TOWARDS A LANGUAGE INDEPENDENT REFACTORING FRAMEWORK

Carlos López, Raúl Marticorena
Área de Lenguajes y Sistemas Informáticos Universidad de Burgos.09006 Burgos, Spain

Yania Crespo, Francisco Javier Pérez
Departamento de Informática Universidad de Valladolid. 47001 Valladolid, Spain

Keywords: Refactoring, metamodel, language independence, object oriented programming, UML.

Abstract: Using metamodels to keep source code information is one of the current trends in refactoring tools. This representation makes possible to detect refactoring opportunities, and to execute refactorings on metamodel instances. This paper describes an approach to language independent reuse in metamodel based refactoring detection and execution. We use an experimental metamodel, MOON, and analyze the problems of migrating from MOON to UML 2.0 metamodel or adapting from UML 2.0 to MOON. Some code refactorings can be detected and applied on basic UML abstractions. Nevertheless, other refactorings need information related to program instructions. "Action" concept, included in UML 2.0, is a fundamental unit of behaviour specification that allows to store program instructions and to obtain certain information related to this granularity level. Therefore, we compare the complexity of UML 2.0 metamodel with MOON metamodel as a solution for developing refactoring frameworks.

1 INTRODUCTION

Language independent refactoring is one of the current trends in refactoring research (Mens and Tourwé, 2004). Defining metrics and refactorings in a language independent way offers a solution to the reuse in development of refactoring tools when they are adapted to new source languages. It is also a rational support in multilanguage integrated development environments.

There are different trends to address these problems: on the one hand, using abstract syntax trees and on the other hand, using metamodels (Demeyer et al., 1999). In this work, we display a proposal using metamodels, showing a previous support and studying the suitability of the UML 2.0 metamodel (OMG, 2004), with the new "action" concept, as new support to refactoring tools.

This paper is structured as follows: Section 2 establishes the current works with metamodels; Section 3 presents the current state of our work, describing the modules of a refactoring framework, language extensions and dependencies; Section 4 shows the standardization problem of the MOON metamodel, and presents a reengineering example, with UML 2.0 as the new candidate to replace it. Finally, in Section 5, we conclude with the pros and cons of using UML 2.0 as the core model of the refactoring framework.

2 RELATED WORKS

Some approaches to the problem of language independence are based on metamodel solutions. In (Tichelaar et al., 2000), FAMIX is defined as a metamodel for storing information, aimed at the integration of several CASE tools. One of these CASE tools is a refactoring assistant tool named MOOSE (Ducasse et al., 2000).

The FAMIX core model specifies the entities and relations that can (and should) be extracted immediately from the source code. The core model consists of the main OO entities: `Class`, `Method`, `Attribute`, `InheritanceDefinition`, etc.

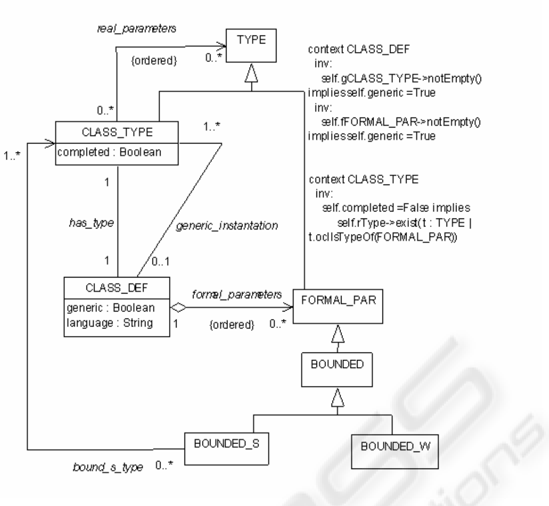
For reengineering purposes, it needs two entities: `Invocation` and `Access` associations. An `Invocation` represents the definition of a `Method` calling another `Method`. An `Access` represents a method body accessing an `Attribute`. These abstractions are needed for reengineering tasks, such as dependency analysis, metrics computation and reengineering operations. FAMIX metamodel does not contain advanced inheritance and genericity features.

