Model-Driven Development Versus Aspect-Oriented Programming
A Case Study

Uwe Hohenstein[1] and Christoph Elsner[2]
[1]*Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, D-81730 Muenchen, Germany*
[2]*Siemens AG, Corporate Technology, Wladimirstr 3, D-91058 Erlangen, Germany*

Keywords: AOP, Aspectj, MDD, XSL-T, Case Study.

Abstract: This case study compares two different technologies – model-driven development (MDD) and aspect-oriented programming (AOP) – both trying to avoid redundant code, but with very different approaches. A real industrial software system, the OpenSOA platform, which had already applied a model-driven XML/XSL-T approach, is used as the basis for implementation and comparison. For evaluation purpose, we have re-implemented the XSL-T implemented logic with a corresponding AOP implementation in AspectJ. Taking into account several criteria, our case study indicates that the AOP implementation reveals its strengths in avoiding redundancy, better testability, and understandability. However, more advanced tooling could significantly improve the position of MDD for the latter. MDD is in turn the more flexible approach, allowing generation of arbitrary artefacts the design demands. As the main issue of the case study, to generate wrapper classes and boilerplate-code, is rather common, we believe that our results have potential to be transferred to other problem settings. Furthermore, we think that our evaluation criteria will help guiding others in making technology choices. We also give an outlook on how combinations of MDD and AOP may leverage the best of both worlds.

1 INTRODUCTION

Model-driven development (MDD) has the goal to develop software systems on a higher abstraction level than code (Stahl and Völter, 2006). Given some high-level form of input, more concrete output is generated, maybe even source code. Code generation not only saves time and effort, but also avoids programming errors and increases programmer productivity (Smaragdakis et al., 2004). Moreover, the input has a higher level of abstraction, is simpler and shorter than the generated code, and makes concepts more explicit. One basic idea of MDD is a voluntary self-restriction, i.e., the input model uses a limited number of concepts that are defined by a metamodel. Common forms of input are domain-specific languages (DSL): graphical, textual, XML, or UML models. A lot of tooling can be used such as Xtext (Xtext) or MPS (MPS) or pure::variants (Beuche,2006).

Aspect-orientation programming (AOP) is quite a different technology that provides new mechanisms to handle crosscutting concerns (CCCs). CCCs are those functionalities that are typically spread across several classes with conventional programming. Those CCCs usually cause duplicated and redundant code. This leads to lower programming productivity, poor quality and traceability, and a lower degree of code reuse. AOP provides new constructs to separate crosscutting concerns. This separation allows for a better modularization, thereby avoiding the well-known symptoms of code tangling and code scattering (Tarr et al., 1999). Aspect-oriented languages such as AspectJ (Kiczales, 2001)(AspectJ, 2014) support the separation of concerns by means of special language constructs. Even other languages such as Scala or Ruby are starting to offer means for handling CCCs such as abstractions or metaprogramming.

Both technologies, MDD and AOP, can be used to avoid redundant code. As (Normén, 2007) states “code duplications smell badly”, and should be avoided. However, there are always cases where they cannot be avoided using conventional programming languages. While MDD uses a generative approach, AOP extends an existing implementation language and modularizes common code in an aspect.

Hohenstein U. and Elsner C..
Model-Driven Development Versus Aspect-Oriented Programming - A Case Study.
DOI: 10.5220/0004999901330144
In *Proceedings of the 9th International Conference on Software Paradigm Trends* (ICSOFT-PT-2014), pages 133-144
ISBN: 978-989-758-037-6

In this paper, we compare both technologies in a real industrial application. The comparison is done for XML-based code generation and the AspectJ language in the context of the OpenSOA project (Strunk, 2007). OpenSOA offers a service-oriented telecommunication middleware platform. It is an open service platform for the deployment and provision of communication services such as capturing user presence, management of calling domains, notifications, administration functionality for the underlying switch technology, and so forth. An OSGi container builds the technical basis.

A specific challenge within the OpenSOA framework is that several message-based interfaces have to be kept consistent. These interfaces are similar, however, having slight differences.

Since the development team spent about 50% of its time to create and maintain interface code and inline documentation, an MDD approach has been implemented to avoid duplicated code in different layers and to achieve consistency between several closely related interfaces and Javadoc comments. The approach relies on XML input and XSL-T transformations producing Java code, similar to (Reichel and Oberhauser, 2004). Although the approach is very helpful, developing and verifying XSL-T transformations has resulted to be tedious.

The content of this paper is to evaluate and compare the usage of AOP in such a typical MDD scenario. We started to re-implement the OpenSOA system with the language AspectJ to avoid code generation and to reduce code – even if it follows a different paradigm. The AspectJ implementation is a (non-obvious) alternative to the existing XSL-T code generation. Moreover, it could easily be integrated into the project infrastructure in contrast to other approaches. Using this basis, several facets of both solutions are compared and discussed.

At first, we compare the classical criterion of "lines of code". This is an indicator for the manual work to be done. "Code" does not only mean Java/AspectJ code but also writing XSL-T transformations and XML input in case of MDD: both comprise effort to be done.

We distinguish two major roles: The *implementer* provides the generative infrastructure, i.e., implements XSL-T scripts or codes aspects in AspectJ. In contrast, the *user* applies this infrastructure by providing XML input or defining pointcuts, respectively.

The more lines of code have to be written, the more work has to be done. But the code does not determine exclusively the effort. Therefore, we qualitatively and quantitatively evaluate several other measures. *Understandability* is a further mean for complexity and maintenance effort: The easier to apply, understand, and maintain a concept, the less effort an implementer or user has. Furthermore, *testability* is important for the implementer to check the correctness of the framework. For instance, XSL-T could generate code that is not accepted by a Java compiler. Further investigation criteria are *usability*, *redundancy avoidance*, and *completeness* of the approach. Our evaluation criteria have a strong industrial background and have been chosen due to their relevance for the involved OpenSOA software developers.

We use the case study to compare in detail the weaknesses and strengths of both approaches with regard to those criteria in order to give some guidance for choosing amongst the technologies and to make the best of both worlds.

