

# Enabling Refactoring with HTN Planning to Improve the Design Smells Correction Activity

Javier Pérez

University of Valladolid; Department of Computer Science  
jperez@infor.uva.es

**Abstract.** Refactorings are a key technique to software evolution. They can be used to improve the structure and quality of a software system. This paper introduces a proposal for generating refactoring plans with hierarchical task network planning, to improve the automation of the bad smells correction activity.

## 1 Introduction

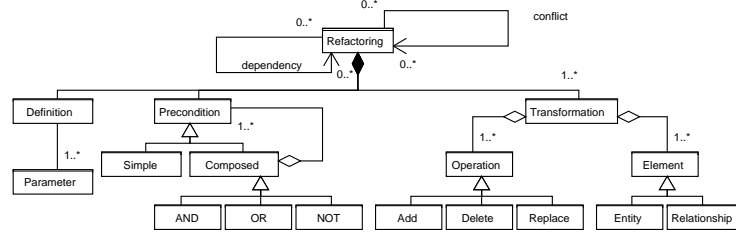
Over the evolution of a software system, its structure deteriorates because the maintenance efforts concentrate more on the correction of bugs and on the addition of new functionalities than on the control and improvement of the software's architecture and design [1]. Bad design practices, often due to inexperience, insufficient knowledge or time pressure, are at the origin of design smells. They can arise at different levels of granularity, ranging from high level design problems, such as antipatterns [2], to low-level or local problems, such as code smells [3].

Design smells are problems encountered in the software's structure, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to true compilation errors and run-time errors in the future. Design Smell management refers to the set of techniques, tools and approaches addressed to detect and to correct or, at least, reduce design smells to improve software quality. Among the activities involved in design smell management, correction and detection are the most significant ones.

## 2 Detection and Correction of Smells

The detection techniques proposed in the literature mainly consist on defining and applying rules for identifying design smells. Meanwhile, correction techniques often consist on suggesting which transformations could be applied to the source code of the system in order to restructure it, by correcting or, at least, reducing its design problems. There has been an increasing number of works dealing with smell detection, and the most successful ones are those based on metrics [4], and on the jointed use of metrics and structural patterns analysis [5].

The correction activity has not been explored as much as the detection one. Most approaches focus on suggesting which are the best redesign changes to



**Fig. 1.** A model for refactoring operations.

perform, and which are the best structures to remedy the smell and reflect the original design intent. The preferred technique for correction is refactoring [3], because the objective is not to remove bugs or errors. In terms of observable behaviour, we aim at leaving the system untouched. The automation of the correction strategies, based on refactorings, faces the problem of precondition fulfillment, as mentioned in [6].

It is rare that the preconditions of the desired refactorings could be fulfilled by the system’s source code at its current state. For example, to allow moving a method from one class to another, one should probably, first, have to move the attributes accessed from it. In these cases, the developer has to plan ahead how to solve this problem. This can be done either by choosing a different refactoring path or by applying other preparatory refactorings to enable the precondition which previously failed. Moreover, violation of preconditions is the most common error developers encounter when trying to apply a refactoring operation [7]. Therefore, suggesting refactorings is not enough to allow for automated correction of design smells.

### 3 Anatomy of a Refactoring Operation

A refactoring can be seen as a conditional transformation [8], which is composed of a precondition and a set of transformations. The precondition establishes the situations under which the refactoring can be executed, while the transformation part specifies the changes that are to be applied to the source code. If the precondition of a refactoring is fulfilled when it is performed, the system’s behaviour is preserved. Figure 1 shows a simplified model for refactoring operations.

Once a smell has been detected and once the refactorings to correct it have been given out, they can’t be immediately applied if their preconditions fail. Therefore, refactoring suggestions don’t suffice to automate the activity of bad smell correction, we need refactoring plans.

We define a refactoring plan as the specification of a refactoring sequence that matches a system redesign proposal and can actually be executed over the current system’s source code. To improve the automation of the smell correction activity, we intend to support the automated generation of refactoring plans.

