Automated Refactoring of Software using Version History and a Code Element Recentness Measure

Michael Mohan and Des Greer
Department of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Northern Ireland, U.K.

Keywords: Search based Software Engineering, Maintenance, Refactoring, Software History, Multi-Objective Optimization, Genetic Algorithms.

Abstract: This paper proposes a multi-objective genetic algorithm to automate software refactoring and validates the approach using a tool, MultiRefactor, and a set of open source Java programs. The tool uses a metric function to measure quality in a software system and tests a second objective to measure the recentness of the code elements being refactored. Previous versions of the software project are analyzed and a recentness measure is then calculated with respect to previous versions of code. The multi-objective setup refactors the input program to improve its quality using the quality objective, while also focusing on the recentness of the code elements inspected. An experiment has been constructed to measure the multi-objective approach against an alternative mono-objective approach that does not use an objective to measure element recentness. The two approaches are validated using six different open source Java programs. The multi-objective approach is found to give significantly better recentness scores across all inputs in a similar time, while also generating improvements in the quality score.

1 INTRODUCTION

Search-Based Software Engineering (SBSE) has been used to automate various aspects of the software development cycle. Used successfully, SBSE can help to improve decision making throughout the development process and assist in enhancing resources and reducing cost and time, making the process more streamlined and efficient. Search-Based Software Maintenance (SBSM) is usually directed at minimizing the effort of maintaining a software product. An increasing proportion of SBSM research is making use of multi-objective optimization techniques. Many multi-objective search algorithms are built using genetic algorithms (GAs), due to their ability to generate multiple possible solutions. Instead of focusing on only one property, the multi-objective algorithm is concerned with a number of different objectives. This is handled through a fitness calculation and sorting of the solutions after they have been modified or added to. The main approach used to organize solutions in a multi-objective approach is Pareto. Pareto dominance organizes the possible solutions into different nondomination levels and further discerns between them by finding the objective distances between them in Euclidean space.

In this paper, a multi-objective approach is created to improve software that combines a quality objective with one that incorporates the use of numerous previous versions of the software code. The element recentness objective uses previous versions of the target software to help discern between old and new areas of code. It will investigate the refactored areas of code to give a value representing how recently these code elements have been added, using the previous versions of the software supplied. To test the effectiveness of the element recentness objective, an experiment has been constructed to test a GA that uses it against one that does not. It may be argued that it is more relevant to refactor the older elements of the code (for instance, if the code has been around longer, it has had a better chance to build up technical debt and become incompatible with its surroundings). However, it is important to note that the purpose of this experiment is not to support either stance. The more important aspects of the code may be different depending on the circumstances and the developer's opinion. The choice has been made in this paper to focus on more recent elements instead of older elements in order to test the effectiveness of the objective itself in doing what it aims, and the objective

Mohan, M. and Greer, D.
Automated Refactoring of Software using Version History and a Code Element Recentness Measure.
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 455-462
ISBN: 978-989-758-300-1

can be tweaked to focus one way or the other depending on the developers needs. In order to judge the outcome of the experiment, the following research questions have been derived:

RQ1: Does a multi-objective solution using an element recentness objective and a quality objective give an improvement in quality?

RQ2: Does a multi-objective solution using an element recentness objective and a quality objective refactor more recent code elements than a solution that does not use the element recentness objective.

In order to address the research questions, the experiment will run a set of tasks to compare a default mono-objective set up to refactor a solution towards quality with a multi-objective approach that uses a quality objective and the newly proposed element recentness objective. The following hypotheses have been constructed to measure success in the experiment.

H1: The multi-objective solution gives an improvement in the quality objective value.

$H1_0$: The multi-objective solution does not give an improvement in the quality objective value.

H2: The multi-objective solution gives significantly higher element recentness objective values than the corresponding mono-objective solution.

$H2_0$: There is no significant difference between the recentness objective value for the multi-objective and mono-objective approaches.