In the following, we present in Section 2 the project OpenSOA, a telecommunication middleware (Strunk, 2007), we used for our case study. Section 3 describes the model-driven approach, based upon XML and XSL-T, which was in productive use. We present in Section 4 an alternative AspectJ solution, which could serve the same purpose. Both approaches are compared in Section 5 using the above mentioned criteria. Moreover, it summarizes the limitations of both technologies and discusses what of our experiences can be generalized beyond the case study. Section 6 presents some related work, before Section 7 concludes the discussion.

2 THE OpenSOA FRAMEWORK

The OpenSOA framework consists of six *services*: DomainManagement, UserManagement, Resource-Management, ProfileManagement, Application-Management, and RoleManagement. These services offer CRUD functionality, i.e., create, find, update, and delete operations. There are 93 operations in total, i.e., 15.5 operations per service in average.

For each of these *Service*s, classes *Service*Skeleton and *Service*TransSkeleton implement essential middleware functionality, while a class *Service*Impl implements the actual business logic. Figure 1 shows the important parts of these classes for the UserManagement service.

The classes *Service*Skeleton provide the entry point for service invocations. CRUD operations such as create expect both a dedicated parameter request object and a service context in its signature: *Op*Reply *op*(*Op*Request req, ServiceContext ctx). Depending on whether persistence in a database is required or not,

```
public final class UserManagementSkeleton extends Service implements UserManagement {
  private UserManagementTransSkeleton trans = null; ...
  public void create(final CreateUserRequest req, final ServiceRequestContext srvCtx) {
    UserIdentity ret = null;
    try {   if (LOG.isDebugEnabled())   LOG.debug("Operation create started: " + req.toString());
          final UserDTO user = req.getUser()
          final boolean returnIdentity = req.getReturnIdentity();
          ret = trans.create(user, returnIdentity, true);
          CreateUserReply reply = new CreateUserReply(ret);
          srvCtx.reply(reply);
          if (LOG.isDebugEnabled())  LOG.debug("Return: " + reply.toString());
          if (LOG.isDebugEnabled())  LOG.debug("Operation create completed successfully");
    } catch (DomainValidationException e) {
      if (LOG.isDebugEnabled())  LOG.debug("missing or wrong arguments.");
      srvCtx.fail(e);
    } catch ...
} }
public final class UserManagementTransSkeleton {
  private UserManagementSkeleton skeleton;
  private UserManagementImpl impl = null;
  public UserIdentity create(final UserDTO user, final boolean returnIdentity, final boolean isValidated) {
    UserIdentity obj = (UserIdentity) skeleton.getOpenJPAConfiguration().getTemplate().execute (
                    new OpenJPACallback() {  public Object doInTransaction(final EntityManager em) {
                                                UserIdentity result = impl.create(user,returnIdentity,em,isValidated);
                                                return result;
                                            } } );
    return obj;
} }
public final class UserManagementImpl {
  private UserManagementSkeleton skeleton = null;
  public com.siemens.project.identity.UserIdentity create(final UserDTO user,final boolean returnIdentity, final EntityManager em,
                                                           final boolean isValidated) {
    if (!isValidated)  //---------- validate params
      DomainValidator.validate(user, "user");
    /* business logic to be implemented by the programmer */
} }
```

Figure 1: Classes *Service*Skeleton, *Service*TransSkeleton and *Service*Impl for the UserManagement service

the call either delegates to the method op(*params*) of class *Service*TransSkeleton or to *op*(*params*,em) of class *Service*Impl. The parameter em provides an OpenJPA EntityManager to perform database operations. In both cases, a list of parameters (denoted as *params*) is extracted from the Request parameter by req.get...(). The *Service*Skeleton class catches technical exceptions and throws various service exceptions such as AuthorizationException or PersistenceDuplicateEntityException.

The *Service*Impl classes provide a code template to be filled out with the real business logic.

Classes *Service*TransSkeleton are used only by services that handle persistence. The class basically delegates to the *Service*Impl methods, but puts some logic around by a template mechanism, especially to let the Impl functionality run in a database session and transaction. The template is obtained by using the *Service*Skeleton object and used to execute an OpenJPACallback. The OpenJPACallback must implement a doInTransaction method, which invokes the *Service*Impl method that contains the logic to be executed in a session and transaction. That is, execute (OpenJPACallback) opens an database connection (which is represented by an EntityManager em in OpenJPA) and starts a transaction around doInTransaction. Moreover, when a database operation fails because of connection problems or database server crashes, a retry is performed taking a new connection, maybe from a failover server in order to achieve high availability.

Further classes *Op*Request and *Op*Reply are used in the signatures of *Service*Skeleton For operations *Op*.

Obviously, similar methods occur in different classes for one single service, having the same name but slightly different signatures. This should not be seen as a deficit of the architecture. A major reason for choosing the design with different signatures is to achieve better testability with shorter test cycles, since *Service*TransSkeleton/Impl can be tested without an OSGi container. Another reason for this type of architecture is to have a class *Service*TransSkeleton for a reusable session and transaction handling.

So, although we consider the architecture appropriate, a lot of method signatures and also code parts have to be kept consistent.

3 THE MDD APPROACH

To ease the development and to handle consistency, an MDD approach has been established in the project. Its goal is to keep related signatures and documentation headers of different parts of a single service consistent.

