TrueRefactor: An Automated Refactoring Tool to Improve Legacy System and Application Comprehensibility

Isaac Griffith, Scott Wahl, Clemente Izurieta
Computer Science Department
Montana State University
Bozeman, MT
{isaac.griffith, scott.wahl}@msu.montana.edu
clemente.izurieta@cs.montana.edu

Abstract

Manual refactoring is a complicated process requiring intimate knowledge of the software design and underlying intended behavior of a system. This knowledge is not always available. Fully automated refactoring, using a meta-heuristic based search that is dependent on software quality metrics and code smells as a guide, eliminates the need for the developer to be intimately connected to the software under modification. Computer applications in industry and engineering benefit significantly from new approaches to self-correcting refactoring software.

TrueRefactor is an automated refactoring tool that significantly improves the comprehensibility of legacy systems. The goal of TrueRefactor is to modify legacy object-oriented systems in order to increase the understandability, maintainability and reusability aspects of legacy software, and to simultaneously generate new UML documentation in order to help developers understand the changes being made.

This paper presents the research behind the design, as well as a technical overview of the implementation of TrueRefactor. We summarize the research goals that TrueRefactor addresses, and identify opportunities where it can be actively utilized.

1 INTRODUCTION

The drive for an automated refactoring tool, utilizing metrics-based code smell detection techniques, stems from the decrease in effort and time spent in manually performing these actions [21]. Currently, there are a number of refactoring tools available to aid developers [15], [11], [17], [16]. Most available tools provide developers with a list of specific refactoring techniques, but require the developer to manually select the code to be refactored [15]. Alternatively, the use of automated search-based refactoring techniques does not depend on developer supplied information and can result in incremental and unsupervised changes.

Traditional search based techniques rely on a suite of software quality metrics to determine the relevance of a mapping between a set of refactorings and their associated code [15]. Two prominent considerations for these techniques are determining the most effective suite of metrics to use for the refactoring fitness evaluation, and selecting the best search technique to explore the potential refactoring solution space. A review of the literature shows a specific focus on understanding the relationships between software quality metrics and their corresponding refactoring selection [16], [13]. Herein we present an approach based on machine learning and metric guided genetic algorithms to automate refactorings of legacy and application code. This approach is demonstrated through the implementation of TrueRefactor.

This paper is structured as follows. Section 2 presents the design and implementation of the TrueRefactor tool. It describes the underlying refactoring mechanism and describes the underlying genetic algorithm fitness function and search technique utilized. Section 3 provides a summary of the method of evaluation for TrueRefactor. Section 4 provides a brief analysis of results provided in Section 3. Section 5 describes the research goal and potential future research which can utilize techniques developed for TrueRefactor. Finally, in Section 6, we provide a conclusion and explore future work as it applies to enhancing the capabilities of TrueRefactor.

2 TRUEREFACTOR DESIGN

TrueRefactor allows a developer to automatically refactor and remove code smells from source code. The basic operation of TrueRefactor is exemplified using a data flow diagram, shown in Figure 1. The input parameter to TrueRefactor is a codebase directory. Each source code file in the directory is parsed and then used to create a control flow graph representing the entire structure of the software. Using this graph, an initial measurement of code smells is obtained, and for each code smell found, a refactoring sequence of algorithms (designed to remove the specific smell) is generated.

A genetic algorithm (GA) is then utilized to search for the best (fittest) sequence of refactorings that removes the highest number of code smells from the original source code. The solution space that the GA searches is comprised of the set of potential refactoring sequences. The best solution is the sequence whose ordering removes

the highest number of code smells. The sequences of refactorings are seeded during the initial code smell measurement process across the originally generated code graph. Decision algorithms are then invoked to create a sequence of refactorings designed to remove said code smells.

Each refactoring sequence potentially becomes a member of sequence of sequences which compose an individual or the population evaluated by the GA. Each individual is then provided a copy of the latest code graph (via an Object Pool [4]). The GA then uses the refactoring techniques associated with the code smells found in this individual to modify its copy of the graph. The modified graph is evaluated for potentially remaining code smells before software quality metrics are evaluated.

After each individual in a generation of the GA is evaluated, a subset of the best individuals is selected for retention. The members of this subset are then used to generate members of the next generation. Refactoring is considered complete when either the amount of remaining code smells has dropped below a threshold value, or when a set number of maximum iterations have been completed. Finally, once refactoring is complete, a new UML class diagram (based on the graph of the best individual set of refactoring sequences) is produced using the Eclipse Modeling Framework and saved to the output directory in XML Metadata Interchange format [24].

