Code-Imp: A Tool for Automated Search-Based Refactoring

Iman Hemati Moghadam
School of Computer Science and Informatics
University College Dublin
Ireland
Iman.Hemati-Moghadam@ucdconnect.ie

Mel Ó Cinnéide
School of Computer Science and Informatics
University College Dublin
Ireland
mel.ocinneide@ucd.ie

ABSTRACT

Manual refactoring is tedious and error-prone, so it is natural to try to automate this process as much as possible. Fully automated refactoring usually involves using metaheuristic search to determine which refactorings should be applied to improve the program according to some fitness function, expressed in terms of standard software quality metrics.

Code-Imp (Combinatorial Optimisation for Design Improvement) is such an automated refactoring platform for the Java language. It can apply a range of refactorings, supports several search types, and implements over 25 software quality metrics which can be combined in various ways to form a fitness function. The original goal of the Code-Imp project was to investigate the use of automated refactoring to improve software quality as expressed by a contemporary metrics suite.

In this paper we present a technical overview of the Code-Imp implementation, and summarise three active research strands involving Code-Imp: *refactoring for testability*, *metrics exploration*, and *multi-level design improvement*.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Software—*Restructuring, reverse engineering, and reengineering*

General Terms

Experimentation

Keywords

Refactoring tool, search-based refactoring, search-based software engineering.

1. INTRODUCTION

There are many refactoring tools available, most of which focus on performing specific refactorings as requested by the developer. In the past number of years, there has been an amount of interest in automated refactoring tools based on Search-Based Refactoring [4, 8, 9, 11]. Using Search-Based Refactoring, it is possible to apply many thousands of refactorings without programmer intervention and hence change the design of the program radically.

In this approach, the process of refactoring is guided by a metrics suite where a refactoring is acceptable if it, apart from preserving the behavior of the system, improves the merit of the design based on the metrics suite. Determining an effective metrics suite as well as a search technique for exploring alternative solutions during the refactoring process are two challenging tasks. A recent review by Raiha provides a broad overview of search-based software design, and specifically search-based refactoring [10].

The remainder of this paper is structured as follows. In Section 2 we present the design and implementation of Code-Imp, as well as describe the supported fitness functions and search techniques. In Section 3 we describe several applications of Code-Imp in terms of previous, present and future work, and finally present our conclusions in Section 4.

2. CODE-IMP DESIGN

We recently rewrote Code-Imp, primarily to support Java 6 input and to provide a more flexible platform as an automated refactoring tool. Our original preference was to base our implementation on the Eclipse JDT framework[1], however this framework maintains a tight integration between the source code and its various internal representations, which is in conflict with our desire to refactor the ASTs extensively before regenerating the source code. In the JDT framework, updates to the ASTs are visible only after rewriting changes to the source code, hence refactorings whose preconditions rely on previous refactorings can only be applied after the source code has been regenerated. However, regenerating the source code after each refactoring is impractical for a large application when many thousands of refactorings are involved, and we were therefore obliged to seek another platform on which to develop Code-Imp.

We finally decided to use RECODER[2], which is a framework for Java source code metaprogramming that supports many kinds of Java analysis and transformation.

Figure 1 depicts the current architecture of Code-Imp. The right side of the figure shows the process of refactoring in detail. Code-Imp first extracts the initial ASTs from