The remainder of this paper is organized as follows. Section 2 discusses related work and gives an overview of the previous studies in SBSM that have incorporated the use of software history. Section 3 describes the MultiRefactor tool used to conduct the experiment along with the searches, refactorings and metrics available in it. Section 4 explains the set up of the experiment used to test the element recentness objective. Section 5 analyses the results of the experiment, looking at the objective values and the times taken to run the tasks. Section 6 concludes the paper and discusses the significance of the findings.

2 RELATED WORK

A few other studies relating to SBSM have used version history of the target software to aid in refactoring. Pérez et al. (Pérez et al. 2013) proposed an approach that involved reusing complex refactorings that had previously been used. They aimed to mine the change history of the software project to find the refactorings used to fix design smells. The position paper introduced a plan to gather and compile the reusable refactorings in a structured way, in order to reapply them in the future. For this, they aimed to extend the ChEOPSJ system (Soetens and Demeyer 2012) and build a refactoring and design smell detector on top of it. They aimed to extend this system to find design smells that have been resolved, trace them back to the refactorings performed and reconstruct the refactoring order.

Ouni et al. (Ouni, Kessentini and Sahraoui 2013; Ouni et al. 2016) implemented an objective as part of a multi-objective solution to encourage refactorings that are similar to those already applied to similar code fragments in the past, by investigating previous versions of the code. They used the Ref-Finder tool (Kim et al. 2010) to find refactorings between versions of code. They also (Ouni, Kessentini, Sahraoui and Hamdi 2013) analyzed "co-change", an attribute that identifies how often two objects in a project are refactored together at the same time, as well as the number of refactorings applied in the past to the code elements. They updated their objective function to provide a value relating to a set of elements as an average of these three measures using refactoring history. An extended study from 2015 (Ouni et al. 2015) investigated the use of past refactorings from other projects to calculate the objective value when the change history for the applicable project is not available. Similarly, Tsantalis and Chatzigeorgiou (Tsantalis and Chatzigeorgiou 2011) have also used previous versions of software code to aid in the removal of design smells in the current code. They used the previous versions of the code to rank refactoring suggestions according to the number, proximity and extent of changes related with the corresponding code smells.

3 MULTIREFACTOR

The MultiRefactor approach[1] uses the RECODER framework[2] to modify source code in Java programs. RECODER extracts a model of the code that can be used to analyze and modify the code before the changes are applied. MultiRefactor makes available various different approaches to automated software maintenance in Java programs. It takes Java source code as input and will output the modified source code to a specified folder. The input must be fully compilable and must be accompanied by any

[1] https://github.com/mmohan01/MultiRefactor

[2] http://sourceforge.net/projects/recoder

necessary library files as compressed jar files. The numerous searches available in the tool have various input configurations that can affect the execution of the search. The refactorings and metrics used can also be specified. As such, the tool can be configured in a number of different ways to specify the particular task that you want to run. If desired, multiple tasks can be set to run one after the other.

A previous study (Mohan et al. 2016) used the A-CMA (Koc et al. 2012) tool to experiment with different metric functions but that work was not extended to produce source code as an output (likewise, TrueRefactor (Griffith et al. 2011) only modifies UML and Ouni, Kessentini, Sahraoui and Boukadoum's (Ouni, Kessentini, Sahraoui and Boukadoum 2013) approach only generates proposed lists of refactorings). MultiRefactor (Mohan and Greer 2017) was developed in order to be a fully-automated search-based refactoring tool that produces compilable, usable source code. As well as the Java code artifacts, the tool will produce an output file that gives information on the execution of the task including data about the parameters of the search executed, the metric values at the beginning and end of the search, and details about each refactoring applied. The metric configurations can be modified to include different weights and the direction of improvement of the metrics can be changed depending on the desired outcome.

MultiRefactor contains seven different search options for automated maintenance, with three distinct metaheuristic search techniques available. For each search type there is a selection of configurable properties to determine how the search will run. The refactorings used in the tool are mostly based on Fowler's list (Fowler 1999), consisting of 26 field-level, method-level and class-level refactorings, and are listed below.

Field Level Refactorings: Increase/Decrease Field Visibility, Make Field Final/Non Final, Make Field Static/Non Static, Move Field Down/Up, Remove Field.

