

# Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Metamodel

Pieter Van Gorp  
Lab on Re-Engineering  
University of Antwerp  
pieter.vangorp@ua.ac.be

Niels Van Eetvelde  
Formal Techniques in  
Software Engineering  
University of Antwerp  
niels.vaneetvelde@ua.ac.be

Dirk Janssens  
Formal Techniques in  
Software Engineering  
University of Antwerp  
dirk.janssens@ua.ac.be

## ABSTRACT

Increasingly more developers are applying refactorings - program transformations that can improve the design of existing source code - to make their software more easily adaptable to new requirements. Because small changes to object-oriented software (such as renaming a class) can require a lot of updates to several source files, tools that automatically update the affected files can save these developers a lot of time. Although refactorings are based on basic OO concepts (the redistribution of classes, variables and methods across the class hierarchy) only, today's development environments have hardcoded them on the abstract syntax trees of programming languages such as Java or C# and do not update middleware deployment descriptors. To facilitate the building of new refactorings and the extension of existing ones to new platforms, we suggest to implement refactorings as declarative specifications on a platform independent metamodel. This paper describes how the metamodel, the graph rewrite language and the architecture of the Fujaba UML tool can be extended to provide the required infrastructure.

## Keywords

Refactoring, Metamodeling, Graph Rewriting, Model Transformation, Middleware, Code Preserver, UML, SDM

## 1. CONTEXT

A *refactoring* is defined as a "behavior preserving program transformation" [1]. Refactorings for OO software are based on the redistribution of classes, variables and methods across the class hierarchy, mainly for the purpose of facilitating future adaptations and extensions [2].

In order to maintain the system's integrity, a *refactoring tool* needs to update the source code references affected by a refactoring. It also needs to make sure that a refactoring is only executed when it is guaranteed not to introduce inconsistencies. Regarding the automatic updating of source code references, current generation refactoring tools do not take into account middleware deployment descriptors which obviously leads to deployment conflicts after refac-

toring. Furthermore, they give no formal guarantees on behavior preservation. Formal proofs rely on the correctness of the pre- and postconditions of the implemented refactorings. Existing refactoring implementations have hardcoded these constraints with third generation programming languages.

*Fujaba* is an open source UML CASE tool that was originally designed for Java code generation from Story Driven Modeling (SDM) specifications [3]. SDM is a visual programming language based on UML and graph rewriting. Graph rewriting is a feasible formalism to reason about the behavior preservation of refactorings [4]. In this paper, we report how we are extending *Fujaba* for implementing refactorings as SDM specifications on a platform independent metamodel without introducing inconsistencies in middleware deployment descriptors.

## 2. METAMODEL REQUIREMENTS

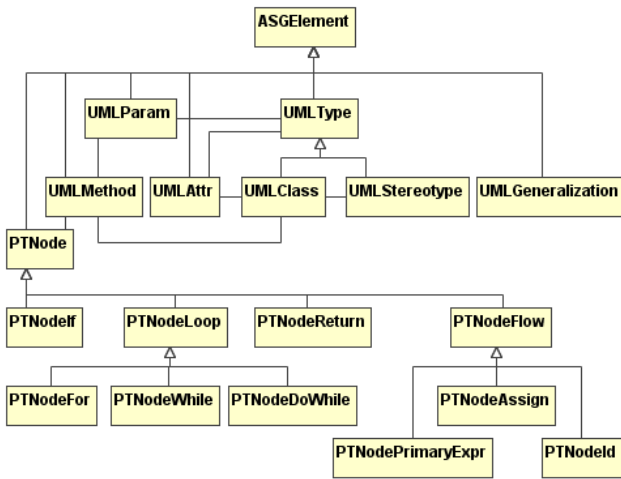
As Don Roberts explains in his Ph.D. thesis [5], building a refactoring tool involves more than implementing the program transformations. The tool should also be able to check the invariants, pre- and postconditions of a (sequence of) refactoring(s) to ensure source-consistency. Therefore, the tool needs a sufficiently expressive metamodel. Similarly, a developer may want to trigger refactorings based on the presence of the bad code smells they can solve. In this section, we evaluate whether *Fujaba's* metamodel is suitable for these purposes.

### 2.1 Evaluation of the Fujaba 4.0 Metamodel

*Fujaba's* metamodel consists of 2 layers of abstraction, physically separated by a lazy parser (see Figure 1).

