DYNAMISM IN REFACTORING CONSTRUCTION AND EVOLUTION
A Solution based on XML and Reflection

Raúl Marticorena
University of Burgos, EPS C/Francisco Vitoria, Burgos, Spain

Yania Crespo
Department of Computing Science, University of Valladolid, Campus Miguel Delibes, Valladolid, Spain

Keywords: Refactoring, evolution, frameworks, refactoring tools, XML, reflection, refactoring dynamic building.

Abstract: Current available refactoring tools, even stand-alone or integrated in development environments, offer a static set of refactoring operations. Users (developers) can run these refactorings on their source codes, but they can not adjust, enhance, evolve them or even increase the refactoring set in a smooth way. Refactoring operations are hand coded using some support libraries. The problem of maintaining or enriching the refactoring tools and their libraries are the same of any kind of software, introducing complexity dealing with refactoring, managing and transforming software elements, etc. On the other hand, available refactoring tools are mainly language dependent, thus the effort to reusing refactoring implementations is enormous, when we change the source code programming language. This paper describes our work on aided refactoring construction and evolution based on declarative definition of refactoring operations. The solution is based on frameworks, XML and reflective programming. Certain language independence is also achieved, easing migration from one programming language to another, and bringing rational support for multilanguage development environments.

1 INTRODUCTION

Refactoring is becoming a natural step in current software development as *"a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"* (Fowler, 2000). Refactoring features are included as options in IDEs or assembled as plugins. Agile development processes include refactoring as a good practice, and nowadays, it is common to teach these concepts in software engineering courses.

Nevertheless, different implementations and solutions are provided for each environment, language and platform. Our work takes a different approach to refactoring development, focusing on a reuse based solution. Refactorings are defined as compound elements that can be assembled and reused to obtain new refactorings from already implemented elements. Some elements are common to different languages and others are particular of each concrete programming language.

In the remainder of this paper, in Section 2 and 3 we give a brief overview about the code managing support using a solution based on metamodels and framewoks, Section 4 describes the refactoring engine that allows refactorings to run and defines refactoring items: queries and transformations. Section 5 proposes a solution using XML and reflection allowing a declarative refactoring definition and dynamic building. Some related works are detailed in Section 6 and conclusions and future work are shown in Section 7.

2 LANGUAGE REPRESENTATION

Refactoring tools need to manage source code information. The main problem is to choose the most suitable representation. Currently, most solutions are based on abstract syntax trees (AST) and well known design patterns as *Visitor* (Gamma et al., 1995). The main problem is that an AST is bound to one language (more concretely to its grammar). This is a big obstacle that hinders reuse in refactoring implementations.

Other trends outline to manage source code in-

Marticorena R. and Crespo Y. (2008).
DYNAMISM IN REFACTORING CONSTRUCTION AND EVOLUTION - A Solution based on XML and Reflection.
In *Proceedings of the Third International Conference on Software and Data Technologies - SE/GSDCA/MUSE*, pages 214-219
DOI: 10.5220/0001890002140219

formation through relational databases but additional problems appear recovering original or modified code. Queries are defined with structured query languages (SQL). Although simple, this allows defect detection but does not support refactoring. Similar problems arise when logical predicates are used to manage the information.

Our proposal uses metamodels as language support. On one hand, metamodels provide with the information as well as previous proposals do. On the other hand, it is also possible to change the current model instance, and obtain the refactored code. Finally, reuse possibilities are enabled from the benefits of a framework based solution, managing commonalities and variations from different languages.

3 USING MOON AS METAMODEL

A minimal object-oriented notation (MOON) (Crespo, 2000) is used as starting point to develop language support and refactoring execution. Source information is stored as instances of graph nodes. The graph can be traversed to check current state (conditions) and run transformations. Once the graph is transformed, can be traversed and refactored code is regenerated. Refactoring process gains in language independence. MOON represents the necessary abstract constructions in refactorings definition and analysis. They are common to a family of programming languages: object-oriented programming languages (OOPL), statically typed with or without genericity. This model language deals with classes, relationships, type system variants, a set of correctness rules to govern inheritance, etc.