In the same way, a new solution based on metamodels is proposed in (Van Gorp et al., 2003). They propose eight additive and language-independent extensions to the UML 1.4 metamodel, which form the foundation of a new metamodel named GrammyUML. This study does not consider the *Action Semantic package* in UML.

Our proposal is based on MOON, Minimal Object-Oriented Notation (Crespo, 2000). MOON represents the necessary abstract constructions in refactorings definition and analysis, just like FAMIX. MOON abstractions are common to a family of programming languages: object-oriented programming languages (OOPL), statically typed with or without genericity. This is the basis for a metamodel-centered solution, with the objective of reusing it in the development and adaptation of refactoring tools.

The MOON metamodel stores: classes, relationships, variants on the type system, a set of correctness rules to govern inheritance, etc. The main difference with FAMIX relies on the type system. The core of MOON metamodel includes classes as `Entity` representing any concept in source code that has a `Type` (self reference, super reference, local variable, method formal argument, class attribute and function result). It also collects the method body description: local variables, formal arguments and instructions. The instructions are classified in the following way: creation, assignment, call and compound instructions.

For example, Figure 1 outlines the MOON metamodel classes related to genericity and their semantic rules expressed in OCL. Types are classified into formal parameters (*FORMAL_PAR*) and types derived from class definitions (*CLASS_TYPE*). Non-generic class definition leads to a 1-1 association between class (*CLASS_DEF*) and type (*CLASS_TYPE*). When class definition is generic, it is said that is the "determining class" of a potentially infinite set of types. Each generic instantiation corresponds to a different type (*CLASS_TYPE*). A generic class definition contains a list of formal parameters.



```
context CLASS_DEF
  inv:
    self.gCLASS_TYPE->notEmpty()
impliesself.generic =True
  inv:
    self.fFORMAL_PAR->notEmpty()
impliesself.generic =True

context CLASS_TYPE
  inv:
    self.completed =False implies
        self.rType->exist(t : TYPE |
t.oclIsTypeOf(FORMAL_PAR))
```

Figure 1: MOON Parametric Types.

All these previous works propose metamodels to support a complete refactoring process. In the next section, the current state of the framework is proposed over MOON, in order to evaluate the suitability of the UML 2.0 metamodel in Section 4.

3 REFACTORING FRAMEWORK

This section establishes the current state of the proposed framework, focusing on the core (in Section 3.1) and the concrete language extensions (in Section 3.2), checked in previous works. The main purpose is to give the "big picture" of the current proposal, explaining the role of the MOON metamodel.

3.1 Framework Core

In previous works, we have designed a framework to detect, to define, and to execute refactorings. Figure 2 shows the main modules that compose it.

Here we present a brief description for each one of them:

Module A represents the metamodel. The metamodel is the basis of the solution for language independence. It must collect the basic elements of any object-oriented language: classes, attributes, methods, client-provider relations, inheritance and genericity. In particular, it is necessary to include information about instructions, assignment instructions and expressions.

Module B defines the refactoring engine. It is composed of a core and a repository. The engine core contains the necessary abstract classes to define the refactorings by composing their inputs, pre-, postconditions and actions. The core establishes how to execute any concrete refactoring, once its components are known. The refactoring repository contains concrete predicates, functions and actions as well as the concrete refactorings as their composition. All the repository elements are defined on the metamodel elements and provide the reuse functionality when a concrete language extension is developed.

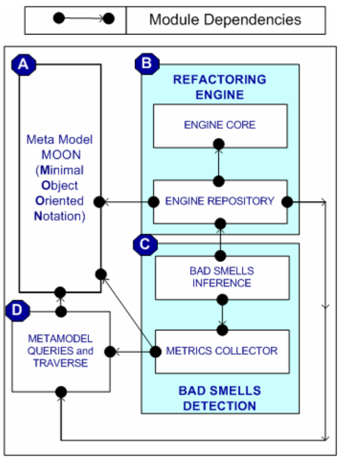


Figure 2: Language Independent Refactoring Framework.