The first layer is equivalent to the UML 1.4 metamodel which contains the coarse OO constructs (such as namespaces, classes, operations and attributes) but excludes all the method body information which is required for maintaining the consistency between the parsed model and the rest of the code when executing a refactoring transformation [6].

The second layer of *Fujaba's* metamodel refines the method body as a partial Java abstract syntax tree. Although the *if*, *for*, *while* and *assignment* constructs could be considered relatively platform independent (they occur in C++ and C# as well), they do not fit our refactoring purposes. On the one hand, one does not need to understand the difference between conditionals and loops: it only matters that a new variable scope is introduced. On the other hand, *Fujaba's* method body syntax tree does not contain the explicit *ac-*



**Figure 1:** Fujaba 4.0 metamodel. *ASGElement* is equivalent to the standard UML *ModelElement* entity. We can reuse *UMLClass*, *UMLGeneralization*, *UMLAttr* and *UMLParam* from the first layer of this metamodel. All *PTNode* subtypes are part of the second layer. They reflect a partial Java abstract syntax tree and do not fit our refactoring purposes.

*cess*, *update* and *call* information that is needed to reason about refactoring.

Therefore, we propose to reuse the first layer of Fujaba’s metamodel but use a new second metamodel layer that is minimal, yet adequate for refactoring.

## 2.2 Running Example

To illustrate the need for a metamodel extension, we highlight the tool requirements for automating a realistic Java middleware refactoring scenario. Our running example will be the “get cart items” operation from the shopping cart class from the EJB implementation of the open source xPetstore application [7]. Enterprise JavaBeans (EJB) is a standard component model for developing the application tier of a web application [8]. EJB components are managed by a container that interfaces with application server resources. Services such as object distribution, resource and transaction management and security are configured by specifying deployment attributes in an XML deployment descriptor.

```

1 /**
2  * @return Return a list of {@link CartItem} objects
3  *
4  * @ejb.interface-method
5  * @ejb.transaction-type
6  * type="NotSupported"
7  */
8 public Collection getCartItems() {
9     try {
10         ItemLocalHome ihome= ItemUtil.getLocalHome();
11         ArrayList items= new ArrayList();
12         Iterator it= _details.keySet().iterator();
13         while (it.hasNext()) {
14             String key= (String)it.next();
15             Integer value= (Integer) _details.get(key);
16             try {
17                 ItemLocal ilocal= ihome.findByPrimaryKey(key);
18                 ItemValue item= ilocal.getItemValue();
19                 ProductValue prod= ilocal.getProduct()

```

```

20         .getProductValue();
21
22         CartItem ci = new CartItem(item.getItemId(),
23             prod.getProductId(),
24             prod.getName(),
25             item.getDescription(),
26             value.intValue(),
27             item.getListPrice());
28
29         items.add(ci);
30     }
31     catch (Exception cce) {
32         cce.printStackTrace();
33     }
34 }
35 // Sort the items
36 Collections.sort(items,
37     new CartItem.ItemIdComparator());
38 return items;
39 }
40 catch (Exception e) {
41     return Collections.EMPTY_LIST;
42 }
43 }

```

## 2.3 Metamodel Extensions for Refactoring

### 2.3.1 Motivating Refactoring Scenario

Suppose we are reading *CartEJB.java* in order to better understand the system. Once we understand the undocumented try block from line 17 to line 29, we decide to extract its body into a new method with a name that fits the intention<sup>1</sup>. We will briefly share some details on the design of xPetstore and the EJB component model in general and then suggest a method name.

We know that the *\_details* attribute represents a hash map from product key strings to the integer amount of items of that type in the cart. Line 17 uses the key of one product in the cart to retrieve a reference to the local item bean from the local item home object. Consequently, the home object checks whether the application server has a cached instance of the involved item and otherwise adds a new object instance for the item *ilocal* bean to an instance pool. The *ilocal* bean retrieves all its data from the underlying database unless it is configured with lazy loading. On line 18, a value object *item* encapsulating all this data is retrieved from the *ilocal* bean. Line 19 navigates to the product bean associated with the item bean and also constructs a value object for it. The data from these two value objects is then used from lines 22 to 26 to construct a value object representing the appropriate amount of cart items of the product from the current while loop iteration. Finally, on line 29, this value object is added to the list of items that our sample method is supposed to compose. What about extracting lines 17 to 27 to a method called *buildCartItemVOfromDB*? The while loop would then look like:

```

13 while (it.hasNext()) {
14     String key= (String)it.next();
15     Integer value= (Integer) _details.get(key);
16     try {
17         CartItem ci= buildCartItemVOfromDB(ihome,
18             key, value);
19         items.add(ci);
20     }
21     catch (Exception cce) {
22         cce.printStackTrace();
23     }
24 }

```

<sup>1</sup>This reengineering activity is commonly referred to as the *refactor to understand* pattern [9].