Method Level Refactorings: Increase/Decrease Method Visibility, Make Method Final/Non Final, Make Method Static/Non Static, Remove Method.

Class Level Refactorings: Make Class Final/Non Final, Make Class Abstract/Concrete, Extract Subclass/Collapse Hierarchy, Remove Class/Interface.

The refactorings used will be checked for semantic coherence as a part of the search, and will be applied automatically, ensuring the process is fully automated. A number of the metrics available in the tool are adapted from the list of the metrics in the QMOOD (Bansiya and Davis 2002) and CK/MOOSE (Chidamber and Kemerer 1994) metrics suites. The 23 metrics currently available in the tool are listed below.

QMOOD Based: Class Design Size, Number Of Hierarchies, Average Number Of Ancestors, Data Access Metric, Direct Class Coupling, Cohesion Among Methods, Aggregation, Functional Abstraction, Number Of Polymorphic Methods, Class Interface Size, Number Of Methods.

CK Based: Weighted Methods Per Class, Number Of Children.

Others: Abstractness, Abstract Ratio, Static Ratio, Final Ratio, Constant Ratio, Inner Class Ratio, Referenced Methods Ratio, Visibility Ratio, Lines Of Code, Number Of Files.

In order to implement the element recentness objective, extra information about the refactorings is stored in the refactoring sequence object used to represent a refactoring solution. For each solution, a hash table is used to store a list of affected elements in the solution and to attach to each a value that represents the number of times that particular element is refactored in the solution. During each refactoring, an element, considered to be most relevant to that refactoring, is chosen and the element name is stored. After the refactoring has executed, the hash table is inspected. If the element name already exists as a key in the hash table, the value corresponding to that key is incremented to represent another refactoring being applied to that element in the solution. Otherwise, the element name is added to the table and the corresponding value is set to 1. After the solution has been created, the hash table will have a list of all the elements affected and the number of times for each. This information is used to construct the element recentness score for the related solution.

To improve the performance of the tool, the recentness scores are stored for each element as the search progresses in another hash table. This allows the tool to avoid the need to calculate the element recentness scores for each applicable element in the current solution at the beginning of the search task. Instead, the scores are calculated as the objective is calculated, for each element it comes across. If the element hasn't previously been encountered in the search, its element recentness value will be calculated and stored in the hash table. Otherwise, the value will be found by looking for it in the table. This eliminates the need to calculate redundant element recentness values for elements that are not refactored in the search and spreads the calculations throughout the search in place of finding all the values in the beginning.

4 EXPERIMENTAL DESIGN

In order to calculate the element recentness objective, the program will be supplied with the directories of all the previous versions of the code to use, in successive order. To calculate the element recentness value for a refactoring solution, each element that has been involved in the refactorings (be it a class, method or field) will be inspected individually. For each previous version of the code, the element will be searched for using its name. If it is not present, the search will terminate, and the element will be given a value related to how far back it can be found. An element that can be found all the way back through every previous version of code will be given a value of zero. An element that is only found in the current version of the code will be given the maximum element recentness value, which will be equal to the number of versions of code present. For each version the element is present in after the current version, the element recentness value will be decremented by one. Once this value is calculated for one element in the refactoring solution, the objective will move onto the next element until a value is derived for all of them. The overall element recentness value for a refactoring solution will be an accumulation of all the individual element values.

In order to evaluate the effectiveness of the element recentness objective, a set of tasks were set up that used the priority objective to be compared against a set of tasks that didn't. The control group is made up of a mono-objective approach that uses a function to represent quality in the software. The corresponding tasks use the multi-objective algorithm and have two objectives. The first objective is the same function for software quality used for the mono-objective tasks. The second objective is the element recentness objective. The metrics used to construct the quality function and the configuration parameters used in the GAs are taken from previous experimentation on software quality. Each metric available in the tool was tested separately in a GA to deduce which were more successful, and the most successful were chosen for the quality function. The metrics used in the quality function are given in Table 1. No weighting is applied for any of the metrics. The configuration parameters used for the mono-objective and multi-objective tasks were derived through trial and error and are outlined in Table 2. The hardware used to run the experiment is outlined in Table 3.

