A Case Study for Program Refactoring

Berthold Hoffmann[1], Javier Pérez[2⋆], and Tom Mens[3]

[1] Universität Bremen, Germany
[2] Universidad de Valladolid, Spain
[3] Service de Génie Logiciel, Université de Mons-Hainaut, Belgium

Abstract. This paper proposes a case for the GraBaTs'08 tool contest. The graph transformation system to be implemented shall (*i*) import a graph-based representation of models of Java programs in a Gxl-based format, (*ii*) allow these models to be transformed interactively with three well-known program refactorings *Encapsulate Field*, *Move Method*, and *Pull-up Method*, and (*iii*) export the resulting models in the same Gxl-format. The case aims to enable comparison of various features of graph transformation tools, such as their expressiveness and their ability to interact with the user.

1 Introduction

Model-driven design of object-oriented software is a very active area of research in software engineering. Since models and programs can be represented as graphs (e.g., diagrams in Uml), and these software artefacts often need to be transformed in the development process, *graph transformation* seems to be an ideal formalism to specify model and program transformations. Refactoring is a particular kind of software transformation by which the structure of software shall be improved whereas its behavior shall be preserved. Many people have already studied how refactoring can be specified with graph transformation. So there should be enough interest in the graph transformation community to propose a case study in this area, which poses challenges for all tools based on graph transformation.

The remainder of this proposal is structured as follows. A brief survey of program refactoring is given in Section 2. Section 3 describes the task in detail. Section 4 describes the evaluation criteria that will be used to compare contributions to the case study and Section 5 concludes. The precise specification of the program model to be used and the refactorings to be implemented is given in the appendix.

2 Refactoring

After the term *refactoring* (of object-oriented programs) was coined by W.F. Opdyke in 1992 [11], the technique became widespread only with the book of M. Fowler.

⋆ Work done while on leave at Université de Mons-Hainaut. This work has been partially funded by the regional government of *Castilla y León* (project VA-018A07).

[3], In his book, Fowler collected and defined the "reference" set of basic program refactorings for JAVA which most interactive development Environments do support. In particular, the Eclipse development environment for JAVA provides very extensive support for program refactoring. In this case study we will base our refactoring examples on how it can be achieved in Eclipse (modulo a number of simplifications).

The relation between refactoring and graph transformation has already been explored (and published) by various authors, most notably with the tools AGG and FUJABA [8, 6, 5, 12], that shows the feasibility of the case we are proposing. Other useful references about the link between (model) refactoring and graph transformation are: [1, 2, 4, 7, 10, 9, 13]

3 The Task

The task is to use a graph transformation tool, called the *Tool* in the following, to implement a refactoring system, which is henceforth called the *System*.

3.1 Required Functionality

The Tool shall be used to generate a System with the following features:

1. The System reads program graphs according to the meta-model specified in B.1, which are given in a serialized format as a GXL file.[4] We have chosen GXL because we think it is the most adequate format for this contest case, given that it is a simple XML-based specification format for graphs, that is well-defined, and known by the graph transformation community, and supported as an input/output format for various tools. [5]. An example graph of a program, modeling local area networks, which conforms to this meta-model can be found in B.2.
2. The System displays this model to its users. We leave it open to the Tool to decide how to display the program graph. Many different result representations are possible: textual, tree-based, graph-based, etc.
3. The System allows users to apply the following refactoring operations of Fowler [3], for which we give precise specifications in Appendix A:
   - Move Method
   - Encapsulate Field
   - Pull-Up Method
4. The refactorings can be applied interactively (i.e. with user guidance when needed).
5. The System writes transformed graphs according to the GXL meta-model.

[4] See Documentation under http://www.gupro.de/GXL/index.html.

[5] For those tools that do not support the format, a converter can be written easily due to the simplicity of the chosen representation, for example, with XSLT.

3.2 Optional Functionality

The features described in subsection 3.1 are required. On top of these, the System may also support the following optional features. The participants' contribution should explain clearly how these criteria have been addressed.

- Reuse and composition of the basic refactorings. Does the System allow to compose simple refactorings into more complex ones? Does the System provide mechanisms to reuse existing refactoring specifications into new ones?
- Automatic checking of properties:
  - Does the system guarantee well-formedness and consistency of the program graph after a refactoring step automatically or does it help the user to manually do so?
  - Is there any support for checking behavior-preservation of a refactoring?
  - Does the System allow to reason about dependencies and constraints between refactoring operations (*e.g.* causal dependencies, constraints imposed on the application order, refactorings that represent mutually exclusive alternatives and so on)?
  - Does the System support opportunities/suggestions of where to apply which refactorings (*e.g.* to remove bad smells).