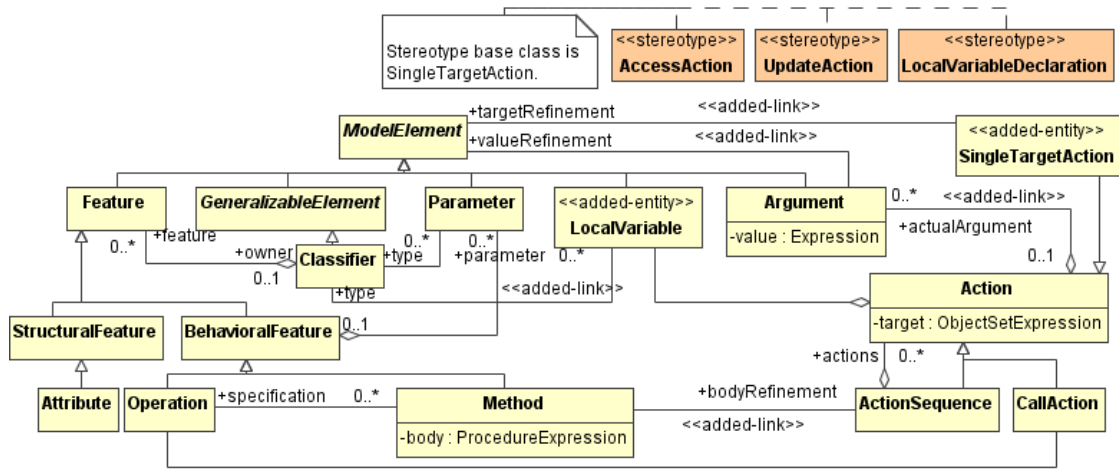


Figure 2: Overall view on the GrammyUML metamodel. The *added-entity* and *added-link* stereotypes highlight what is added to the UML 1.4 metamodel.

Note that we could move the declaration from *ihome* inside the try block as it is only used there, but we decide not to do it because retrieving the home interface of a bean is a resource-intensive operation.

### 2.3.2 Refactoring Implementation

Let us now consider the internals of this extract method refactoring. The precondition of this refactoring states that (i) the signature of the method that needs to be created may not result in a name conflict in the inheritance hierarchy of the containing class, (ii) there should be no return statement among the extracted statements, and (iii) within the selected source fragment, there should be at most one update to a local variable. The postcondition states that (i) there should exist a method with the chosen name in the containing class, (ii) each local variable that is used in the extracted fragment but not declared in it should be a parameter of the new method, (iii) each formal parameter that is used by the extracted statements should be a parameter of the new method, (iv) all checked exceptions that are thrown from the extracted statements (or from a method called by these statements), and that are not handled by the extracted statements, should be listed in the signature of the new method, and finally (v) in the original method, the extracted statements should be replaced by a method call to the new method.

To be able to express these constraints, our refactoring metamodel should include a *return action* (e.g. for precondition ii), an *update action* (e.g. for precondition iii), a *local variable declaration action* (e.g. for precondition ii), an *access action* (e.g. for precondition ii and iii), a *throw action* and a *handle action* (e.g. for precondition iv), and finally a *call action* (e.g. for precondition iv and v).

“Extract method” is an example of a refactoring that can only be executed by looking at the source code. It should be noted that this is not the cause of the need for all of our metamodel extensions. The “pull up method” refactoring for example can be triggered from a class diagram and also requires reasoning about method calls and accesses or updates to attributes. A part of its precondition for instance states that the method under consideration should not make any references (accesses or updates) to attributes that are part of

the subclass. Line 12 of our running example accesses the attribute *\_details* and calls the *keySet* operation for this object. Our sample method can not be pulled up because the superclass of *EJBCart* (i.e. *javax.ejb.SessionBean*) does not own or inherit the *\_details* attribute. Without our metamodel extension, it would be impossible to check this precondition. Similarly, call behavior reasoning is required for another part of the precondition: no methods that are defined in the subclass should be called.

