# Overview of the Refactoring Discovering Problem

Javier Pérez

Universidad de Valladolid
jperez@infor.uva.es

**Abstract.** Refactoring support is currently aimed at automatically finding and applying individual transformation steps. Some techniques exist to suggest wider design changes that can be made to a system in order to improve certain design characteristics. Therefore, the sequence of refactorings needed to perform the proposed changes must be found by the developer. Support for complex refactoring sequences generation and execution could help on high level software systems redesigning, allowing the inclusion of very high level refactorings as a regular technique in the software development process. This paper presents which formalisms, techniques and tools we are exploring to address this problem, and shows the conclusions reached so far.

## 1 Introduction

Current approaches to refactoring automation [1] are focused on finding and applying individual transformation steps. When a refactoring opportunity has been detected, development tools are used to automatically apply the refactoring and test its effects.

This refactoring opportunity detection usually deals with a single transformation at a time. Some techniques exist to suggest wider design changes that can be made to a system in order to improve certain design characteristics. Formal Concept Analysis can be used as in [2] to propose hierarchy reorganisation. Metrics can help detecting Bad Smells [3] so that the developer can remove them. In these cases, the desirable change is automatically sketched, but to achieve it, just some guidelines are provided. The sequence of refactorings needed to perform the proposed change must be found by the developer.

Therefore automatic support to systematically build these refactoring sequences is needed to fully automatise the process of improving software system design. So, given a change proposal to improve a system's design, this work aims at providing automatic support for planning the refactoring sequence which could make that change effective.

This paper presents which formalisms, techniques and tools we are exploring to address this problem, and shows the conclusions reached so far. The paper is organised in the following manner. In Section 2, the main problem, its subproblems and the research strategies are presented. In Section 3, a software system which will be used to extract examples from it and as a case study, is presented. Later on, in Section 4, we introduce the underlying formalisms we are exploring to model the problem. We introduce how programs and refactorings can be represented within the chosen formalism in Section 5, and the refactoring discovering algorithm is overviewed in Section 6. Conclusions and future work can be found in Section 7.

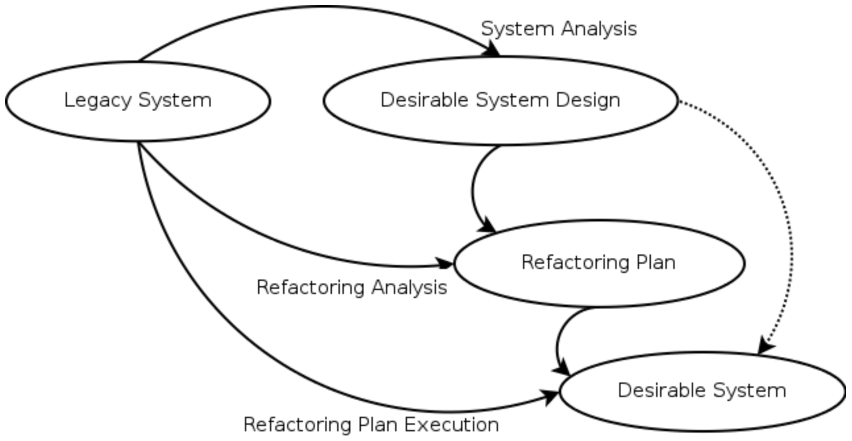

**Fig. 1.** Overview of the Refactoring Plan Problem

## 2 Refactoring Plans: Problem Description

Refactorings [4, 5] are structural transformations that can be applied to a software system to perform design changes without modifying its behaviour. So, running refactoring transformations should obey high level design modification proposals. Efforts to include refactorings as a regular technique in the software development process, have led to refactoring support being commonly integrated into development environments. Refactorings are offered to the programmer as small transformations which can be launched over the source code. The programmer is still responsible of translating a design changing proposal, such us reorganising an inheritance hierarchy, into the appropiate set of refactorings that fulfils that proposal. Another automatisation step has to be taken to help on high level redesigning. It is not only required to support single refactoring operations within development tools, but to automatise planning and execution of complex refactoring sequences.

### 2.1 Defining Refactoring Plans

We pretend to introduce a new concept, which we have named "Refactoring Plan", and which can be described in the following terms:

**Definition 1.** *A “Refactoring Plan” will be an specification of a refactoring sequence which matches a system redesign proposal, so that it can be automatically executed to modify the system in order to obtain that desirable system re design without changing the system’s behaviour.*

The research presented here aims at supporting Refactoring Plans, allowing its inclusion as a regular technique in the software development process (Fig. 1). A set of formalisms, techniques and tools providing automatic generation of refactoring plans will be our main research results, and for that, results need to answer the following questions.