TrueRefactor has a basic graphical user interface. The user interface, shown in Figure 2, is simple and can be controlled by entering basic information such as the location of the source code to be refactored, and the output locations for refactored UML diagrams and log

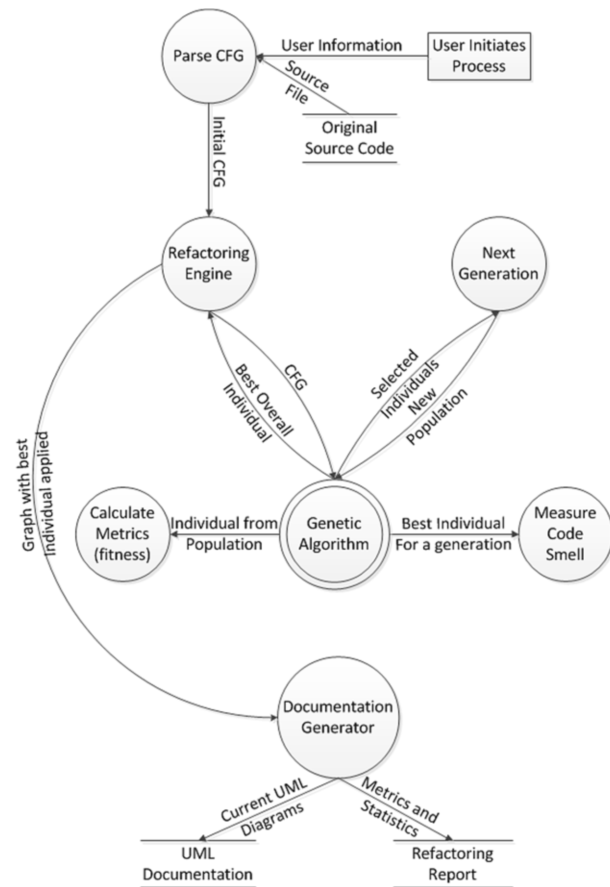

Figure 1. Basic Operation of TrueRefactor.

