

Write Once, Deploy N: a Performance Oriented MDA Case Study

[Position Paper]

Pieter Van Gorp
Formal Techniques in Software Engineering
Universiteit Antwerpen, Belgium
pieter.vangorp@ua.ac.be
<http://www.win.ua.ac.be/fots/>

ABSTRACT

To focus the comparison of languages for model checking and transformation on criteria that matter in practical development, there is an urgent need for more, and more realistic, case studies. In this position paper, we first present the problem of developing distributed database applications that are optimized for concurrent data access. The problem constraints are avoiding vendor lock-in, making a proper separation of concerns, and enabling tool support for domain evolution. Then, we derive the requirements and tradeoffs for designing a language for model refinement and code generation based on the presented problem. After applying a conceptual transformation language to our case study, we derive general conclusions on composition, sequencing, inheritance, and design by contract for such languages.

Keywords

MDA, EJB, Performance Tuning, Portability

1. INTRODUCTION

In Model-Driven Software Engineering, the primary artifact for development are models, rather than conventional source code. By restricting modelling languages to well-defined domains, the complexity resulting from repetitive logic can be made implicit. Similarly, mappings from abstract to concrete metamodels can be used to abstract away the use of vendor specific logic. Whereas source code is input to a black-box compiler, models are input to white-box consistency checkers and transformation tools. Hence the need for developer-friendly languages for interacting with such tools. OMG has standardized on the OCL for model checking and is currently working on standard languages for transforming models to other models and to source code ([Object Management Group, 2002, 2004](#)). Criteria for objectively comparing alternative languages for model checking and transformation are still premature. For example, several industrial submit-

ters have conflicting opinions about the declarative nature of a model transformation language ([Gardner et al., 2003](#)).

Without realistic case studies, it is not clear what language criteria really matter in practical MDA development. In this paper, we present the problem of developing distributed database applications that are optimized for concurrent data access. The problem constraints are avoiding lock-in on vendor extensions of a particular J2EE application server ([Brown et al., 1999](#)), making a proper separation of concerns, and enabling tool support for domain evolution.

This paper is organized as follows. Section 2 introduces our running example: a performance oriented middleware pattern. Section 3 derives the requirements for a transformation language for model refinement and code generation. In Section 4, we discuss the tradeoffs of designing a transformation language by developing parts of a code generator for our case study with a conceptual transformation language. We conclude this paper by generalizing the early results of our case study to desirable properties of transformation languages for practical MDA development and discussing future work.

2. WRITE ONCE, DEPLOY N (WODN)

Distributed server components are deployed on an application server that delivers the middleware services from a platform like J2EE or .NET. These services are configured by attaching deployment attributes to the component sources. After inheriting from the appropriate component model classes (or interfaces), the remaining Java or C# code can focus on business logic, rather than non-functional aspects such as transaction demarcation, persistence, caching, and clustering.

2.1 Towards Portable Server Applications

The J2EE platform makes the distinction between vendor independent and vendor specific deployment attributes. The design goal “Write Once, Deploy Anywhere” has been accomplished for the vendor independent deployment descriptor and the Java source files. However, any realistic server component will require the usage of vendor specific files with at least some network distribution information and in most cases an object-relation mapping before it can be deployed on an application server. The good news is that this information tends to be very similar across all the components of

a server application. Along with the repetitive structure of their Java sources, this makes J2EE components a natural candidate for code generation.

Today's MDA tools have built-in code generators for the leading server component models and their application servers (Compuware, 2004; Bohlen et al., 2004). They reduce complexity and initial development time by providing reasonable defaults for most deployment properties. Kleppe et al. describe a mapping for Enterprise JavaBeans that characterizes how existing MDA tools generate code for server components: each platform independent entity will eventually have one corresponding server deployment (Kleppe et al., 2003). Application server migration comes down to generating default deployment information for the new (version of the) product.

2.2 Performance Issues

While the default deployments generated by today's MDA tools are extremely useful for rapid prototyping on different application servers, it is a waste of server resources to run them as such in a production environment. Still, this antipattern occurs more than one may expect, even outside the case of generated systems.