Module C is composed of two submodules: Metric Collector and Bad Smells Inference. It is responsible for detecting bad smells using metrics. To remove the detected bad smells, it is possible to suggest a refactoring set defined in the refactoring repository. In (Fowler, 2000), relations between bad smells and refactoring have been proposed. The idea of collecting metrics from a language independent metamodel presents clear advantages from the reuse point of view.

Module D isolates the queries and traversals on metamodel elements. These queries and traversals are necessary in the refactoring repository and in the metric collector. This module is reused, avoiding duplicated code in modules B and C. Furthermore, Visitor and Strategy (Gamma et al., 1995) design patterns are applied to add new operations without changing the metamodel classes on which they operate.

3.2 Framework Extension

The framework introduced here has been validated with the implementation of two concrete languages extensions: Java and Eiffel, because to reuse the framework core, a concrete language extension is required. In Figure 3, we show some examples of Java extensions plugged in the framework core.

Module extension A is responsible for picking up source code information and transforming it into language extension instances. There are particular language features, for example in Java, native and transient modifiers that are represented in java extension on the metamodel core. Their classes implement abstract methods defined on the core module.

Module extension B defines the specific features of each refactoring, according to the particular selected language.

Module extension D uses language extensions to regenerate the code. Each concrete language extension in module extension A has the semantic to walk through elements using visitor classes.

4 EXAMPLE: UML 2.0 AS METAMODEL

MOON can evolve to be fitted for a standard metamodel. UML is currently embraced as the standard in object oriented modelling languages.

When talking about requirements for using a

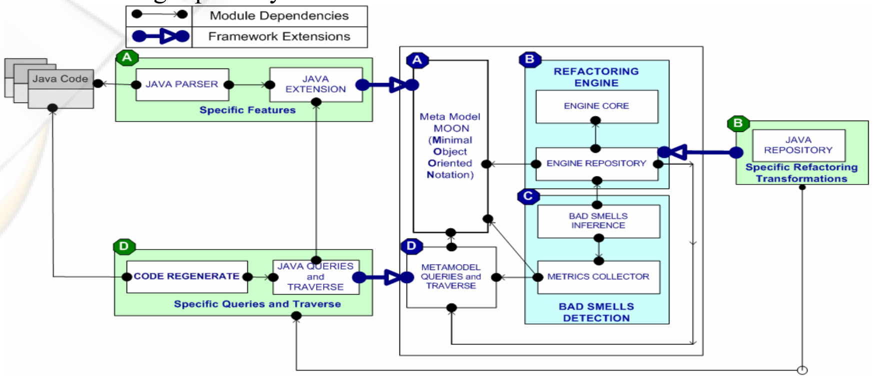


Figure 3: Refactoring Framework with Java Extensions.

metamodel in the refactoring context, a question is missing: can the method body (instruction, local variable, etc…) be stored with UML? The action concept, defined in UML 2.0, is the fundamental unit of behavior specification. An action takes a set of inputs and converts them into a set of outputs. Actions could store method body information, hence UML 2.0. metamodel can become a candidate to module A (see Figure 2). Furthermore, UML 2.0 includes template mechanisms, which provide support for generic types available in programming languages. Module A extensions could be solved with UML profiles (OMG, 2004). Profiles are mechanisms that allow metaclasses from existing metamodels to be extended to adapt them for different purposes. Profiles include the ability of tailoring the UML metamodel for different language features, such as Java, C#, C++, Eiffel, etc. The Profile mechanism is consistent with the OMG Meta Object Facility (MOF).

In the following subsections we present an example. It guides the mapping from program language constructions to UML abstractions, including advanced features: actions and templates.

4.1 Statement

It is very difficult to select an example that contains all language features. Our example considers the parameterized factory methods when the Abstract Factory design pattern (Gamma et al., 1995) is applied. A generic factory (*GenericFactory*) with subtyped bound is defined, avoiding to use an inheritance hierarchy with Factory classes to create suitable product objects.