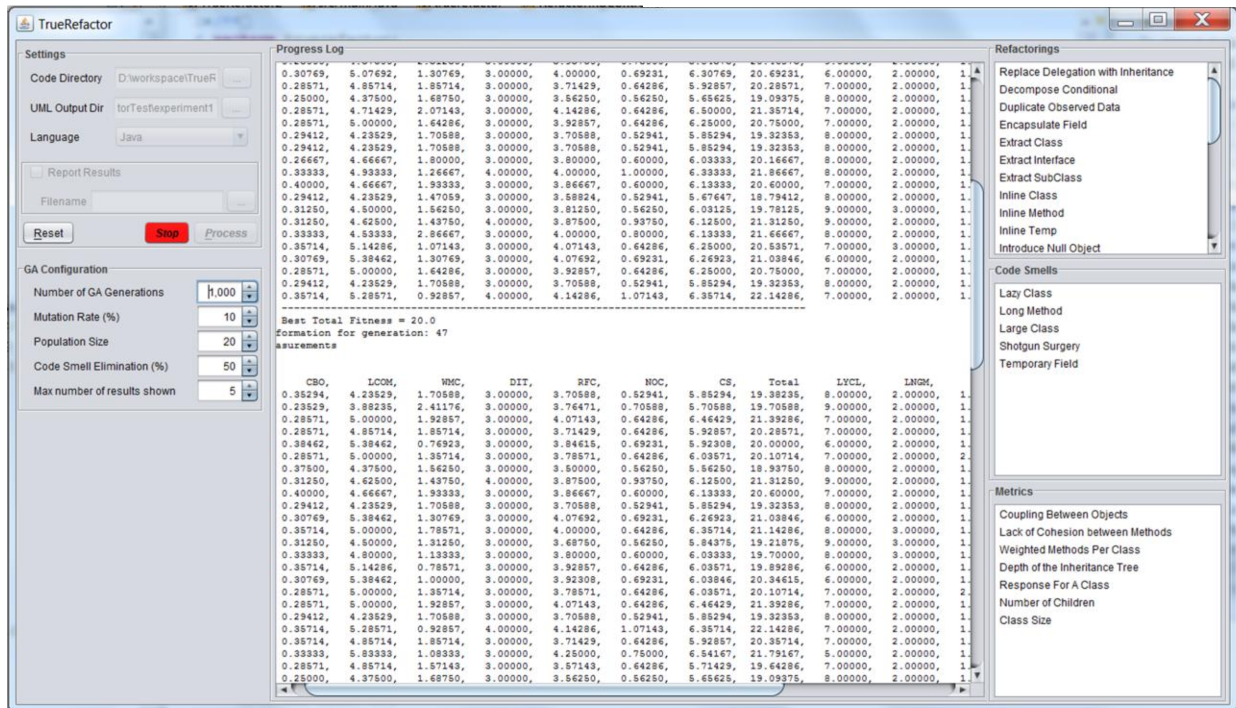

Figure 2. Screenshot of the TrueRefactor User Interface During Operation

files. Other parameters for the genetic algorithm can also be entered and/or modified via the graphical interface (in the GA Configuration section as shown in the middle of the left column of Figure 2), but if values are left blank or unchanged (from the defaults) the program will automatically analyze the source code in order to generate necessary parameters for the GA. The center pane of the graphical interface provides a dynamic log that tracks the operation of the GA. The information displayed is also stored in a more convenient form in the log output directory.

2.1 Code Smell Detection

We implemented code smell detection and decision algorithms for 5 code smells, described as follows: 1) *Lazy Class:* A class that does not do enough to justify its existence; 2) *Long Method:* A method that is very long, quite complex, and is responsible for more than one behavior; 3) *Large Class:* A class that attempts to take too much responsibility onto itself; 4) *Shotgun Surgery:* Condition when you must make a lot of small changes to a lot of classes in order to make a single large modification; and 5) *Temporary Field:* An instance variable whose use is dependent on a specific set of circumstances. A brief description of the code smell detection algorithms can be found in Table 2.

The decision algorithms were based on Fowler's description of which refactorings could be applied for a given code smell (see [8]). This generates a sequence of refactorings (if the pre- and post-conditions of each refactoring is met), which should resolve the code smell. Thus, the selection of refactorings is dependent on the code smells currently implemented.

2.2 Implemented Refactorings

TABLE I. REFACTORINGS SELECTED

Refactoring Level	Refactoring
Class-Level	Inline Class
	Collapse Hierarchy
	Extract Class
Method-Level	Move Method
	Extract Method
	Pull Up Method
	Push Down Method
Field-Level	Self-Encapsulate Field
	Encapsulate Field
	Move Field
	Pull Up Field
	Push Down Field

TABLE 2. CODE SMELLS DETECTION ALGORITHMS

Associated Code Smell	Algorithm Description
Lazy Class (LYCL)[a]	Uses a combination of CS, WMC, CBO, DIT, and number of methods to detect lazy class.
Temporary Field (TMPF)[a]	Uses the number of instance variables which are referenced by methods of the same class.
Long Method (LNGM)[b]	Uses a combination of WMC and CS to compare the method to determine if it is too long.
Large Class (LGCL)[b]	Uses CBO, WMC, CS and DIT to find a larger than normal class.
Shotgun Surgery (SHOSUR)[b]	Uses LCOM and CBO to find inter-class complexity to find this smell.

a. Descriptions based on algorithms found in [14]
b. Descriptions based on algorithms found in [19]

As an automated refactoring tool, TrueRefactor does perform the actual refactorings, but currently only supports the modification of UML rather than code. The modification of the source code is scheduled to be added later in the tool's development. The reasoning behind designing our own refactoring engine, instead of utilizing refactoring engines such as those in IDE's like eclipse, was to prevent reliance on or attachment to any specific tools. It also allows us the ability to utilize refactorings those engines do not include.

In order to utilize the decision algorithms associated with the code smell detection, a variety of refactoring techniques were needed. Each refactoring itself has a set of pre- and post-conditions which must be met. As a part of the refactoring algorithm design if these pre- and post-conditions are not met then the system will not continue with the refactoring and will not attempt to resolve the code smell. With code smell resolution in mind we have implemented 12 refactorings from the following three categories (see Table 1 for a concise list):

2.2.1 Class-Level Refactorings:

Inline Class: Replaces all instances of a class by moving the internals of that class where the instances are used; *Collapse Hierarchy:* Removes a non-leaf class from its inheritance hierarchy by moving its fields and methods up or down the hierarchy; *Extract Class:* Finds related components within a class C that are not related to any other components of C and forms a new class containing them.