*Question 1.* Given a software system as the source of a transformation, a redesign proposal as the target, and a set of refactorings that can be used as transformation operations:

1. Does a refactoring plan, which transforms the source into the target using the provided refactorings, exist?
2. when a refactoring plan exists, can it be generated and executed automatically?

### 2.2 Subproblems and Research Strategy

Addressing the automatic generation of refactoring plans problem involves solving the following subproblems:

– Definition and formalisation of the “Refactoring Plan” concept.
– Representation and formalisation of Software and Refactorings.
– Elaboration of a refactoring discovering algorithm.
– Validation of proposed subproblem solutions through existing tools and prototype implementation.

In addition to that, the research has been planned to progress through two stages. Differences between both stages are on the type of description of the transformation target and the information amount it provides about the desirable software system. In both stages, the source description will be the current system source code.

For the first stage, the target will be the desirable system source code, assuming the refactoring plan has already been applied. Goals of the first stage are proposing and validating a solution to the automatic generation of refactoring plans problem. Results of this stage can be applied to problems where the need is not to perform the system transformation, already done, but testing if a refactoring sequence exists, so the target system preserves the behaviour of the original system.

Once the first stage is over, the second stage will focus on how much information can be removed from the target system, so the refactoring plan can still be computed. Extensions and improvements of the developed solutions will be needed. Goals of the second stage are defining the kind of descriptions that can be used as targets of a transformation. Application of second stage results will be providing full support for automatic generation of refactoring plans for available types of target descriptions.

While in [6] an approach to the refactoring problem is presented, it is just addressed to deal with situations covered by the first stage. In the following sections we will present researching being done within the first stage too.

## 3 Case Study: a “Lazy Class” in the JFreeChart Project

A real system has been chosen to be used as the case study to test researching directions. This software system is the project **JFreeChart**, hosted in **www.sourceforge.net** In [3] this project has already been used, validating metrics to detect Bad Smells. The example focus on a class that suffers from the Bad Smell “Lazy Class”, which has been discovered through metrics analysis. The Bad Smell “Lazy Class”, identifies a class which is not holding enough responsabilities so it must be deleted and its elements must been moved to another class. It is suggested to use refactorings such us “move method”, “remove class” and “collapse hierarchy”.

This case study demonstrates the need for an automatic generation of refactoring plan support. Despite of the simple change proposed, the refactoring plan needed to accomplish it takes 16 refactoring steps and 7 different refactoring types.

## 4 Problem Formalisation

Intrinsic characteristics of refactorings include modification of the system structure and behaviour preserving transformations. An underlying formalism must be chosen, so that it focuses on description and manipulation of structural information, and so that certain condition checking can be performed too when modelling transformations. Graph transformation ([7, 8]) is the general formalism that has been selected. In [6] an approach to the refactoring problem is presented which directly uses the program abstract syntax tree as the software representation mean.

Previous works exist which make use of graph transformation for refactoring formalisation. In [9, 10] the graph transformation approach is shown to be valid for refactoring formalisation. Programs and refactorings are represented with graphs in a language independent way, using a version of abstract syntax trees with the appropiate expressiveness level needed for the problem. This kind of representation, called **Program Graphs**, will be used as the basis for developing a specific graph representation for this research.

Two main directions can be found within the field of graph transformations, that can be useful addressing the refactoring discovering problem: Rule Driven Systems and Programmed Graph Rewriting Systems.

Other approaches in refactoring formalisation exists. The work presented in [11, 12] relies on first order logic formalisation. In this work, refactorings are described in terms of preconditions, postconditions and transformation relationships between them. It is specially interesting because this kind of descriptions could also be used in a graph rewriting approach.

### 4.1 Rule Driven Systems

Rule Driven Systems are graph rewriting systems where transformations are executed following the derivation sequences stated by graph grammar rules. Rules are randomly selected and a graph grammar controls the transformation process.

An early approach to the refactoring plan generation problem based on graph grammars was presented in [13]. The problem was modeled as a formal language problem. Checking whether the target system could be derived or not from the source system using the available set of defined refactorings, was compared to the membership problem of formal language theory. The idea was to take advantage of the graph parsing capabilities of the graph grammar systems.