### 2.3.3 Proposed Extensions

GrammyUML is a minimal and backward compatible extension of the standard UML metamodel that is adequate to reason about refactoring (compose primitive refactorings, verify preservation of behavioral properties, and trigger refactorings based on code smells) [10]. As Fujaba’s first metamodel layer is largely equivalent to the UML standard, all GrammyUML extensions apply to Fujaba as well. All primitives in the method body that are needed to reason about refactoring are modeled as variants of the standard UML “action” entity. *CallAction* is a standard UML construct that can be used to model call behavior in Fujaba. Accesses and updates can apply to attributes, parameters and local variables. The latter are not yet part of Fujaba’s metamodel. We add a *LocalVariable* construct and associate it with its type. *LocalVariableDeclaration* defines where a local variable is defined. Its scope reaches to the last *Action* of the containing *ActionSequence*. The try block from line 9 to line 39 for instance is an *ActionSequence* that implicitly defines the scope of *ihome*, *items* and *it*. Also note the anonymous local variables introduced by *\_details.keySet()* on line 12 and *new CartItem.ItemIdComparator()* on line 37. Next to access, update and call behavior, we also need the notion of a return statement, and need to be able to reason about exceptions. Note that Fujaba’s second metamodel layer would provide us a *PTNodeReturn* construct but it does not link this construct to the object that is returned. We provide this link in the *SingleTargetAction* construct whose *targetRefinement* can be reused by all its subclasses or stereotypes (*AccessAction*, *UpdateAction*, *LocalVariableDeclaration*, *ReturnAction* and *ThrowAction*). Figures 2 and 3 summarize the proposed metamodel extensions.

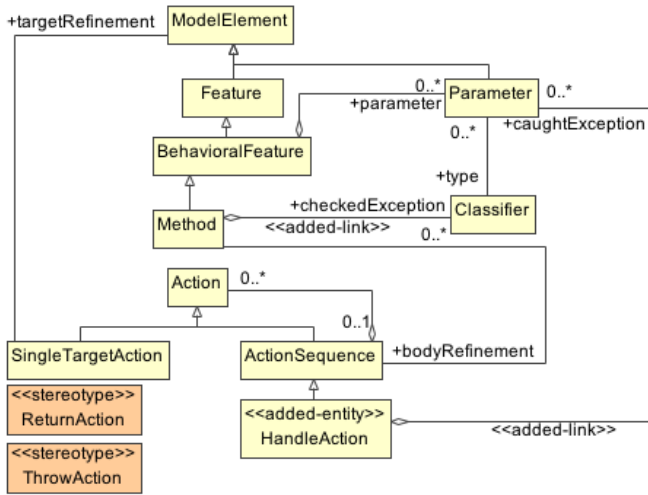


Figure 3: GrammyUML metamodel fragment to model exception throwing and catching and returning the flow of control from a method.

### 3. IMPLEMENTING REFACTORINGS IN SDM

In this section we describe a small experiment with Fujaba’s graph rewriting language. For the sake of understandability, we concentrate on the relatively simple *Pull Up Method* refactoring.

#### 3.1 Story Driven modelling

Story Driven Modelling (SDM) is a visual language for behavioral modeling based on UML activity diagrams, UML collaboration diagrams and graph theory [11]. Based on our small refactoring experiment, we identify some shortcomings of SDM for expressing our sample refactorings and suggest how this problem could be overcome.

#### 3.2 Expressing Pull Up Method

From section 2 we recall that *Pull Up Method* has three important preconditions: (i) no references (accesses or updates) should be made to an attribute that is defined in the subclass, (ii) no methods that are defined in the subclass should be called and (iii) no method with the same signature should already exist in the superclass. Because of the lack of update and access information in the metamodel, we only implemented the third precondition. The story diagram expressing this precondition is shown in Figure 4. It has one parameter node: *method*, which is the method to be pulled up. The diagram consists of four patterns: In the first story, the unbound *superclass:UMLMethod* node, representing any class in the program is bound to the superclass of the class containing *method*. The three other stories represent a loop over each *methodFromSC* in the superclass (story 2) and a comparison between the signature of *methodFromSC* and the signature of *method* (story 3 and 4). The signatures are compared by comparing the names and type of each parameter in the signature.