Although a model language can represent common points and general variants, it does not include all features of the programming language family. It is necessary to support common abstractions, variability and extension points for peculiarities and exceptions for particular features. With this aim, frameworks (Fayad et al., 1999) emerge as a suitable solution. Language particular features are extended and hooked in their concrete framework extensions. For example, a Java extension framework is developed to store concepts as exceptions, interfaces, try-catch blocks, etc, not included in MOON. In this sense, a taxonomy of concepts is established on the model elements:

1. General: are elements contained in most of statically typed languages. For example, *Class*, *Method* or *Attribute* are common concepts included in languages as Eiffel, C++, Java, C#, etc. Common features can be represented as MOON core features and reused on several languages.
2. Extensible: are present in most of the languages in the family but each one of them gives a different semantic value. For example, access modifiers, inheritance rules, etc. MOON concepts with variations are abstracted in the metamodel core and redefined for each language.
3. Particular: language concepts that are not included in MOON core. Language extension framework includes these concepts to support the whole set of language features.

4 RUNNING REFACTORINGS

An overview of refactoring definitions, formal as well as semi-formal definitions as in (Roberts, 1999) and (Opdyke, 1992), or textual definitions using “recipes” (Fowler, 2000), etc. leads to a clear separation of refactoring elements: queries for checking preconditions or postconditions and transformational actions.

A refactoring tool must include all these elements and put refactorings in motion on extracted code. Next we describe the basics of our refactoring engine and its concrete elements.

4.1 Refactoring Engine

A language independent framework has been defined and used to allow a simple reuse schema, as can be seen in Fig. 1. The refactoring engine runs refactoring definitions and obtains a new model state. The running algorithm is common to all refactorings, with hook methods to engage concrete elements. *Template Method* (Gamma et al., 1995) is taken as basis on the `Refactoring` class. `Run` method plays the role of template method running the interaction between elements.

With this schema, a refactoring can be seen as pieces implemented with classes from repositories: predicates, functions and actions. *Command* role (Gamma et al., 1995) is played by action classes. An action can be undone and logged when any exception is thrown in the refactoring process. Each refactoring is implemented as an extension of the `Refactoring` class. `Predicate`, `Function` and `Action` abstract classes must be extended by concrete classes in the repositories. Refactoring constructors take inputs and assemble preconditions, actions and postconditions in the correct order.

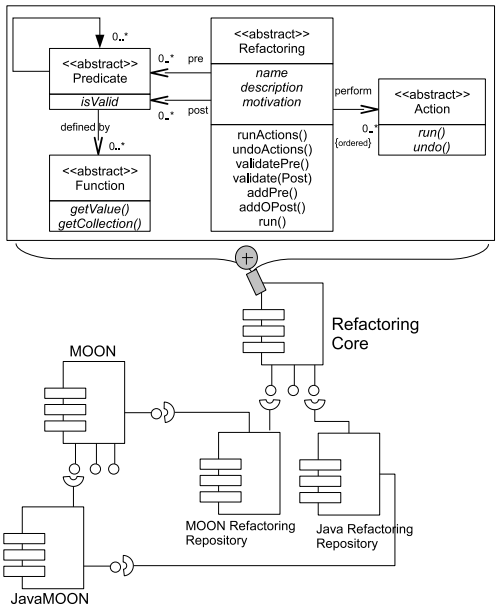

Figure 1: Refactoring engine framework.

4.2 Queries and Transformations

Refactoring elements, queries and transformations, are usually implemented as concrete classes collected in repositories. Although model extensions contain information of real code, most classes work with the MOON core metamodel abstractions. Queries are usually predicates based on functions and other predicates. Actions change the graph state, from previous state to the refactored state. Both of them are classified following the taxonomy described in Section 3.

Same queries or actions are reused when implementing refactoring operations for different languages if they can be seen as language independent. For example, the precondition `ExistParameterWithSameName` or the action `MoveAttributeAction`, are reusable for several languages.

The main advantage of this approach appears when same refactorings are implemented for different object-oriented programming languages. In an ideal case, a refactoring operation can be reused as a whole, but in most cases, the refactoring elements are reused and properly composed introducing variations and language dependent parts.