2.2.2 Method-Level Refactorings:

Move Method: Moves a method from one class A to another class B; *Extract Method:* Takes a set of related statements in some method A and uses them to generate a

new method B; *Pull Up Method* and *Push Down Method:* Move a method up or down, respectively, in an inheritance hierarchy.

2.2.3 Field-Level Refactorings:

Self-Encapsulate Field and *Encapsulate Field:* Provides accessor and mutator for a field and declares the field as private, then replaces direct accesses to the field with calls to the generated methods; *Move Field:* Moves a field from a class C to a more appropriate class; *Pull Up Field* and *Push Down Field:* Moves a field up or down, respectively, within an inheritance hierarchy.

2.3 Fitness Evaluation

Unlike other similar tools, we have opted not to define the fitness evaluation of individuals based solely on software quality metrics. Although we are attempting to improve the understandability, maintainability, and reusability of the code by improving the readings of selected software quality metrics, TrueRefactor's primary goal is to remove source code smells. As the GA reduces the number of code smells, we also observe a general improvement in the software as measured by the quality metrics.

3 TOOL EVALUATION

This section presents a summary of the method used to evaluate TrueRefactor. In order to evaluate the refactoring capability and code smell removal functionality in TrueRefactor, we developed a test program. The program design included functionality to support a simple navigation tool for a virtual vessel in a 2D setting. It was designed such that allowing us to inject code smells in each experiment would a very quick and easy process. From this codebase we created three separate experimental variations of the source code that maintained the same functionality. For each variation of the source code, we injected code smells and ran TrueRefactor. We then compared the results of each experiment to the results of running the tool across the original unmodified code base. We found both, a decrease in the amount of code smells present in each experimental codebase, and reduced measurements in the metrics used as surrogates for comprehensibility (as measured through the surrogate product metrics).

Along with an evaluation of the decrease in both code smell and metrics values and the comparisons to the control group, we used UML comparison as well. The tool generates both an initial UML class diagram and a final (after refactoring) class diagram. By visually comparing and through analysis the final diagram can be used to determine whether the final product is less complex and more comprehensible.

Below are the results from one of the experiments. Figures 3-6 represents the results from the application of TrueRefactor across the experimental code base with codes smells injected.

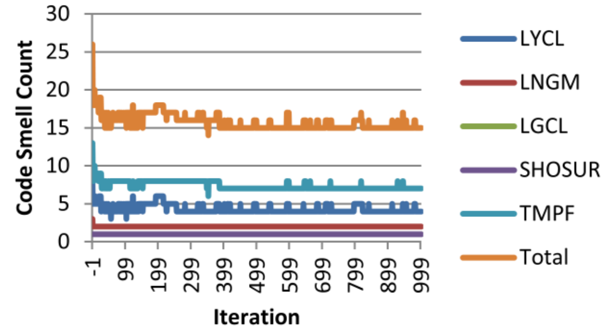

Figure 3. Code smell counts of the fittest individual of a population for each iteration of the genetic algorithm

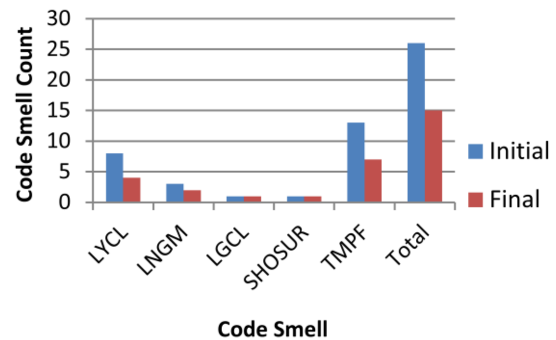

Figure 4. Comparison between smell counts before and after the genetic algorithm processes the source code

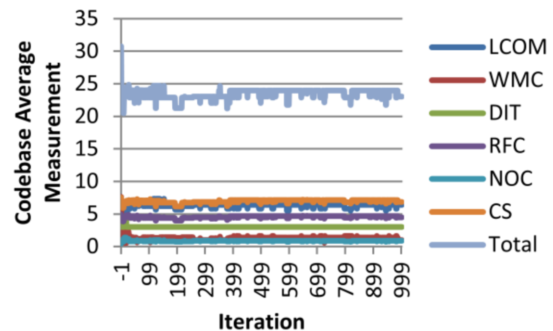

Figure 5. Average CK-metric values of the subject's codebase after every iteration of the genetic algorithm