The basic idea consists of specifying services in an XML-based description language in one place. The user has to specify a file *Service*.xml for each service. The corresponding meta-model is a pre-defined XML-Schema. Figure 2 presents such a file.

```
<service name="UserManagement" persistence="true"
                                        eventing="false">
  <operation name="create" id="#User01" deprecated=”false”
             transaction="true">
    <description>
     <![CDATA[ * <p>Create a new user with all settings from
                        UserDTO.</p>
               * <p>The working domain is the user's domain</p>]]>
    </description>
    <return type="com.siemens.project.identity.UserIdentity">
     <description> <![CDATA[returns the identity object of the user
                                  created.]]>
     </description>
    </return>
    <parameter name="user" type="com.siemens.project.UserDTO">
      <validation nullAllowed="false"/>
      <description> <![CDATA[DTO containing all information about
                                    the new user.]]>
      </description>
     </parameter>
     <exception type=
        "com.siemens.project.exception.DomainValidationException">
       <description> <![CDATA[if an argument value is invalid
              * that means also null or empty if not explicitly allowed.]]>
       </description>
       <logmessage> <![CDATA[missing or wrong arguments.]]>
       </logmessage>
     </exception>
     <!--other exceptions -->
  </operation>
  <!--other operations -->
<service>
```

Figure 2: XML sample service description.

This XML input is taken for generating code by means of several XSL-T scripts. In particular, the documentation and the Javadoc description of parameters are generated in a consistent manner. The XML input specifies an XML element <service> with a certain name. An attribute persistence=true controls the persistence infrastructure for using the OpenJPA persistence framework to access a DBS. Similarly, eventing=true prepares an eventing mechanism in the business logic.

Each <service> element specifies <operation>s with <parameter> types, <return> type, and <exception>s in XML. Several XML attributes affect the code generation:

- deprecated=true lets @deprecated occur in the Javadoc behind a parameter.
- transaction=true adds a session and transaction management. We call such an operation *transactional* in the following.
- Parameters can be validated by specifying a <validation> such as nullAllowed or emptyAllowed; checks are added on parameter values, e.g., whether null or empty strings are allowed.
- A <description> can be added to most XML parts to be used in Javadoc documentation.

3.1 XSL-T Scripts for Code Generating

There are three basic XSL-T transformations that are responsible for generating the code for the three types of classes mentioned before:

- TransSkeleton.xsl generates the complete code for *Service*TransSkeleton classes.
- Skeleton.xsl generates the complete code for *Service*Skeleton classes.
- Impl.xsl generates the code frames for *Service*Impl classes, which have to be completed with business logic by programmers.

The overall principle of generation is straight-forward. Each service in *Service*.xml results in three Java classes *Service*TransSkeleton, *Service*Impl, and *Service*Skeleton.

Each XSL-T implementation simply transforms XML elements and attributes to Java code and produces the classes. Each <operation> results in a corresponding Java method in each class, however, having slightly different signatures and implementations for the classes. The <parameter>s describe the signature of methods.

The <description> is used for adding a consistent documentation including Javadoc. <description> can occur at several levels (<operation>, <exception>, <parameter>).

Details about the output generated from the XSL-T scripts are described in the following. Figure 3 presents an excerpt of a script to generate a signature with documentation. These lines show how verbose and unreadable the XSL-T code is.

3.2 Classes *Service*impl

The Java code for *Service*Impl classes and its methods are directly derived from the XML specification. *Service*Impl is the only class that is not fully generated. The user has to implement the business logic. Some specific points are (cf. Fig. 1):

- *Signature changes*: The create method obtains two additional parameters EntityManager em and boolean isValidated if transaction=true and<validation nullAllowed= "false"/>, respectively, are specified for any

```
<xsl:param name="svc_name"/>
<xsl:template match="//service">
  <xsl:text>/* * </xsl:text>
  <xsl:value-of select="$svc_name"/>
  <xsl:text>Skeleton.java * * Copyright (c) 2008 Siemens... */
  ...
  <xsl:for-each select="operation">
    <xsl:text /* * Operation definition for message based service
interface. * *
            @param req    the service method's request object. *
            @param srvCtx the service methos's context object. *
    </xsl:text>
    <xsl:if test=@deprecated="true">
      <xsl:text> * @deprecated </xsl:text>
    </xsl:if>
    <xsl:text>*/ public void </xsl:text> <xsl:value-of select="@name"/>
      <xsl:text (final </xsl:text> <xsl:value-of select="@name"/>
      <xsl:text Request req, final ServiceRequestContect srvCtx) {
    </xsl:text>
    <xsl:if test="return/@type != 'void'">
      <xsl:value-of select="return/@type"/> <xsl:text ret;/>
        ...
    </xsl:if>
  </for-each>
  ...
</xsl:template>
```

Figure 3: XSL-T excerpt from Skeleton.xsl.

operation. The first parameter em enables the method to use OpenJPA's EntityManager functionality. The second parameter allows invokers to switch a parameter validation on or off.

- *Additional code fragments:* The validation of parameters of the form "if (!isValidated)...", if turned on by <validation>, is added at the beginning of the method. For instance, nullAllowed=false checks whether a parameter is null, then throwing a DomainValidationException.
- *import statements:* All required imports are generated, according to what classes are used.
- *JavaDoc*: The informal <description> text occurs in comments, particularly Javadoc @param and @return clauses are filled with the operation's <description> text as well as @throws for <exception> specifications. This avoids checkstyle warnings, which are reported in quality metrics. If an operation is marked with deprecated=true, then @deprecated will be added in Javadoc.

3.3 Classes *Service*transskeleton

This type of class is only required for persistent classes, i.e., services that are specified as persistence=true. Their methods are allowed to access the database via OpenJPA. In contrast to *Service*Impl classes, the generated classes possess a complete implementation. The following points are specific:

- *Signature changes*: The signatures differ since there is no parameter em.
- Again, headers with Javadoc are generated, taking into account the different signature.
- The same holds for *import statements*.
- *Code variants:* The XML service description controls the code generation. For example, if transaction=true is specified for a method, OpenJPA is used to execute the database statements, and a session and transaction template is put around the logic, which also takes care of a retry in case connection problems.