5 REFACTORING CONSTRUCTION AND EVOLUTION

Most available refactoring tools implement their refactoring operations as a fixed composition of parts which are hand coded in source files with a particular programming language. Although better design solutions have emerged last years (Frenzel, 2006) (JetBrains, 2006), refactorings are difficult to change, maintain and reuse. A simplified view of refactoring as the composition of queries and transformations leads to think about it in a declarative way using ordered elements and additional information.

If we are able to manage refactoring pieces as isolated parts, they can be assembled at execution time using advanced programming mechanisms. With this aim, we migrate our initial proposal, of building and running refactorings in hand coded modules, towards a dynamic assembly, as can be seen in Fig. 2. Refactorings elements are not directly referenced in the code, but are declared in XML files. For example, a refactoring as *Rename Class*, has its counterpart as `RenameClass.xml` (see Appendix).

Refactoring engine has been modified to load the file on demand. Graphical user interfaces can be generated, asking the user to introduce refactoring input parameters. In our previous experience, each refactoring needed a customized window (hand coded GUI), whereas dynamic solution solves the problem in one step. Preconditions, actions and postconditions are also parsed from the XML definition. These parts are extracted and assembled from their repositories. The refactoring is build at runtime, reusing the engine presented in Section 4.

5.1 Refactoring Building Process

In order to include a new refactoring in our tool according to this new approach, contributors must:

1. Define refactoring parts.
2. Review parts contained in the refactoring repositories.
3. Build refactoring XML file:
   - include general information
   - select inputs
   - select preconditions (optional)
   - select actions
   - select postconditions (optional)
   - include code examples (optional)

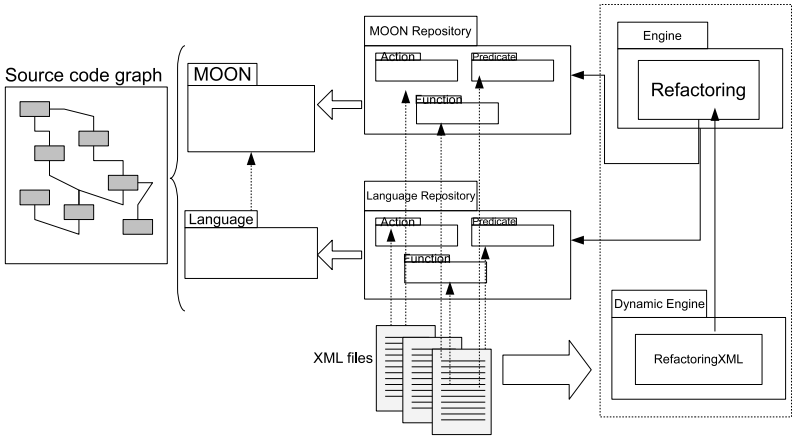

Figure 2: Dynamic refactoring building.

Next we detail some part of this process, as well as the process for building and executing the refactoring operation, once the refactoring definition data have been correctly uploaded.

5.2 XML Elements

Refactoring elements, previously described, have their counterpart in the XML file (see Fig. 2). The refactoring inputs are identified as elements of the MOON or extension model, using a type mark. A qualified name is also given that allows the input item to reference other elements (pre/postconditions and actions). Special inputs as root elements are used as a guide to the graphical interface. When refactorings need inputs derived from other inputs, an attribute `from` is included.

Preconditions and postconditions reference predicates to be evaluated. These items are looked up in current repositories (MOON and language repositories). If they do not exist, a definition and implementation subprocess must be completed before following the refactoring construction. Parameters are added using previous input names. It is mandatory that all parameters come from inputs, or their values can be derived from inputs. Actions use the same strategy, including target elements to be transformed. Optionally, postconditions are included to check the correct execution. We always consider the model (current graph) as default input. Finally, the XML file is validated with a DTD to check its correctness.

5.3 Assistant Tool and Reflection

Since XML manual construction is prone to failures, a graphical interface is provided to build the XML file. The assistant or wizard helps the user to build correct refactorings, through five steps: enter general refactoring information, select inputs, add pre/post and actions, include some code examples and confirm changes.