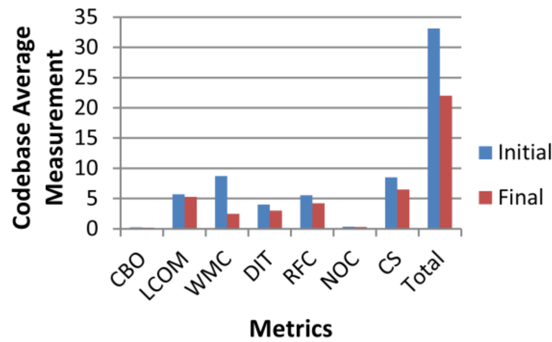

Figure 6. Comparison of initial and final metric measurements after the genetic algorithm processes the source code

4 ANALYSIS

In this section we describe and analysis the results of the experiment. In both Figures 3 and 5 the curves come down quickly and then flatten out, yet the GA continued until 1000 iterations was reached. This stems from a design decision in which 1000 iterations is the minimum amount of iterations and has been selected to help ensure that the best sequence of refactorings has been found. In Figure 4 the measured code smell at the initiation of the GA and at the end can be seen, with a reduction in overall code smell counts. Figure 6 shows a similar view concerning the comparision of initial and final measurements for the surrogate metrics. This data (along with the other experiments) leads us to the conclusion that the software is capable of removing code smell through search base refactoring. Yet, additional code smells and refactorings are still required to begin testing TrueRefactor on larger scale problems.

5 APPLICATIONS OF TRUEREFACTOR

In this section we investigate potential applications of TrueRefactor, related research and future plans.

5.1 Current Work

The goal of TrueRefactor is to provide a tool that can be used to explore the concept of automated refactoring as it applies to legacy software and engineering applications. Initially we wanted to explore the possibility of increasing the overall understandability and reusability (comprehensibility) of a software product. The improvement of the underlying quality aspects of the software is measured by removing code smells and observing a decrease in measured surrogate quality metrics. In order to test the abilities of TrueRefactor, a system was designed, and code smells were injected into the source code. During each experiment the tool performed adequately and was able to find and remove injected code smells. A full description of the experiment can be found in [9].

5.2 Exploring the Relationship between Metrics, Refactoring and Code Smells

Although Fowler and others [8], [14], [19], [13] have provided the underlying ideas connecting code smells and refactoring, the exact relationship connecting metrics, refactoring, and code smells together is still elusive. Utilizing the underlying techniques of TrueRefactor we plan to explore these relationships further to define new mapping possibilities as well as connections to other negative evolutionary side effects such as anti-patterns, increases in test requirements [10], increases in coupling, grime buildup, or decay in general.

5.3 Exploring Refactoring

The process of refactoring requires that an assumption be made –if the refactoring is carried out correctly then the behavior of the software will be preserved even though the static structure of the software is changed. In order for automated refactoring tools such as TrueRefactor to be generally accepted by industry and working developers, a rigorous mathematical framework (a set of mathematical relationships and definitions utilized to provide a succinct and unambiguous description of a given entity, relationship, or object) is needed. We would like to utilize this tool as a seminal base to begin the development of such a framework.

6 CONCLUSIONS AND FUTURE WORK

In this paper we have presented TrueRefactor, an automated refactoring tool. This tool currently works with the Java language and focuses on the elimination of code smells as a means to improve the comprehensibility of existing software. This tool also focuses on the improvement of the design of the software through the generation of improved UML class diagrams (as compared to the UML diagram generated for the code base prior to starting refactoring). This research is a step toward understanding that automated refactoring is currently limited if it is to be used as the only means to improve software. Manual intervention is still needed, and hybrid approaches continue to provide the best-in-class solutions. Facilitating the transition from initial code to modified code is an eventual goal of this project. Focusing on mitigating these automated difficulties will be an avenue for continued and future research.

In order to further enhance the viability of TrueRefactor and of automated refactoring in general, several key issues need to be addressed. First is the ability to handle multiple languages by developing and utilizing parsing technology which provides a sufficient and unified interface to the parse tree. Second, we need the ability to generate complete working code. Third, the ability to generate more detailed documentation of the automated changes so that developers can understand what changes the software has undergone. Finally, the ability to allow developers to control the removal of code smells from the code is required. Once these improvements have occurred the application of these methods to larger projects will help to validate and refine automated refactoring as a whole.

