Modularity-Oriented Refactoring

Sérgio Bryton, Fernando Brito e Abreu
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal
bryton@di.fct.unl.pt, fba@di.fct.unl.pt

Abstract

Refactoring, in spite of widely acknowledged as one of the best practices of object-oriented design and programming, still lacks quantitative grounds and efficient tools for tasks such as detecting smells, choosing the most appropriate refactoring or validating the goodness of changes.

This is a proposal for a method, supported by a tool, for cross-paradigm refactoring (e.g. from OOP to AOP), based on paradigm and formalism-independent modularity assessment.

1. Introduction

Refactoring is the process of modifying software implementation to improve its internal structure, without altering its external behavior [1]. Implicitly, this definition encompasses four tasks:

(1) Find where to apply a change;
(2) Choose how to perform the modification;
(3) Guarantee that the external behavior of the impacted software module was not altered;
(4) Make sure that the systems' internal structure has effectively improved.

For each of the previous tasks, an immediate question arises, respectively:

(1) Which part of the code should be changed?
(2) Which kind of refactoring should be applied?
(3) How to evaluate that the external behavior of the resulting software part remains the same?
(4) How to evaluate the alleged improvements?

While for assessing the stability of the external behavior, automated regression tests will do a good job, for the remainder questions, the answer is not so straightforward.

A serious evaluation of improvements should be formal and quantitatively based. Without a metrics-based assessment (before and after a refactoring is applied) it would not be possible to rank the set of applicable refactorings in terms of the actual improvements they would cause.

Curiously, while some researchers claim that metrics are a simple approach for the quality assessment of source code [2] or that no set of metrics rivals informed human intuition[1], others get puzzled when confronted to the fact that although refactorings are used to improve the quality of their systems, a supposedly adequate set of metrics indicates that this process often has the opposite result [3].

As stated before, refactoring implies improvement of the internal structure of a system, but this improvement can be achieved on several quality characteristics, the most usually claimed being maintainability, reusability and efficiency [4].

The proposed method, Modularity-Oriented Refactoring (MORe) will enable cross-paradigm refactoring automation based on paradigm and formalism independent modularity metrics, as well as the development of a tool to support it.

The remainder of this proposal is organized as follows. At section 2 a brief description of the state of the art is presented, followed by the research objectives, current work and work plan at sections 3, 4 and 5 respectively. Section 6 presents some conclusions.

2. State of the art

2.1 Refactorings and code smells

Code smells are ways to describe warning signs about potential problems in code [1, 5]. One of the purposes of refactoring is precisely eliminating code smells.

Several catalogues of refactoring have been proposed, being the most widely accepted those from Fowler [1] and Kerievsky [6]. There are also some very interesting publications regarding refactoring for aspect-oriented programming, such as those from Laddad [7] and Monteiro [5].

Simon et al. agree with the difficulty of identifying where to apply each refactoring and propose object-oriented cohesion-based metrics to solve this problem.

However, they do not evaluate the quality improvements after the refactoring is applied [8].

Tahvildari et al. proposed a taxonomy for design flaws, a reengineering strategy [9], and a framework to detect design flaws and re-engineer them [10, 11] for object-oriented systems using, among other, classical modularity metrics.

In spite of the work done so far, the relation among code smells, refactoring, and the affected quality characteristics has a lot of room for improvement, namely if considered that these quality properties are independent of paradigm and adopted language and should be measured as such. Therefore, there is a wide berth for research on cross-paradigm quantitatively-based refactoring.

2.2 Tools

Ideally, tools should provide a fully automated refactoring process, whilst giving the developer a chance to select the most appropriate decisions at their discretion. To fully achieve this purpose, tools must be able to detect, decide upon changes, and assess the results achieved, in a quantitative way, as seen before. However, even though at least 31 refactoring tools for 10 different languages exist [12], none of these seems to fully support the aforementioned requirement. Nevertheless, two of them are worth mentioning for being a step ahead: JDeodorant [13] and TRex [14].

JDeodorant is an Eclipse plugin that identifies Feature Envy bad smells in Java projects and resolves them by applying the appropriate Move Method refactoring upon ranking them. The whole process is grounded on dissimilarity metrics and no changes to the source occur until the decision to refactor is taken. No post-refactoring assessment is made, and the tool is very limited in scope, since only one smell for Java is supported. Still, the process is quite elegant and promising.

TRex is also an Eclipse plugin that automates the application of refactorings and the detection of refactoring opportunities for test suites specified using the Standardized Tree and Tabular Combined Notation (TTCN-3) [15]. The whole process is grounded on specific metrics for test suites and pattern-based analysis. Refactorings can be applied in two different ways: either the developer invokes the refactoring from the code location, or the refactoring is invoked directly by a quick fix which is provided by the analysis results of the automated quality assessment. As for this tool, besides being too domain-specific, no post-refactoring assessment is performed. However, this process is also quite elegant and aiming systems with thousands of lines of code.