Referenced elements from repositories are validated using a reflection mechanism. If the classes cannot be loaded, they are not shown in the wizard screens. Thus, final definition exclusively contains runnable pieces of code. Once the XML refactoring file is completed, it is saved in one specific directory where dynamic refactoring engine can find it and parse it (see Fig. 2).

Our tool dynamically detects available refactorings, reading suitable XML files stored in the directory, and shows the applicable set of refactorings depending on current selected elements in the model (i.e. packages, classes, methods, etc). When the user wants to use these refactorings, he/she must select the refactoring and a dynamic window is created on the fly. Inputs, types and names, are obtained defining the screen layout. The user must enter new values for each one of these fields. From these entry points, preconditions, actions and postconditions are extracted from XML files, recovered from repositories, parameterized with previous inputs, and used to feed the refactoring engine framework (see Fig. 1 and 2). Lately, refactored code is regenerated from the new model state.

5.4 Current Development

We have developed a prototype as concept proof of our proposal with a set of eight refactorings. The refactoring set was previously implemented using libraries, where each refactoring is codified and implemented on the basis of them. Changes to these refactorings implied a new development process, editing source codes and rebuilding refactoring libraries.

Our current development migrate from closed

support definition to new XML file definition. Refactorings are not hand coded, but they are assembled from elements of the refactoring repository. It is always necessary a complete set of queries and transformations, but if this requirement is fulfilled, refactoring definition is completed.

The tool guides the construction process, showing current elements in the refactoring repository and allowing their selection and coupling, using a wizard. A second phase appears when the user needs to run the refactoring on real code. Now, previous pieces work together to put the refactoring in motion. Although there are not different results between the solutions (hand coded or generated), maintenance problem is solved in a smooth way. Future changes are applied using the assistant again, in few seconds.

Some features are difficult to remove. The person who assembles refactorings needs to know language particular features and the repositories. Nevertheless this problem was also present in previous solutions. On the other hand, both of them, static vs. dynamic methods, can be simultaneously included in the same tool without problems.

6 RELATED WORKS

Some approaches to the language independence problem in refactoring are based on metamodel solutions as FAMIX (Tichelaar, 2001) and its refactoring environment named Moose (Nierstrasz et al., 2005). This tool includes a refactoring framework that allows some kind of source analysis. Nevertheless, code changes remain language specific using Refactoring Browser (Roberts, 1999) for Smalltalk and a text-based approach for Java.

In the same line, (Van Gorp et al., 2003) develop a new solution based on metamodels. Authors propose eight additive and language-independent extensions to the UML 1.4 metamodel, which form the foundation of a new metamodel named GrammyUML, very similar to the Fujaba metamodel (Burmester et al., 2003), since they use the visual language named SDM (Story Driven Modelling) to describe refactoring preconditions. Some problems about representation completeness which force to add new features to the SDM language appear. Refactoring operations are run as graph rewriting in Fujaba. Neither reuse of refactoring elements, nor declarative refactoring were faced in this work.

In (Kniesel, 2006), author works in refactoring composition with the aim to simplify a sequence of refactorings. The main goal is to ensure atomic execution of a refactoring chain by creating a single equivalent conditional transformation. The proposal, named ConTraCT, was applied to the Java language, since language independence was not faced. There is no XML support, although conclusions of that work could be considered valid for our proposal.

The aim of independence of source core model, target programming language and code manipulation language, using XML as unique tool is described in (Mendoça et al., 2004). Although XML use gives a great independence, queries and updates are rewritten for each new schema (language programming), and refactoring construction requires to work with XML and other derived technologies, without aided support.

Refactoring tools available today as Refactoring Browser (Roberts, 1999), Refactor-Pro (Inc., 2007), Eclipse (Frenzel, 2006), etc, also include a refactoring engine, but it is not available a smooth way to define, reuse, construct and evolve refactorings. Language independent support is weak or completely absent, some steps are given by Eclipse offering a language independent part mainly at refactoring scheme and GUIs construction.

