Before("execution(*org.apache.openjpa.persistence
         .PersistenceProviderImpl.createEntityManagerFactory(..))
          && args(.., p)")
  public void determineDBS(Properties p) {
    String str = p.getProperty("openjpa.ConnectionProperties");
    if (str != null) {
      if (str.contains("Solid"))
         db = "SOLID";
      else if (str.contains("mysql"))
         db = "MYSQL";
      else if (str.contains("postgresql"))
         db = "POSTGRES";
  }
  @Around("call(* org.apache.openjpa.meta.FieldMetaData
    .getValueStrategy(..)) && !within(com.siemens.ct.aspects.*)")
  public Object changeStrategy(JoinPoint jp) {
    FieldMetaData fmd = (FieldMetaData) jp.getTarget();
    int strat = fmd.getValueStrategy();
    if (db.equals("SOLID") && strat == STRATEGY_IDENTITY) {
      fmd.setValueSequenceName("system");
      return STRATEGY_SEQUENCE;
    } ... // similar for other DBSs
    return strat;
} }
```

A `@Aspect` annotation lets the Java class `KeyGenerationAspect` become an aspect. Annotations are used instead of the AspectJ language. This was important for us because we could rely on a standard Eclipse setup with an ordinary Java compiler.

There are two advices: The first one `determineDBS` determines the DBS and the second one `changeStrategy` changes the strategy if necessary. Both advices exchange information about the DBS in use by means of an aspect-local variable `db`.

Since the method `determineDBS` is annotated with `@Around`, it defines an advice to be executed around those join points that are specified by the pointcut string: Any execution of the method `PersistenceProviderImpl.createEntityManager-Factory` with a `Properties` parameter. The `args(..,p)` clause requires at least a `Properties` parameter and binds a variable `p` to that parameter. The variable also

occurs in the method signature and allows the advice to access the value. Thus, `p.getProperty("openjpa.ConnectionProperties")` yields the connection properties, i.e., the comma-separated list we are interested in so that we can extract the type of DBS. The result is stored in an internal variable `db`.

The `changeStrategy` advice uses this information about the DBS to switch from strategy `identity` to `sequence` in case of solidDB. Hence, the aspect can simply be used to share and exchange information even if different parts of code, even of different JARs, are intercepted. The technical problem how to determine the type of DBS is solved in an easy way.

The `@Around` advice `changeStrategy` intercepts any call of `FieldMetaData .getValueStrategy`, which returns the strategy. Due to `@Around`, the original logic is replaced in such a way that we decide when to switch the strategy in the advice.

Please note that `!within(com.siemens.ct.aspects.*)` is necessary: Whenever `getValueStrategy` is called, the call is implicitly changed to calling the `@Around` method, which performs `strat = fmd.getValueStrategy()` inside. This means this call is again intercepted, resulting in an infinite recursion. `!within` excludes any call within the aspect from being intercepted.

The parameter `JoinPoint jp` gives access to context information about the join point, especially the target object on which the method is invoked (`jp.getTarget()`). This is a `FieldMetaData` object in this case, which allows determining the current strategy by means of `getValueStrategy()`. Instead of returning the original strategy, e.g., `identity`, we can switch for solidDB to `sequence` and set the sequence name to the system sequence.

4.2 Solving the Failover Problem

As explained in Section 3.2, OpenJPA is unable to connect to the solidDB DBS with a dual-node URL `jdbc:solid://h1:1315,h2:1315/usr/pw`. Our first problem was to detect the reason why.

Refining the log4j level especially for OpenJPA produces an overwhelming but useless output of OpenJPA activities such as initialization activities, analyzing mapping specifications, named queries etc.

Debugging works only, if the source code is available. Even with IDE support, the problem is hard to detect with debugging, especially since several dynamic method invocations interrupt the execution flow: OpenJPA has a pluggable connection pool and loads dynamically the one chosen. And the connection pool dynamically invokes the JDBC driver for the selected DBS.