For the tasks, six different open source programs are used as inputs to ensure a variety of different domains are tested. The programs range in size from relatively small to medium sized.

These programs were chosen as they have all been used in previous SBSM studies and so comparison of results is possible. The source code and necessary libraries for all of the programs are available to download in the GitHub repository for the MultiRefactor tool.

Table 1: Metrics used in the software quality objective.

Metrics	Direction
Data Access Metric	+
Direct Class Coupling	-
Cohesion Among Methods	+
Aggregation	+
Functional Abstraction	+
Number Of Polymorphic Methods	+
Class Interface Size	+
Number Of Methods	-
Weighted Methods Per Class	-
Abstractness	+
Abstract Ratio	+
Static Ratio	+
Final Ratio	+
Constant Ratio	+
Inner Class Ratio	+
Referenced Methods Ratio	+
Visibility Ratio	-
Lines Of Code	-

Table 2: GA configuration settings.

Configuration Parameter	Value
Crossover Probability	0.2
Mutation Probability	0.8
Generations	100
Refactoring Range	50
Population Size	50

Each one is run five times for the mono-objective approach and five times for the multi-objective approach, resulting in 60 tasks overall.

Table 3: Hardware details for the experiment.

Operating System	Microsoft Windows 7 Enterprise Service Pack 1
System Type	64-bit
RAM	8.00GB
Processor	Intel Core i7-3770 CPU @ 3.40GHz

The inputs used in the experiment as well as the number of classes and lines of code they contain are given in Table 4. Table 5 gives the previous versions of code used for each input, in order from the earliest version to the latest version used (up to the current version being read in for maintenance). For each input, five different versions of code were used overall. Not all sets of previous versions contain all the releases between the first and last version

Table 4: Java programs used in the experiment.

Name	LOC	Classes
Beaver 0.9.11	6,493	70
Apache XML-RPC 3.1.1	14,241	185
JRDF 0.3.4.3	18,786	116
GanttProject 1.11.1	39,527	437
JHotDraw 6.0b1	41,278	349
XOM 1.2.1	45,136	224

Table 5: Previous versions of Java programs used in experiment.

Beaver	0.9.8	0.9.9	0.9.10	pre1.0 demo
Apache XML-RPC	2.0	2.0.1	3.0	3.1
JRDF	0.3.3	0.3.4	0.3.4.1	0.3.4.2
Gantt Project	1.7	1.8	1.9	1.10
JHotDraw	5.2	5.3	5.4b1	5.4b2
XOM	1.1	1.2b1	1.2b2	1.2

In order to find the element recentness score for the mono-objective approach to compare against the multi-objective approach, the mono-objective GA has been modified to output the element recentness score after the task finishes. At the end of the search, after the results have been output and the refactored population has been written to Java code files, the recentness score for the top solution in the final population is calculated. Then, before the search terminates, this score is output at the end of the results file for that solution. This way the scores don't need to be calculated manually and the element recentness scores for the mono-objective solutions can be compared against their multi-objective counterparts.

For the quality function the metric changes are calculated using a normalization function. This function causes any greater influence of an individual metric in the objective to be minimized, as the impact of a change in the metric is influenced by how far it is from its initial value. The function finds the amount that a particular metric has changed in relation to its initial value at the beginning of the task. These values can then be accumulated depending on the direction of improvement of the metric (i.e. whether an increase or a decrease denotes an improvement in that metric) and the weights given to provide an overall value for the metric function or objective. A negative change in the metric will be reflected by a decrease in the overall function/objective value. In the case that an increase in the metric denotes a negative change, the overall value will still decrease, ensuring that a larger value represents a better metric value regardless of the direction of improvement. The directions of improvement used for the metrics in the experiment are given in Table 1. In the case that the initial value of a metric is 0, the initial value used is changed to 0.01 in order to avoid issues with dividing by 0. This way, the normalization function can still be used on the metric and its value still starts off low. Equation 1 defines the normalization function, where $_m$ represents the selected metric, $_{C_m}$ is the current metric value and $_{I_m}$ is the initial metric value. $_{W_m}$ is the applied weighting for the metric (where 1 represents no weighting) and $_D$ is a binary constant (-1 or 1) that represents the direction of improvement of the metric. $_n$ represents the number of metrics used in the function. For the element recentness objective, this normalization function is not needed. The objective score depends on the relative age of the code elements refactored in a solution and will reflect that.