To be able to bind *paramFromMethod* to the correct parameter of *method* in story 4 (i.e. the parameter with the same index as *paramFromMethodFromSC*), we use SDM’s *qualified associations*: the *param* link between *method* and *paramFromMethod* in story 4 is qualified with the index of the parameter. In our

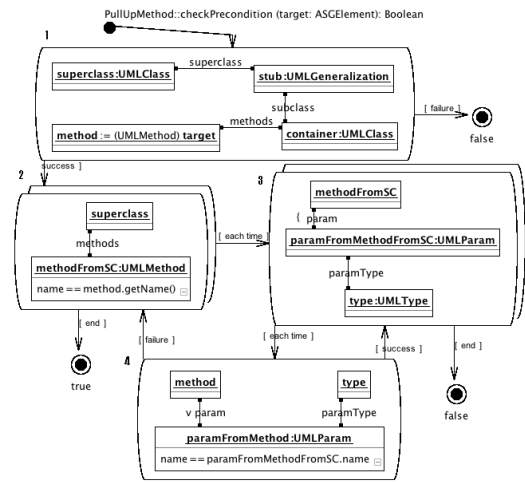


Figure 4: Story pattern for precondition (iii) of *Pull Up Method*

example, the value of this index is determined by evaluating the expression `paramFromMethodFromSC.getIndex()`. Although this constraint can be defined in the Fujaba environment, it is currently not possible to visualize it on a story diagram.

The control flow of the `checkPrecondition` procedure is straightforward: if the evaluation of the comparison is successful, the next parameter is tried. The loop continues until all parameters are compared. If all parameters have identical names and types, the superclass contains a method with the same signature and the evaluation of the precondition returns false. If, however, the evaluation of story 4 fails, the method signatures are different, and the next method of the superclass is tried. This explains the *failure* transition between story 4 and story 2. We experienced that SDM currently does not support this ‘nested loop break’ construction, so for our experiment, we had to work-around this by implementing the comparison in pure java code.

If the precondition returns true, the execution step of the refactoring now moves the method from its containing class to the superclass. In a story diagram this is expressed by breaking the *methods* association link between the method and its containing class, and creating a new one with the superclass. SDM allows these graph rewriting operations by adding *create* and *destroy* modifiers to the associations between the nodes. The diagram for the refactoring is shown in Figure 5. To illustrate how Fujaba generates code out of the story diagrams, the java code of the execute story is given below.

```

1 public void execute(ASGElement target)
2 {
3     boolean fujaba__Success = false ;
4     Iterator fujaba__IterContainerRevSubclassStub=null ;
5     UMLClass container, superclass = null ;
6     UMLGeneralization stub = null ;
7     UMLMethod method = null ;
8     try
9     {
10        fujaba__Success = false ;
11        // check object is really bound
12        JavaSDM.ensure ( target != null ) ;
13        // ensure correct type
14        JavaSDM.ensure ( target instanceof UMLMethod ) ;
15

