Composite Refactorings: The Next Refactoring Rubicons

Mohsen Vakilian Ralph E. Johnson
University of Illinois at Urbana-Champaign
{mvakili2, rjohnson}@illinois.edu

Abstract

The industry crossed the first refactoring rubicon, namely Extract Method, more than a decade ago. Today, all mainstream Integrated Development Environments (IDEs) support this refactoring, and empirical studies have shown that Extract Method is one of the most frequently used automated refactorings. Although complex refactorings are more tedious and error-prone, studies have shown that programmers use the automated refactorings mostly for performing simple changes. We argue that new interaction models are needed to support high-level composite refactorings. Because of the challenges involved in automating such complex refactorings, we consider composite refactorings as the next refactoring rubicons.

1. The First Refactoring Rubicon

The first refactoring tool was developed for Smalltalk in 1997 to make the refactoring process more efficient and reliable [6]. In 2001, Martin Fowler declared that two Java refactoring tools crossed the rubicon by supporting the Extract Method refactoring [1]. Automating the Extract Method refactoring required more complex analysis than Rename and was considered as a sign that an IDE was serious about refactorings. Several empirical studies have shown that Extract Method is one of the most frequently used automated refactorings [4, 5, 7]. This is good news. Mainstream refactoring tools have crossed the rubicon, and the Extract Method refactoring has crossed the chasm.

2. The Second Refactoring Rubicon

Today, mainstream IDEs support more complex refactorings that change the class hierarchy, affect many files and lines of code, or perform sophisticated analyses to compute the change, e.g. Extract Superclass and Infer Generic Type Arguments, and Push Down. The research community has proposed tools for automating even more sophisticated changes such as those for parallelism, optimization, security, and migration to new libraries.

However, studies have shown that programmers have not adopted the existing automated refactorings that perform complex changes [4, 5, 7]. Besides, current IDEs do not support higher-level refactorings that change the architecture of the system or refactor the code to implement design patterns like Composite, Visitor, or Strategy [2, 3].

Composite refactorings are high-level refactorings that consist of several other refactorings. Refactoring to design patterns is one kind of composite refactorings [3]. Composite refactorings are more tedious and error-prone to perform manually than low-level ones. However, IDEs lack good support for such high-level refactorings. Composite refactorings are inherently complex and thus their automations tend to be difficult to learn and predict. Designers of automated refactorings face conflicting forces especially for composite refactorings. For instance, on one hand, the automated refactoring should be general enough to be applicable to many pieces of code. On the other hand, the more powerful transformations tend to have higher configuration costs. These are some of the factors that make the design of a suitable interaction model for composite refactorings challenging. Thus, we consider automated composite refactorings as the next refactoring rubicons.

The first refactoring rubicon was mostly an engineering problem, because an implementation of the Extract Method refactoring was available for Smalltalk [6]. However, designing an appropriate interaction model for composite refactorings is still an open research problem.

We recently studied the refactoring composition patterns that programmers use in practice. A refactoring composition pattern is a recurring set of automated refactorings that make up a bigger refactoring. We found that composition patterns allow programmers to reduce the configuration and learning overhead of automated refactorings. Our findings led us to take a composition-based approach to automating refactorings. We built a prototype of the Extract Superclass refactoring, which is one of the more complex and less often used automated refactorings, as a composition of several refactorings. Our preliminary evaluations showed that our approach is promising. Most programmers appreciated the high control that our interaction model provided and confirmed our hypotheses about the lower configuration and learning costs [8].

References

[1] M. Fowler. Crossing Refactoring's Rubicon. http://martinfowler.com/articles/refactoringRubicon.html.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] J. Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

[4] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 2006.

[5] E. Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. *IEEE Trans. Software Eng.*, 2011.

[6] D. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theor. Pract. Object Syst.*, 1997.

[7] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, Disuse, and Misuse of Automated Refactorings. *To Appear in ICSE*, 2012. URL http://hdl.handle.net/2142/27730.

[8] M. Vakilian, N. Chen, S. Negara, R. Zilouchian Moghaddam, and R. E. Johnson. Composite Refactorings. 2012. URL http://hdl.handle.net/2142/30851.