[1]http://www.eclipse.org/jdt
[2]http://sourceforge.net/projects/recoder

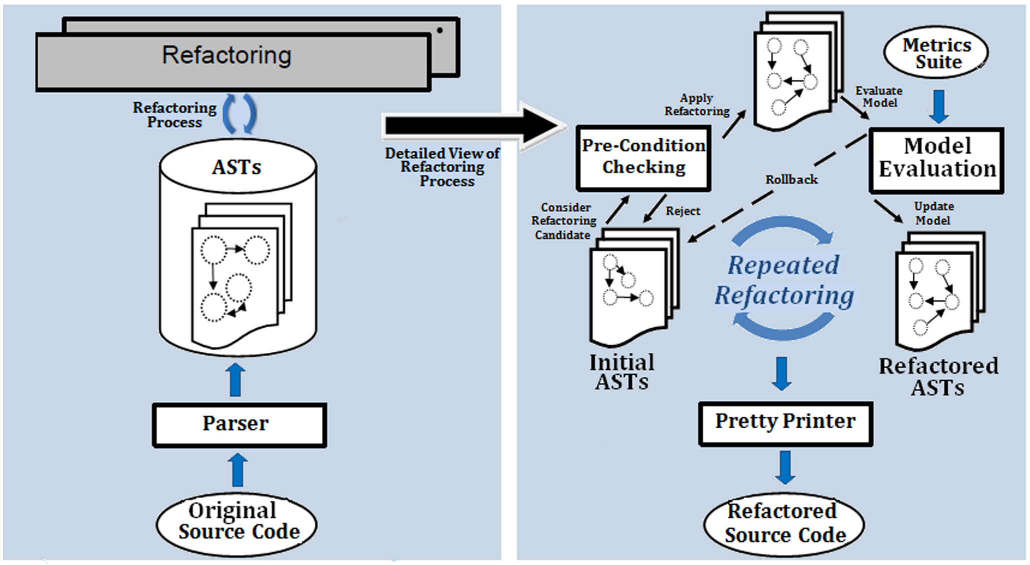

Figure 1: Overall architecture of the Code-Imp automated refactoring framework

the source code. In terms of memory consumption, the ASTs generated require approximately 10 times more memory than the original source code. Code-Imp searches the ASTs for candidate refactorings. A refactoring is acceptable if it satisfies all pre- and post-conditions as well as complying with the demands of the search technique in use[3]. This process is repeated many times. After the final refactoring is applied, the ASTs are pretty printed to source code files.

During the refactoring process, a rollback mechanism is supported by logging each change to the ASTs. The change history service makes it possible to perform a rollback at different levels of granularity. For example, at the finest level of granularity, individual refactorings can be reversed. At a coarser level of granularity, a composite refactoring such as a pull up method (which also contains a pull up attribute refactoring) can be reversed.

The current version of Code-Imp is not integrated with any IDE and is used from the command-line.

2.1 Refactorings

Code-Imp currently supports fourteen design-level refactorings, as described in the following three categories:

1. **Method-Level Refactorings:**
   Push Down Method: Moves a method from some class to those subclasses that require it. *Pull Up Method:* Moves a method from some class(es) to their immediate superclass. *Decrease/Increase Method Visibility:* Changes the visibility of a method by one level, e.g. private to package or public to protected.

2. **Field-Level Refactorings:**
   Push Down Field: Moves a field from some class to those subclasses that require it. *Pull Up Field:* Moves a field from some class(es) to their immediate superclass. *Decrease/Increase Field Visibility:* Changes the visibility of a field by one level, e.g. private to package or public to protected.

3. **Class-Level Refactorings:**
   Extract Hierarchy: Adds a new subclass to a non-leaf class C in an inheritance hierarchy. A subset of the subclasses of C will inherit from the new class. *Collapse Hierarchy:* Removes a non-leaf class from an inheritance hierarchy. *Make Superclass Abstract:* Declares a constructorless class explicitly abstract. *Make Superclass Concrete:* Removes the explicit abstract declaration of an abstract class without abstract methods. *Replace Inheritance with Delegation:* Replaces an inheritance relationship between two classes with a delegation relationship; the former subclass will have a field of the type of the former superclass. *Replace Delegation with Inheritance:* Replaces a delegation relationship between two classes with an inheritance relationship; the delegating class becomes a subclass of the former delegate class.

[3]In the case of a hill-climb, this means improving the quality of the design based on the metrics suite. In the case of, e.g. simulated annealing, a drop in quality may also be accepted.

2.2 Fitness Function

In order to guide the search, it is necessary to evaluate if a refactoring has improved the design of the program. To this end, we implemented a set of twenty seven software quality metrics. The fitness function can then be defined based on any combination of these metrics. Metrics can be combined using one of two optimality approaches namely *weighted-sum* and *pareto optimality*.

In the weighted-sum approach, the metrics values are simply added using weights that represent the relative importance assigned to the different metrics. Choosing weights is at best an immense challenge and at worst is a theoretically unsound operation on metrics defined on an ordinal scale. In pareto optimality, a refactoring is only regarded as an improvement if it improves at least one metric and does not degrade any other.

Choosing an appropriate metrics suite to form the fitness function and deciding on the optimality algorithm to use depends on the objective of the refactoring process, and we have used both in our experiments.

A list of the metrics implemented in Code-Imp is provided below. For space reasons, we omit references for the metric definitions.

