Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A Case Study to Evaluate the Suitability of Graph Transformation Tools for Program Refactoring

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

[1] Departamento de Informática, Universidad de Valladolid, Spain
e-mail: {jperez|yania}@infor.uva.es
[2] Fachbereich Mathematik und Informatik, Universität Bremen, Germany
e-mail: hof@informatik.uni-bremen.de
[3] Institut d'Informatique, Faculté des Sciences, Université de Mons, Belgium
e-mail: tom.mens@umons.ac.be

Received: date / Revised version: date

Abstract This article proposes a case study to evaluate the suitability of graph transformation tools for program refactoring. In order to qualify for this purpose, a graph transformation system must be able to (*i*) import a graph-based representation of models of Java programs , (*ii*) allow these models to be transformed interactively with well-known program refactorings, and (*iii*) export the resulting models in the same graph-based format used as input. The case study aims to enable comparison of various features of graph transformation tools, such as their expressiveness and their ability to interact with the user. The model of Java programs is presented and some examples for translating Java source code into the model are provided. The refactorings selected for the case study are specified in detail.

Key words: refactoring – graph transformation – case study – Java program graphs

1 Introduction

Model-driven design of object-oriented software is a very active area of research in software engineering. Analysis and design models as well as programs can be represented as graphs (*e.g.*, diagrams in UML), and these software artefacts often need to be transformed during the development process. Therefore, *graph transformation* seems to be an ideal formalism to specify model and program transformations [15].

Refactoring is a particular kind of software transformation that aims to improve the structure of software while preserving its behaviour. Many authors have already studied how refactoring can be specified with graph transformation [1,2, 5,8,9,11,13,17]. Hence there is 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.

This paper describes a case study that can be used to test certain features of graph transformation tools and approaches. Specifically, it is focused in those features that will be desirable to support formal program-refactoring: expressiveness, the ability to specify complex graphs and complex rule execution control, etc. This case study was originally proposed for a contest organized as part of the International Workshop on Graph-Based Tools (GraBaTs 2008). Five graph transformation tools contributed to the contest, providing solutions for this "Refactoring Case".

The remainder of this article is structured as follows. A brief introduction of program refactoring is given in Section 2. The case study task is described in detail in Section 3. The precise specification of the program model to be used as well as some examples of translating Java code into the program model are given in Section 4. Three refactoring specifications to be implemented are detailed in Section 5. Section 6 describes the evaluation criteria that will be used to compare contributions to the case study and Section 7 concludes with a detailed discussion of the proposal.

2 Refactoring

Refactoring is an object-oriented source code transformation technique [12]. It is aimed at modifying the structure and the design of a system, while at the same time, keeping its observable behaviour unchanged. Some examples of refactorings are: renaming a class, which is used to change the name of a class so that the new name represents the class' responsability better than the old one; moving a method from one class to another, so that related services can be grouped together in the same class and thus, cohesion and coupling can be improved, etc. Improving the evolvability and reusability of a system are among the main objectives of refactoring an already working system.

⋆ Work partially done while on leave at Université de Mons.

After the term *refactoring* (of object-oriented programs) was coined by W.F. Opdyke in 1992 [12], the technique became widespread only with the book of M. Fowler [4]. In his book, Fowler collected and defined the "reference" set of basic program refactorings, which most interactive development environments support, and described them using examples written in JAVA.

In particular, the ECLIPSE development environment for JAVA provides very extensive support for program refactoring. The way that the ECLIPSE development environment implements refactorings differs to some extent from Fowler's reference. In fact, this is also the case for other available refactoring tools. In this case study, we include the specification of three refactoring operations which shall be implemented with a graph transformation system. The specification is mainly inspired on how the refactorings 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 [5, 8, 9, 13], thus illustrating the feasibility of the case study we are proposing. This relation has been motivated by the need to address some of the challenges faced by the current tool-support for refactorings, such as: composition of refactorings [7], searching refactorings [13], planning refactoring sequences [14], computation of conflicts and dependencies of refactorings [10], etc. These challenges would greatly benefit from a formal support provided by techniques such as graph transformation.

3 The Case Study at a Glance

The purpose of the case study (called the *Case* in the remainder of this article) is to allow researchers to compare the following features of graph transformation tools:

- **The ability to specify complex graph models**, because refactorings must deal with the representation of models of programs, including the program source code itself, which can lead to complex graph models.
- **The expressiveness of graph transformation rules** and, in particular, the ability to specify and verify refactoring pre- and postconditions, as well as the ability to apply the refactoring itself. For example, in order to specify and check preconditions, negative application conditions for rules, relatively complex logical expressions, path expressions, or other sophisticated mechanisms may be needed.
- **The expressiveness of control constructs**, because some *nontrivial* refactoring applications may require complex control flow.
- **Usability and understandability**, e.g., by providing interactive support for selecting, applying and controlling transformations. Such support is required to gather specific information used to achieve the application of a particular refactoring.
- **Genericity of the refactoring specification.** A refactoring can be customized for a particular program element. Nevertheless, in order to be productive in a software development platform, as well as to increase its reusability, it must be implemented in a generic way.
- **Extensibility**, i.e., the ease with which the proposed solution can be used to accommodate and implement new refactorings.
- **Performance requirements**, such as the speed of importing and exporting data, the speed of executing refactorings, the amount of memory used, and so on.

3.1 Rationale

The *Case* is proposed with the following considerations in mind:

1. The refactorings required for the *Case* range from moderately complex to rather complex. As such, they may serve as a "stress test" for contemporary tools.
2. What graph transformation tools could contribute extra (with respect to *ad hoc* implementations of refactoring) is the ability to formally reason about refactoring properties.

The main task to achieve in the *Case* is to use a graph transformation tool, called the *Tool* in the following, to implement a refactoring system, which is henceforth called the *System*. The specification of some requirements for the *Tool* to be implemented follows.

3.2 Required Functionality

The *Tool* shall be used as the basis on top of which a *System* with the following features shall be developed:

1. The *System* reads program graphs according to the meta-model specified in Figure 3. A set of sample graphs conforming to this meta-model have been obtained from a program that models local area networks. These samples can be found in subsection 4.1.
2. The *System* displays this program graph to its users. We leave it open to the *Tool* to decide how to display the program graph. Many different representations are possible: textual, tree-based, graph-based, etc.
3. The *System* allows users to apply the following refactorings of Fowler [4]: *Move Method*, *Encapsulate Field* and *Pull Up Method*. Their specification, together with a motivation of why we selected exactly these refactorings, is given in Section 5.
4. The refactorings can be applied interactively (*i.e.*, with user guidance when needed).
5. The *System* exports transformed graphs according to the given meta-model.

Since different graph transformation tools support different input and output formats, we suggest to deal with input and output models in a serialized format as GXL[1] files. This is not a mandatory requirement, however, the usage of a common format is desirable because it will ease comparison and

[1] See the full GXL documentation under `http://www.gupro.de/GXL`.

evaluation of results. GXL has been chosen for its simplicity and broad support. 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 [2].

3.3 Optional Functionality

The features described in subsection 3.2 are mandatory. In addition, the *System* may also support the following optional features.