3.4 Classes *Service*skeleton

The *Service*Skeleton classes are completely generated according to the XML service specification, which controls the code generation. We again mention some specific points:

- *Signature changes*: Compared to the other classes, signatures are changing again, e.g., operations possess a Request-object, which bundles parameter values instead of having individual parameters. This means that the parameters for invoking trans.create must be extracted from such a Request. Depending on the context, the right list of parameters is filled in.
- The relevant *import statements* are added, too. Again, headers with Javadoc are generated, taking into account the different signature.
- *Additional class fields:* If persistence=true is set for a service, then the class is prepared to use OpenJPA by providing an internal field OpenJPA-Configuration openJPAConf with get/set methods. Similarly, if eventing=true is specified for a service, the class is prepared for handling events by adding a field EventingComponent myEC with get/set methods. Any class with a transactional method also obtains an internal field *Service*TransSkeleton trans.
- *Code variants:* Transactional methods such as create basically delegate to trans.create. Non-transactional methods directly delegate to the *Service*Impl class.
  There are six exception types that can be specified for a method by means of <exception>: DomainValidationException, AuthorizationException, Domain-PersistenceException etc. Every specified exception is caught, logged and re-thrown. Special database exceptions DataAccessException and Persistence-Exception are handled for transaction=true. In particular, several subtypes of PersistenceException are distinguished in order to throw service-specific exceptions such as UserDuplicateEntity-Exception or UserEntityNotFoundException.The <logmessage> element for <exception> is used as text in LOG.debug().

4 AspectJ APPROACH

The most popular AO language is certainly AspectJ (AspectJ, 2014). AspectJ programming is essentially done by adding *aspects* to Java source code. The main purpose of aspects is to concentrate cross-cutting functionality. To this end, an aspect can intercept certain points of the program flow, called *join points,* and add logic by *advices*. Examples of join points are method and constructor calls or executions, attribute accesses, and exceptions.

Join points are syntactically specified by means of *pointcuts*. Pointcuts identify join points in the program flow by means of a signature expression. A specification can determine exactly one method by describing the complete signature including final, private, static, return and parameter types etc. 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 (..).

The following aspect has a before advice that adds logic before executing those methods that are captured by the pointcut myPC:

```
aspect MyAspect {
  pointcut myPC():
        execution(*MyClass*.get*(..));
  before() : myPC() { // advice:
    Java code to be executed before myPC join points }
}
```

4.1 General Principle

Using AspectJ, we re-implemented the software system. We were able to replace the code generation with a pure homogeneous language approach. There is no XML input and no XSL-T transformation. It is just AspectJ code.

The basic idea is to let developers start with manually writing the *Service*Impl classes instead of *Service*.xml descriptions, including Javadocs and the business logic. The signatures in *Service*Impl now need to be specified as required, i.e., including em and isValidated parameters (which are added by XSL-T, cf. Section 3.2, if specified). This has to be done only once in the Impl classes.

AspectJ is used to add all the missing parts for the whole implementation. The aspects are described in more detail in the subsequent subsections.

4.2 One Transskeletonaspect

A TransSkeletonAspect aspect is responsible for implementing the functionality of TransSkeleton classes (see Section 3.4 and Figure 1), which provide the session and transaction handling. Instead of specifying transaction=true for specific methods, a pointcut executeInTx() determines the transactional methods to which the logic of doInTransaction() should be applied, i.e., all public methods of Impl classes that possess an EntityManager parameter:

```
pointcut executeInTx(EntityManager em)
 : execution(public *com.siemens.project.*Impl.*(..)) && args(em);
```

A single around advice can then add the logic:

```
Object around(EntityManager em) : executeServiceInTx(em) {
  Object ret = null;
  EntityManagerFactoryImpl emf  = openJPAConf.getEMFactory();
  em = emf.getEntityManager();
  ... retry loop around ...
  EntityTransaction tx  = em.getTransaction();
  ret = proceed(em);  /* exec Impl-method instead of
                         doInTransaction(em) */
  ... commit or rollback on tx
  return ret;
}
```

The advice obtains an EntityManager em, starts and ends a new transaction, invoking the intercepted method with proceed() in between, and putting the redo logic around (not shown here). Hence, the logic is done in a central place and becomes much easier since we get rid of the complicated OpenJPACallback template mechanism as shown in Figure 1 and explained in Section 2. Please note this code is now defined once and no longer part of every transactional method. The pointcut defines where the code has to be executed.

4.3 Skeletonaspect for each *Service*

In principle, there is no need for Skeleton classes since it is possible to put the logic around the Impl methods. However, we are faced with the problem that the Skeleton methods are invoked from outside. Moreover, the signatures refer to service-specific *Op*Request and *Op*Reply objects. Thus, we are forced to keep the Skeleton classes. However, we are able to factor out common functionalities in aspects. The following code remains to be written for the user management service, for example:

```
public class UserManagementSkeleton   extends Service {
  private UserManagementImpl impl = null;
  private UserManagementTransSkeleton  trans = null;
  public void create(final CreateUserRequest req,
                     final ServiceRequestContext srvCtx) {
    if (LOG.isDebugEnabled()) {
        LOG.debug("Op create started");
    } // -> added by single before advice
    UserDTO user = req.getUser();
    boolean id = req.getReturnIdentity();
    UserIdentity ret = impl.create(user,id,true);
    CreateUserReply reply = new CreateUserReply(ret);
    srvCtx.reply(reply);
    if (LOG.isDebugEnabled()) {
        LOG.debug("Op create succeeded");
      } // -> added by after return advice
    return ret;
} }
```

This is basically the Skeleton-method without

logging functionality (see the strikethrough) and exception handling, both being extracted into aspects. In the original code, a method of the TransSkeleton or Impl is invoked inside depending on the transactional setting. Here, we call the Impl-method directly since the TransSkeleton behaviour (if necessary) is put around by means of an aspect. Thus, the reference TransSkeleton trans is no longer needed.

It remains to manually specify the signature, unpack parameters from a CreateUserRequest, and invoke methods impl.*op* of *Service*Impl classes.