According to Laddad [14], one myth about AOP is to be good only for logging and tracing. AOP is indeed useful for tracing (but we disagree with the word “only”). We want to show how AO allows for a better and spontaneous controlling of tracing that is more dedicated to the problem to solve; overwhelming and useless trace output can be avoided. Thanks to load-time weaving in Eclipse, tracing can be done in a few minutes: Add the `aspectjrt` JAR-file to the classpath, provide an `aop.xml` file specifying relevant packages, use `-javaagent` in Eclipse, and implement the following advice:

```
@Before("execution(* *.*(..,String,..))")
public void myTrace(final JoinPoint jp) {
  Object[] args = jp.getArgs();
  for (Object a : args) {
    if (a instanceof String && arg!=null
    && ((String)a).contains("jdbc:solid:"))
    System.out.println("* In: " +   jp.getSignature() + "->"
                                          + a.toString());
} } }
```

This `@Before` advice intercepts any execution of any method (`execution(* *.*)`) with a `String` parameter (`(..,String,..)`) and checks whether the string contains a solidDB URL. If it does, it prints out that URL. The parameter `JoinPoint jp` gives access to context information about the join point. For instance, `jp.getSignature()` can be used to print out the intercepted method signature, and `jp.getArgs()` returns the passed parameter values.

These simple changes are done in a few minutes and lead to the following clear output:

```
* In: void org.apache.openjpa.lib.conf.Value setString(String)
->  DriverClassName=solid.jdbc.SolidDriver  ,Url=jdbc:solid://h1:
   1315,h2:1315/usr/pw,defaultAutoCommit=false,initialSize=35
...
* In: Options org.apache.openjpa.lib.conf.Configurations.parse
  Properties(String)
-> DriverClassName=solid.jdbc.SolidDriver,Url=jdbc:solid://h1:
   1315,h2:1315/usr/pw,defaultAutoCommit=false,initialSize=35
* In: boolean solid.jdbc.SolidDriver.acceptsURL(String)
-> jdbc:solid://h1:1315
* In: Connection solid.jdbc.SolidDriver.connect(String,
  Properties)
-> jdbc:solid://h1:1315
...
```

The bold parts are important: They show the transition from a good to a bad URL. Hence, the problem lies in the method `Configurations.parseProperties()`: The URL is correct before execution, but truncated afterwards. To detect this problem, AO tracing is much more effective than debugging. Thanks to a problem-specific tracing, the reason for problems can be detected immediately.

Since the problematic method is now known, we can fix the problem in a second step. Looking at the OpenJPA code, we see what goes wrong in method `parseProperties`. As already explained in Section 3.2, the code separates the units by using a comma. Then, if no "=" is found in a unit, the unit is ignored, what exactly happened to the second part of the dual-node URL.

An aspect can correct the URL. Having a pointcut trapped the execution of this `parseProperties` method, an `@Around` advice can implement an instead-of behavior: Instead of executing the original method, we use our "corrected" implementation without touching the original source code directly:

```
@Around("execution(public static Options org.apache.openjpa.lib
    .conf.Configurations.parseProperties(String)) && args(s)")
public Object parseProperties(String s) {
```

```
    Options opts;
    parse properties string s correctly and set the return value
opts;
    return opts;
}
```

4.3 Missing Connection Property

Similarly, we can add the `solid_tf_level` connection property by modifying the JDBC driver: The following advice intercepts the execution of `SolidDriver.connect(...)` and adds the required `solid_tf_level` property to the `Properties` parameter:

```
@Before("execution(* solid.jdbc.SolidDriver.connect
                  (..,String,..,Properties,..)) && args(url, pr)")
public void addSolidTfLevel(String url, Properties pr) {
  if (url != null && url.contains("solid"))
    pr.setProperty("solid_tf_level","1");
}
```

The part `(..,String,..,Properties,..)` specifies the parameters of interest. The `args` clause binds variables `url` and `pr` to them. The variable `url` is used to determine the DBS platform and `pr` to set the `solid_tf_level` property. Again, the JDBC driver, an external JAR file, is modified.