Implementers of the *Case* should explain clearly whether and how these criteria have been addressed.

Composition and reuse of refactorings:

- Does the *System* allow to compose complex refactorings from simpler ones?
- Does the *System* provide mechanisms to reuse existing refactoring specifications, or parts of them?

Automatic checking of formal properties, i.e., the ability to formally reason about refactoring transformations. Such formal support will depend a lot on the chosen approach (tool and language), and may be used to verify properties such as consistency, completeness, correctness, termination, behaviour preservation, and many more. It can answer many interesting questions such as:

- Does the *System* guarantee well-formedness and consistency of the program graph after a refactoring step in an automatic way, or does it help the user to manually do so?
- Does the *System* provide any support for checking the behaviour-preservation property of a refactoring [3]?
- Does the *System* allow reasoning 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 detecting opportunities or giving out suggestions of where to apply which refactorings (*e.g.* to remove bad smells)?
- Does the *System* provide other kinds of formal support that could be beneficial in the context of refactoring?

3.4 Challenges

The following challenges make the implementation of the *Case* non-trivial:

- The program graphs to be transformed are reasonably complex. The given meta-model provides enough detail to allow checking for 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, likely involving complicated patterns and sophisticated application conditions, and / or complex strategies for rule application.

3.5 The LAN example

A program to be refactored is provided as an example, thus the solution implemented can be checked and evaluated with this example in order to complete the *Case*. The example describes a (toy) simulation of a LAN. We have selected this example because it is widely considered as a good case study. It has been accepted, and it is used frequently, by both the software evolution and graph transformation community as a reference example [3,6].

The example simulates a LAN connecting two different types of nodes: workstations and printservers. All nodes in the LAN are connected in a ring topology, so each node in the network is always connected to two other nodes, a previous and a next one. Nodes in the network send packets to the next node in the ring using a `send` method. Each node confirms the reception of a packet with an `accept` method. Depending on the type of node, each one performs a different task with the packet. Workstations simply forward the packet while printservers extract the content of the packet to print it. Each packet can contain two types of documents: ASCII or PostScript. The only difference between them is that PostScript documents can contain an array of figures while ASCII ones cannot.

Figure 1 shows a bird's-eye overview of the system structure in terms of packages and classes. Figure 2 shows more details of its classes in terms of their operations and attributes.

Additionally we provide files with the meta-model and the example models (from the LAN program), which can be downloaded from `http://www.infor.uva.es/~jperez/GraBaTs2008/refactoringCaseProposal.tar.gz`. The content of this file includes:

- a meta-model specifying the program graph representation for Java programs. The meta-model is provided in XMI format and in ArgoUML[4] format.
- various snapshots of the LAN example source code. Each snapshot is the result of applying one of the three refactorings requested in this *Case*.
- serialized versions –in GXL format– of some program graphs representing the LAN example according to the GXL models (See Section 4.1).

The specification of the program graph representation for Java programs, with additional textual documentation of the meta-model, is given in Section 4. Some examples are provided as well, to illustrate how to represent some particular source code fragments in the Java program graph format.

[2] For those tools that do not support this format, a converter can be easily written, due to the simplicity of the chosen representation (*e.g.* with XSLT).

[3] As a suggestion, a description of these behaviour-preserving conditions can be taken and adapted from [11] if they are needed.

[4] ArgoUML is an open-source UML modeling tool that can be downloaded from `http://argouml.tigris.org`

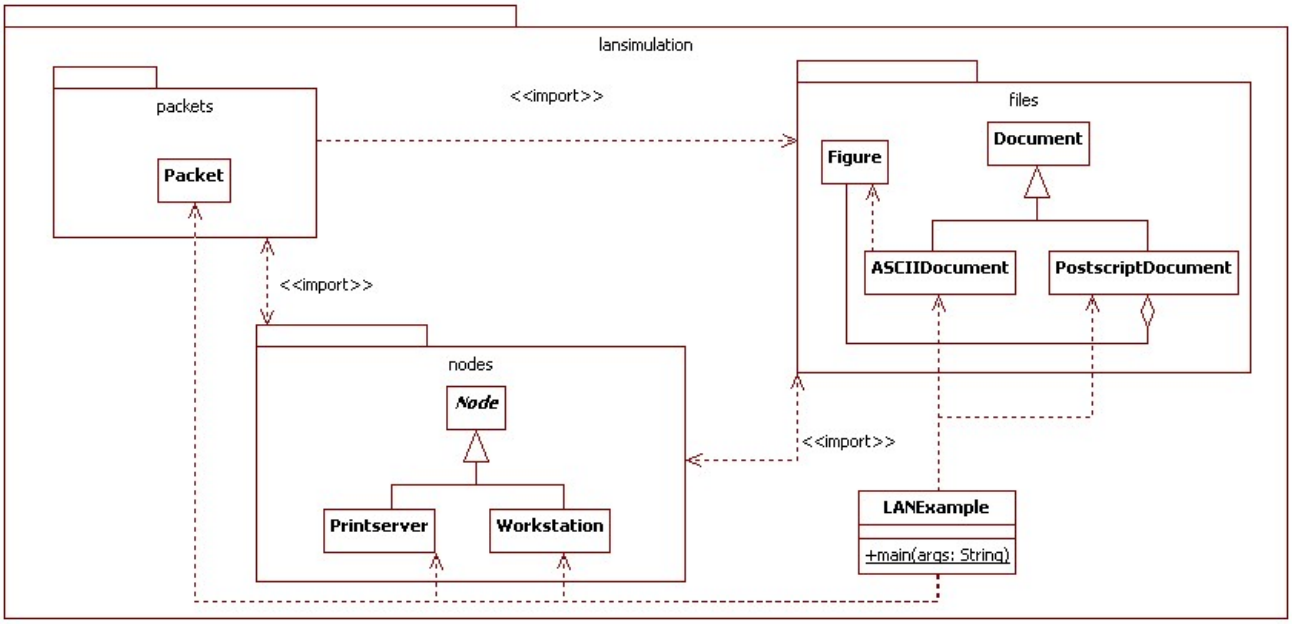
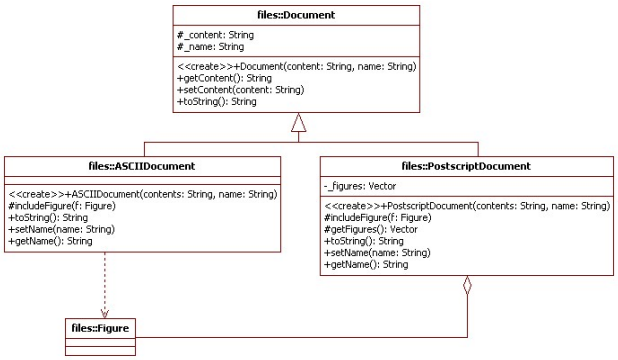
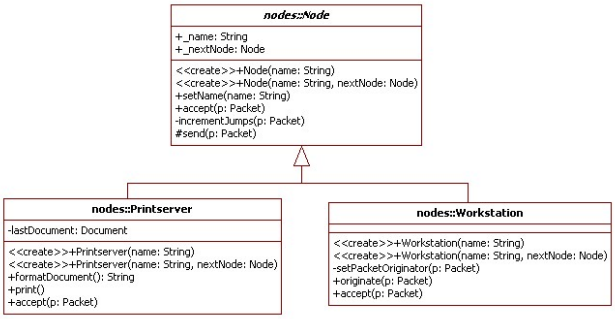