If persistence is required, get/set methods for OpenJPAConfiguration and a corresponding internal field need to be added. This can simply be implemented in a dedicated superclass Persistence:

```
public class Persistence {
  private OpenJPAConfiguration  openJPAConf = null;
  public OpenJPAConfiguration getOpenJPAConfiguration()
    { return this.openJPAConf; }
  public void setOpenJPAConfiguration (OpenJPAConfiguration conf)
    { this.openJPAConf = conf; }
}
```

The following statement puts the Persistence superclass on top of persistent Skeleton classes and let derived classes inherit the above functionality:

```
declare parents: UserManagementSkeleton, ... : Persistence
```

Similarly, another superclass Eventing and a declare parents statement are added if eventing is enabled. Please note there is no problem with multiple inheritance: Aspects can add two superclasses, Persistence and Eventing, to a Skeleton class.

A single SkeletonAspect aspect keeps all these declare parents statements and also concentrates the logging functionality in corresponding before/afterReturning advices:

```
public class SkeletonAspect {
  declare parents: ... /* as above */
  private static final Logger LOG
                          = Logger.getLogger(SkeletonAspect.class);
  before() : call(public * com.siemens.project.impl.*Skeleton.*(..)) {
    if (LOG.isDebugEnabled())
      LOG.debug("Operation " + thisJoinPoint.getSignature()
                + " started");
  }
  afterReturning() { Log.debug() for successful operation ... }
}
```

4.4 Aspect for Exception Handling

Another aspect takes care of exception handling, which was originally part of Skeleton classes. This aspect defines several advices. Each advice adds a further try-catch block around the invocation of Impl methods:

```
public aspect ExceptionAspect {
  Object around() :
  call(... any Skeleton method with a DomainValidationException ...) {
    Object ret = null;
    ServiceRequestContext srvCtx = (ServiceRequestContext)
                                          thisJoinPoint.getArgs()[1];
    try {
      ret = proceed();
    } catch(DomainValidationException e) {
      if (LOG.isDebugEnabled())
        LOG.debug ("missing or wrong arguments.");
      srvCtx.fail(e);
    }
    return ret;
  } // ... for other exceptions
}
```

The ServiceRequestContext, which is used to signal a failure, is obtained by accessing the second parameter of the joinpoint by means of thisJoinpoint.getArgs()[1].

DataAccessException and PersistenceException, which are thrown in case of transactional methods (transaction=true), are handled similarly, however, transforming exception types:

```
catch (PersistenceException e) {
  if (e instanceof PersistenceDuplicateEntityException) {
    if (LOG.isDebugEnabled())
      LOG.debug("Domain entity already exists in the database.");
    srvCtx.fail(new DomainDuplicateEntityException
                                                (e.getMessage()));
  } else if ... other exceptions ...
}
```

4.5 Validation Logic

Validation logic such as

```
if (!isValidated) DomainValidator.validate(user,“user”)
```

is inserted whenever a validation is required. This adds a check for nullness for the given parameter name in the method of the Impl class. In XSL-T, this is specified for an operation by means of

```
<parameter name="user" type="com.siemens.project. UserDTO">
<validation nullAllowed="false"/>
```

The same behaviour can be achieved by a before advice that adds the nullness check before method execution. The problem is how to get the parameter object to be checked, i.e., user above. As the kinds of validation checks the programmer would like to perform is known in advance, we can simplify the code by only referring to the position of the parameter in the signature. For example, we provide pointcuts validateNotNullAtPosition*i* that allow for adding a check for a certain position *i*. An advice can access the parameter at this position:

```
public aspect ValidationAspect {
  pointcut validateNotNullAtPos0 (Object o, boolean isVal) :
  execution(...) && args(o,..,isVal);
  before(Object o, boolean isVal) : validateNotNullAtPos0(o, isVal)  {
    if (!isVal) {
      MethodSignature sig = (MethodSignature)
                                thisJoinPointStaticPart.getSignature();
      String name = sig.getParameterNames()[0];
      DomainValidator.validate(o,name);
} } }
```

The parameter name, to be added to DomainValidator.validate, is obtained by means of reflection (MethodSignature); the isValidated parameter always occurs last and can simply be bound to a variable isVal.

To make code more readable, an annotation @Validate("user",nullAllowed="false") can mark every method to be validated: An aspect intercepts any usage of this annotation and inserts the validation logic. This makes usage easier.

5 COMPARISON

We compare the originally existing MDD with the new AOP approach with regard to several comparison criteria. The criteria have been selected due to their relevance for the OpenSOA developers. At first, we investigate the classical quantitative criterion of "lines of code". This is a measurement for the manual work to be done. "Code" here does not only mean Java or AspectJ code but also XSL-T transformations and XML input in case of MDD: This comprises effort to be done as well. Further, qualitatively evaluated, criteria are usability, understandability, testability (which all affect development time), and redundancy. We took those criteria without any weights since they all together have an impact on development time and cost. We asked the developers but did not obtain a precise weighting.

Please also note we ignored performance since the performance is mostly affected by database accesses. Anyway, the types of pointcuts we use are very simple and usually do not cause performance issues.

The results are partially subjective in the sense that the assessment of the original MDD infrastructure is done by the involved software developers.

5.1 Lines of Code

The XSL-T approach requires XML input files *Service*.xml. That is the specification effort for a user to apply the infrastructure for the six services ApplicationManagement, DomainManagement etc. All these XML files have **4339 lines** in total.

To provide the generative infrastructure, the implementer has to implement three XSLT scripts: TransSkeleton.xsl (220 lines), Skeleton.xsl (499 lines), and Impl.xsl (384 lines). We have mentioned briefly the classes Request/Reply for Skeleton operations. These are generated as well by XSL-T scripts RequestObject.xsl (205 lines) and ReplyObject.xsl (113 lines). These are **1421 lines** for code generation.

In total, ***5760*** (= 4339 + 1421) lines are required for the XSL-T approach.

In the AspectJ solution, an implementer has to code advices in AspectJ, while a user applies this infrastructure by defining pointcuts or placing annotations.

The user has to manually implement a class *Service*Impl. From a logical point of view, the specification parts in *Service*.xml are directly put into code in *Service*Impl.java; These are **1208** lines for 93 methods without business logic (which we do not count in either approach).