1. **Cohesion Metrics:**
   Low-level Similarity Class Cohesion, *Similarity Class Cohesion*, *Normalized Hamming Distance*, *Cohesion Among Methods of Class*, *Tight Class Cohesion*, *Loose Class Cohesion*, *LCOM1* to *LCOM15*, *Sensitive Class Cohesion Metric*, *Information-Flow-Based Cohesion*, *Class Cohesion.*

2. **Coupling Metrics:**
   Response for Class, *Direct Class Coupling*, *Data Abstraction Coupling*, *Coupling Factor*, *Coupling Between Objects*, *Message Passing Coupling*, *Instability*, *Non-inheritance ICP*, *Inheritance ICP*, *Information-Flow-Based Coupling.*

3. **Other Metrics:**
   Class Interface Size, *Number of Method*, *Data Access Metric.*

2.3 Search Techniques

The first version of Code-Imp used a variety of local and metaheuristic search techniques, namely *hill-climbing*, *simulated annealing* and *genetic algorithms.* The new version of Code-Imp currently supports just two flavours of hill climbing: *first-ascent hill-climbing (HCF)* and *steepest-ascent hill-climbing (HCS).* These are both local search algorithms where the search examines neighbouring solutions. In HCF the search moves to the first higher quality neighbour found, whereas in HCS all neighbouring solutions are examined and the search moves to the solution of highest quality.

While our best individual refactoring results were achieved using simulated annealing, the most reliable search technique proved to be hill climbing [8].

3. APPLICATIONS OF CODE-IMP

In this section, we describe some applications of Code-Imp in terms of previous work, current projects and future plans.

3.1 Summary of Existing Results

The goal of the original Code-Imp project was to investigate if it was possible to use automated refactoring to improve software quality as expressed by a contemporary metrics suite. Of course this is a very ambitious goal, which was motivated by a belief that automated refactoring can achieve more than automation of individual refactorings. The key results of this work are summarised below.

The effectiveness of different search techniques was investigated by an experimental comparison of simulated annealing, genetic algorithms and multiple ascent hill-climbing [7]. The results showed that overall, multiple-ascent hill climbing performed best over a set of five medium-sized open source Java programs. The success of hill climbing unfortunately suggests that the refactoring process was not achieving much in the way of design exploration, i.e., it was ‘giving the program a lick of paint’ rather than radically changing its design. We return to this challenge in section 3.4.

We investigated if automated search-based refactoring could improve a program’s design [8] in terms of three external software quality attributes, namely Flexibility, Understandability and Reusability, as defined by the QMOOD metrics suite [2]. It was found that medium-sized programs could indeed be improved significantly using the QMOOD Understandability function as fitness function. A minimal improvement was achieved using the QMOOD Flexibility function while the QMOOD Reusability function was found to be ill-formed and caused refactoring process to include a large number of empty classes. This latter result hinted at another application for Code-Imp — as a platform for investigating the properties of the software metrics themselves. Our current work in this area is described in section 3.3.

3.2 Refactoring for Testability

The increasing popularity of Test-Driven Development has led to a greater emphasis on *testability* as a desirable property of software. To test if automated refactoring could be used to improve testability, we created a small, uncohesive Java application and used Code-Imp to improve its cohesion using the LSCC metric [1]. We then conducted an experiment with industrial software engineers to determine which program they found easier to write test cases for. On inspection, Code-Imp definitely appeared to have improved the design of the program, though the study of the software engineers’ responses was inconclusive on the testability question [5].

This result prompted us to consider as future work the possibility of automated refactoring to improve the design of the software for subsequent automated test case generation. In this case, the goal of the transformation is not to refactor the program for the benefit of the developers, but to transform it so as to enable more effective automated test case generation[4]. Test cases are then generated for the transformed program, but applied to the original one. It is interesting that in this case the “refactorings” used need not preserve behaviour, but must preserve certain test adequacy criteria such as branch coverage or statement coverage [3].

3.3 Exploring Software Metrics

Existing work on search-based refactoring uses software quality metrics to guide the transformation process that optimises the program. In recent work, we have investigated the possibility of turning this on its head and using the refactoring process to discover new properties of the metrics themselves [6].

In these experiments we used Code-Imp to refactor ten real-world open source Java applications, up to 88K lines of code in size[5]. We used a semi-random refactoring process, and measured the effects on a collection of structural cohesion metrics. We were able to measure properties such as metric volatility, propensity for positive change and correlation between the metrics.

The most interesting observations were in the area of metric conflict. Although all cohesion metrics are, in some sense, measuring the same property (cohesion), we discovered that some of them are strongly in conflict with each other. Not only can we detect the conflict, but by studying the refac-

[4]This idea is appealing as it avoids that *bête noire* of automated refactoring — the problem of explaining the refactored program to the programmer.

[5]This shows the robustness of Code-Imp in dealing with non-trivial Java applications.