7 REFERENCES

[1] M. Afenzeller, S. Winkler, S. Wagner, and B. Andreas, Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications, Chapman & Hall/CRC Taylor & Francis Group, Boca Raton, FL , pp. 1–70, 2009.

[2] M. Bowman, L. C. Briand, and Y. Labiche, "Multi-Objective Genetic Algorithms to Support Class Responsibility Assignment," Proc. of the IEEE International Conference on Software Maintenance (ICSM 2007), pp. 124-133, 2007.

[3] F. Budinsky, D. Steinburg, E. Merks, R. Ellersick, and T. J. Grose, Eclipse Modeling Framework: A Developer's Guide, Pearson Education, Inc., Upper Saddle River, NJ, pp. 89-280, 2004.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons Ltd., Hoboken, NJ, pp. 25-193, 1996.

[5] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Trans. on Software Engineering, vol. 35(6), pp. 476-493, 1994.

[6] C. Chisalita-Cretu, "A Multi-Objective Approach for Entity Refactoring Set Selection Problem", Second International Conference on the Applications of Digital Information and Web Technologies, pp. 790–795, 2009.

[7] N. E. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, Revised, 2nd ed, PWS Publishing Co., Boston, MA, 1998.

[8] M. Fowler and K. Beck, Refactoring: Improving the Design of Existing Code, Addison-Wesley, New York, pp. 27–100, 2000.

[9] I. Griffith, S. Wahl and C. Izurieta, "Evolution of Legacy System Comprehensibility through Automated Refactoring," unpublished.

[10] C. Izurieta, J. M. Biemanm, "Testing Consequences of Grime Buildup in Object Oriented Design Patterns," 1st ACM-IEEE International Conference on Software Testing, ICST '08, Lillehammer, Norway, 2008.

[11] M. Harman and L. Tratt. "Pareto Optimal Search Based Refactoring at the Design Level," In Proceedings of the Conference on Genetic and Evolutionary Computation, 2007.

[12] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," The Journal of Systems and Software, vol. 23(2), pp. 111–112, doi: 10.1016/0164-1212(93)90077-B, 1993.

[13] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," Proc. 20th IEEE International Conference on Software Maintenance (ICSE '04), pp. 350- 359, 2004.

[14] J. Munro, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code," Proc. 11th IEEE International Software Metrics Symposium (METRICS '05), pp. 15 doi: /10.1109/METRICS.2005.38, 2005.

[15] M. O'Keeffe and M. O. Cinneide, "Search-Based Software Maintenance," Proc. of the 10th European Conference on Software Maintenance and Reengineering, CSMR 2006, pp. 10, doi: 10.1109/CSMR.2006.49, 2006.

[16] O. Raiha, "A Survey on Search-Based Software Design," Computer Science Review, 4(4):203-249, 2010.

[17] F. Otero et al., "Refactoring in Automatically Generated Programs," Proc. Symposium on Search Based Software Engineering, 2010.

[18] R. S. Pressman, Software Engineering: A Practitioner's Approach, 7th ed., McGraw-Hill, New York, pp. 613–44, 2010.

[19] N. Roperia, JSmell: A Bad Smell Detection Tool for Java Systems, UMI Microform 1466306, 2009.

[20] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 3rd ed., Pearson Education, Inc., Upper Saddle River, NJ, 2010.

[21] J. Schumacher, N. Zazworka, F. Shull, C. Seaman and M. Shaw, "Building Empirical Support for Automated Code Smell Detection," Proc. 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10), doi:/10.1145/1852786.1852797, 2010

[22] O. Seng, J. Stammel, and D. Burkhart, "Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems," Proc. Conference on Genetic and Evolutionary Computation, pp.1909-1916, 2006.

[23] N. Tsantali and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," IEEE Trans. on Software Engineering, vol. 35(3), pp. 347–367, 2009.

[24] XML Metadata Interchange, Object Modeling Group, 2005, <http://www.omg.org/spec/XMI/ISO/19503/PDF/>.

[25] Unified Modeling Language, Version 2.3, Object Modeling Group, 2010, <http://www.omg.org/spec/UML/2.3/>.

[26] Java Compiler Compiler (JavaCC) <http://javacc.java.net/>

[27] Eclipse Modeling Framework Project (EMF). <http://www.eclipse.org/modeling/emf/>.