The infrastructure is given by aspects. One aspect TransSkeletonAspect handles the transactional behaviour for transactional methods. The decision which methods are transactional is done by means of method pointcuts. An around advice puts the transactional logic around the relevant methods of Impl-classes. This aspect has **259** lines.

A SkeletonAspect aspect adds Persistence and Eventing super classes by means of two declare parents pointcuts. Moreover, the aspect introduces logging with before/afterReturning advices. This aspect requires **12** lines of code. The two new superclasses Persistence and Eventing have **17** lines (9 and 8 lines).

For each *Service*, a *Service*Skeleton class must be implemented due to external usage. These are 93 methods with about 8 lines in average, which sums up to **744** lines.

An ExceptionAspect adds exception handling. It comprises 2 lines for the aspect declaration itself and 12 lines for each of 6 the exception types. Handling transactional exceptions requires additional 21 lines. This sums up to **95** lines.

One ValidationAspect handles the validation code for at most two positions: 8 * 2 positions à 13 lines. These are additional **208** lines.

Hence, the AspectJ approach requires ***2543*** lines thus saving more than 3600 lines, i.e., nearly 60%.

Unfortunately, this calculation does not consider the 94 Request and 57 Reply classes for Skeleton operations. In the XSL-T approach, these 10418 and 4176 lines of code, respectively, are generated. But in the AspectJ approach, there is no mean to produce or to avoid these classes: We have to manually implement those 14594 lines of code: The previously calculated advantage of AspectJ is lost!

However, the classes contain a lot of trivial comments (28 lines for Request and 15 lines for Reply classes in average), i.e., 3487 lines could be left out. Since the classes are simple JavaBeans with a constructor, a get-method, and toString method, specifying the attributes is enough; Eclipse or any other IDE can generate the code by a mouse-click.

This requires additional time to handle the IDE, but reduces the lines of code by further 735 lines (94*6 + 57*3). But the AOP approach still requires 10372 lines for handling Request/Reply-classes.

5.2 Understandability

There is another point that concerns the development time for providing the infrastructure: understandability. It also affects the evolution of the system.

XSL-T is quite different from an object-oriented programming language such as Java, since it defines a set of rules that apply to a given XML document recursively. Reading those rules and understanding the overall behaviour is not easy even if one is familiar with XML and XPath. In particular, the rule-base approach makes it difficult to write or to extend XSL-T scripts. Moreover, programmers must handle a couple of unintuitive and error-prone details of XSL-T, such as a special handling of zero-parameter methods or leaving out a "," after the last parameter in parameter lists. Other MDD frameworks such as (Xtend2,XPand) provide a better support.

These drawbacks are not present in the AspectJ approach. Indeed, its major advantage is its homogeneity: There is one language to learn, AspectJ, which extends well-known Java by a few constructs such as pointcuts and advices the semantics of which is clear and understandable. Advices, in turn, are implemented in pure Java. Having a little knowledge about AspectJ, it should be no problem to understand the advices we have presented.

The disadvantage is that some conceptual points cannot be handled appropriately. One example is adding validation logic, which becomes less intuitive because we cannot directly handle the parameter position (cf. subsection 4.6). Furthermore, we cannot generate Skeleton and Request/Reply classes easily. These parts must be hand-coded. And finally, import statements must be added manually or generated by using IDE support. In contrast, those parts are completely generated in the XSL-T approach.

5.3 Testability

Testability is the major disadvantage of the XSL-T approach. Since code is generated, syntactical correctness is not immediately visible. Thus, the effort to check correctness is high. Several cycles of generating code, compilation, testing, and debugging are necessary in order to check ultimate correctness. Moreover, debugging of XSL-T is very limited.

Moreover, the correct behaviour must be proven by unit testing. This means particularly that any variation within XML service descriptions has to be checked and unit tested. This is difficult and increases complexity with the number of possible combinations. One possible but challenging approach is to generate unit tests as part of the XML-based generation. However, also because of the complexity of the XSL-T language, only manual testing of main use cases was performed for OpenSOA. The (inappropriate) strategy, we noticed in practice, is thus to let developers generate code and detect problems during tests; having their feedback, implementers can fix the problems. In turn, a new rollout of the MDD infrastructure is required, leading to slow turn-around cycles for bugfixing.

Using AspectJ, syntactical correctness is immediately given for both the infrastructural advices and the pointcuts thanks to special plug-ins such as AJDT for the Eclipse IDE. As a direct consequence of the integrated language approach and corresponding compiler support, any syntax errors in wildcards or aspects are detected by a compiler. The plugin also issues a warning if a pointcut does not match any joinpoint in the code base. Only the correct behaviour has to be checked, but can be achieved by running unit tests in an ordinary Java IDE. Moreover, debugging AspectJ is similar to Java code thanks to IDE support.

5.4 Usability

In the XSL-T approach, it is very straightforward to write input .xml files. Moreover, an XML schema exists and indicates any syntactical errors in input files. Only the code generator has to be started to produce Java code.

In AspectJ, applying "code generation" means to specify corresponding pointcuts, e.g., to apply exception handling or the transaction template to methods. Despite not being part of the ordinary Java language, pointcuts are easy to understand. In fact, we only use a small subset of AspectJ pointcuts, more or less using obvious wildcard expressions in the sense of "all method of a *Service* class". Anyway, the simplest way is to enumerate methods. Applying aspects is mostly a one-line pointcut. Moreover, excellent support of the Eclipse AJDT plugin let one determine the effect of aspects immediately, e.g., where an advice will be inserted. Using annotations to apply an aspect certainly yields to a better separation of infrastructure and usage.

5.5 Redundancy

The XSL-T transformations are partially redundant because the redundancy of signatures moves from code to XSL-T scripts: Generating similar classes Impl, Skeleton, TransSkeleton etc. with similar methods requires similar XSL-T transformations. Furthermore, the exception handling in the generated code is crosscutting and scattered around classes in the final outcome.