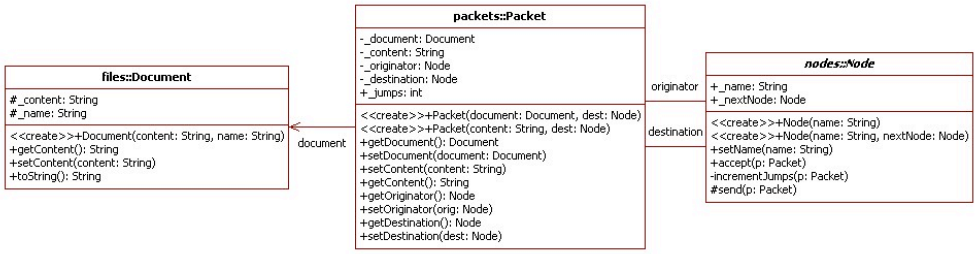
Figure 1. A bird's-eye view of the LAN example.



(a) Classes in the files package



(b) Classes in the nodes package



(c) Packets, Documents and Nodes

Figure 2. Details on classes and their relations of the LAN example.

4 Meta-model for Program Graphs

We propose to use the meta-model of Figure 3 to represent JAVA programs. The selection of a meta-model could have been left open, but we have considered appropriate to provide a reference meta-model. This way, the different graph transformation tools sharing the same meta-model will be more easily comparable. This model is based on other works that addressed the problem of formalising refactorings with graph transformation [11,13]. We have tailored these meta-models, instead of choosing an already existing one, in order to produce a meta-model with an adequate level of detail for this *Case*. The JAVA AST generated by ECLIPSE, for example, would have been too much detailed.

In order to keep the *Case* feasible, without removing too much of its inherent complexity, we decided to restrict ourselves to a subset of JAVA programs. In particular, the following concepts are not included in the meta-model and thus should not be represented by solution providers:

1. Inner classes and anonymous classes
2. Exception handling
3. Generic types (JAVA 5 or later)
4. Annotations (JAVA 5 or later)
5. Type casting

It should also be noticed that there is not an specific node type to represent constructors. These methods are represented as ordinary methods (except that they follow a specific name convention).

The restrictions imposed by the meta-model of Figure 3 are completed with the following well-formedness constraints, derived from the JAVA language specification:

- **Constraint 1:** Classes can not extend to/from interfaces.
- **Constraint 2:** Classes do not have multiple inheritance (classes can not extend multiple classes).
- **Constraint 3:** The belongsTo relations from Variable are mutually exclusive.
- **Constraint 4:** The extends, belongsTo and expression relations should be acyclic.
- **Constraint 5 and 6:** The link relations from Access or Update are mutually exclusive.
- **Constraint 7:** The this and super attributes of a call entity cannot be simultaneously true.

These constraints are expressed as forbidden subgraphs in Figure 4. Essentially, a forbidden subgraph expressed a conjunction of graph patterns that are prohibited, i.e., not allowed to occur in a well-formed JAVA program graph model.

The classification of Expression subtypes in the metamodel of Figure 3 is probably incomplete, but it includes enough synutactic elements to model programs for code refactoring purposes. Please consider that there are many expression relations between the subtypes of the Expression type, which 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.

Another concern that deserves attention is dynamic binding. In JAVA, it is not possible to know statically which will be the exact type of an entity at run time. This difficulty is inherent to the language, hence it imposes a limitation over the expected representation of JAVA programs. Only the static types of entities, those which are well-known from the declaration statements, have to be represented.

4.1 The meta-model by examples

The files and graphs for the GXL model of the whole LAN program are too big to fit in the paper. They are included within the downloadable material.

Nevertheless, in this section we present some graph examples that show the different abstraction levels of the graph representation. A graph representing the highest level entities of the LAN example –classes and packages– is displayed in Figure 5. This graph models the system structure shown in Figure 1.

A JAVA code excerpt of the Document class is shown in the following listing. The graph representation for this class structure, ignoring method bodies, is depicted in Figure 8.

```
package lansimulation.files;

public class Document {
  protected String _content;
  protected String _name;

  public Document(String content, String name) {
    setContent(content);
    _name = name;
  }
  public String getContent() {
    return _content;
  }
  public void setContent(String content) {
    _content = content;
  }
  public String toString() {
    return getContent();
  }
}
```

Another JAVA code fragment, namely the incrementJumps method of class Node, is shown in the following listing. Its corresponding graph representation is displayed in Figure 6.

```
  private void incrementJumps(Packet p){
    p._jumps = p._jumps + 1;
    System.out.println("Traversing node "+this.
        _name);
  }
```

The listing of a constructor method for class Node is shown below. Its graph representation is depicted in Figure 7.

```
  public Node(String name, Node nextNode) {
    setName(name);
    this._nextNode = nextNode;
  }
```

Let us now compare both method's body representations. The former example, incrementJumps, differs from the latter