This approach deals well with visual languages parsing [14]. In these problems, the graph rules generate graphs, belonging to the language defined by a graph grammar, from an initial state or node. Derivation paths are well defined. But, for the refactoring discovering problem, an exhaustive refactoring analysis must be done to state things like: *"To remove Bad Smell P, a sequence of the kind R1 R2+ R3* (R4|R5) is always executed."*, so that a graph grammar with a well known derivation tree can be formulated. Given that, over a software system, the number of different instances of different refactorings which can be applied for each derivation step is almost unpredictable, another graph grammar direction was searched.

The AGG tool [15, 16], representing this kind of graph transformation approach, was used in the experiments. AGG is a graph grammar tool that supports graph grammar concepts needed for the refactoring plan generation problem, such as typed attributed graph grammars ([17, 18]), negative application conditions, graph parsing, and rule layering. Rule layering allows for a basic flow control definition for rule execution.

### 4.2 Programmed Graph Rewriting Systems

Programmed Graph Rewriting Systems present a more general graph rewriting approach. Structured programming, algorithms, etc.can be use in addition to the definition of a graph grammar. The General graph rewriting tool PROGRES [19, 20], will be used in the forthcoming experiments, because it offers more control over the rule selection and application algorithm and more expressiveness than the grammar based systems. Within this approach, the refactoring plan generation problem can be modeled as a state space search problem. The transformation source can be seen as the start state and the target as the final state. Refactorings are treated as state changing operations. The problem here is the combinatorial explosion, which must be faced with heuristics.

## 5 Software and Refactoring Representation

To represent software systems and refactorings, the graph of [9] has been chosen. To formalise refactorings, a formal graph representation of Object-Oriented (OO) software is needed. It must support elements of OO paradigm (classes, fields, methods, ...) and their structural relationships. In order to represent refactorings below method signature level, also method bodies must be represented. As mentioned in Sec. 4, the formalism presented in [9] presents a graph rewriting system based on directed typed graphs which is already aimed at formalisation of refactorings. This formalism defines a representation of refactorings as graph transformation rules. As an important feature related to refactorings, it is designed to allow representation of behaviour preserving invariants too. Syntactic correctness of the generated graphs are assured with the use of constraints and well formedness rules. To allow representation of refactoring transformations which must appear in a specific context to be performed, it supports the specification of context-sensitive rules through the definition of negative application conditions, node embeddings and the use of path expressions.

We have develop an extension to Program Graphs called **Java Program Graphs**, which includes Java concepts such as visibility, interfaces and packages, in order to allow running experiments for real systems such as the case study presented in Sec. 3. Despite of the formalism [9] is designed for graph grammar systems, representation ideas can be transferred to a programmed graph rewriting modelisation of the problem.

## 6 Refactoring Rules and Refactoring Discovering Algorithm

We are describing programmed graph refactoring rules in a similar way refactorings are defined in [11, 12]. The programmed graph rewriting approach allows to build complex refactoring descriptions. Structured programming can be used for example,to define *if...then...else* conditions, and it makes possible to describe complex refactorings that takes more than one step to be performed. With this approach, refactorings can be decomposed into preconditions, postconditions, transformations and transformation relationships between pre and postconditions. Therefore, more expressiveness and execution control than the graph grammar only approach allows to define a more guided algorithm to find the source-target transformation path.

To address the refactoring plan problem with a programmed graph rewriting approach, a basic state space algorithm is being developed. This definition provides an overview of the refactoring discovering process.

**Definition 2.** *Basic refactoring discovering algorithm. Iteratively, refactorings are being selected and applied to the source graph until it is transformed into the target graph. At the end of each iteration, source and target graphs are checked to test whether they are isomorphic or not.*

*Refactorings which their precondition sets don't hold on the source graph will never be selected. Refactorings which their postcondition sets don't hold on the target graph, can be selected but their postconditions that not hold will be kept as awaiting postconditions. Refactorings that make a awaiting postconditions no longer being held on the source graph, will be selected preferably and will remove that postconditions from the awaiting list when applied.*

*The refactoring plan will be found if and only if, the graphs become isomorphic, with no postcondition to be held on the target graph.*

The first heuristic that has been formulated is based on refactoring postconditions and prioritises the selection of refactorings which postconditions hold on the target graph. Refactorings which make that previous selected refactorings could hold their awaiting postconditions, are given priority too in each iteration selection process. Results from Demeyer's work ([21]) where metrics are used to detect quantitative changes that refactorings produce are being explored to develop complementary heuristics.