4.4 Further Problems

We applied AspectJ in a similar manner to solve other deficits of OpenJPA. We are not going into technical details, because the techniques are similar.

One problem occurred with class loading in OpenJPA. In some use cases, we ran into out-of-memory exceptions sporadically. Our analysis showed that thousands of class loader objects are created by OpenJPA. Unfortunately, the garbage collector places those objects in the system space, which means that the objects are destroyed too seldom. Using AspectJ, we detected the places where the class loaders are created and where they are used. The result was surprising: OpenJPA effectively uses only one of those class loaders. To solve the useless creation of class loaders, we defined an aspect that intercepts any constructor call. Instead of calling the original constructor, an around advice creates a class loader object only for the first time. Any further call returns that singleton.

Another memory problem is concerned with OpenJPA's query compilation cache. This cache is indispensable for achieving an acceptable performance since it relieves OpenJPA from analyzing and transforming JPQL queries again and again. Its size is configurable. If the cache is exceeded, an old query is dropped, however, this query is still kept in a second hidden cache with a fixed upper size of 1000. Since we have several database projects, each obtaining such a cache with hundreds of old queries, we again ran into high memory consumption. An aspect helped us to reduce the second cache to 0.

Furthermore, we also had some performance problems due to wrong connection pool settings. An aspect helped us to monitor whenever a JDBC connection is

requested and released; the difference determines the number of currently active connections. Moreover, the aspect detects whenever a connection is requested directly via JDBC, thus bypassing OpenJPA; there is a danger of not having closed the connection. This monitoring is done for all databases in the system. Hence, we get detailed statistics of connection usage.

5 Experiences

5.1 General Experiences

The first lesson we learned is not really an experience, but rather a confirmation of our approach: The recommendation is to start doing as early as possible, not spending too much time on product selection. We decided to quickly choose a Hibernate substitute because the real problems are anyway hard to detect even with an extensive evaluation of products. The problems are occurring when starting the doing – and they will certainly arise. In our case, we checked the most important issues carefully and early. However, the severe problems came up quite late during the replacement. It is nearly impossible, in our opinion, to check all problems for several candidates.

Anyway, there is no need to worry about potential or suddenly arising problems. Even if hard problems occur unexpectedly, AO is a very powerful mechanism to overcome them.

The wrapping approach, i.e., implementing the "old" Hibernate interface on top of OpenJPA turned out to be very helpful and reduced the replacement time drastically. But there is a difference between syntactic and semantic success. It is quite easy to get the replacement compile-clean. The harder problems occur at runtime during the testing, e.g., the different behavior in Hibernate and OpenJPA when storing new objects with an existing key. And performance is not portable anyway.

Especially for achieving the same semantic behavior, testing turned out to be important. Without a huge test suite with several thousands of JUnit test cases, the replacement would presumably have failed. Thanks to the test suite, we could immediately check the correct behavior after replacement. We can remember only very few errors that came up after finishing and testing the replacement.

5.2 Convincing Project Management

Unfortunately, our project managers are not keen on using AO or having AspectJ in their projects: There is always the fear of having uncontrollable behavior if several developers use AOP. Our experiences go along with a recent study of AO adoption [15] within non-academic projects, which indicates that the majority of the interviewed developers are "early adopters" (according to [16]) of this technology. The current stage of adoption is that occasionally developers learn the AO concepts and try to apply them *in non-critical* phases of development projects, e.g., for architectural checks or performance monitoring, as in [17). Very rarely the project management deliberately decides to use AO technologies in a project. This keeps the obstinate myths living: "AO is good only for logging/tracing" [14].