Tyler Jewell investigated this issue and observed that the reason for this performance problem is that a default deployment must have its cache and transactions configured conservatively for concurrent read-write data access while production systems tend to have as much as 85% read-only data access, only 10% read-write data access and 5% batch update access (Jewell, 2001).

A solution that is often suggested is to limit the use of the EJB component model to transactional read-write data access and take a shortcut to the database layer in the case of read-only or batch-update data access (Tate and Flowers, 2002). As with a lot of performance hacks, this approach spoils the integrity of the overall architecture.

Jewell demonstrated that the problem can also be solved without dismissing the EJB component model. The proposed "Write Once, Deploy N times" (WODN) pattern fully utilizes application server implemented optimizations like lazy loading and distributed cache invalidation by deploying the same Java sources one time for each data access scenario. The read-only deployment is not configured with the strict transaction settings from the write deployments. Moreover, the application server sends selective invalidation messages from the write deployments to the read-only cache.

2.3 Goal

It has not yet been investigated how performance patterns, such as WODN, can be efficiently implemented on various application servers. We will demonstrate that model-driven engineering is a promising approach as it can reconcile performance optimization with portability, separation of concerns and tool support for domain evolution.

3. TRANSFORMATION LANGUAGE

In this section, we start from a concise taxonomy of model transformation to subsequently derive domain specific language constraints for the different transformation forms.

3.1 Model Transformation Taxonomy

Model transformation can be roughly divided into two sub-domains: rephrasing and translation (Visser et al., 2004).

Rephrasing is transforming a program into a different program in the *same* language. The source and target metamodel of the transformations are the same. *Refactorings* (i.e. restructurings for object-oriented software) are probably the most widely known examples of rephrasings (Du Bois et al., 2004). A *restructuring* is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics) (Chikofsky and Cross, 1990). Note that with a narrow definition of functionality, program optimizations are restructurings as well. Moreover, the aspect of behavior preservation is not a transformation language property, yet a transformation specification property.

Translation Based on the above definitions, we can define a translation as the transformation of a program across different metamodels. In program *refinement* (and its inverse *analysis*) a chain of translations is applied with metamodels at different relative abstraction levels. Translations are also applied to maintain consistency between specifications in different languages, yet at the same level of abstraction. In this context, Schür has demonstrated how correspondence rules between the graph grammars of the two such languages can be used for bidirectional consistency maintenance (Schürr, 1995).

3.2 Language Requirements

Languages for rephrasing need to support model transformations within the same metamodel, whereas languages for translation need to support mappings between abstract and concrete metamodels. Graph rewriting is an interesting formalism for specifying restructurings as its formal theory supports correctness proofs (Mens et al., 2002) and existing tools can be used to execute the constraint and transformation specifications (Van Gorp et al., 2003). In figure 1, the transformation of the Pull Up Method refactoring is specified in the UML / graph rewriting language called Story Driven Modeling (Fischer et al., 1998).

As illustrated by our case study, languages for refinement play a more crucial role in the generative MDA process: how can we elegantly specify the mapping from a platform independent business model to a detailed component specification that is optimized for heavy server load?

1. Definitely, the transformation language needs to support mappings across different metamodels. Our case study requires a domain metamodel that has no notion of the performance pattern, neither from other aspects such as object to relational mapping.

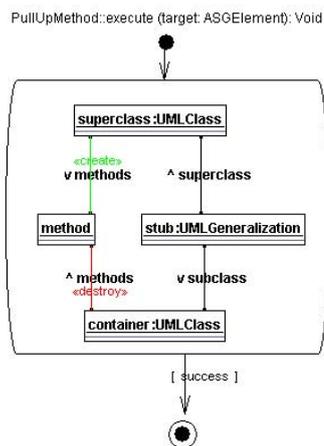


Figure 1: An executable refactoring specification in the Story Driven Modeling language. The `execute` method of the “The Pull Up Method” class has one parameter, `target`, representing the model element on which the transformation applies. The node `method` contains this parameter after casting it to `UMLMethod`. The edges `methods`, `subclass` and `superclass` specify a pattern that checks if the method has a superclass. If so, the link to the containing class is destroyed and a link to the superclass is created. Note that this approach loads the metaclasses (MOF M2), on which the transformation is specified, as plain model classes (MOF M1). This does not form a problem for the Fujaba tool, since there are no structural differences between M1 and M2. This does not hold for the current UML and MOF standards.