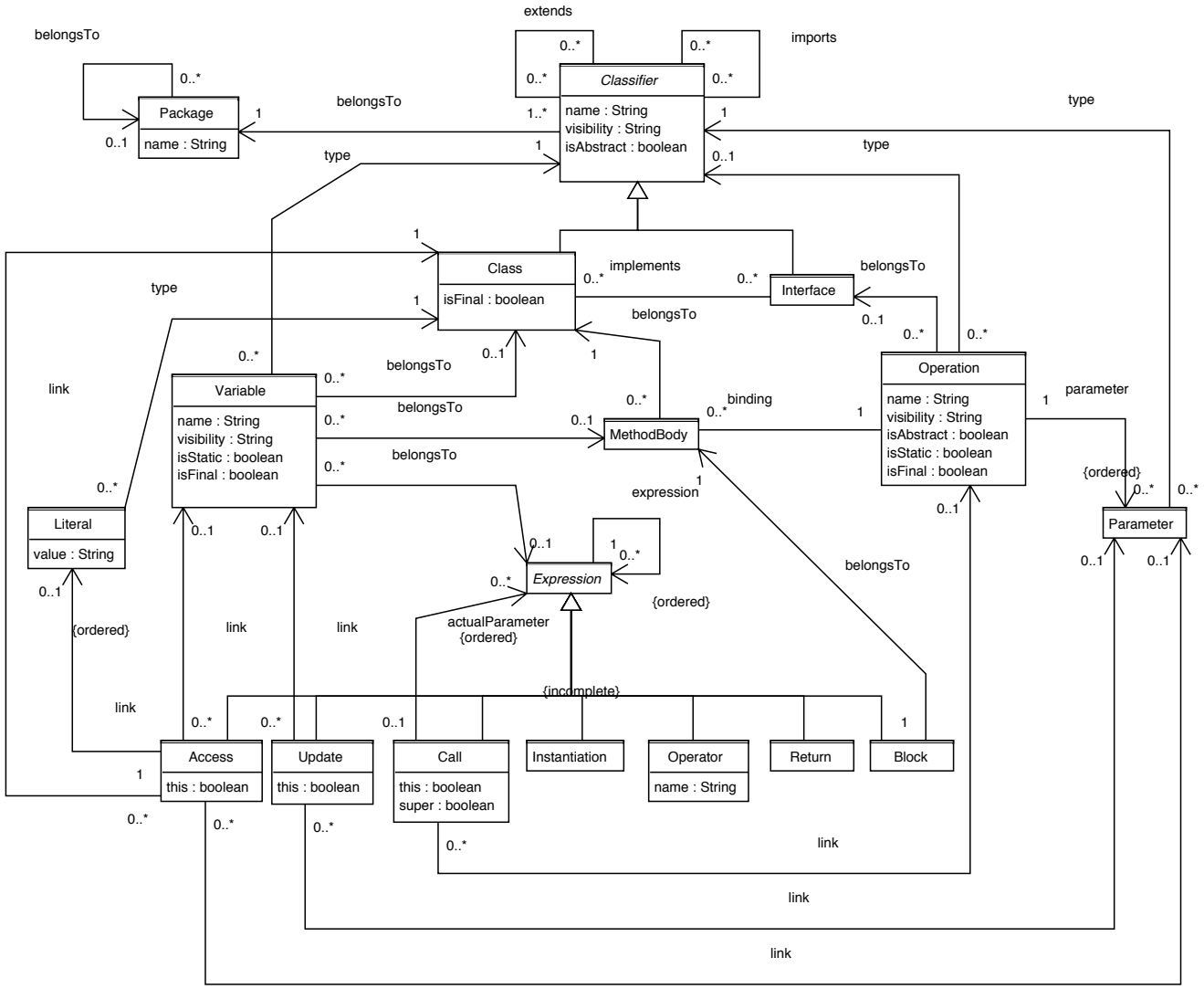


Figure 3. Meta-model for graph representation of Java programs

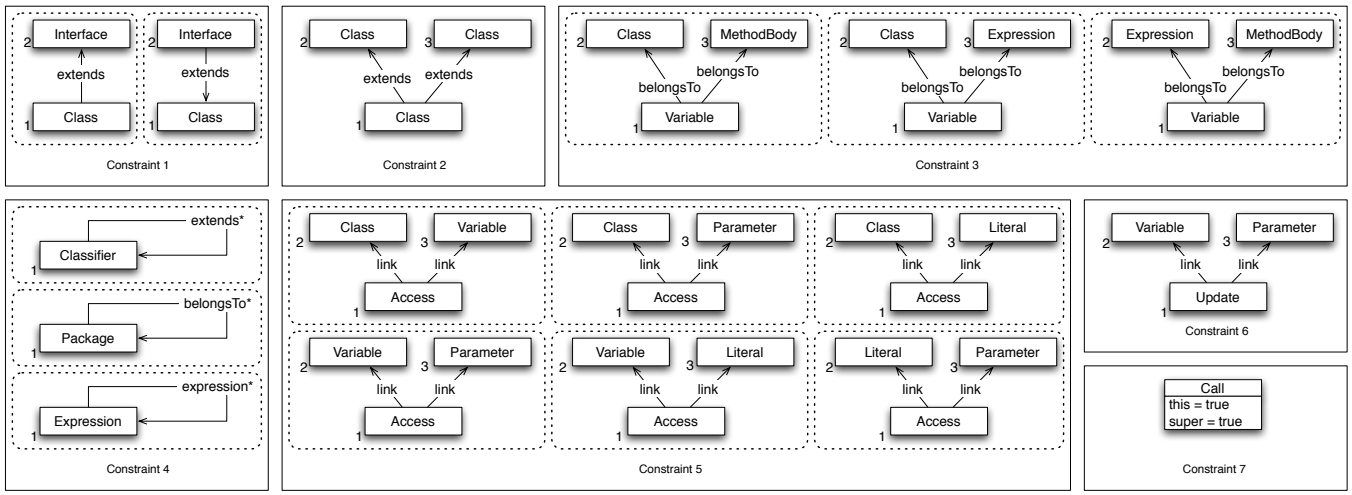


Figure 4. Forbidden subgraphs that represent additional constraints to the Java Program Graphs meta-model. Contraints 1, 3, 4 and 5 are composed of multiple forbidden subgraphs (separated by dotted lines), indicating that each of these subgraph patterns is not allowed to occur in a well-formed Java graph model.

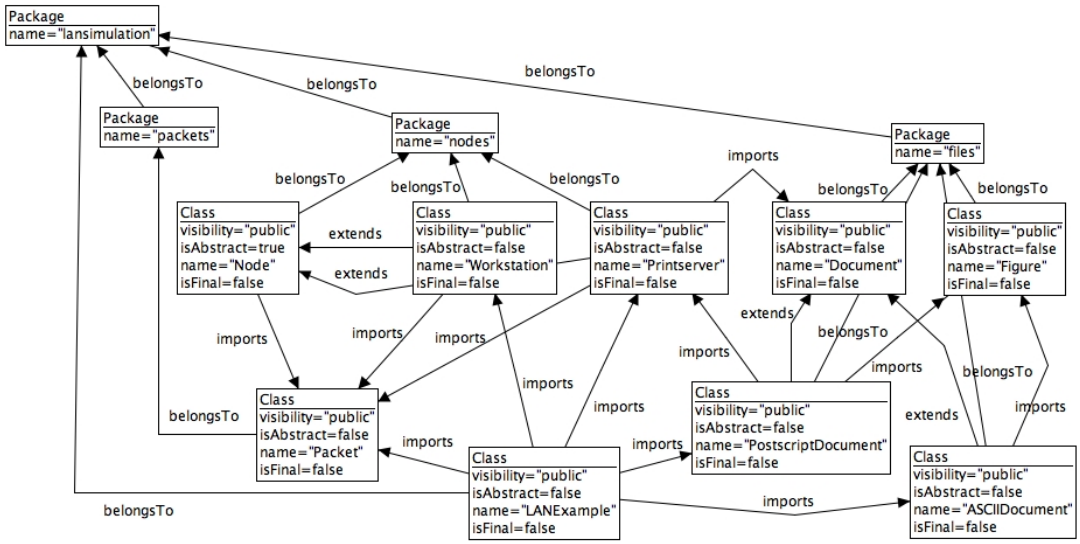


Figure 5. A graph example representing packages' and classes' nodes, corresponding to the LAN system structure of Figure 1.