```

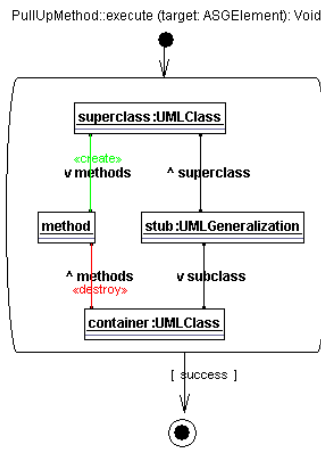


Figure 5: The execution story diagram of Pull Up Method

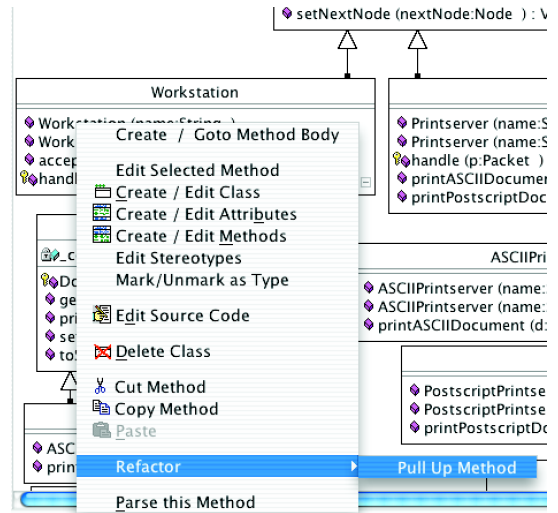


Figure 6: A refactoring plugin for Fujaba

```

16 // explicite type cast
17 method = (UMLMethod) target ;
18 // bind container : UMLClass
19 container = method.getParent () ;
20 JavaSDM.ensure ( container != null ) ;
21
22 // bind stub : UMLGeneralization
23 fujaba__IterContainerRevSubclassStub
24 = container.iteratorOfRevSubclass () ;
25 while ( !(fujaba__Success) &&
26 fujaba__IterContainerRevSubclassStub.hasNext () )
27 {
28     try
29     {
30         stub = (UMLGeneralization)
31         fujaba__IterContainerRevSubclassStub.next () ;
32
33         // bind superclass : UMLClass
34         superclass = stub.getSuperclass () ;
35         JavaSDM.ensure ( superclass != null ) ;
36         // check isomorphic binding
37         JavaSDM.ensure
38         (!(container.equals (superclass))) ;
39         // delete link
40         container.removeFromMethods (method) ;
41         // create link
42         superclass.addToMethods (method) ;
43         fujaba__Success = true ;
44     }
45     catch (JavaSDMException fujaba__InternalException)
46     {
47     }
48 }
49 }
50 catch (JavaSDMException fujaba__InternalException)
51 {
52     fujaba__Success = false ;
53 }
54 }

```

To complete the experiment, the java code from the diagrams was reused in a small gui plugin for Fujaba, which allowed us to apply the refactoring on a class diagram (see Figure 6).

### 3.3 Extending SDM for refactoring

Even with such a small experiment we found that it was not possible to express every feature of a refactoring in SDM. Because SDM supports *statement activities*, every problem expressing a constraint graphically can always be solved by implementing this constraint in pure java code. However, to be able to express refactorings in an

efficient and elegant way, one needs to add some new features to the SDM language. In the next paragraph we will suggest an extension, which will be needed for expressing the sample refactorings in this paper, but might be useful for other applications too. Of course, more extensions will be needed in the future, as the implementation of other refactorings will raise the opportunity for adding new constructs to the language.

The extension we propose here is the possibility of using *Parameterized graph expressions* [4] in SDM. This is extremely useful for expressing pre- and postconditions of refactorings. For example: precondition (iii) of *Extract Method* requires a check that a local variable is updated only once inside the extracted block. This means that there will be at most one path from the ActionSequence node that represents the method body, to an UpdateAction node with the LocalVariable node as its TargetRefinement. To express this constraint using SDM, one would need an infinite number of stories. If we allow regular expressions on the links in a story diagram the expression



Figure 7: A regular expression pattern in SDM

would be sufficient to express all the possibilities. *MB* and *V* are nodes of type ActionSequence and LocalVariable, and *UA* is an UpdateAction node. When this graph expression is evaluated, *MB* and *V* are bound to their respective parameters. Then a path between the two end nodes, that satisfies the regular expressions and unbound nodes (like *UA*) in the pattern, is determined. If no such path is found, the evaluation returns false. SDM already supports the definition of an arbitrary path between two nodes in a story. So this concept might serve as a basis for implementing the regular expression extension.

## 4. ARCHITECTURAL REQUIREMENTS

To ensure that the parsed source code will be regenerated appropriately, two new components are required in the Fujaba architecture.

We call these new components the “Code Preserver” and the “Refinement Repository”. These components complement the lexer, the parser, the metamodel, and the code generator.

## 4.1 Code Preserver

### 4.1.1 Definition

A *Code Preserver* is a development tool component that stores all the required source code files from which a model is extracted in such a way that the complete system can be regenerated from a transformation of the input model. A code preserver does not require a metamodel of all system properties and can preserve the original code layout.

### 4.1.2 Motivation

First of all, for the sake of simplicity, we want to minimize the amount of extensions to Fujaba’s metamodel as much as possible (without sacrificing source consistency). However, if we want to regenerate arbitrary method bodies with a conventional code generator, we would need a metamodel that contains all syntactically possible source code constructs (cascaded method calls, local variable declarations, type casts, type checks, ...). Otherwise, some (fragments of) source code statements would get lost.

In addition to this problem, code generators assume a fixed code layout for all instances of a particular metamodel element. This is undesirable in the context of refactoring, where developers don’t want to lose their layout each time they execute a refactoring.

### 4.1.3 Other Applications

To manage the rapid evolution of (and the number of alternatives between) today’s middleware component models, we want to minimize the work to add support for new XML deployment descriptors to our refactoring framework. When developing an application to evaluate the performance of a new component model (for which there are no code generators available yet), one may want to execute refactorings to evolve from a small running example to a more realistic prototype. With a code preserver, it would suffice to extend the Fujaba parser to integrate the new source code syntax in our refactoring tool. Without a code preserver, we would also need to write a new code generation template.

With a code preserver, it would suffice to extend the Fujaba parser to handle the new source code syntax. Without a code preserver, we would also need to write a new code generation template.