In Figure 4, a structural solution using design patterns is shown, where the generic class with subtyped bound (*GenericFactory*) and generic instantiations (*FactoryConcreteProd1*, *FactoryConcreteProd2*) are highlighted. Both of them are used by client class (*FactoryTest*) to create new instances of concrete products (*ConcreteProduct1*, *ConcreteProduct2*).

Code 1 defines (in Java 1.5) the *GenericFactory* class using reflective programming by means of `java.lang.Class<T>` class, while Code 2 collects the piece of code associated to *FactoryTest* in Figure 4. In the shown codes, reserved words have been emphasized.

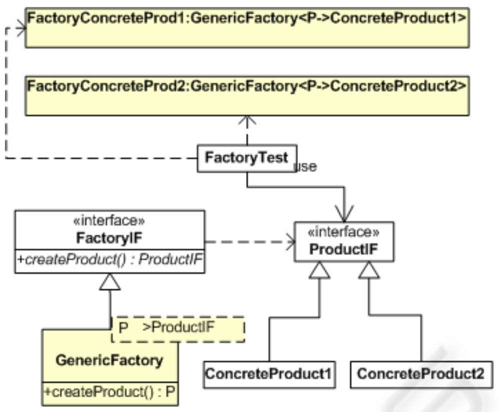

FactoryConcreteProd1:GenericFactory<P->ConcreteProduct1>
FactoryConcreteProd2:GenericFactory<P->ConcreteProduct2>
FactoryTest
use
«interface»
FactoryIF
+createProduct() : ProductIF
«interface»
ProductIF
P >ProductIF
GenericFactory
+createProduct() : P
ConcreteProduct1
ConcreteProduct2

Figure 4: Generic Factory Method.

```
public class GenericFactory
 <P extends ProductIF>
 implements FactoryIF{

 private Class<P> c;

 public GenericFactory
  (Class<P> c) {
  this.c = c;
 }

 public P createProduct(){
  P product = null;
  try{product=c.newInstance();}
  catch(InstantiationException
        e){  }
  catch(IllegalAccessException
        e){  }
  return product;
 }
}
```

Code 1: Generic Factory Definition Java 1.5

```
FactoryIF      factory1      =      new
GenericFactory<ConcreteProduct1>(Concre
teProduct1.class);
GenericFactory     factory2     =     new
GenericFactory<ConcreteProduct2>(Concre
teProduct2.class);
```

Code 2: Generic Factory Instantiations.

4.2 UML Mapping

We outline the relevant problems of mapping a factory method `createProduct` body to UML abstractions: exception handlers, instruction sequences, call instructions and parametric types. In these mappings the related sections of the "UML 2.0 Superstructure" (OMG, 2004) are indicated.

To model an operation we can associate an activity diagram (Booch et al., 1999). An action flow of the operation is represented, so that all diagram elements are semantically linked to an underlying model with expressive richness.

UML 2.0 introduces new functionality, allowing to catch exceptions and manage them inside an activity diagram. In the behaviour specification, and more precisely in the activity section, we find the *ExceptionHandler* class. This element specifies the body (action sequence) to be executed in case that exceptions happen during the running of a protected node. A protected node groups an activity set that could throw one or more exceptions. Graphic notation of both of them can be observed in Figure 5. To map these concepts the following UML superstructure sections are used:

- ExtraStructuredActivities (in Activities) mapping Exception Handler.
- Activities mapping Instruction sequence.

In the `GenericFactory` class context, Figure 5 shows an activity diagram that defines the algorithm of the `createProduct` operation. This operation has a signature with a return value of *`P`* parameter type. `GenericFactory` class has an attribute `c:java.lang.Class<P>` that allows us to accomplish the creation of new concrete products.