## 7 Conclusions and Future Work

This work presents the initial state to support the automated elaboration of Refactoring Plans. The main requirements of the problem and the directions to explore have been stated. The case study that will be used to explore the initial proposals has been introduced It has been found that graph transformations offer good mechanisms to be used as the underlying formalism, specifically the programmed graph rewriting approach. A format for representing Java programs has been developed. We have defined also, a mean for specification of refactorings as programmed graph rewriting rules in terms of preconditions, postconditions and transformation rules. Finally, we have introduced a general overview of our refactoring discovering algorithm.

Main future tasks will be directed to:

– Further definition of the “Refactoring Plan” concept
– Extend the available set of refactorings.
– Validate proposals implementing the algorithm in the PROGRES tool.
– Analyse termination and correctness conditions of the refactoring discovering algorithm.
– Analyse, improve and extend heuristics of the state space search algorithm.
– Test another approaches such as the first order logic refactoring framework.

## References

1. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Transactions on Software Engineering **30** (2004) 126–139
2. Prieto, F., Crespo, Y., Marqués Corral, J.M., Laguna, M.A.: Applying formal concepts analysis to the construction and evolution of domain frameworks. In: 6th International Workshop on Principles of Software Evolution (IWPSE 2003). Helsinki, Finland.ISBN 0-7695-1903-2., IEEE Computer Society (2003) 139–148
3. Crespo, Y., López, C., Marticorena, R., Manso, E.: Language independent metrics support towards refactoring inference. In: 9th ECOOP Workshop on QAOOSE 05 (Quantitative Approaches in Object-Oriented Software Engineering). Glasgow, UK. ISBN: 2-89522-065-4. (2005) 18–29 http://pisuerga.inf.ubu.es/clopez/refactoring/.
4. Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (1992) also Technical Report UIUCDCS-R-92-1759.
5. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Object Technology Series. Addison-Wesley (1999)
6. Dig, D., Comertoglu, C., Marinov, D., Johnson., R.: Automatic detection of refactorings in evolving components (2006)
7. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume I: Foundations. World Scientific (1997)
8. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume II: Applications, Languages and Tools. World Scientific (1999)
9. Mens, T., Van Eetvelde, N., Demeyer, S., Janssens, D.: Formalizing refactorings with graph transformations. Journal on Software Maintenance and Evolution: Research and Practice **17(4)** (2005) 247–276
10. Eetvelde, N.V., Janssens, D.: Refactorings as graph transformations. Technical report, Universiteit Antwerpen (2005)
11. Koch, H.: Ein refactoring-framework für java. Diploma thesis, CS Dept. III, University of Bonn, Germany (2002)
12. Kniesel, G., Koch, H.: Static composition of refactorings. Science of Computer Programming **52** (2004) 9–51 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.
13. Pérez, J.: Automated elaboration of refactoring plans. In: Preproceedings of the Summer School on Generative and Transformational Techniques in Software Engineering. (2005)
14. Bardohl, R., Minas, M., Schurr, A., Taentzer, G.: Application of Graph Transformation to Visual Languages. [8] 105–180
15. Taentzer, G.: Agg: A graph transformation environment for modeling and validation of software. [16] 446–453

16. Pfaltz, J.L., Nagl, M., Böhlen, B., eds.: Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers. In Pfaltz, J.L., Nagl, M., Böhlen, B., eds.: AGTIVE. Volume 3062 of Lecture Notes in Computer Science., Springer (2004)
17. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. [18] 161–177
18. Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G., eds.: Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings. In Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G., eds.: ICGT. Volume 3256 of Lecture Notes in Computer Science., Springer (2004)
19. Münch, M.: Programmed graph rewriting system progres. [20] 441–448
20. Nagl, M., Schürr, A., Münch, M., eds.: Applications of Graph Transformations with Industrial Relevance, International Workshop, AGTIVE'99, Kerkrade, The Netherlands, September 1-3, 1999, Proceedings. In Nagl, M., Schürr, A., Münch, M., eds.: AGTIVE. Volume 1779 of Lecture Notes in Computer Science., Springer (2000)
21. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding refactorings via change metrics. In: OOPSLA. (2000) 166–177