Our xPetstore sample is developed with the open source xDoclet code generator [12] and the Poseidon UML tool [13]. xDoclet generates skeleton classes and the deployment descriptors for the EJB component model from JavaDoc-annotated domain model sources. Poseidon visualizes the domain model classes as UML class diagrams. Poseidon’s model is stored in an XMI file. Instead of parsing and updating the XML deployment descriptors themselves, we need to update the annotated java sources that define the input model for xDoclet. We can obtain the new deployment descriptors by deleting them and regenerating them with xDoclet. Additionally, to maintain consistency with Poseidon, we need to update its XMI file. This illustrates how a code preserver can facilitate the integration of a set of special-purpose UML tools: Fujaba takes care of generating model transformation code from graph rewrite rules whereas xDoclet takes care of generating middleware framework code from stereotyped class diagrams that are visualized by Poseidon.

The fragment below illustrates the structure of *xpetstore.xmi*:

```

1 <UML:Class xmi.id='a1936' name='CartEJB'...
2   isRoot='false' isLeaf='false' isAbstract='true'...>
3   ...
4   <UML:Classifier.feature>
5     ...
6     <UML:Operation xmi.id='a2006' name='getCartItems'
7       isSpecification='false' ownerScope='instance'...
8       isLeaf='false' isAbstract='false'>
9       <UML:ModelElement.taggedValue>
10         ...
11         <UML:TaggedValue xmi.id='a2008'...
12           dataValue='@return Return a list of
13             {@link CartItem} objects&#10;&#10;
14             @ejb.interface-method&#10;
15             @ejb.transaction-type&#10;
16             type='quot;NotSupported&quot;'>
17             <UML:TaggedValue.type>
18               <UML:TagDefinition xmi.idref='a91' />
19             </UML:TaggedValue.type>
20           </UML:TaggedValue>
21         </UML:ModelElement.taggedValue>
22       <UML:BehavioralFeature.parameter>
23         <UML:Parameter xmi.id='a2009' ...
24           kind='return'>
25           ...
26         </UML:Parameter>
27       </UML:BehavioralFeature.parameter>
28     </UML:Operation>
29   ...
30 </UML:Classifier.feature>
31 ...
32 </UML:Class>

```

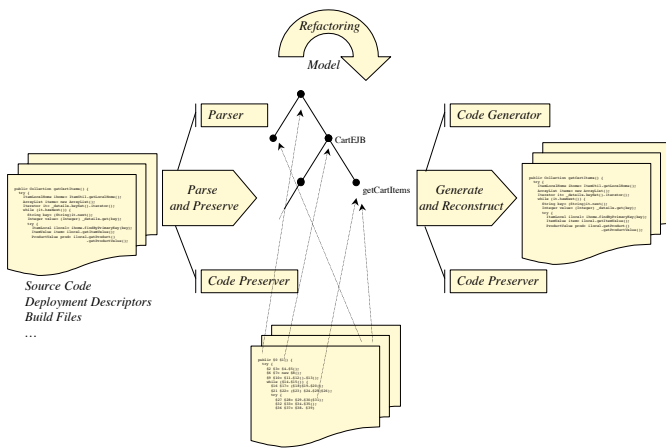
For the sake of readability, some fragments are suppressed as three dots. A code preserver could help us to keep track of the model references (e.g. *getCartItems* in the fragment above) and preserve all other information (like *@ejb.interface-method* and *@ejb.transaction-type*) without including the information into a dedicated metamodel. Of course, we would still need to write (or reuse) an XMI parser but this would also be the case if we would use a conventional code generator.

### 4.1.4 Overall Architecture

Figure 8 visualizes how the code preserver builds dynamic code templates with lexer input while the sources are parsed into a model. After refactoring, these templates are used to rebuild the files for the updated model. The design and implementation of the code preserver are beyond the scope of this paper.

### 4.1.5 Closing Remark

We should note that the abstraction level of a refactoring metamodel depends on the set of supported refactorings. Moreover, there is no ultimate refactoring metamodel as new refactorings are continuously being discovered. The code preserver can help to defer the inclusion of a metamodel entity until its semantics is an explicit part of the problem domain (i.e. the supported refactorings in our case). For example, currently we have not included type checks and type casts into our metamodel. Thanks to the code preserver, we do support the update of the referred class name when the rename class refactoring is executed. If we would ever need to implement a refactoring whose refactoring contract explicitly makes use of type casts we would include a dedicated *type cast action* in our metamodel.