A variety of techniques can be used to reason about refactorings and assist the generation of refactoring plans. Analysis of dependencies and conflicts can be performed to find out which refactorings can enable or disable other refactoring's preconditions [9]. First-order logic inference can help composing refactoring sequences [8]. Automated planning [10], can integrate all these techniques.

## 4 Enabling Refactoring with HTN planning

Automated planning [10] is an artificial intelligence technique to generate sequences of actions that will achieve a certain goal when they are performed. We think that automated planning is a technique suitable to be used in the generation of refactoring plans.

For a typical automated planner, the current state of the world is represented as a set of logical terms which are changed through application of operators. Operators are composed of a precondition which specifies the conditions under which they can be applied, and two separate sets of actions which specify how the operator modifies the state of the world. These lists enumerate the terms the operator will add to and delete from the current state. A goal is a list of terms which represents a certain state of the world we want to achieve. A planner computes a plan as a sequence of operator instances that changes the world to achieve the desired goals in the final state.

Among all the existing planning approaches, we think that hierarchical task network (HTN) planning provides the best balance between search-based and procedural-based strategies, for the problem of refactoring planning. We have explored other approaches, such as partial-order backwards planning, only to discover that combinatorial explosion and lack of expressivity disallow their application in the refactoring planning domain.

HTN planning [10], introduces the concept of "task", which models actions composed by simple operators or by other tasks. Task networks allow to include domain knowledge describing which subtasks should be performed to accomplish another one. HTN planning and forward search allows very expressive domain definitions which lead to very detailed domains with a lot of domain knowledge which can guide the planning process in a very efficient way.

Starting from a set of refactorings, the refactoring dependencies, and the system's source code and a redesign proposal, an HTN planner can obtain a refactoring plan matching the redesign proposal, while solving the problem of failing preconditions.

To search for refactoring plans we use the representation from [8], which turns the system's AST into a set of logical terms. This set builds up the planner's state of the world. Refactorings are modeled with tasks and operators. Tasks hierarchies allow to specify the algorithm of a refactoring along with the dependencies with other refactorings. Thus, using tasks and subtasks dependencies, we model which refactorings should be executed in order to enable the precondition of another one. An HTN planner can be tailored to search for plans which

achieve a certain design structure, or which enable application of a desired set of refactorings.

## 5 Conclusions

This paper introduces a proposal to enable refactoring application through automated planning, more precisely HTN planning. Automation support for refactoring planning can improve any practice which uses them and many software evolution techniques, particularly the correction of design smells. We are currently preparing experiments to show the feasibility of this approach.

## Aknowledgements

I want to thank Tom Mens, Naouel Moha and Carlos López who have helped me reviewing the state of the art in design smells management.

This work has been partially funded by the regional government of Castilla y León (project VA-018A07).

## References

1. Frederick P. Brooks, J.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Publishing Company, Reading, MA , USA (1975)
2. Brown, W.H., Malveau, R.C., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley (March 1998)
3. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Object Technology Series. Addison-Wesley (1999)
4. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer (2006)
5. Moha, N.: DECOR : Détection et correction des défauts dans les systèmes orientés objet. PhD thesis, Université des Sciences et Technologies de Lille; Université de Montréal (August 2008)
6. Trifu, A., Reupke, U.: Towards automated restructuring of object oriented systems. Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on (March 2007) 39–48
7. Murphy-Hill, E., Black, A.P.: Breaking the barriers to successful refactoring: observations and tools for extract method. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, New York, NY, USA, ACM (2008) 421–430
8. Kniesel, G.: A logic foundation for conditional program transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn (January 2006)
9. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and Systems Modeling **6**(3) (September 2007) 269–285
10. Ghallab, M., Nau, D., Traverso, P.: Automated Planning; Theory and Practice. Morgan Kaufmann (2004)