Automated Refactoring

Lisa Maria Kritzinger
Johannes Kepler University Linz
1255353
Email: kritzinger@gmx.net

Peter Feichtinger
Johannes Kepler University Linz
1056451
Email: shippo@gmx.at

Abstract

Refactoring, the restructuring of a software system without changing its semantics, is essential in software evolution. Manual refactoring can be time-consuming and error-prone, so tool support is desirable when making large changes. In this article, we will explore a number of publications on automating different refactoring tasks, from just making code more compact to introducing objects into a C codebase. ***[PF] Should the abstract be longer?***

Additionally, we will provide comparisons which will show the advantages and disadvantages of the different approaches, regarding the performance of the software after refactoring, the applicability of a certain approach or the simplicity of the application of an approach.

Keywords

Software restructuring, automatic refactoring, tool support, software evolution.

1. Introduction

Refactoring is the process of restructuring a software system without changing its semantics. It is used to increase readability and maintainability of software, reduce its complexity, or change the architecture of a system. Refactoring is essential in software evolution, because as a system is adapted to new requirements it inevitably becomes more complex and drifts away from its original design. This makes maintenance more difficult and reduces the software quality. Refactoring can then help to bring the system back to its original design and into a more maintainable state, improving code quality in the process.

However, manual refactoring without any tool support can be error-prone and time-consuming. There are various tools available for supporting elementary refactorings like renaming a variable or introducing an additional parameter to a function, often built into the used *Integrated Development Environment* (IDE) itself. But even with tool support, manual refactoring can still be too complicated or just tedious, and tool support for automating refactoring tasks is desirable in a number of cases.

In this article we're going to highlight some recent and not-so-recent contributions in the field of automatic refactoring, ranging from simple tasks like making code more compact [1], to more complicated tasks like introducing object-orientation into a C codebase [2]. We will also investigate the benefit from refactoring a software through replacing conditionals by using polymorphism [3] as well as the approach of using design differencing for automated refactoring [4].

2. Background

[LK] do we need this section?

3. Automated Refactoring Approaches

[LK] TODO short introduction

3.1. Restructuring Legacy C Code into C++

[LK] PF: TODO definition and comparison

3.2. Conditionals vs Polymorphism

[LK] TODO definition and comparison

3.3. Design Differencing

[LK] TODO definition and comparison

3.4. Spartanizer

[LK] PF: TODO definition and comparison

4. Conclusion

Text

Acknowledgments

Text

References

[1] Y. Gil and M. Orrù, "The Spartanizer: Massive automatic refactoring," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, M. Pinzger, G. Bavota, and A. Marcus, Eds. IEEE Computer Society, 2017, pp. 477–481.

[2] R. Fanta and V. Rajlich, "Restructuring legacy C code into C++," in *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999*. IEEE Computer Society, 1999, pp. 77–85.

[3] S. Demeyer, "Maintainability versus performance: What's the effect of introducing polymorphism?" Lab. on Reengineering (LORE), Universiteit Antwerpe, Tech. Rep., 2002.

[4] I. H. Moghadam and M. Ó Cinnéide, "Automated refactoring using design differencing," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, T. Mens, A. Cleve, and R. Ferenc, Eds. IEEE Computer Society, 2012, pp. 43–52.