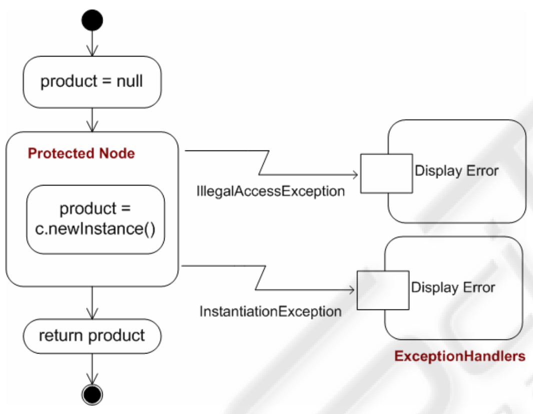


Figure 5: Method body createProduct.

Actions can be contained in activities which provide their contexts. Activities specify control and data sequence restrictions over the actions, as well as nesting mechanisms in control structures.

Each activity is defined by a set of actions which provide precise semantics. The mapping of the instructions `c.newInstance()` with the call action (*`CallOperationAction`*) is represented in the object diagram of Figure 6. This action obtains its input (*`InputPin`*) from the ouput (*`OutputPin`*) of the reading actions (*`ActionInputPin`* and *`ReadStructuralFeatureAction`*). In that way, it has the object reference contained in the attribute `c` (*`StructuralFeature`*) in `GenericFactory` class.

The UML superstructure sections used to map the call instruction `c.newinstance()` have been actions and classes.

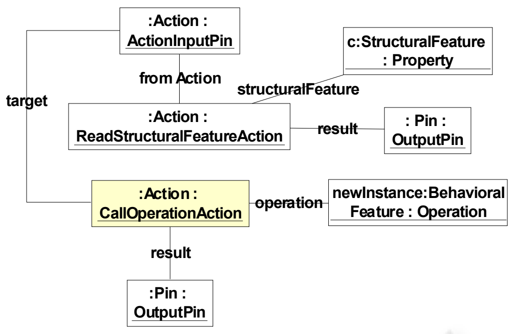

:Action : ActionInputPin
c:StructuralFeature : Property
from Action
structuralFeature
target
:Action : ReadStructuralFeatureAction
result
: Pin : OutputPin
:Action : CallOperationAction
operation
newInstance:Behavioral Feature : Operation
result
:Pin : OutputPin

Figure 6: Call instruction c.newinstance.

In the UML 2.0 superstructure we can find a section focused on auxiliar constructors defining mechanism sets. One of these mechanisms is the use of templates as support to parameterize classifiers (*`Classifiers`*), packages (*`Packages`*) and operations (*`Operations`*). We can also find mechanisms to define templates, formal parameters (*`TemplateParameters`*) and to tackle the generic instance process. This subsystem has 20 classes.

Before introducing the example instantiation, we give a brief description of metaclasses and relations among the participants:

`TemplateableElement` is the abstraction that supports the template definition. It can contain a *`TemplateSignature`* that specifies a formal parameter sequence (*`TemplateParameter`*). *`TemplateableElement`* can also contain links to other Classifiers:*`TemplateableElement`*, related to the generic instantiations, by the substitution of the formal parameters (*`TemplateParameters`*) with the current parameters (*`ParameterableElement`*).

`TemplateParameter` refers to a *`ParameterableElement`* that it is exposed as a formal parameter in the template.

`TemplateParameterSubstitution` associates the current parameters with the formal parameters as part of the *`TemplateBinding`* relation.

`RedefinableTemplateSignature` specializes *`TemplateSignatures`* and *`RedefinableElement`* to allow adding new formal parameters in the definition context of templates over classifiers.

A classifier (*`Classifier`*) is a specialization of *`TemplateableElement`* and of *`ParameterableElement`*.

In Figure 7 we show the mapping of the generic factory definition with its bounded formal parameter by the subtype of *`ProductIF`*.

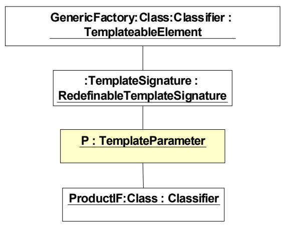

GenericFactory:Class:Classifier : TemplateableElement
:TemplateSignature : RedefinableTemplateSignature
P : TemplateParameter
ProductIF:Class : Classifier

Figure 7: Generic definition.