2. It is desirable that different aspects of the refinement process can be specified independently. This decomposition of transformation enhances their understandability and reusability.
3. It must be possible to parameterize the refinement process *between* the abstract and concrete metamodels, instead of *on* one of them. This makes the variability of code generation components explicit.
4. Application servers from the same component model often extend the standard with the same conceptual services. However, these services are configured by different deployment parameters, in different files. It must be possible to model the specialization of this common transformation behavior in the refinements for these similar application servers. This is especially relevant in the context of the J2EE platform, where there is a practical boundary on the level of standardization that vendors are willing to commit.
5. A refinement language should support the resolution of evolution conflicts where possible. Among the possible scenario’s are manual manipulation of generated artifacts and integration with other software engineering tools. Ideally, a transformation language allows variability in the conflict resolution, such that tools can prompt for user interaction.
6. It should be possible to query transformation specifications stored in a “refinement repository” (Van Gorp et al., 2003). This is required when, for example,

a Rename Entity refactoring is executed on a business model. After regenerating all derived models and code, one needs to execute a series of primitive Rename Class refactorings for updating all the manual code that makes use of the generated classes. Note that a query on a refinement repository returns transformation models, that are in turn specifications on the metamodels of the generated software. Along with the previous item, this feature motivates why strict imperative transformation languages do not suffice.

7. Transformation specifications should be repository independent. This can be achieved by applying the MDA principle of code generation from platform independent models to transformation models themselves. As an initial experiment, we have defined a UML profile for specifying refactorings as graph rewritings (similar to figure 1.) We use the AndroMDA tool to drive code templates (model to code transformations) that generate repository access code from instances of the profile (Bohlen et al., 2004). Figure 2 displays a part of such a code template we are developing for JMI based repositories.

JMI is the Java implementation of the MOF standard for repository access, which should make our transformations already executable on a number of repositories (Java Community Process, 2002). However, we can extend our repository support to non-MOF repositories like EMF by adding a new set of model to code transformations (Eclipse Foundation, 2002).

4. GENERATING WODN COMPONENTS

As discussed in Section 2.3, our goal is to use model-driven engineering techniques to reconcile performance optimization with portability, separation of concerns and tool support for domain evolution. In this section, we illustrate the tradeoffs of designing a transformation language by developing parts of a code generator for the “Write Once, Deploy N” pattern with a conceptual transformation language. We restrict this paper to generating all vendor specific artifacts from an abstract business model. Pattern and vendor independent client access (from which a model checker derives whether read-only, read-write or batch-update needs to be called) is outside the scope of this paper.

4.1 Designing PIM & PSM Metamodels

Figure 3 presents the metamodel for the pattern and vendor independent specification of models from the problem domain. It does not add any properties to the standard UML model elements. Each “Entity” should be automatically refined to a persistent component.

At the next level of abstraction, we want to specify the transaction and caching attributes for the three participants in the WODN pattern. Figure 4 displays the metamodel for specifying such models, without locking in on concrete vendor attributes. This enables us to reuse the pattern refinement across code generators for different application servers.

Another aspect of our database application is persistence. Let us consider the mapping to a relational database. The

```

1  ## Velocity Template
2  ##
3  import java.util.*;
4
5  import javax.jmi.model.MofClass;
6  import javax.jmi.reflect.RefPackage;
7  import javax.jmi.reflect.RefClass;
8
9  /**
10 * Code generated by JCMTG
11 */
12 public class $class.name {
13
14 ...
15 /**
16 * Find JMI Class Proxy with specified name in specified
17 * package (or subpackage).
18 */
19 private RefClass jcmtg_findClassProxy(String name,
20                                     RefPackage pkg) {
21     Collection classes = pkg.refAllClasses();
22     for (Iterator it = classes.iterator();
23          it.hasNext(); ) {
24         RefClass c = (RefClass) it.next();
25         if (((MofClass) c.refMetaObject()).getName()
26             .equals(name))
27             return c;
28     }
29     // not found in package
30     // now look in subpackages
31     Collection subpkgs = pkg.refAllPackages();
32     for (Iterator it = subpkgs.iterator();
33          it.hasNext(); ) {
34         RefClass c = jcmtg_findClassProxy(name,
35                                     (RefPackage) it.next());
36         if (c != null) return c;
37     }
38     return null;
39 }
40
41 #set ($transOps=$transform.getTransformationOperations($class))
42 #foreach ($transOp in $transOps)
43 #set ($transFlow=$transform.getTransformationFlow($transOp))
44 #parse ("templates/TransFlow.vsl")
45 #end
46 }