AspectJ, from its nature, has a much better separation of concerns for handling the transaction skeletons and exception handling. The overall redundancy is less. There are no longer several similar classes, it is essentially the Impl Java class; the logic of other generated classes becomes part of aspects. However, there are some limitations. For instance, the Skeleton methods have to be manually written (with IDE support for generating import's). Even if some common logic can again be concentrated in aspects, e.g., by putting superclasses on top of classes, we cannot avoid these classes.

5.6 Completeness

The XSL-T approach allows for generating code including Javadoc comments and import statements.

In contrast, the AspectJ solution is not able to handle necessary import statements. The AO approach simply relies on IDE support such as "Organize import" functionality; which however often is just a mouse-click. Similarly, comments and Javadocs have to be manually added. From a logical point of view, those parts move from *Service*.xml to *Service*ManagementImpl.java, i.e., put directly into code. In the XSL-T approach, Javadoc is generated into several classes, but this is not necessary here: There will be only one Java class, besides additional aspects.

5.7 Comparison Summary

The results we obtained with our case study indicate that AspectJ reveals its major strengths in avoiding redundancy and better testability, while MDD with XSL-T is a more complete and flexible approach. In fact, XSL-T allows for generating arbitrary artefacts the design demands, whereas AspectJ cannot provide this functionality and would require changes in the design. AOP in turn is better understandable and readable, however, we see that other MDD tools offer more advanced and integrated features.

Table 1 provides a rough summary of the comparison results.

Table 1: Summary of comparison results.

	AspectJ - AOP	XSLT - MDD
Lines of Code	- (requires add. OO classes)	o (duplicated XSLT code)
Under-standability	+ (straight forward)	- (complex syntax/semantic)
Testability	+ (directly testable)	o (difficult for generated code)
Usability	o/+ (reasonable)	- (difficult)
Redundancy	+ (nearly not redundant)	o (partially redundant)

5.8 Limitations and Generality

As our case study focuses on a specific software framework, our study cannot serve as an extensive guide for the selection among the technologies for arbitrary use cases and software projects. Nevertheless, we think that our case study results can be of value for practitioners being in the situation to choose among them.

As our problem of generating wrapper classes and boilerplate-code is rather common, we believe that our results have potential to be transferred to other problem settings. Furthermore, we think that the dimensions our evaluation is based on will help others to guide their decision making when choosing amongst the technologies or to take benefit from the best of both worlds.

Whereas in our solution, understandability speaks in favour of AOP, we see that more advanced and integrated tooling could significantly improve the position of MDD here. More advance generator languages, for example Xtend2 (Xtend2), provide a more straight forward generation approach, without recursive generation rules, but with mature editor support and even debugging functionality.

6 RELATED WORK

There are several case studies and a large body of papers that either only evaluate the benefits and liabilities of MDD (e.g., Kapteijns et al., 2009, Lussenburg et al., 2010) or AOP (e.g., Kästner et al., 2007). For example, (Kästner et al., 2007) take the Berkeley DB as a case study and refactored the code into 38 features. While other studies, e.g., (Lee, 2006), suggested that features of a product line be implemented by aspects, they find that AspectJ is not suitable to implement most of their features. Even if this work is not a comparison, it shows deficiencies of the language AspectJ, not necessarily of AO or AOP, with respect to their case study. In

contrast, (Hohenstein, 2005) shows how to successfully apply aspects to implement a persistence framework, which is usually controlled by code generation based upon annotations or XML.

Our work, in contrast, aims at a comparison of AOP with MDD, in order to support the selection among the technologies. Only few work exists that explicitly makes such a comparison. (Stein and Hanenberg, 2006) argue that AOSD and MDD are alike since both adapt an input system in order to receive an augmented output system, however, using different approaches, weaving and transformation, respectively. They discuss the technical differences by means of an example. (Kaboré and Beugnard, 2007) compares AOP and MDD with regard to a better separation of concerns. They only investigate how to describe and how to apply both, concluding that a model-driven approach offers more flexibility.

(Liu et.al, 2006) use a heart pacemaker product line to elaborate on modelling crosscutting variability with AO. They state that AO can benefit the MDD of product lines. The study identifies desired characteristics of AO modelling techniques for product lines and proposes similar evaluation criteria to ours such as feasibility, degrees of variability, evolution, tool support, and cost, however, miss to investigate those in their case study.

(Anastasopoulos and Muthig, 2004) use a mobile phone software product line to systematically evaluate AOP as a product line technology. Their result is that AOP is especially suitable for variability across several components. The study discusses several factors and the effort for various activities: implementing reusable code, reacting to evolutionary changes, reusing code, resolving variations, and testability. Our study discusses similar points, however, at a deeper level using a real industrial case study.

Indeed, there is further significant work on combining AOP and MDD. For instance, (Henthorne and Tilevich, 2007) notices that the generated code is not always adequate for a task at hand, and mentions following in-house coding conventions and missing import features as examples. These are particular problems we handle. Generating AspectJ code helps to give flexibility.

(Pinto et al., 2009) combine both approaches by describing an MDD approach that generates aspect-oriented models. That is, aspects are part of the outcome. This is especially useful to handle unanticipated variabilities by means of aspects as the MDD/AOP approach of (Völter and Groher, 2007) illustrates. In our work, we explicitly compare the two technologies, to avoid increasing the overall technical complexity and dependencies of the developed software, in our case, the OpenSOA framework.

7 CONCLUSION

In this paper, we compared two completely different approaches, model-driven development (MDD) and aspect-oriented programming (AOP) with AspectJ, by means of a real industrial software system and thus investigating several criteria. While MDD, here applying XSL-T, is straight forward and well-understood for code generation, the usage of AOP is not so obvious, but can serve the same purpose in a different manner (Stein and Hanenberg, 2006).

We achieved some interesting results during an aspect-oriented re-implementation of the original XSL-T system. AOP is principally able to handle code generation and has some advantages over XSL-T: AspectJ is better understandable and usable, especially from an implementer's point of view. There is a huge advantage for testing, in particular, checking the syntactic and semantic correctness. We also notice a better separation of concerns and avoidance of redundancy, for instance, if logic is put around existing code (transactional skeleton) or after/before (logging). The most striking limitations appear if new classes have to be introduced. This is the main reason why the pure AspectJ-based solution requires more lines of code (LoC).