The remainder tools focus on implementing well the refactorings for their corresponding target language. They do not provide assistance in detecting smells, they do not help on choosing the right refactoring and they do not assess the final result. Basically, the user decides everything, while the tool tries to implement the decisions quick and cleanly. We are not aware of tools that perform cross-paradigm quantitative modularity-based refactorings.

3. Research Objectives and Approach

3.1 Research Objectives

The research objective of this dissertation is to enable a fully automated and quantitatively grounded refactoring process, focused on modularity benefits. We expect to contradict Opdyke claim that a refactoring tool can help a designer by providing the right set of refactorings and by ensuring that each refactoring is applied correctly, but it cannot decide which refactorings to apply [16]. Habra [17] and Tahvildari et al. [9-11] corroborate the opinion that metrics can identify potential refactorings and estimate the refactoring effect.

The main expected contributions of this work are:

(1) A cross-paradigm and language independent method for refactoring, based on modularity metrics, called Modularity-Oriented Refactoring (MORe). The latter follows the Meta-Model Driven Measurement (M2DM) approach [18] and uses the PIMETA metamodel [19].

(2) The MORe tool, an Eclipse plug-in, which will fully implement the MORe method for refactoring Java and AspectJ systems.

3.2 Research Approach

The classical scientific method depends upon theory formation, followed by experimentation and observation, to provide a feedback loop to validate, modify and improve the theory. This procedure can be followed, and it is also appropriate, for software engineering research [20-22].

This thesis will follow the scientific method; first an evaluation of the state of the art will be conducted; then the MORe method and the MORe tool will be proposed and developed and, finally, to assess the claims presented in this paper, a statistical validation of the achieved modularity improvements will be performed, based on a set of case studies, the GoF Design Patterns implemented both in Java and AspectJ [23]. For this purpose, a set of paradigm-independent metrics will be collected on the original manual refactorings and on

those resulting from the application of the MORe method.

4. Current Work and Preliminary Results

A lot of significant references have already been gathered and these seem to underline the actuality and relevance of this research work.

PIMETA offers provisions for performing a formal comparison of software systems using the M2DM approach. Given the expressiveness and preciseness of OCL for expressing modularity metrics and the simplicity of the PIMETA semantics, combined with its intended multi-paradigm instantiation ability, PIMETA appears to be an adequate ground for basing paradigm and language independent modularity assessments. The benefit of independence allows comparisons across paradigm boundaries.

Grounded on PIMETA, several paradigm and language independent modularity metrics have already been defined and are being used to analyze modularity benefits from AOP over OOP. These metrics, and eventually others that may be required, will be used to identify code smells based upon their effect on modularity, to evaluate the impact of candidate refactorings on modularity, and to evaluate the overall system modularity, before and after refactorings take place.

There is also ongoing work on the capability of visualizing modularity with graphs. This visual capability has proven of value when it comes to analyze the dependencies among the elements in a software system and validate the metrics results. It may also be useful to visualize the proposed changes, before approving them. The visualization technique has also been used by Simon et Al. [8] to identify code smells.

The MORe method, visible at Figure1, consists in a sequence of seven steps. At each step, a result is produced (r1 through r7). To achieve the results for each step, several artifacts (a1, a2 and a3) and previous results are combined.

At the first step, the modularity metrics (a2) are collected from the source code (a1), to obtain a modularity assessment for the source code (r1). Then, at step number 2, the previous modularity metrics (a2) are used along with code smell catalogues (a3), to identify a set of code smell detection metrics (r2). At the third step, the code smell detection metrics (r2) will be collected from the source code (a1) to identify the code smells (r3). Then, at step number 4, the refactorings to apply (r4) at the code smells will be chosen based on a modularity evaluation, with the modularity metrics (a2), of each possible refactoring, according to the code smell catalogues (a3), for each code smell (r3). At step number 5, the best refactorings (r4) are applied to the code smells (r3) and the refactored code is obtained (r5). Then, at step number 6, another modularity assessment (r6) is obtained by applying the modularity metrics (a2) at the refactored code (r6). Finally, the modularity improvements (r7) are identified by comparing the source code modularity assessment (r1) and the refactored code modularity assessment (r6).

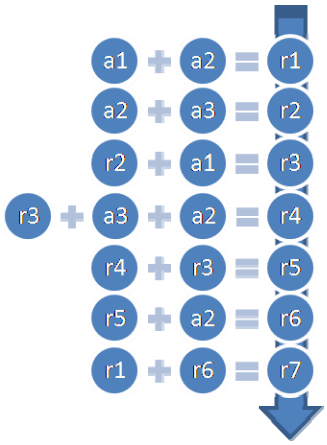

Figure 1 - The MORe method

The MORe tool architecture is being outlined. It is not intended to implement the refactorings. Instead, the refactoring features available at the Eclipse IDE will be triggered for this purpose. The MORe tool will be focused on instantiating the PIMETA with the system under assessment, conducting the required measurements, analyzing the results, and interacting with the user. From these four features, the first two have already been developed and are going through intensive validation, although not yet as an Eclipse plugin.