Well, we were able to convince our project management of using our AspectJ-based solution. Since we represented a focused team, we did not use the approach of

[18] and other authors who describe several stages for the adoption of AOP in order to guide single developers getting familiar with AO. This approach suits well, if a critical mass of developers can be convinced, which then in turn influence decisions of their management. We acknowledge the practical benefit of this approach, but it did not apply for our case. Even the approach we proposed in [19] could not be applied, because the advantages of AO we showed are not relevant in this project.

Rather, we faced the lucky situation that we had to tackle critical problems which imposed a lot of pressure: The replacement must have been successful in a short time, switching to yet another candidate than OpenJPA was not feasible because it could pose again uncertainties. Moreover, there was a lack of adequate alternative solutions to overcome the explained problems. The only alternative seemed to patch source code: This implies that the sources are available and that building the 3rd party library is feasible. This could go for a single version of OpenJPA, but did not work with the solidDB JDBC driver. Hence, our project management was slightly forced to accept AO.

However, AspectJ in its "originally intended" form is still unacceptable, because the infrastructure would require a lot of significant changes: As a new language, AspectJ requires a special compiler, for instance given by the Eclipse AJDT plug-in. Nonetheless, we have used AspectJ, but it is important that we have used aspects that are implemented as ordinary Java classes. All the AspectJ concepts such as aspects, pointcuts and advices are specified as annotations. Instead of using load-time weaving (cf. Section 4), which caused some problems with the class loading of the underlying OSGi container, we preferred an explicit instrumentation. The aspect classes are compiled with the Java compiler and then applied to existing JAR files in a separate step, particularly to 3rd party JAR files such as OpenJPA or JDBC drivers. Both steps require the predefined `iajc` taskdef to invoke the AspectJ compiler in Ant build scripts. The result is a new JAR, e.g., `myopenjpa.jar`, which must be used instead of the original one. Please note building the new JAR file requires only a single build file and a single additional build step. As a consequence, no source code and no knowledge about the build process is required for modifications to a 3rd party tool's JAR file. Integration into an external build process, for example by using a tool like Cruise Control with daily builds and overnight test reports, does not pose any problems and can be done by exchanging the JAR files. And finally, scaling problems with AspectJ for large projects such as long compile-times, as reported by [17], are avoided.

6 Conclusions

This paper reports on problems that occurred in a concrete replacement scenario in an industrial telecommunication project where the object-relational persistence framework Hibernate has been replaced with OpenJPA due to licensing and patent problems.

At a first glance, the Hibernate replacement has appeared as a straightforward task, because there are only syntactic differences in the APIs and in the mapping specifications of both frameworks. In fact, putting the Hibernate interface on top of OpenJPA reduced code changes to simply exchanging packages. This kept the

replacement effort low. However, harder problems occurred and endangered the success of the replacement. For example, OpenJPA does not offer Hibernate's native key generation strategy and OpenJPA prevents a failover between two solidDB database servers. This functionality is important for the telecommunication middleware, and hence, solutions are indispensable!

For these harder problems, we have presented the successful adoption of aspect-orientation (AO), especially AO programming with AspectJ [12]. Particularly, with this approach we bridged the gap of functionality and handled deficits of internal functionality. The key to success was not only AspectJ, but the special capability to apply aspects to external JAR files the source code of which is unavailable. By this technique, we were able to correct the behavior of OpenJPA and JDBC drivers. Aspects can change the behavior, however, leave the source code and original JARs intact. Thus, the essential and novel value of our AO approach is a method to address the challenges of integrating 3rd party software, keeping the original software untouched and being able to manage the concerns of replacement in a maintainable manner.

It is AspectJ that let the replacement succeed with simple solutions in short time. In contrast to [20], we were satisfied with the power of the AspectJ language. Indeed, AspectJ is a powerful language and we are simply using this power to easily solve critical problems quickly. Moreover, there is a lack of adequate alternative solutions. The only alternative seems to patch the source code explicitly – if available at all. The effort for changing the source code, adding data exchange between unrelated classes, and building the JAR leads to more complexity, error proneness, and effort than our AO-based approach. Moreover, we are unsure whether the problems could be solved with conventional techniques since the source code of JDBC drivers is usually not available.