3.3 Rationale

This case is proposed with the following considerations in mind:

1. The refactorings required for this case range from moderately complex to rather complex.
2. What graph transformation tools could contribute extra (with respect to "ad hoc" implementations of refactoring) is the ability to formally reason about formal properties of refactoring, such as parallel and sequential dependencies between refactorings, termination properties, and many more.
3. The Case is non-trivial, so it may serve as a "stress test" for contemporary tools.

3.4 Provided Material

We provide files with the meta-models and example models (the LAN program), which can be downloaded from

http://www.infor.uva.es/~jperez/GraBaTs2008/refactoringCaseProposal.tar.gz

Textual documentation of these items is enclosed in the Appendix:

- A meta-model as the specification of the program graphs representation for Java programs. See B.1.
- A meta-model of the Gxl serialized format for program graphs.
- Program graphs representing the LAN example according to the Gxl models. See B.2.

3.5 The Purpose of the Case Study

The case will explore the following features of a graph transformation tool:

- Its ability to specify graph models
- The expressiveness of its transformation rules
- The expressiveness of its control constructs
- Its interactive support for selecting, applying, controlling and parameterizing transformations.
- More advanced support, such as the ability to formally reason about refactoring transformations.

3.6 Challenges

The challenges of this task will include the following aspects:

- The program graphs to be transformed are reasonably complex, as they involve many static properties of programs, not only syntax (hierarchical structure), but also scope rules for names and type rules in a programming language.
- The transformations are rather complex, involving complicated patterns and sophisticated application conditions, and / or complicated strategies for rule application.

4 Evaluation Criteria

In the context of model refactoring, the tool contest should aim to measure many different criteria, some more practically oriented, others more theoretically oriented:

- Expressiveness: For complex refactorings, mechanisms are often needed to express control flow among primitive rules, or to parameterize transformations. For checking the preconditions of refactorings, relatively complex logical expressions or path expressions may be needed.
  - How "self-explanatory" is the specification of the refactoring operations?
  - Which language concepts help to specify the operations?
  - Can primitive refactoring operations be be reused to build other refactorings?
  - Can refactorings be composed to build larger refactorings?
  - Can refactorings be parameterized?
- *User Interaction*
  - How can refactoring selection, parameterization, and application be triggered by the user?
  - Is the System robust against malicious or accidentally incorrect user input?
  - Is there help to indicate the place(s) *where* a chosen refactoring could be applied?

    - Is here help to indicate *which* refactorings could be applied at a given place?
    - Does the system provide feedback to the user on potentially interesting relations between refactorings? (other alternatives, potential conflicts, etc.)
- Genericity: Is it possible to specify refactorings in a generic way, so that they can be applied to different meta-models? To which extent can the refactoring rules be parameterized?
    - This is a quite important question in the light of domain-specific languages: ideally, it should be relatively easy to support refactorings for these languages.
    - This question is also important to support language migration. If the meta-model evolves due to changes in the language specification, all models should be migrated to the new meta-model. If refactoring operations are defined in a sufficiently generic way, this task may be simplified.
- Extensibility: How easy is it do add new refactorings in the tool? (a) For a graph transformation expert; (b) for a novice user
- Usability: How easy is it to use the tool for performing refactorings? What is the learning curve? This also includes user interface aspects such as the ability to implement a refactoring plug-in, the ability to apply refactorings via context-sensitive menus or by selecting the appropriate model elements that one wishes to refactor, user interaction during the application of the refactoring and so on.
- Formal properties: Which formal properties of the tool can be exploited to reason about the model refactorings, and to ensure their correctness? Some examples: confluence analysis, termination analysis, sequential and parallel dependency analysis.
    - Are all decidable preconditions of a refactoring checked before it is applied?
    - How does the System handle preconditions that are *undecidable*? (An example is the condition that the method bodies to be "pulled up" shall be equivalent.)
    - Does the System guarantee that the refactored graph does again conform to all constraints of the model?
    - Can conflicts between refactorings be determined? (E.g, overlapping matches of operations?)
    - How to express constraints on and between refactoring operations?
    - How to deal with postconditions? Due to the fact that these may depend on input provided by the users during the refactoring, they may not be necessarily be convertible into an equivalent precondition.