```

Figure 2: *Code Template fragment*. Any ordinary text, such as the Java code from lines 3 through 11, will be copied verbatim to the generated file. Line 12 shows how properties from model elements can be accessed with the \$ notation. Lines 41 through 45 illustrate the use of scripting commands for assignments, control flow and delegation to other scripts.

persistence aspect can be refined independently from the transaction and caching aspect module.

The next level of abstraction joins the above two aspects and maps them to a concrete application server. Instead of building a full-fledged metamodel for each application server and refining our models one more time, we choose to transform the models to text. Our motivation lies in the fact that there is already a code generator that transforms annotated Java files into all component sources of the leading J2EE application servers ([xDoclet Team, 2004](#)). By generating code as input for xDoclet we can reuse the mappings from one bean description to the remote and local bean interfaces, their abstract factories (remote and local home interface) and possibly a primary key and data transfer class. Although this simplifies the presentation and implementation of our first case study iteration, xDoclet's design may not meet requirements 5 to 7 from Section 3.2.

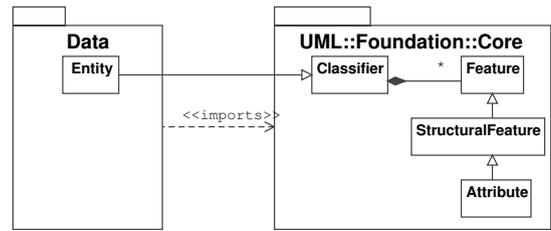


Figure 3: *The Data metamodel*.

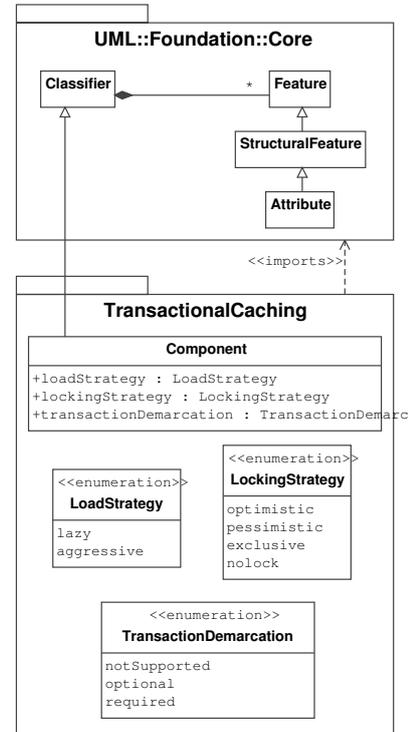


Figure 4: *The TransactionalCaching metamodel*.

4.2 Transformation Choreography

Figure 6 displays a control flow diagram of the proposed refinement from pattern and vendor independent models to concrete component files. The notation is based on UML activity diagrams. Models are transformed from one metamodel to the other. Metamodels are displayed as regular states (with rounded rectangles) and transformations are displayed as object flow states (with regular rectangles). Using synchronization bars, one can visualize the independent aspects of the refinement process.

4.3 Transformation Signature

Transformations can have more than one input metamodel. Consider for example the `Components2xDoclet` transformation on figure 6. Generally, transformations that join (or “weave”) require one input metamodel per view (or “aspect”). It is also possible that a transformation has multiple output metamodels. Although not strictly required to solve the case study, it may be desirable to output an automated text description as documentation on a model refinement. In

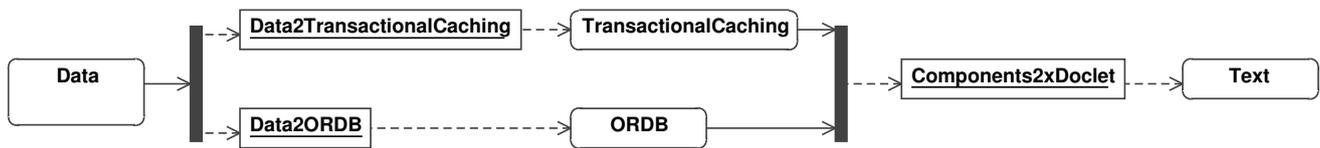


Figure 6: The overall refinement process for the “Write Once, Deploy N” pattern.