$$\sum_{m=1}^{n} D.W_m \left(\frac{C_m}{I_m} - 1\right) \quad (1)$$

The tool has been updated in order to use a heuristic to choose a suitable solution out of the final population with the multi-objective algorithm to inspect. The heuristic used is similar to the method used by Deb and Jain (Deb and Jain 2013) to construct a linear hyper-plane in the NSGA-III algorithm. Firstly, the solutions in the population from the top rank are isolated and written to a separate sub folder. It is from this subset that the best solution will be chosen from when the task is finished. Among these solutions, the tool inspects the individual objective values, and for each, the best objective value across the solutions is stored. This set of objective values is the ideal point $\overline{z} = (z_1^{max}), (z_2^{max}), \ldots, (z_M^{max})$, where (z_i^{max}) represents the maximum value for an objective, and an objective i = 1, 2, ..., M. This is the best possible state that a solution in the top rank could have. After this is calculated, each objective score is compared with its corresponding ideal score. The distance of the objective score from its ideal value is found, i.e. $(z_i^{max}) - f_i(x)$, where $f_i(x)$ represents the score for a single objective. For each solution, the largest objective distance (i.e. the distance for the objective that is furthest from its ideal point) is stored, i.e. $fmax(x) = max_{i=1}^{M}[(z_i^{max}) - f_i(x)]$. At this point each solution in the top rank has a value, $fmax(x)$, to represent the furthest distance among its objectives from the ideal point. The smallest among these values, $min_{j=0}^{N-1} fmax(x)$ (where N represents the number of solutions in the top rank), signifies the

solution that is closest to that ideal point, taking all of the objectives into consideration. This solution is then considered to be the most suitable solution and is marked as such when the population is written to file. On top of this, the results file for the corresponding solution is also updated to mark it as the most suitable. This is how solutions are chosen among the final population for the multi-objective tasks to compare against the top mono-objective solution.

For the element recentness objective, the recentness value of each element refactoring is calculated and then added together to get an overall score. Accumulating the score instead of getting an average recentness value avoids the solution applying a minimal number of refactorings in order to keep a low average and thus possibly yielding inferior quality improvements. Accumulating the individual values will encourage the solution to refactor as many recent elements as possible, and it will prioritize these elements, but it will also allow for older elements to be used if they improve the quality of the solution. Equation 2 gives the formula used to calculate the element recentness score in a refactoring solution using the hash table structure. $_m$ represents the current element, A_m represents the number of times the element has been refactored in the solution and R_m represents the recentness value for the element. $_n$ represents the number of elements refactored in the refactoring solution.

$$\sum_{m=1}^{n} A_m.R_m \qquad (2)$$

5 RESULTS

Fig. 1 gives the average quality gain values for each input program used in the experiment with the mono-objective and multi-objective approaches. In all of the inputs, the mono-objective approach gives a better quality improvement than the multi- objective approach. For the multi-objective approach all the runs of each input were able to give an improvement for the quality objective as well as look at the element recentness objective. For the mono-objective approach, the smallest improvement was given with GanttProject, and for the multi-objective approach, it was Apache XML-RPC. For both approaches, XOM was the input with the largest improvement. The mono-objective Beaver results were noticeable for having the most disparate range in comparison to the rest.

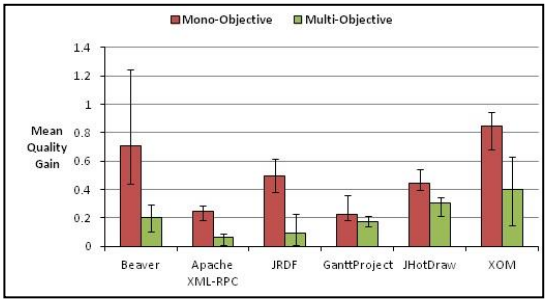

Figure 1: Mean quality gain values for each input.