4.1 Tasks for Live Contest

The developers of the System will be asked to do the following:

1. Refactoring of another, maybe bigger program graph conforming to the same GXL meta-model. (This checks for generality of the refactorings, and for scalability of the System.)
2. Specifying one or several other refactoring operations and apply them to the program graph of the Case, as well as the program graph of point 1 above. (This checks for extensibility.)
3. As an alternative to 1. we can provide an evolved meta-model (reflecting a new version of the programming language), and the refactorings need to be migrated to conform to this new meta-model.

5 Conclusions

The task specified in this proposal is of medium size, but quite demanding (for tools and their users). The case is representative of a quite popular application area – program refactoring. So we think it will attract several people to provide solutions.

References

1. E Biermann, Karsten Ehrig, C. Köhler, Gabriele Taentzer, and E. Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In O. Nierstrasz, editor, *Proceedings of International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 219–257. Springer, 2006.
2. Paolo Bottoni, Francesco Parisi-Presicce, G. Mason, and Gabriele Taentzer. Specifying coherent refactoring of software artefacts with distributed graph transformations. In P. van Bommel, editor, *Handbook on Transformation of Knowledge, Information, and Data: Theory and Applications*, pages 95–125. Idea Publishing Group, 2005.
3. Martin Fowler. *Refactoring—Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, Reading, MA, 1999.
4. Lars Grunske, L. Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Daniel Varro. Using graph transformation for practical model driven software engineering. In V. Gruhn S. Beydeda, M. Book, editor, *Model-driven Software Development*, pages 91–118. Springer, 2005.
5. Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Daniel Varro. *Using Graph Transformation for Practical Model Driven Software Engineering*, volume II: Model-driven Software Development, pages 91–117. Springer, 2005.
6. Tom Mens. On the use of graph transformations for model refactoring. In Joost Visser Ralf Lämmel, João Saraiva, editor, *Generative and transformational techniques in software engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 219–257. Springer, 2006.
7. Tom Mens. On the use of graph transformations for model refactoring. In *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2006.

8. Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, pages 269–285, September 2007.
9. Tom Mens, Gabriele Taentzer, and Olga Runge. Analyzing refactoring dependencies using graph transformation. *Journal on Software and Systems Modeling*, 2007. To appear.
10. Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.
11. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
12. Javier Pérez and Yania Crespo. Exploring a method to detect behaviour-preserving evolution using graph transformation. In Tom Mens, Kim Mens, Ellen Van Paesschen, and Maja DHondt, editors, *Proceedings of the Third International ERCIM Workshop on Software Evolution*, pages 114–122. ERCIM, October 2007. Informal Workshop proceedings.
13. J. Zhang, Y. Lin, and J Gray. Generic and domain-specific model refactoring using a model transformation engine. In *Model-driven Software Development – Research and Practice in Software Engineering*. Springer, 2005.

A Precise refactoring specifications

We provide simplified definitions of the following Fowler refactorings, based on the way they are implemented in Eclipse. We have chosen this implementation because it is probably the most popular and best-known.

Participants are asked to implement the following three refactoring operations, but the third one (*Pull Up Method*) can be considered as optional in lack of time. The given specifications have been extracted manually (by inspecting the source code and testing the refactoring itself) from Eclipse 3.3.1 and 3.3.2. These specifications are simplified and modified versions of the actual implementations, because we think they are better suited to fulfill the case objectives than the originals. Both the Eclipse originals and the given specifications should be taken by participants as initial guidelines. Participants are invited to further extend their refactoring implementations to approximate the Eclipse variant as close as possible, in order to come to a "realistic" solution, and in order to "stress test" the Tool as much as possible.

Encapsulate Field

operation: encapsulateField(Class container, Variable var, String getterName, String setterName, Boolean useAccessorsAlways)

preconditions: The variable *var* must exist within the class *container*.
If the getter or setter names being proposed already exist within the container class, the methods and the variable must be all non-static or all static.

mechanics: An instance variable is turned private and getter and setter methods are created to access it. All references to the encapsulated field (excluding those within the container class) are updated to use the accessors.

- If the "useAccessorsAlways" option is set to true, references to the variable within the container class will be turned to calls to accessors too.