XSL-T has advantage if several code generators are producing several output files based upon the same input file. This leads to the mentioned LoC advantage. Thus, XSL-T is more extensible and has potential for creating further classes, in particular Request/Reply classes in this case study. Finally, XSL-T results in a rather weak understandability. This, however, seems to be a consequence of the technology choice than of the MDD approach in general. By using MDD approaches with more intuitive languages and mature IDE support based around Eclipse Ecore (e.g., (Xtext, Xtend2)), we believe the implementation and the evaluation would improve in this category. In particular, there are tools available that can be used to produce Java code, at least classes and method signatures as a model. This can build a basis to take the Java *Service*Impl file as input and produce Request and Reply classes. Indeed, (Heidenreich et al., 2009) even developed an Ecore metamodel for Java 5.0 together with a parser and printer, so that plain Java statements could be produced.

A combination of XSL-T and AspectJ also seems to be a promising approach to combine the advantages of each technology. This particularly fits smoothly to the existing implementation. That is why we intend to investigate the combination of both approaches, i.e., following (Henthorne and Tilevich, 2007) to generate aspects within the code to get the best out of both worlds.

REFERENCES

Anastasopoulos, M., Muthig, D., 2004. *An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology*. In ICSR 2004.

AspectJ. *Eclipse AspectJ Homepage*. http://eclipse.org/aspectj/, visited 2014-03-18.

Beuche, D., 2006. *Variant management with pure::variants*. Technical report, pure-systems GmbH, 2006. http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf, visited 2014-05-25.

Groher, I., Krüger, C., Schwanninger, C., 2008. *A Tool-Based Approach to Managing Croscutting Feature Implementations*. 7th Int. Conf. on AOSD, Brussels 2008.

Hohenstein, U., 2005. *Using Aspect-Orientation to Add Persistency to Applications*. Proc. of Datenbank-systeme in Business, Technologie und Web (BTW), Karlsruhe 2005.

Heidenreich, F. Johannes, J. Seifert, M. Wende, C., 2009. *Closing the Gap Between modelling and Java*. In Proc. of 2nd Int. Conf. on Software Language Engineering, Springer, Lecture Notes in Computer Science, 2009.

Henthorne, C., Tilevich, E., 2007. *Code Generation on Steroids: Enhancing Code Generators via Generative Aspects*. 2nd Int. Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (IWICSS'07).

Kästner, C., Apel, S., Batory, D., 2007. *A Case Study Implementing Features Using AspectJ*. In Proc. Int. Software Product Line Conference (SPLC), Kyoto 2007.

Kaboré, C., Beugnard, A., 2007. *Interests and Drawbacks of AOSD compared to MDE – A Position Paper*. 3rd Workshop on Aspects and Models, at 21st ECOOP 2007.

Kapteijns, T., Jansen, S., Houet, H., Barendse, R., 2009. *A Comparative Case Study of Model Driven Development vs Traditional Development: The Tortoise or the Hare*. In CTIT Proc. of 5th European Conf. on Model Driven Architecture, 2009.

Kiczales, G. et al., 2001: *An overview of AspectJ*. Proc. of 15th ECOOP , 2001.

Lee, K., 2006. *Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development*. In Proc. Int. Software Product Line Conference, 2006.

Liu, J., Lutz, R. Rajan, H., 2006: *The Role of Aspects in Modeling Product Line Variabilities*. In Proc. of 1st Workshop on Aspect-Oriented Product Line Engineering, GPCE, Portland (Oregon) 2006.

Lussenburg, V., van der Storm, T., Vinju, J., 2010. *Mod4J: A Qualitative Case Study of Model-Driven Software Development*. In: Model Driven Engineering Languages and Systems. Warmer Lecture Notes in Computer Science Volume 6395, 2010.

Mezini. M. Ostermann, K., 2004. *Variabilility Management with Feature-Oriented Programming and Aspects*. In Proc. of 12th Int. Symp. On Foundations of Software Engineering (FSE), Newport Beach (CA), 2004.

MPS. *JetBrains :: Meta Programming System*. http://www.jetbrains.com/mps/, visited 2014- 05-25.

Normén, F., 2007. *Remove code smell with AOP*. http://weblogs.asp.net/fredriknormen/archive/2007/11/29/remove-code-smell-with-aop.aspx, visited 2014-05-25.

Pinto, M., Fuentes, L. Fernández, L., Valenzuela, J., 2009. *Using AOSD and MDD to Enhance the Architectural Design Phase*. In: Proc. OTM'09.

Reichel, C. Oberhauser, R., 2004. *XML-Based Programming Language Modeling: An Approach to Software Engineering*. In: SEA 2004.

Smaragdakis, Y., Huang, S., Zook, D. 2004. *Program Generators and the Tools to Make Them*. In SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press 2004.

Strunk, W., 2007. *The Symphonia Product-Line*. Java and Object-Oriented (JAOO) Conf, Arhus, Denmark, 2007

Stein, D., Hanenberg, S., 2006. *Why Aspect-Oriented Software Development and Model-Driven Development are not the Same – A Position Paper*. Electr. Notes Theor. Comput. Sci. 163(1), 2006.

Stahl, T. Völter, M., 2006: *Model-Driven Software Development*. Wiley&Sons, 2006.

Tarr, P., Osher, H., Harrison, W., Sutton, S., 1999: *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. In 21st Int. ICSE 1999.

Völter, M., Groher, I., 2007: *Product Line Implementation using Aspect-Oriented and Model-Driven Software Development*. In: 11th Int. Software Product Line Conference (SPLC), Kyoto (Japan) 2007.

Xtend2. *Eclipse Xtend 2 Homepage*. http://www.eclipse.org/Xtext/#xtend2, visited 2014-03-18.

Xtext. *Eclipse Xtext Homepage*. http://www.eclipse.org/Xtext/, visited 2014- 05-25.