Fig. 2 shows the average element recentness scores for each input with the mono-objective and multi-objective approaches. For all of the inputs, the multi-objective approach was able to yield better scores coupled with the recentness objective. The values were compared for significance using a one-tailed Wilcoxon rank-sum test (for unpaired data sets) with a 95% confidence level (α = 5%). The element recentness scores for the multi-objective approach were found to be significantly higher than the mono-objective approach. The scores tended to vary with both the mono-objective and multi-objective approaches. The exception to this in the XOM input which had a more refined set of results for both approaches. Also, for this input, in comparison to the others, the multi-objective approach didn't give as much of an improvement in the element recentness score in relation to its mono-objective counterpart. For the mono-objective GanttProject scores, one of the tasks gave an anomalous result of 784 (the other values were between 212 and 400) that was greater even than the average multi-objective score for the input, at 764.8.

Fig. 3 gives the average execution times for each input with the mono-objective and multi-objective searches. The times for the mono-objective and multi-objective tasks mostly mirrored each other. For most input programs, the mono-objective approach was faster on average, with the exception being Beaver which takes slightly longer. The Wilcoxon rank-sum test (two-tailed) was used again and the values were found to not be significantly different. The times seemed to increase in relation to the number of classes in the project, although the mono-objective GanttProject time was slightly smaller than JHotDraw, an input with fewer classes. The multi-objective GanttProject times stand out as taking the longest, with the longest task taking almost 71 minutes to run. The average time for the multi-objective GanttProject tasks was just under 64

minutes, whereas the average time for the next largest input, JHotDraw, was only 41 minutes. Whereas the inputs had similar times for the mono-objective and multi-objective approaches, for GanttProject the multi-objective tasks took quite a bit longer (over 28 minutes longer on average).

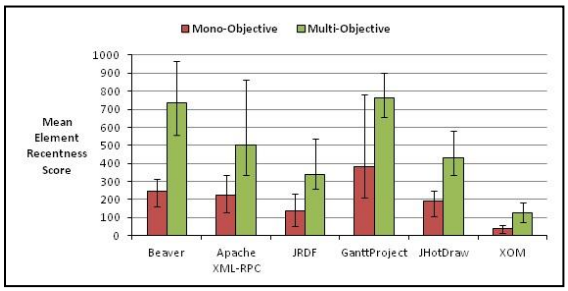

Figure 2: Mean element recentness scores for each input.

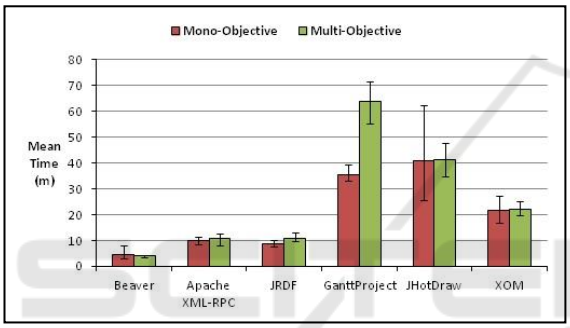

Figure 3: Mean times taken for each input.

6 CONCLUSION

In order to test the aims of the experiment and derive conclusions from the results a set of research questions were constructed. Each research question and their corresponding set of hypotheses looked at one of two aspects of the experiment. **RQ1** was concerned with the effectiveness of the quality objective in the multi-objective setup. To address it, the quality improvement results were inspected to ensure that each run of the search yielded an improvement in quality. In all 30 of the different runs of the multi-objective approach, there was an improvement in the quality objective score, therefore rejecting the null hypothesis. **RQ2** looked at the effectiveness of the element recentness objective in comparison with a setup that did not use a function to measure element recentness. To address this, a non-parametric statistical test was used to decide whether the mono-objective and multi-objective data sets were significantly different. The recentness scores were compared for the multi-objective approach against the basic approach and the multi-objective element recentness scores were found to be significantly higher than the mono-objective scores, rejecting the null hypothesis $\mathbf{H2_0}$. Thus, the research questions addressed in this paper help to support the validity of the element recentness objective in helping to focus refactorings on recent elements in a software program with the MultiRefactor tool, while in conjunction with another objective.