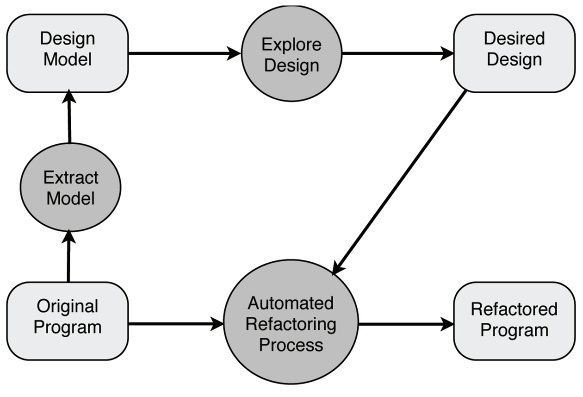

Figure 2: Multi-level Refactoring using Code-Imp

torings that caused it we were able to explain precisely the nature of the conflict [6].

3.4 Muti-Level Automated Refactoring

The Code-Imp search currently operates at source code level. Precondition checking and refactoring execution on ASTs is time consuming, and this limits how extensively a program can be refactored. Another issue with any fully-automated refactoring tool is that the programmer only sees the end result of the refactoring process, and this is likely to lead to comprehension problems. In order to address both these issues, we plan in future work to extend Code-Imp to use *multi-level refactoring* as outlined in figure 2. Multi-level refactoring proceeds in two-stages as follows:

1. **Design Exploration:** A design model is extracted from the source code and transformed to a better design in terms of some metrics suite. As most of the program detail has been abstracted away, precondition checking and refactoring execution become much faster. This will enable a far more extensive search of the design space than has been hitherto possible. At the end of this process, a number of possible optimal designs are presented to the programmer, who selects one as the design they wish to use.

2. **Full Refactoring:** In this detailed refactoring phase, the source code is refactored under the general guidance of a metrics suite, but crucially using the design selected in (1) as the ultimate goal. When this process completes, the resulting program will have the same functional behavior as the original, and a design close to the one chosen by the programmer in stage (1).

The proposed Design Exploration process can be based existing work in this area, e.g. that of Simons, Parmee and Gwynllyw [12]. The Full Refactoring phase is essentially what Code-Imp does now, but with an augmented fitness function (the desired design). The synergy between these two approaches is clear, with each one ameliorating the key problem of the other.

4. CONCLUSION

In this paper we have presented Code-Imp, an automated refactoring platform for Java programs. Previous work with the original version of Code-Imp focussed on automated design improvement. Code-Imp has since been entirely reengineered to support Java 6 and is now built upon the RECODER platform. In our current work we have used Code-Imp to investigate the relationships between software metrics and to explore the possibility improving the testability of a program. The main aspect of future work we described in that of multi-level automated refactoring.

Automated refactoring is most promising in areas where programmers do not have to understand the output of the refactoring process. The major challenges lie in automated design improvement where, even if improvement is technically possible, the difficulty in comprehending the final result may militate against its use.

5. REFERENCES

[1] J. Al Dallal and L. Briand. A precise method-method interaction-based cohesion metric for object-oriented classes. *ACM Transactions on Software Engineering and Methodology*, 2010.

[2] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Software Eng.*, 28(1):4–17, 2002.

[3] M. Harman. Open problems in testability transformation. *IEEE International Conference on Software Testing Verification and Validation Workshop*, 0:196–209, 2008.

[4] M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, July 2007.

[5] M. Ó Cinnéide, D. Boyle, and I. Hemati Moghadam. Automated refactoring for testability. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*, March 2011.

[6] M. Ó Cinnéide, L. Tratt, M. Harman, I. Hemati Moghadam, and S. Counsell. Analysing software metrics with search-based refactoring. In *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO) (submitted)*, July 2011.

[7] M. O'Keeffe and M. O. Cinnéide. Search-based refactoring: an empirical study. *J. Softw. Maint. Evol.*, 20(5):345–364, 2008.

[8] M. O'Keeffe and M. Ó. Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[9] F. Otero et al. Refactoring in automatically generated programs. *Proceedings of the Symposium on Search Based Software Engineering*, 2010.

[10] O. Raiha. A survey on search-based software design. *Computer Science Review*, 4(4):203 – 249, 2010.

[11] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, pages 1909–1916, Seattle, Washington, USA, 8-12 July 2006. ACM.

[12] C. Simons, I. Parmee, and R. Gwynllyw. Interactive, evolutionary search in upstream object-oriented class design. *IEEE Transactions on Software Engineering*, 36(6):798–816, Nov. 2010.