- if the visibility of the variable is not public, the user is asked about the desired visibility of the accessor methods: when protected or default, user can choose between the original visibility or public, when private, the user can choose any visibility. Only methods created by the refactoring are given the chosen visibility, if an accessor method already exists and it is used in the refactoring, its visibility remains unchanged.
- Eclipse offers two more options: one to specify where to insert the newly created methods, within the source code text, and other to generate method comments. None of these options is interesting for the Case.

Move method

operation: moveMethod(Class source, Class target, MethodBody method, Boolean useDelegation)

preconditions: The target should be reachable from the code that calls the moved method. Eclipse implements it with the following restriction: either there is an instance variable of the target's type within the source class, or one of the method parameters is the same type of the target class.

The moved method should not have a call to "super".

Static methods are not meant to be moved by this refactoring.

Name conflicts can be caused by methods with the same name as the moved method, which exists either in the target class or in its super-classes. The second conflict is not really a conflict, the moved method overrides a parent's one, but it is a clear problem for behavior preservation. For this Case, we will specify that methods that cause name conflicts can't be moved. [6]

This refactoring cannot be applied to methods in interfaces.

description: A method body is moved from a source class to a target class.

- If the "useDelegation" option is set to false, the method is moved to the target class, and all the references/calls to the method are updated.
- If the "useDelegation" option is set, the method is copied to the target class and a delegating call is added within the original method body. Updating of references is not needed.
- To update references:
  - if the method is being moved to a target class, and this target class is referenced in the source class by an instance variable, then the call to the moved method, which is being performed through a reference to the source class (referenceToSourceClass.method(p) call in the caller code) should be substituted by an access to the reference of the source class, followed by an access to the instance variable of the target class and followed by the method call (referenceToSourceClass.referenceToTargetClass.method(p)).
  - if the method is being moved to a target class, which is the type of one of the method's parameters (p), the call referenceToSourceClass.method(p) is substituted in the caller code by p.method().
  - If members of the target class are accessed within the body of the method being moved, references can be updated, for example, from referenceTo-TargetClass.member() to simply member().

[6] In Eclipse, moving methods is allowed even in these situations. We leave it open to the developers to implement the Eclipse variant.

- If the method being moved accesses members from the source class, the source class must be passed as an argument. Therefore, a parameter (p) of the source class type is added to the moved method, and references are updated in the caller code from referenceToSourceClass.method(p) to p.method(referenceToSourceClass).
- If the method is overridden within subclasses of the source class, or overrides methods of a superclass of the source class, the "useDelegation" version of the refactoring is applied.
- If the "useDelegation" version of the refactoring is applied, Eclipse marks the old method as "deprecated". This particular behavior does not need to be implemented during this case study.
- if the visibility modifiers of the moved method, or those of the members accessed from it, prevent that the moved method could be executed, all the visibility modifiers involved are recursively changed to public.
- The "mark method as deprecated" option is not meant to be implemented.

Pull Up Method

operation: pullUpMethod(Class source, Class target, MethodBody method, Boolean makeAbstract, Class[] keepMethods)

preconditions: The source class must have a superclass to which members could be pulled up.
The target class must be a superclass of the source class.
A method with the same signature can not be declared within the target superclass.
A constructor is a special kind of method that is not meant to be pulled up with this refactoring.
All elements referenced from within the moved method must be accessible for the method in its new location (superclass).
All occurrences of the method to be be pulled up, within subclasses of the target superclass, must share the same operation signature as the pulled up method.
All occurrences of methods within subclasses of the target superclass and with the same operation signature, must have the same return type.

mechanics: In Eclipse, the "pull up" refactoring is implemented in a slightly different way, It can be used to pull up many members at a time. In order to simplify the case, we will restrict this refactoring to pull up a single method [7]. The pulled up method is moved to the superclass.

- If the "use destination type where possible" is set, all the references to the pulled up method that do not depend on members of the source class, are updated to point to the target superclass.
- The user must specify which occurrences of the method within the subclasses of the target superclass are being kept and not removed. Methods which are not deleted will override the pulled up method in their respective classes. If no method is specified, the default behavior will be to remove all occurrences. In Eclipse, this is implemented differently: methods have to be marked for deletion explicitly, but we think implementing the refactoring this way is more interesting for the GT tool contest case.

[7] Nevertheless, we leave it optional to the developers to implement the more complex variant as a composite refactoring

– If the user specifies the "makeAbstract" option, an abstract version of the pulled method is created in the target superclass and the method body is not pulled up. Stub methods (methods with empty bodies) have to be created in subclasses which do not implement the pulled up method and are not abstract.
– Visibility of private methods must be changed to protected.