7 CONCLUSIONS

The framework and solution described in this paper reduce the need to build refactorings from scratch, and evolve previous refactorings with a relative low effort, as long as repository items are available. Our proposal allows developers to enrich the refactoring offer as long as to tune, enhance, or upgrade the already offered refactoring operations. The approach provides certain kind of dynamism in refactoring construction and evolution. Furthermore, if refactorings can be seen as compound actions, our solution supports the use of refactorings as high level actions leading to more complex refactoring operations.

We developed the framework with the refactoring engine and the language independent treatment, based on the metamodel core with extensions points as hooks and the particular extensions for each language (Java delivered, C# and Eiffel in process). The proposal presented in this paper on dynamic construction and evolution based in XML and reflection is available as a prototype.

Starting from our previous experiences developing this prototype, we are currently developing a plug-in for Eclipse with Java, with the same functionalities. In the same line, .NET platform is being included in the tool. Models and frameworks should be extended through hook methods, to include .NET features.

Furthermore, new possibilities emerge. Develop-

ers can build complex transformations required for evolution process, not including “preconditions” and building their own non preserving program behavior transformations. This can aid in evolution process, automating complex transformations.

REFERENCES

Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J. P., Wagner, R., Wendehals, L., and Zündorf, A. (2003). Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. In *Proc. of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, Satellite Event of the joint Conferences ESEC/FSE 2003*, pages 51–56.

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

Fayad, M., Schmidt, G., and Johnson, R. (1999). *Building Applications Frameworks: Object-oriented Foundations of Framework Design*. Wiley Computer Publishing.

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

Frenzel, L. (2006). The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. Eclipse Magazine.

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

Inc., D. E. (2007). Refactor Pro for Visual Studio .NET.

JetBrains (2006). IntelliJ IDEA :: The Most Intelligent Java IDE. http://www.jetbrains.com/idea/. Java IDE.

Kniesel, G. (2006). A Logic Foundation for Conditional Program Transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn.

Mendoça, N. C., Maia, P. H. M., Fonseca, L. A., and Andrade, R. M. C. (2004). RefaX: A Refactoring Framework Based on XML. In *20th IEEE International Conference on Software Maintenance (ICSM’04)*, pages 147 – 156.

Nierstrasz, O., Ducasse, S., and Gîrba, T. (2005). The Story of Moose: An Agile Reengineering Environment. In Wermelinger, M. and Gall, H., editors, *ESEC/SIGSOFT FSE*, pages 1–10. ACM.

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA.

Roberts, D. B. (1999). *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA.

Tichelaar, S. (2001). *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern.

Van Gorp, P., Van Eetvelde, N., and Janssens, D. (2003). Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Meta model. In *Proceedings of 1st Fujaba Days*.

APPENDIX

Refactoring example with its suitable XML definition: *Rename class*, included in (Opdyke, 1992).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE refactoring
SYSTEM "refactoringDTD.dtd">
<refactoring name="Rename Class" version="1.1">
  <information>
    <description>Change name class.
    </description>
      <motivation>...</motivation>
  </information>
  <inputs>
     <input type="moon.core.ClassDef"
        name="Class" root="true"/>
     <input type="moon.core.Name"
        name="OldName" from="Class"/>
     <input type="moon.core.Name"
        name="NewName" />
  </inputs>
  <mechanism>
  <preconditions>
     <precondition
     name="NotExistsClassWithSameName">
        <param name="Class"/>
        <param name="NewName"/>
     </precondition>
  </preconditions>
  <actions>
       <action name="RenameClassAction">
          <param name="Class"/>
          <param name="NewNombre"/>
     </action>
     <action name="RenameConstructorAction">
          <param name="Class"/>
          <param name="NewName"/>
     </action>
     <action name="RenameJavaFileAction">
          <param name="Class"/>
          <param name="NewName"/>
     </action>
  </actions>
  <postconditions>
     <postcondition
     name="NotExistsClassWithSameName">
          <param name="Class"/>
          <param name="OldName"/>
     </postcondition>
  </postconditions>
  </mechanism>
</refactoring>
```