Figure 8 shows the generic instantiation of a concrete product factory of `ConcreteProduct1`. Those objects that are useful as nexus between both diagrams have been highlighted.

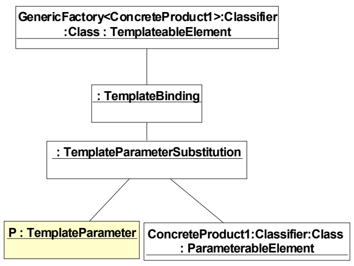

GenericFactory<ConcreteProduct1>:Classifier :Class : TemplateableElement
: TemplateBinding
: TemplateParameterSubstitution
P : TemplateParameter
ConcreteProduct1:Classifier:Class : ParameterableElement

Figure 8: Generic instantiation.

5 CONCLUSIONS

UML can be used to store source code, even though there is a high complexity in the metamodel structure. Table 1 shows a comparison of the two metamodels, MOON and UML, divided in subsystems/sections and number of classes. The number of related classes in the UML metamodel is three times higher than in the MOON metamodel.

Table1: Comparative MOON vs. UML metamodel.

MOON		UML	
Subsystem	Number of classes	Sections	Number of classes
Module	24	classes	55
Inheritance	7		
Genericity	5	templates	20
Instructions	20	actions	54
		activities	52
Total	56	Total	181

The paper focuses on UML abstractions which are needed to represent code information, classes and activity diagrams. Although the displayed example in Section 4 achieves a mapping to UML abstractions (generic classes, exceptions, etc.), the experiment is limited because it does not include all abstractions in object oriented languages. In this sense, we have identified some abstractions that are not represented in the UML metamodel, as typecast and multiple bounds of parametric types. Both features are supported on the MOON metamodel, but MOON does not support concepts such as exceptions, conditionals, loops, etc. Besides, MOON supports three type variants in genericity giving a suitable support to this feature in statically typed object oriented languages.

Due to the previous advantages and the minimal core size in the MOON metamodel, we think that an UML approach is only appropriate from the point of view of a standardization effort, and reuse in other abstraction levels, such as the design level. Therefore, we propose a new design solution with the UML metamodel extension as a future direction, extending the current MOON metamodel in the same way as we have done with programming languages. This should provide full support to a refactoring process reusing the previously designed framework.

REFERENCES

Booch, G., Rumbaugh, J. y Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison Wesley.

Crespo, Y. (2000). *Incremento del potencial de reutilización del software mediante refactorizaciones.* PhD thesis, Universidad de Valladolid. Available at http://giro.infor.uva.es/Publications/2000/Cre00/.

Demeyer, S., Tichelaar, S., and Steyaert, P. (1999). *FAMIX 2.0 - the FAMOOS in-formation exchange model.* Technical report, Institute of Computer Science and Applied Mathematic. University of Bern.

Ducasse, S., Lanza, M., and Tichelaar, S. (2000). *MOOSE: an extensible language-independent environment for reengineering object-oriented systems.* In Proceedings on constructing Software Engineering Tools (CoSET 2000).

Fowler, M. (2000). *Refactoring. Improving the Design of Existing Code*. Addison Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J.(1995). *Design Patterns. Elements of Reusable Object Oriented Software.* Addison Wesley.

Mens, T. and Tourwé, T. (2004). *A survey of software refactoring.* IEEE Trans. Softw. Eng., 30(2):126–139.

OMG 2004. *Unified Modeling Language: Superstructure version 2.0.* http://www.uml.org.

Tichelaar, S., Ducasse, S., Demeyer, S., and Nierstrasz, O. (2000). A meta-model for language-independent refactoring. In Proceedings ISPSE 2000, pages 157–167. IEEE.

Van Gorp, P., Stenten H., Mens, T., and Demeyer, S. (2003) *Towards automating source-consistent UML Refactorings.* In Proceedings of UML 2003 - The Unified Modeling Language. Springer-Verlag, 2003