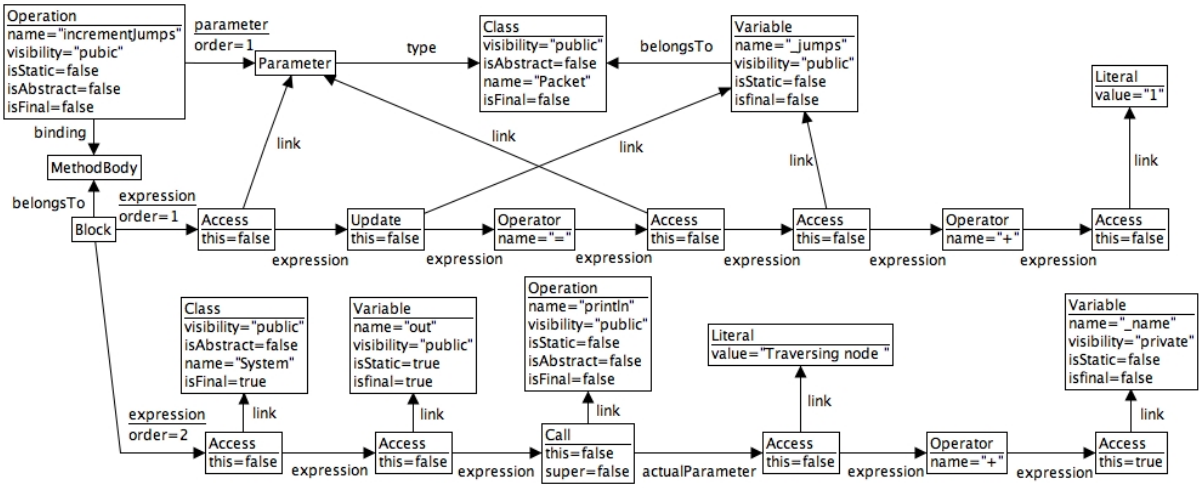


Figure 6. A graph example representing the incrementJumps method from class Node.

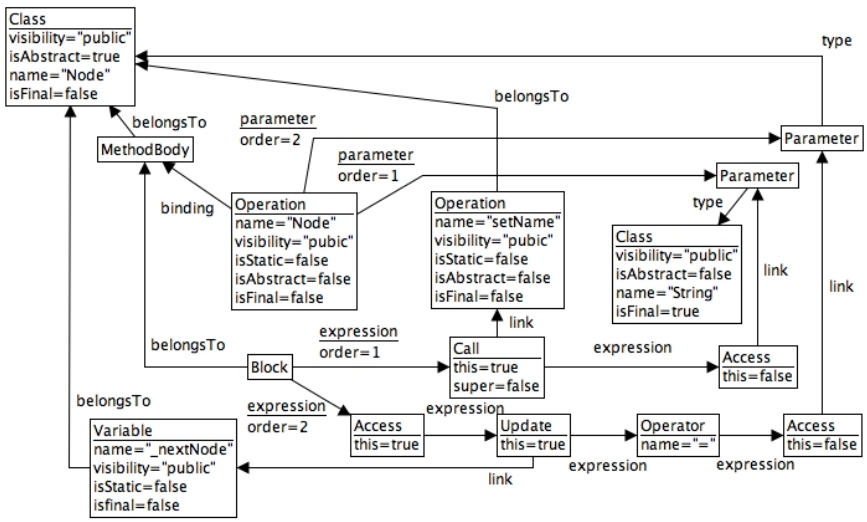


Figure 7. A graph example representing a constructor method from class 'Node'.

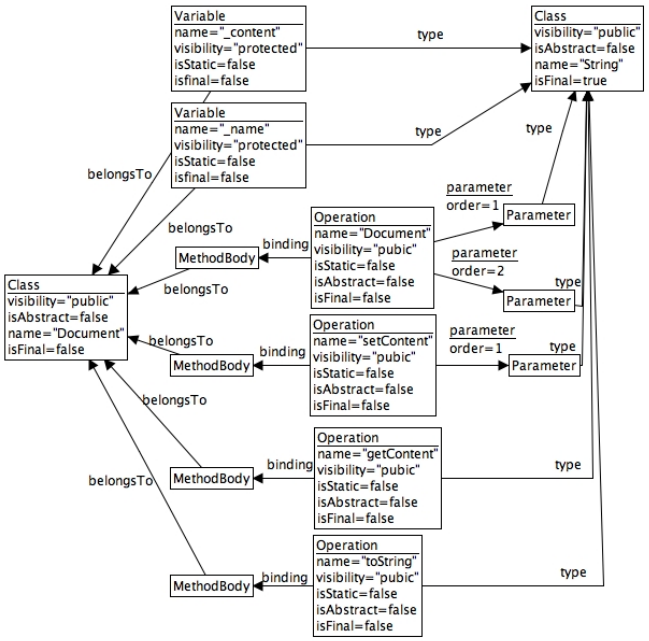


Figure 8. Graph representation of the Document class

that it is a constructor method. The name convention for constructor methods is observed. Both examples also differ in the number of arguments, for which order labels in edges are shown. The incrementJumps method includes feature access through a reference to a separate object (reference p in the example). The constructor method includes two different manners of self references, explicit, using this (this._nextNode in the example), and implicit, call without an explicit receiver (setName(name) in the example).

5 Precise refactoring specifications

We provide definitions of a small subset of three carefully selected 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. However, solutions to this *Case* can choose to implement another particular refactoring, if they consider it is needed to demonstrate a specific interesting feature of a certain graph transformation tool.

To keep the *Case* feasible, it consists of implementing only the following three refactorings: *EncapsulateField*, *MoveMethod* and *PullUpMethod*. 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, in order to better fulfill the *Case* objectives than the originals. Both the Eclipse originals and our own specifications should be taken as initial guidelines by implementors of solutions. Implementors of the *Case* are invited to further extend their refactoring implementations to approximate the Eclipse variant as closely as possible, in order to come to a "realistic" solution. Implementors of the *Case* are also invited to implement other refactorings, such as *Push Down Method*, *Inline Method*, *Rename Method* and so on.

5.1 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:

- The visibility of a variable field is turned private (i.e., the variable is encapsulated) and getter and setter methods are created to access it, if necessary.
- All references to the encapsulated field (excluding those within the container class) are updated to use the accessor methods.
- If the useAccessorsAlways option is set to true, references to the variable within the container class will be transformed into calls to accessor methods 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, the 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.
- If the variable is a static field, the newly created methods must be specified with the static modifier when creating non-existing accessors.
- Eclipse offers two more options: one to specify where to insert the newly created methods, within the source code text, and another to generate method comments. None of these options is interesting for the *Case* as they do not impact the program graph.

5.2 Move method

operation:

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

preconditions:

- The target class should be reachable from the code that calls the moved method. Eclipse implements this by applying 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 of the same type as the target class.

– The method to be moved 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 assume that methods that cause name conflicts cannot be moved [5].

mechanics:

– The corresponding method body is moved from the source class to the 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 to true, the method is copied to the target class and the original method body in the source class is turned into a delegating call to the new method. Updating of references is not needed.
– To update references:
  – if the moved method is going to be accessed through an instance variable of the source class, then those calls to the moved method such as: referenceToSourceClass.method(p)) will have to be substituted, to access that instance variable, by an expression similar to this: referenceToSourceClass.instanceVariableOfTargetsType.method(p).
  – if one of the method's parameters, say p, which belongs to the type of the target class, is going to be used to access the moved method, 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 from referenceToTargetClass.member() to simply member().
  – If the method being moved accesses instance members from the source class, a reference to the source class must be available from within the method. Therefore, a parameter, say 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 tags the old method as deprecated. This particular behavior does not need to be implemented for the *Case*.
  – 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 changed to public in a chained way.