ACKNOWLEDGEMENTS

The research for this paper contributes to a PhD project funded by the EPSRC grant EP/M506400/1.

REFERENCES

Bansiya, J. & Davis, C.G., 2002. A Hierarchical Model For Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering.*, 28(1), pp.4–17. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=979986.

Chidamber, S.R. & Kemerer, C.F., 1994. A Metrics Suite For Object Oriented Design. *IEEE Transactions on Software Engineering.*, 20(6), pp.476–493.

Deb, K. & Jain, H., 2013. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Non-Dominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation.*, 18(4), pp.1–23. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6600851%5Cnhttp://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6600851.

Fowler, M., 1999. *Refactoring: Improving The Design Of Existing Code*,

Griffith, I., Wahl, S. & Izurieta, C., 2011. TrueRefactor: An Automated Refactoring Tool To Improve Legacy System And Application Comprehensibility. In *24th International Conference on Computer Applications in Industry and Engineering, ISCA 2011.*

Kim, M. et al., 2010. Ref-Finder: A Refactoring Reconstruction Tool Based On Logic Query Templates. In *International Symposium on Foundations of Software Engineering, FSE 2010.* pp. 371–372. Available at: http://dl.acm.org/citation.cfm?id=1882291.1882353.

Koc, E. et al., 2012. An Empirical Study About Search-Based Refactoring Using Alternative Multiple And Population-Based Search Techniques. In E. Gelenbe, R. Lent, & G. Sakellari, eds. *Computer and Information Sciences II.* London: Springer London, pp. 59–66. Available at: http://link.springer.com/ 10.1007/978-1-4471-2155-8 [Accessed December 3, 2014].

Mohan, M. & Greer, D., 2017. MultiRefactor: Automated Refactoring To Improve Software Quality. In *1st International Workshop on Managing Quality in Agile and Rapid Software Development Processes, QuASD 2017*. p. in press.

Mohan, M., Greer, D. & McMullan, P., 2016. Technical Debt Reduction Using Search Based Automated Refactoring. *Journal Of Systems And Software.*, 120, pp.183–194. Available at: http://dx.doi.org/10.1016/j.jss.2016.05.019.

Ouni, A. et al., 2015. Improving Multi-Objective Code-Smells Correction Using Development History. *Journal of Systems and Software.*, 105, pp.18–39. Available at: http://www.sciencedirect.com/science/article/pii/S0164121215000631.

Ouni, A., Kessentini, M., Sahraoui, H. & Boukadoum, M., 2013. Maintainability Defects Detection And Correction: A Multi-Objective Approach. *Automated Software Engineering.*, 20(1), pp.47–79.

Ouni, A. et al., 2016. Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study. *ACM Transactions on Software Engineering and Methodology.*, 25(3).

Ouni, A., Kessentini, M., Sahraoui, H. & Hamdi, M.S., 2013. The Use Of Development History In Software Refactoring Using A Multi-Objective Evolutionary Algorithm. In *Genetic and Evolutionary Computation Conference, GECCO 2013.* pp. 1461–1468. Available at: http://dl.acm.org/citation.cfm?doid=2463372.2463554.

Ouni, A., Kessentini, M. & Sahraoui, H., 2013. Search-Based Refactoring Using Recorded Code Changes. In *European Conference on Software Maintenance and Reengineering, CSMR 2013.* pp. 221–230.

Pérez, J., Murgia, A. & Demeyer, S., 2013. A Proposal For Fixing Design Smells Using Software Refactoring History. In *International Workshop On Refactoring & Testing, RefTest 2013.* pp. 1–4.

Soetens, Q.D. & Demeyer, S., 2012. ChEOPSJ: Change-Based Test Optimization. In *European Conference on Software Maintenance and Reengineering, CSMR 2012.*

Tsantalis, N. & Chatzigeorgiou, A., 2011. Ranking Refactoring Suggestions Based On Historical Volatility. In *15th European Conference on Software Maintenance and Reengineering, CSMR 2011.* pp. 25–34.