Another advantage becomes obvious. Although we exchanged the solidDB JDBC driver twice and switched from OpenJPA version 0.9.7 to 1.1.0 during the effort, we did not touch the aspects, they are stable and still work correctly with the newer versions.

In future work, we want to apply AO for other purposes in the project. For example, we currently use a model-driven approach to generate code from XML specifications, i.e., several Java classes are generated by XSL-T transformations. We want to investigate whether AspectJ could be an alternative, although others decline appropriateness [21]. We hope that such a solution could be easier to use, better understandable, and evolvable.

References

1. Strunk, W.: The Symphonia Product-Line. In: Java and Object-Oriented (JAOO) Conference (2007)
2. Elrad, T., Filman, R., Bader, A.: Theme Section on Aspect-Oriented Programming. CACM 44(10) (2001)
3. Murphy, G., Walker, A.R., Robillard, M.: Separating Features in Source Code: An Exploratory Study. In: Proc. of 23rd Int. Conf. on Software Engineering (2001)
4. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: Proc. of the 17th Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002 (2002)

5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
6. Burke, B.: Implementing Middleware Using AOP. In: Proc. 4th Conf. on Aspect-Oriented Software Development (AOSD), Chicago (2005)
7. Laddad, R.: Aspect-Oriented Database Systems. Springer, Heidelberg (2004)
8. Rashid, A.: Persistence as an Aspect. In: [22]
9. Hohenstein U.: Using Aspect-Orientation to Manage Database Statistics. In: [23]
10. Kienzle, J., Gélineau, S.: AO Challenge – Implementing the ACID Properties for Transactional Attributes. In: Proc. of 5th Int. Conf. on Aspect-Oriented Software Development, Bonn, Germany (2006)
11. Coady, Y., Kiczales, G.: Back to the Future: A Retrospective Study of Aspect Evolution in Operating System Code. In: [22]
12. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming, 2nd edn. Manning, Greenwich (2008)
13. Vines, D., Sutter, K.: Migrating Legacy Hibernate Applications to OpenJPA and EJB 3.0., http://www.ibm.com/developerworks/websphere/techjournal/0708_vines/0708_vines.html
14. Laddad, R.: AOP@Work: Myths about AOP, http://www-128.ibm.com/developerworks/java/library/j-aopwork15
15. Duck, A.: Implementation of AOP in Non-Academic Projects. In: [23]
16. Joosen, W., Sanen, F., Truyen, E.: Dissemination of AOSD expertise support documentation. AOSD-Europe Deliverable No.: AOSD-Europe-KUL-8
17. Wiese, D., Meunier, R.: Large Scale Application of AOP in the Healthcare Domain: A Case Study. In: Industry Track of 7th Int. Conf. on Aspect-Oriented Software Development (AOSD), Brussels (2008)
18. Kiczales, G.: Adopting AOP. In: Proc. 4th Conf. on Aspect-Oriented Software Development; AOSD 2005, Chicago. ACM Press, New York (2005)
19. Wiese, D., Hohenstein, U., Meunier, R.: How to Convince Industry of Aspect-Orientation? In: Industry Track of 6th Int. Conf. on Aspect-Oriented Software Development, AOSD 2007, Vancouver (2007)
20. Ostermann, K., Mezini, M., Bockisch, C.: Expressive Pointcuts for Increased Modularity. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 214–240. Springer, Heidelberg (2005)
21. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features Using AspectJ. In: Proc. Int. Software Product Line Conference (SPLC), Kyoto. IEEE Computer Society, Los Alamitos (2007)
22. Aksit, M.: Proc. of 2nd Int. Conf. on Aspect-Oriented Software Development. In: AOSD 2003 (2003)
23. Chapman, M., Vasseur, A., Kniesel, G.: Proc. of Industry Track of 3rd Conf. on Aspect-Oriented Software Development (AOSD), Bonn (2006)