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

5.3 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 can be pulled up.
– The target class must be a superclass of the source class.
– A method with the same signature cannot 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 pulled-up method must be accessible for the method in its new location (superclass).
– All occurrences of the method to 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: The pulled up method is moved to the 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 with a multiple choice dialog where the user can select which methods have to be kept and which ones have to be deleted. For this *Case*, the *System* is not required to provide this particular GUI, but a way to define the sets of methods to be kept and deleted has to be provided.
– 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.

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

5.4 Refactoring the LAN example

As mentioned before, the additional downloadable material includes 9 snapshots of the LAN example. Each snapshot cor-

responds to the application of a single step from the following refactoring sequence:

0. Original version of the LAN simulation system
1. EncapsulateField(Node, _name, getName, setName, true)
2. EncapsulateField(Node, _nextNode, getNextNode, setNextNode, true)
3. EncapsulateField(Printserver, lastDocument, getLastDocument, setLastDocument, false); option: private → protected
4. MoveMethod(Printserver, Document, formatDocument, false)
5. MoveMethod(Node, Packet, incrementJumps, false)
6. EncapsulateField(Packet, _jumps, getJumps, setJumps, true)
7. PullUpMethod(PostscriptDocument, Document, setName, false, {})
8. PullUpMethod(PostscriptDocument, Document, getName, false, {})
9. PullUpMethod(PostscriptDocument, Document, includeFigure, true, {PostscriptDocument, ASCIIDocument})

This sequence of refactorings we suggest covers most of the options, and it tests most of the features, of the refactoring specifications given. It can also be used as a reference sequence. Once this refactoring sequence is completed (according to the refactoring specification provided in this section), the resulting classes are shown in Figure 9. In order to properly compare the *Tools*, the *System* shall reproduce this sequence, obtaining the same results. The detailed effect of each refactoring, each resulting source code snapshot, can be found in the downloadable materials.

6 Evaluation Criteria

In the context of model refactoring, the *Case* aims at measuring many different criteria, some practically oriented, others more theoretically oriented:

Expressiveness: As was introduced in Section 3, 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. To assess the expressiveness of the used mechanisms, the following questions may be used as a guideline:

- How "self-explanatory" is the specification of the refactorings?
- Which graph transformation language concepts help to specify the refactorings proposed? Which control constructs are needed?
- Can primitive refactoring operations be reused to build other refactorings?
- Can refactorings be composed to build larger refactorings?
- Can refactorings be parameterized?

Usability and understandability: To assess this criterion, we can ask the following questions:

- 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.
- How can refactoring selection, parameterization, and application be triggered by the user? Can the user interact with the refactoring process during its application?
- Is the *System* robust against malicious or accidentally incorrect user input?
- Does the *System* provide any help to indicate the place(s) *where* a chosen refactoring could be applied?
- Does the *System* provide any support 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? (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? Both questions are important, in order to make the refactoring specifications less dependent from the chosen graph metamodel, and hence become more robust to evolution of this metamodel.

Extensibility: How easy is it do add new refactorings in the *Tool*? (a) For a graph transformation expert; (b) for a novice user. In order to check for extensibility, solution providers shall consider to specify one or several other refactoring operations.

Performance: This relates to questions pertaining memory consumption and speed of execution:

- How much memory is used to specify graph models, refactorings, and the application of refactorings?
- How long does the importing and exporting of data take?
- How fast is the verification of refactoring preconditions and the actual application of the refactoring?

Formal properties: Which formal properties of the tool can be exploited to reason about the model refactorings, and to ensure their correctness, consistency and behaviour preservation? Specific questions related to formal properties are:

- Are all decidable preconditions of a refactoring checked before it is applied?
- How does the *System* handle preconditions that are undecidable? An example of this is the condition stating that the method bodies to be "pulled up" shall be equivalent. This condition can not be proved or disproved.
- Does the *System* guarantee that the refactored graph does again conform to all constraints of the model, and that it preserves behaviour?
- How can postconditions be expressed? Due to the fact that these may depend on input provided by the users during the refactoring, they may not necessarily be convertible into an equivalent precondition.

The developers of the *System* could consider to apply some of the specified refactoring operations to another, maybe bigger program graph, conforming to the same GXL meta-model, in order to check for generality of the refactorings, and for scalability of the *System*.