**Figure 8: The role of the code preserver within the overall architecture.**

## 4.2 Refinement Repository

### 4.2.1 Definition

A *refinement repository* is a development tool component that exposes the model to model refinement transformations of an MDA code generator.

### 4.2.2 Motivation

In this paper, we explore to what extent refactorings can be expressed on platform independent metamodels without sacrificing consistency with the underlying sources and configuration files. In the case of generated software, it is important to know the dependencies between domain entities and their derived component model specific classes.

First of all, refactorings on a domain entity should trigger a regeneration of all derived sources and configuration files. This regeneration can be implemented with existing black box code generators such as xDoclet.

Secondly, all manually written code that makes use of the generated classes needs to be updated as well [14]. Suppose, for example, we rename *Cart* to *ShoppingCart* in the problem domain of our xPetstore sample. Under the covers, this high level refactoring would be decomposed into the primitive “rename class” refactorings for *CartEJB*, *CartLocalHome*, *CartValue*, ... To execute such high level refactorings, a refactoring tool would need to query the code generator’s “refinement repository” in order to learn about the name dependency from domain entities that are stereotyped as *EJB* to the name of model elements representing their bean class, local home class, value object class, ...

## 5. FUTURE WORK

First of all, this experiment calls for more validation. Among other things, we need to implement a GrammyUML parser for Java, a code preserver and a refinement repository. As a first step, we are extending the open source AndroMDA code generator with Fujaba’s SDM. We have selected AndroMDA because of its standard JMI repository and its support for various middleware component models [15]. In this project, we will implement our suggested SDM extensions, along with lessons learned from a review of the MOF QVT submissions [16]. Another interesting project would be to

extend Fujaba’s lazy parser with the proposed metamodel extensions. We also have to investigate whether and how the UML 2.0 diagram exchange format can be parsed to GrammyUML because the current XMI standard does not include such information.

In our future work on Model Driven Refactoring with GrammyUML we may cover additional refactorings, additional formalisms and additional languages.

We are working on both additional primitive OO refactorings and high level composed refactorings supporting design and architecture evolution.

We are also evaluating how the emerging XQuery and XUpdate standards can be used to implement refactorings on XML representing GrammyUML models. Our goal is to compare our graphical (SDM), in-memory implementation (in Fujaba) with a textual (XML), database implementation mainly in terms of expressiveness and scalability.

An interesting validation case for the new code preserver architecture is implementing refactorings for C++ programs. Our approach would preserve not only the C++ programmer’s code conventions concerning white-spaces and newlines, but would also preserve hand-written forward declarations across `cpp` and header files (which are often designed as API documentation).

## 6. ACKNOWLEDGEMENTS

We would like to thank Matthias Bohlen, the lead engineer behind AndroMDA, for his valuable feedback on the draft of this paper.

## 7. REFERENCES

- [1] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [2] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. *Language Descriptions, Tools and Applications (LDTA)*, 2002.
- [3] University of Paderborn Software Engineering Group. Fujaba. <http://www.uni-paderborn.de/cs/fujaba/>, August 2003.
- [4] Tom Mens, Niels Van Eetvelde, Dirk Janssens, and Serge Demeyer. Formalising refactorings with graph transformations. *Fundamenta Informaticae*, 2003.
- [5] Don Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
- [6] Sander Tichelaar Serge Demeyer and Patrick Steyaert. Famix 2.0 – the famoos information exchange model. <http://www.iam.unibe.ch/famoos/FAMIX/>, 09 1999.
- [7] Herve Tchepannou. xPetstore. <http://xpetstore.sourceforge.net/java2html/xpetstore-ejb/xpetstore/services/cart/ejb/CartEJB.java.html>, August 2003.
- [8] Java Community Process. Enterprise JavaBeans specification, August 2003.
- [9] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*, chapter 5, pages 103–107. Morgan Kaufmann, 2002.
- [10] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In *Proceedings of the 6th International Conference on « UML » – The Unified Modeling Language.*, 2003.
- [11] 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 6<sup>th</sup> International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.
- [12] xDoclet Project Team. xDoclet. <http://xdoclet.sourceforge.net/>, September 2003.
- [13] Gentleware. Poseidon for UML. <http://www.gentleware.com/>, September 2003.
- [14] Matthias Bohlen. AndroMDA 3.0 vision document: Moving to the agile world. <http://www.andromda.org/developerdocs/>, July 2003.
- [15] AndroMDA Project Team. AndroMDA. <http://andromda.sourceforge.net/>, September 2003.
- [16] 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, July 2003.