B Documentation of (Meta-) Models

We enclose documentation of the material mentioned in section 3.4.

B.1 Meta-model for Program Graphs

We propose to use the meta-model of Figure 1 to represent JAVA programs. Some restrictions over the JAVA code can be considered in order to simplify the representation.

– The following concepts will not be modeled:
  • Inner classes and anonymous classes

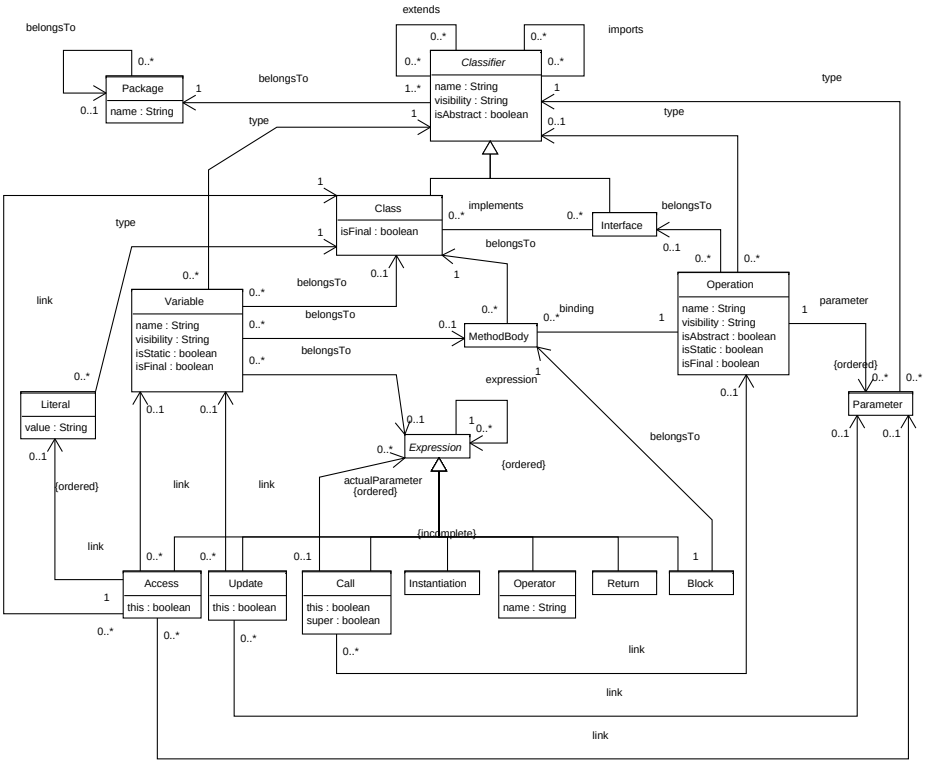

Fig. 1. Meta-model for graph representation of Java programs

  - Exception handling
  - Generic types (JAVA 5 and later)
  - Annotations (JAVA 5 or later)
  - Type casting
- Constructor methods are represented as ordinary methods (except that they follow a specific name convention).

In addition to the meta-model of Figure 1, the following constraints, derived from the JAVA language specification, apply. They are expressed in natural language since they are difficult to express visually:

- Classes can not extend to/from interfaces.
- Classes do not have multiple inheritance.
- The extends, belongsTo relations should be acyclic.
- The belongsTo relations from variable are mutually exclusive.
- The link relations from Access or Update are mutually exclusive
- The "extends", "belongsTo" and "expression" relations
- The "this" and "super" attributes of a "call" entity can't be simultaneously true.
- The classification of expression "subtypes" is probably incomplete, but it is a minimal set
- Please consider that there are many expression relations between the subtypes of the expression entity, that are not allowed by the definition of the Java language. These constraints are not included in the model for the sake of simplicity but they should be taken into account.
- ...

B.2 Gxl Model for the LAN Program

The files and graphs are to big to be fitted in in the paper. They can be downloaded from

http://www.infor.uva.es/~jperez/GraBaTs2008/refactoringCaseProposal.tar.gz

B.3 A Program to be Refactored

Again, these files and graphs are to big for this paper, but can be downloaded from

http://www.infor.uva.es/~jperez/GraBaTs2008/refactoringCaseProposal.tar.gz