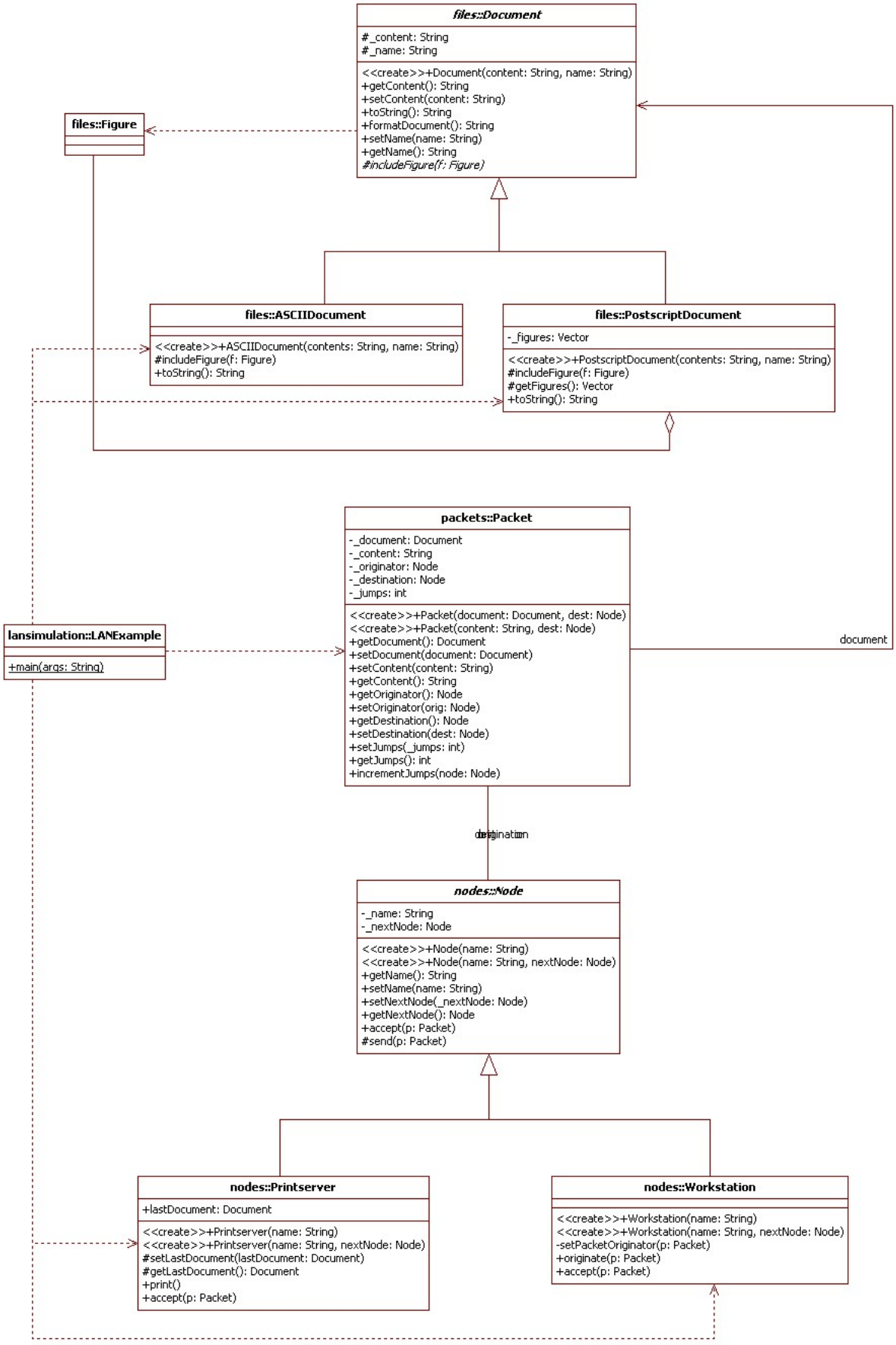


Figure 9. The resulting LAN example after having applied the refactoring sequence.

Evaluation matrix for the Refactoring case study							
	Feature	Approach					
		The *Tool*, *e.g.*	VMTS	MoTMoT	Fujaba	AGG	PROGRES
General	GT language						
	Import/Export format						
	Changes to metamodel						
	User interaction						
	Embedded code						
	External libraries						
	Rule composition						
	Visualization						
Completeness	Encapsulate field						
	Move Method						
	Pull Up Method						
Performance	Memory Consumption						
	Speed: Refactoring						
	Speed: Import/Export						
	Speed: Visualization						
Appeal	Understandability						
	Conciseness						

Scale for the above criteria	
-2	Bad
-1	Meager
0	Average
1	Good
2	Excellent

Table 1. Features to be used in order to evaluate *Tools* according to this case study.

As already said, this *Case* was proposed for GraBaTs 2008. In addition to the evaluation criteria presented in this article, Contest organizers proposed the evaluation matrix shown in Table 1. This table includes columns for a generic *Tool* and for the *Tools* presented in the live contest. [6]

7 SWOT analysis

This article proposed a case study where the task specified can be considered of medium size, but quite demanding (for tools and their users). The *Case* is representative of a quite popular application area – program refactoring.

To conclude, we discuss the *Case* using a SWOT (Strengths, Weaknesses, Opportunities, and Threats) analysis:

- **Objective:** The specific subject to evaluate, and the rationale behind this.
- **Strengths:** Positive internal attributes that benefit the goal and are within the authors' control.
- **Weaknesses:** Factors that negatively affect the achievement of the objective and are beyond the authors' control. They can be understood as points that can be improved in the future.
- **Opportunities:** External factors beyond the authors' control that could be helpful to achieving the objective.
- **Threats:** External factors beyond the authors' control that are harmful to the goal if they occur.

The usual format to document this analysis is by filling in the Strengths, Weaknesses, Opportunities, and Threat cells, within a 2x2 matrix We compile and discuss these points in the following sections and we summarise them in Table 2.

7.1 Objective

The goal of the *Case* is to provide a framework to evaluate and to compare graph transformation tools. Specifically, it is focused on the evaluation of:

- the **expressiveness** of the tool and its graph language in regards of rule specification, and its features for using complex rule execution control;
- the **extensibility** of the tool and its language in terms of support for reuse and composition in rule specification;
- the **usability and understandability** of the tool (also called *appeal)*, especially for non graph-transformation-savvy users. This can be expressed in terms of conciseness, ease of use, and suport for user interaction (*e.g.*, to allow user guidance during the transformation); process;
- the **genericity** of the developed refactoring rules, and more precisely whether they can be used or easily adapted for other program graph meta-models;
- **performance requirements** of the tool;
- **formal support** for analysis of rule properties;

[6] The solutions submitted to GraBaTs 2008 for this *Case* can be found in `http://www.fots.ua.ac.be/events/grabats2008/solutions.html`, and the result of the live contest can be obtained from `http://www.fots.ua.ac.be/events/grabats2008/awards.html`.

7.2 Strengths

The relations between graph transformation and refactorings have already been stated. The source code of a program has a straightforward graph representation as an abstract syntax tree (AST). Refactoring operations can be understood as structural transformations of an AST.

The *Case* used an *ad-hoc* AST-like meta-model that can be used as a reference meta-model, against which each solution can be compared and evaluated. The use of this simplified AST forces all the solution providers to follow a common graph representation with a level of detail that is sufficiently close to the JAVA source code,

Refactoring operations have enough complexity to stress test the characteristics of many graph transformation tools, such as expressiveness, extensibility, performance, genericity and usability.

Preconditions and mechanics of refactoring operations are rather complex, so the expressiveness of a graph transformation tool and a graph transformation language is a key attribute in order to develop a proper solution.

Refactorings vary in their complexity, and thus, their mechanics are implemented by simple or rather intrincated algorithms. The *Case* tests the tool's support for complex control of transformation rules. A graph transformation tool with the ability to specify control over the execution of transformation rules will undoubtably allow more satisfactory solutions than tools or approaches that do not feature this ability.

The semantics of refactorings have been extracted from their implementation in the ECLIPSE development tool. Therefore, the correct semantics of a refactoring can be verified easily by executing it with ECLIPSE over real source code or the toy examples given.

A downloadable set of source code examples is given, so the different solutions can be evaluated and compared against the same reference example. The proposed reference example (the LAN example) is considered as a good case study, accepted by both the software evolution community and the graph transformation community [3,6].

7.3 Weaknesses

The particular chosen subset of the JAVA language, meta-model representation and selected refactorings proposed in the *Case*, can favour some *Tools* against others. More precisely, the proposal of a specific meta-model for JAVA program graphs, could favour the solutions developed with some graph transformation tools and graph language formalisms against others. Perhaps, a more open proposal could allow the developers to submit a solution fulfilling the *Case* requirements, while using a meta-model more adequate to each specific graph transformation tool.

Refactorings, while referring to the same transformation kind, vary greatly in their preconditions and mechanics. The results of any graph transformation tool comparison may be affected by the choice of a particular set of refactorings. The refactorings selected do not include opportunities, for example, to evaluate amalgamation features, such as copying subgraphs [16]. Some refactorings, such as *Push Down Method*, would offer these opportunities. Support for meta-rules, subgraphs, graph shapes, etc. is an interesting feature to this domain, since those tools implementing it will produce more understandable and concise graphs and rules.

Another concern regarding the refactoring specifications is the abscence of postconditions in them. The addition of postconditions would have make the *Case* more complete and would allow testing more features of graph transformation tools. Preconditions and mechanics of the ECLIPSE refactorings can be extracted from the definitions in [4], and by testing ECLIPSE itself or examining its source code. Unfortunately, postconditions are not available from these sources.

The complexity and huge size of the graphs involved in the *Case* makes it difficult to compare the graphs produced by the different *Tools*. In future refinements or revisions, the *Case* should advise to check certain small parts of the graphs for evaluation and *Tool* comparison purposes.