5. Work Plan and Implications

The work plan is divided into three phases, which are expected to take place between January 2007 and October 2009, when the thesis is expected to be delivered.

The first phase aims at obtaining as much insight as possible on the problem in hands. The main activity here will be an intensive literature review on four areas of research: (i) refactoring code-smells, (ii) catalogues, tools and meta-models, (iii) quantitative-based refactoring and (iv) refactoring from OOP to AOP.

On the second phase, the thesis proposals will be implemented, and at the third phase, the results will be validated with a case study, the GoF Design Patterns implemented both in Java and AspectJ, as mentioned earlier.

6. Conclusions

The MORe method will bring more quantitative arguments which, supported by a tool, will enable cross-paradigm refactorings, namely from OOP to AOP. We aspire at obtaining at least as good results as those produced by informed human intuition.

The strategy followed by the MORe method can be used by similar methods aiming to automate refactoring, regarding different quality properties.

The results achieved will provide a good ground for classifying existing refactorings, according to their effect on modularity, and proposing new ones.

It is also expected that the relations between bad smells, metrics, refactorings and modularity can be clearly identified.

7. References

[1] M. Fowler, et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[2] H. Neukirchen and M. Bisanz, "Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites," *Proc. of 19th IFIP TestCom'2007 and FATES'2007*, Springer, 2007, pp. 228-243.

[3] K. Stroggylos and D. Spinellis, "Refactoring: Does it improve software quality?," *Proc. of 5th International Workshop on Software Quality*, ACM Press, 2007.

[4] *ISO 9126 Standard: Information Technology - Software Product Evaluation - Software Quality Characteristics and Metrics*, 2001.

[5] M.P. Monteiro and J.M. Fernandes, "Towards a Catalogue of Refactorings and Code Smells for AspectJ," *Transactions on Aspect-Oriented Software Development I*, vol. LNCS, no. 3880, 2006, pp. 214-258.

[6] J. Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2004.

[7] R. Laddad, *Aspect-Oriented Refactoring*, Addison-Wesley, 2006.

[8] F. Simon, et al., "Metrics Based Refactoring," *Proc. of CSMR'2001*, IEEE Computer Society Press, 2001, pp. 30-38.

[9] L. Tahvildari and K. Kontogiannis, "A Metric-Based Approach to Enhance Design Quality through Meta-pattern Transformations," *Proc. of CSMR'03*, IEEE Computer Society Press, 2003.

[10] L. Tahvildari, "Quality-Driven Object-Oriented Re-engineering Framework," *Proc. of ICSM'04*, IEEE Computer Society Press, 2004.

[11] M. Salehie, et al., "A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws," *Proc. of ICPC'06*, IEEE Computer Society Press, 2006.

[12] M. Fowler, "Refactoring home page," 2008; http://www.refactoring.com/.

[13] M. Fokaefs, et al., "JDeodorant: Identification and Removal of Feature Envy Bad Smells," *Proc. of ICSM'2007*, 2007, pp. 519-520.

[14] H. Neukirchen and B. Zeiss, "Automation of refactoring and refactoring suggestions for TTCN-3 Test Suites," *Proc. of 1st Workshop on Refactoring Tools held in conjunction with ECOOP'2007*, 2007.

[15] *Methods for Testing and Specification (MTS) - The Tree and Tabular Combined Notation version 3 (TTCN-3): Core Language*, *ETSI*, .

[16] B. Opdyke, *Refactoring Object-Oriented Frameworks*, Technical Report UIUCDCS-R-92-1759, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1992.

[17] N. Habra and M. Lopez, "On the Use of Measurement on Software Restructuring," *Proc. of International ERCIM Workshop on Software Evolution*, 2006.

[18] F.B. Abreu, *Using OCL to formalize object oriented metrics definitions*, Technical Report ES007/2001, Software Engineering Group, INESC, 2001.

[19] S. Bryton and F.B. Abreu, "Towards paradigm-independent software assessment," *Proc. of QUATIC'2007*, IEEE Computer Society, 2007.

[20] M.V. Zelkowitz and D. Wallace, "Experimental Validation in Software Engineering," *Journal of Information and Software Technology*, vol. 39, no. 11, 1997, pp. 735-743.

[21] W.F. Tichy, et al., "Future Directions in Software Engineering," *SIGSOFT Software Engineering Notes*, vol. 18, no. 1, 1993, pp. 35-48.

[22] M. Goulão and F.B. Abreu, "Modeling the Experimental Software Engineering Process," *Proc. of QUATIC'2007*, IEEE Computer Society Press, 2007.

[23] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," *Proc. of OOPSLA'02*, ACM Press, 2002, pp. 161-173.