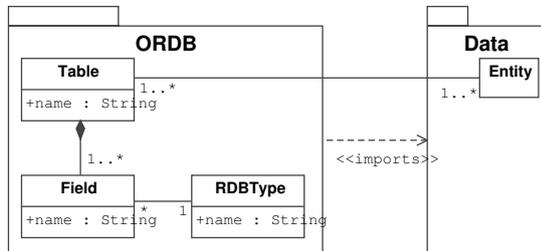


Figure 5: The ORDB metamodel.

such scenario, users can zoom in to the next level of models or zoom to the text description.

4.4 Transformation Rules

Figure 7 illustrates how specific OCL postconditions can be used to specify the relationship between the abstract and concrete metamodels in a purely declarative fashion. The rule states that for all entities in the abstract domain (Data), there should be a corresponding read-only component in the concrete domain (TransactionalCaching). The rules for the read-write and batch update components follow the same approach, yet set other values for the deployment attributes. The rule from the Data2ORDB transformation can be specified declaratively as well: for each entity in the Data domain, there should be an entity linked to table with the appropriate fields in the ORDB domain.

```

1 Transformation Data2TransactionalCaching
2   from: {Data},
3   to: {TransactionalCaching} {
4     ...
5   Rule Entity2RO_Component () {
6     postcondition:
7       Entity.allInstances->forall(e |
8         Component.allInstances->exists(c |
9           e.Classifier = c.Classifier and
10          c.lockingStrategy = LockingStrategy::noLock and
11          c.transactionDemarcation =
12            TransactionDemarcation::optional
13        )
14      )
15    }
16    ...
17  }
  
```

Figure 7: The *Entity2RO_Component* rule from the *Data2TransactionalCaching* transformation. By using specific OCL constructs, such as `exists`, one can rely on the transformation engine to automatically satisfy the constraint. Line 11 shows how a syntactic shorthand for comparing all the fields of a common superclass.

The postconditions can be monitored by the transformation engine and trigger a semi-automatic resolution process for failed assertions. The degree of automation depends on the phase the software lifecycle. If the *exists* predicate of the postcondition fails in the first iteration of the development cycle, the engine can simply instantiate an object that satisfies all the comparisons. After this initial instantiation, fully automatic resolution of such failed assertions is probably not feasible anymore. However, by proper inspection of the assertions, a tool may still present a set of corrective transformations on existing model elements to the user.

4.5 Traceability Models

A traceability model consists of a set of tuples that maintain the relationship between abstract and concrete model elements. These tuples allow users to browse from abstract to concrete concepts and visa versa after the transformations have been applied. We are investigating how these tuples can be used for maintaining the consistency across abstraction layers while each layer is subject to evolution.

4.6 Integrated Code Templates

Model (to model) transformation is currently one of the most active points of research to improve the power of existing MDA tools (Sendall and Kozaczynski, 2003). Due to the lack of proper model refinement techniques, early MDA tools try to transform abstract models directly into code, using template languages as illustrated by figure 2. This leads to too much complexity in the templates, as illustrated by figure 8 (Bohlen, 2004).

Still, code template languages provide a very intuitive means to generate code from models at a low level of abstraction. Therefore, we propose to integrate model transformation languages with code template languages. A code template can be considered as an imperative rule: instead of specifying a match pattern, it is explicitly called by other rules. Therefore, code templates can only access model elements through their formal parameter list.

4.7 Platform Specialization

In the last step of the transformation process, `Components2xDoclet` maps information from the `TransactionalCaching` and `ORDB` metamodels to an input file for the `xDoclet` code generator. The idea behind `xDoclet` is that deployment attributes are specified in special documentation tags within a source file, instead of in separate XML deployment descriptors.