One evident weakness of the *Case* is the lack of a tool to generate the GXL representation of a given JAVA program. Since the development of such converter is not the objective of the *Case*, the solution providers will have to rely just on the toy example provided.

The given meta-model, which is an *ad hoc* representation of JAVA programs, can be misunderstood by the solution providers. A more standard meta-model, such as an EMF model of the JAVA AST, could have been proposed to tackle this problem.

The interactiveness of the solution can be more dependent of the graph transformation tool used than the particular implementation developed by the solution provider.

7.4 Threats

The lack of a tool to convert JAVA programs to the GXL meta-model specification, can discourage a solution provider to develop and debug a solution to the *Case*, or to apply the *Case* to other programs than the ones that were provided.

Solution providers may choose to focus on different refactorings, or different variants of the same refactoring, which could make it difficult to compare the different tools and solutions.

The performance of the solutions may depend on the platform/machine on which the *Tool* and *Case* are deployed. The results can be different if we use another machine configuration (e.g. CPU speed, memory configuration, ...), or even another operating system. This is a threat that needs to be taken into account when doing the comparison.

Also with respect to *Tool* comparison, the solution providers may provide graph examples of many different types in terms of size and the subgraphs they focus on. This will make it difficult to compare the different solutions.

	Positive	Negative
Intermal	**Strengths** – Formalisation of refactorings as graph transformations is a well-studied problem. – A common AST-like meta-model for graph representation of JAVA source code is given. – Code refactorings are sufficiently complex (they are not "toy examples"). – Each refactoring implies a different subproblem with different characteristics and degree of complexity. – Possibility to evaluate tool support for complex control of transformation rules. – The refactoring semantics have been extracted from the refactorings implemented in the Eclipse IDE. – A downloadable set of source code examples is given.	**Weaknesses** – The chosen programming language, meta-model and set of refactorings can favour certain tools against others. – The chosen set of refactorings may don't offer oportunities to evaluate every graph transformation tool feature –*i.e.* subgraph copying–. – Complexity and size of the graphs involved. – The abscence of postconditions. – Lack of a tool to convert JAVA source code to JAVA program graphs. – The proper instantiation of the meta-model can be misunderstood. – The interactiveness of the solutions is highly dependent of the graph transformation tools used.
External	**Opportunities** – The *Case* is suitable to future refinement and revision. – It can serve as a basis for a benchmark of similar case studies. – It specifies a framework to test different aspects of graph transformation tools. – It can be revisited easily, by changing the meta-model or the set of refactorings selected. – It is open enough to allow the solution providers to focus on different aspects of their solutions. – It can be used to test the scalability of graph transformation tools. – It can be used to demonstrate practical application of graph transformation tools; – It can be used to assess the formal reasoning ability of graph-transformation-based *Tools*.	**Threats** – The lack of a source code to graph converter may discourage a solution provider to develop a solution for the *Case*. – Solution providers may decide to focus on different (variants of) refactorings, making it difficult to compare their solutions. – Solution providers may focus on different subgraphs and parts of their solutions, making it difficult to compare them. – Solution providers may use different platform/machine, making it difficult to compare the performance of their solutions.

Table 2. SWOT analysis of the case study.

7.5 Opportunities

The *Case* can set the basis for a test benchmark for graph transformations tools. It is obviously open for further refinement and revision. Different programming languages, different meta-models, and different refactorings can lead to evaluate many different aspects of graph transformation tools and languages. Future revisions of this *Case* can use it as a template and can ask for different refactorings. Authors of graph transformation tools can develop solutions for other refactorings, for which their tools fit best, and thus show their features and capabilities within a common framework. However, the *Case* is sufficiently open to allow the solution providers to focus on those aspects in which their tool or solutions can perform best.

Although not explicitly stated, the *Case* can also be used to test the scalability of graph transformation tools, specially in regards of subgraph matching, because the program graphs involved in the process can easily contain millions of nodes and edges.

While it is given as an optional requirement, the implementors have the chance to demonstrate how their graph transformation tools can provide practical application of graph transformation theory.

A formal approach to source code refactoring can assess the ability to reason about formal properties of refactorings such as parallel and sequential dependencies between refactorings, termination properties, and many more.

Acknowledgements. Javier Pérez and Yania Crespo have been partially funded by the regional government of *Castilla y León* (project VA-018A07) and by the spanish government (*Ministerio de Ciencia e Innovación*, project TIN2008-05675). Tom Mens is partially funded by the *Ministère de la Communautée française - Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique* (Action de Recherche Concertée AUWB-08/12-UMH).

The authors thank all those who have contributed to this case study: the organisers of GraBaTs 2008, for which this case study was originally developed, the participants that submitted solutions for it, and last but not least Pieter Van Gorp and Erhard Weinell.

References

1. E. Biermann, K. Ehrig, C. Köhler, G. Taentzer, and E. Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 219–257. Springer, 2006.
2. P. Bottoni, F. Parisi-Presicce, G. Mason, and G. 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. S. Demeyer, F. Van Rysselberghe, T. Girba, J. Ratzinger, R. Marinescu, T. Mens, B. Du Bois, D.Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall, and M. El-Ramly. The LAN-simulation: A refactoring teaching example. *Principles of Software Evolution, Int'l Workshop on*, 0:123–134, 2005.
4. M. Fowler. *Refactoring—Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, Reading, MA, 1999.
5. L. Grunske, L. Geiger, A. Zündorf, N. Van Eetvelde, P. Van Gorp, and D. 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.
6. D. Janssens, S. Demeyer, and T. Mens. Case study: Simulation of a LAN. *Electr. Notes Theor. Comput. Sci.*, 72(4), 2003.
7. G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51, 2004. Special issue on Program Transformation, edited by Ralf Lämmel, ISSN: 0167-6423, digital object identifier http://dx.doi.org/10.1016/j.scico.2004.03.002.
8. T. Mens. On the use of graph transformations for model refactoring. In J. Visser R. Lämmel, J. Saraiva, editor, *Generative and transformational techniques in software engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 219–257. Springer, 2006.
9. T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, pages 269–285, September 2007.
10. T. Mens, G. Taentzer, and O. Runge. Analyzing refactoring dependencies using graph transformation. *Journal on Software and Systems Modeling*, 2007. To appear.
11. T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.
12. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
13. J. Pérez and Y. Crespo. Exploring a method to detect behaviour-preserving evolution using graph transformation. In *Proceedings of the Third Int'l ERCIM Workshop on Software Evolution*, pages 114–122. ERCIM, October 2007. Informal Workshop proceedings.
14. J. Pérez and Y. Crespo. Perspectives on automated correction of bad smells. In *IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 99–108, New York, NY, USA, 2009. ACM.
15. Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, László Lengyel, Tihamér Levendovszky, Ulrike Prange, Dániel Varró, , and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, October 2005.
16. N. Van Eetvelde and D. Janssens. Extending graph rewriting for refactoring. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *2nd Int'l Conf. Graph Transformation (ICGT'04)*, number 3256 in Lecture Notes in Computer Science, pages 399–415. Springer, 2004.
17. 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.