The vendor lock-in problem with `xDoclet` is that the deployment attributes are still completely vendor specific and there is no elegant mechanism to refine user-defined properties to these low-level properties. We will illustrate how this prob-

```

1 #foreach ( $op in $class.operations )
2 #if ($transform.getStereotype($op) == "FinderMethod")
3 * @ejb.finder signature="{transform.findFullyQualifiedName(\
4 $op.getType())} ${transform.getOperationSignature($op)}"
5 #set($viewtype = "")
6 #set($viewtype = $transform.findTagValue($op.taggedValues, \
7 "@andromda.ejb.viewType"))
8 #if($viewtype == "local" || $viewtype == "remote" || \
9 $viewtype == "both")
10 * view-type="$viewtype"
11 #end
12 #set($querystring = "")
13 #set($querystring = $transform.findTagValue($op.taggedValues, \
14 "@andromda.ejb.query"))
15 #if($querystring == "")
16 #set($querystring = "SELECT DISTINCT OBJECT(c) FROM \
17 $class.name AS c")
18 #if($op.parameters.size() > 0 )
19 #set($querystring = "${querystring} WHERE")
20 #foreach($prm in $op.parameters)
21 #set($querystring="${querystring} c.$prm.name = \
22 ?$velocityCount")
23 #if($velocityCount != $op.parameters.size())
24 #set($querystring = "${querystring} AND")
25 #end
26 #end
27 #end
28 #end
29 * query="$querystring"
30 #end##if op.stereotype = "FinderMethod"
31 #end##foreach operation

```

Figure 8: *Fragment from EntityBean.vsl*. This code template illustrates the complexity resulting from a too abstract input metamodel.

lem can be overcome by using a transformation language with polymorphism.

In our case study, we need to map the vendor neutral properties from the enumerations in the `TransactionalCaching` domain to concrete properties for our target application servers, say JBoss and BEA WebLogic ([JBoss Group, 2004](#); [BEA Systems, 2004](#)).

Figure 9 displays how the abstract `Components2xDoclet` transformation is specialized by a concrete transformation for each target application server. These vendor specific transformations implement the `Join2ClassTags` and `Join2MethodTags` rules by outputting the xDoclet tags that realize the properties from the transactional caching and persistence aspects on WebLogic and JBoss respectively. Compared to conventional inheritance, transformations play the role of classes while rules play the role of methods.

Figures 9 and 10 are intended to give an idea of the conceptual structure of the abstract transformation and the concrete transformation for WebLogic.

5. CONCLUSIONS AND FUTURE WORK

In this position paper, we presented a complex middleware pattern as a realistic case study for model-driven development. Based on this case study, we developed a conceptual transformation language.

We can already derive some preliminary conclusions on the desirable characteristics of such languages: refinement languages need to support mappings from abstract to concrete metamodels. Parallelism in the refinement process enables

```

1 abstract Transformation Components2xDoclet
2 from: {TransactionalCaching,ORDB},
3 to: {Text} {
4
5 Rule Join2JavaFile (TransactionalCaching::Entity e1,
6 ORDB::Entity e2) {
7 // code template fragment for Java imports
8 // code template fragment for conventional Javadoc
9 #call Join2ClassTags(e1,e2);
10 ...
11 // code template fragment for iterating over methods
12 ...
13 #call Join2MethodTags(e1,e2);
14 ...
15 ...
16 }
17
18 abstract Rule Join2ClassTags (TransactionalCaching::Entity e1,
19 ORDB::Entity e2) {
20 // code template for vendor independent xDoclet class
21 // tags like @ejb.finder, @ejb.home, @ejb.interface, ...
22 }
23 ...
24 }

```

Figure 9: *Fragment from Components2xDoclet.transfo*.

the separation of concerns. Transformation choreography diagrams can elegantly visualize such parallelism. Inheritance hierarchies can be used to make variability in the refinement process explicit to the user of the transformation library.

OCL postconditions are a promising approach to the declarative specification of model refinements. By carefully inheriting from common superclasses in the design of metamodels for PIMs and PSMs, one can rely on syntactic shorthands for the specification of such postconditions. Research on model refinement should focus on semi-automatic evolution conflict resolution strategies. The use of traceability models seems promising for this purpose. Finally, at a low level of abstraction, imperative code templates can take over the refinement process from the declarative model (to model) transformations. The transition between these paradigms needs some further investigation. More specifically, there is a need for rules that declaratively bind model elements and imperatively call the code templates upon the result.

We will validate these results using the framework of the ATLAS Transformation Language ([Bézivin et al., 2003](#)).

```

1 abstract Transformation Components2WL
2 inherits Components2xDoclet {
3
4 // Join2JavaFile inherited, not overridden
5
6 Rule Join2ClassTags (TransactionalCaching::Entity e1,
7 ORDB::Entity e2) {
8 #call super.Join2ClassTags(e1,e2);
9 // code template for WebLogic specific xDoclet class
10 // tags like @weblogic.persistence, @weblogic.cache,
11 // @weblogic.invalidation-target, ...
12 }
13 ...
14 }

```

Figure 10: *Fragment from Components2WL.transfo*.

References

- Object Management Group. MOF 2.0 Query / Views / Transformations RFP ad/2002-04-10, October 2002. URL http://www.omg.org/cgi-bin/apps/do_doc?ad/02-04-10.pdf.
- Object Management Group. Draft MOF Model to Text Transformation RFP ad/2004-01-13, January 2004. URL <http://www.omg.org/cgi-bin/apps/doc?ad/04-01-13.pdf>.
- Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A review of OMG MOF 2.0 Query / Views / Transformations submissions and recommendations towards the final standard, 2003.
- William Brown, Raphael Malveau, Hays McCormick, Thomas Mowbray, and Scott W. Thomas. Vendor lock-in, December 1999. URL <http://www.antipatterns.com/vendorlockin.htm>.
- Compuware. OptimalJ, February 2004. URL <http://www.compuware.com/products/optimalj/>.
- Matthias Bohlen et al. AndromDA, February 2004. URL <http://andromda.sourceforge.net/>.
- Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Object Technology Series. Addison – Wesley, 2003.
- Tyler Jewell. Unlocking the true power of entity EJBs, 2001. URL <http://www.onjava.com/pub/a/onjava/2001/12/19/eejbs.html>.
- Bruce A. Tate and Braden R. Flowers. *Bitter Java*, chapter 8.4.4 and 8.4.5. Manning, 2002.
- Eelco Visser et al. Program-transformation.org – a taxonomy of program transformation, February 2004. URL <http://www.program-transformation.org/Transform/ProgramTransformation>.
- Bart Du Bois, Pieter Van Gorp, Alon Amsel, Niels Van Eetvelde, Hans Stenten, Serge Demeyer, and Tom Mens. A discussion of refactoring in research and practice. Technical report, University of Antwerp, January 2004.
- Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1): 13–17, 1990.
- Andy Schürr. Specification of graph translators with triple graph grammars. In *Proceedings 20th Workshop on Graph-Theoretic Concepts in Computer Science WG 1994*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1995.
- Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002. Proc. 1st Int’l Conf. Graph Transformation 2002, Barcelona, Spain.
- Pieter Van Gorp, Niels Van Eetvelde, and Dirk Janssens. Implementing refactorings as graph rewrite rules on a platform independent metamodel. In Holger Giese and Albert Zündorf, editors, *Proceedings of the 1st International FUJABA Days*, pages 17–24. University of Kassel, Oktober 2003.
- T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.
- Java Community Process. Java metadata interface (JMI) specification – JSR 000040, June 2002. URL <http://java.sun.com/products/jmi/>.
- Eclipse Foundation. The Eclipse modeling framework (EMF) overview, September 2002. URL <http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/emf-home/docs/overview.pdf>.
- xDoclet Team. xDoclet, February 2004. URL <http://xdoclet.sourceforge.net/>.
- Shane Sendall and Wojtek Kozaczynski. Model transformation - the heart and soul of model-driven software development. *IEEE Software, Special Issue on Model Driven Software Development*, 20(5):42–45, 2003.
- Matthias Bohlen. Metamodel Decorators, February 2004. URL <http://team.andromda.org/tiki/tiki-index.php?page=MetaModelDecorators>.
- JBoss Group. The JBoss application server, 2004. URL <http://www.jboss.org/>.
- BEA Systems. Weblogic server, 2004. URL <http://www.bea.com/products/weblogic/server/>.
- Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, October 2003.