Using a Many-Objective Approach to Investigate Automated Refactoring

Using a Many-Objective Approach to Investigate Automated Refactoring

-M. Mohan, D. Greer

School of Electronics, Electrical Engineering and Computer Science, Queens University Belfast, BT71NN, UK

Abstract

Context: Software maintenance is expensive and so anything that can be done to reduce its cost is potentially of huge benefit. However, it is recognised that some maintenance, especially refactoring, can be automated. Given the number of possible refactorings and combinations of refactorings, a search-based approach may provide the means to optimise refactorings.

Objective: This paper describes the investigation of a many-objective genetic algorithm used to automate software refactoring, implemented as a Java tool, MultiRefactor.

Method: The approach and tool is evaluated using a set of open source Java programs. The tool contains four separate measures of software looking at the software quality as well as measures of code priority, refactoring coverage and element recentness. The many-objective algorithm combines the four objectives to improve the software in a holistic manner. An experiment has been constructed to compare the many-objective approach against a mono-objective approach that only uses a single objective to measure software quality. Different permutations of the objectives are also tested and compared to see how well the different objectives can work together in a multi-objective refactoring approach. The eight approaches are tested on six different open source Java programs.

Results: The many-objective approach is found to give better objective scores on average than the mono-objective approach and in less time. However, the priority and element recentness objectives are both found to be less successful in multi/many-objective setups when they are used together.

Conclusion: A many-objective approach is suitable and effective for optimising automated refactoring to improve quality. Including other objectives does not unduly degrade the quality improvements, but is less effective for those objectives than if they were used in a mono-objective approach.

Keywords: search-based software engineering, maintenance, automated refactoring, refactoring tools, software quality, many-objective optimisation, genetic algorithms.

1. Introduction

Search-Based Software Engineering (SBSE) attempts to apply search heuristics to solve complex problems in software development. SBSE has been used to automate various aspects of the software development cycle including software design, project management, software release planning, model verification and software testing [1].

Software maintenance is one of the most expensive phases of the software process, according to some taking up an estimated 75% [2] of development time. Refactoring is a special case where the structure of a software program is changed without affecting the functionality. While, refactoring may make software evolution easier, the value of the product in relation to its execution is often unchanged, from the perspective of a customer. Thus, minimising the cost of refactoring is desirable. Search-Based Software Maintenance (SBSM) treats the refactoring process as a combinatorial optimisation problem. The software code represents the search space of the problem and the refactorings can be applied across this search space to explore possible solutions. Metaheuristic search techniques can be used to seek optimal solutions within the search space, without resorting to an impossible exhaustive search.

Multi-objective algorithms are used where more than one objective function is to be optimised simultaneously. Usually approaches that generate multiple possible solutions, such as evolutionary algorithms, can be applied in such circumstances. Multi-objective algorithms have been sparsely used in SBSM, e.g. [3]. Numerous multi-objective search algorithms are built using genetic algorithms (GAs) due to their ability to generate multiple possible solutions. The main approach used to organise solutions in a multi-objective approach is Pareto dominance [4]. Pareto dominance organises the possible solutions into different non-domination levels and further discerns between them by finding the objective distances between them in Euclidean space. If there are more than three

objectives used, Pareto dominance becomes inefficient [3], [5], [6] as an increasingly larger fraction of the population becomes non-dominated and the solutions generated will be less diverse. For instance, Deb and Saxena [6] demonstrated that the NSGA-II approach is vulnerable to a large number of objectives. Where more than three objectives are to be optimised the algorithms have become known as many-objective [7]. These overcome the limitations of multi-objective algorithms using various techniques like objective reduction [7], alternative preference ordering relations [8], decomposition [9], use of a predefined multiple targeted search [10] and the incorporation of a decision maker's preferences [11].

In previous work [12], a quality objective was constructed to measure quality in a software program. The function itself is described later in section 4 but is made up of a weighted sum of the metrics shown in Table 1 as adapted from the list of metrics in the QMOOD [13] and CK/MOOSE [14] and other metric suites. Details of these can be found in previous work [12].

Table 1 – Available Metrics in the MultiRefactor Tool

QMOOD Based Metrics	CK Based Metrics	Others
Class Design Size	Weighted Methods Per Class	Abstractness
Number Of Hierarchies	Number Of Children	Abstract Ratio
Average Number Of Ancestors		Static Ratio
Data Access Metric		Final Ratio
Direct Class Coupling		Constant Ratio
Cohesion Among Methods		Inner Class Ratio
Aggregation		Referenced Methods Ratio
Functional Abstraction		Visibility Ratio
Number Of Polymorphic Methods		Lines Of Code
Class Interface Size		Number Of Files
Number Of Methods		

In addition to the main quality objective, three supplementary objectives are introduced. These are: i) a priority objective allowing user defined preferences for classes to be refactored; ii) a refactoring coverage objective where covering more areas of code is favoured and iii) an element recentness objective where newer elements are favoured for refactoring.

An experiment is set up to run all four objectives together and measure how successful they are as an overall framework for maintaining software. The objectives are also compared by using different permutations of them together and analysing whether certain combinations of objectives are more successful than others. In order to run the different multi-objective setups, an adaptation of the multi-objective algorithm NSGA-II [15] is used, and to run the four objectives in one many-objective solution, an adaptation of the many-objective algorithm NSGA-III [10] is used in its place. NSGA-III replaces the crowding distance functionality used in NSGA-II with an alternative approach to maintain the diversity in the chosen solutions. The adaptation of the algorithm used for experimentation is discussed in Section 4.

The experimentation is split into two parts. The first part is concerned with running the many-objective search with all 4 objectives and the mono-objective counterpart with just the quality objective to compare against. For the second part, different permutations of the three supplemental objectives are combined with the quality objective to see how they interact with each other. Each individual objective is tested with the quality objective in a multi-objective solution. Then, the different permutations of the objectives are tested with each other and the quality objective in a three-objective search. Overall, there are six different permutations to test along with the mono-objective and many-objective variations inspected in part 1 of the experimentation. In all cases the quality objective is present as part of the process, in order to improve the state of the code itself while allowing the other objective(s) to work in conjunction with it. In order to judge the outcome of the experimentation, the following research questions have been formulated.

RQ1: Does a many-objective solution using the priority, refactoring coverage and element recentness objectives with the quality objective give an improvement in quality?

RQ2: Does a many-objective solution using the priority, refactoring coverage and element recentness objectives with the quality objective have a better effect on the 3 objectives than a mono-objective solution that only uses the quality objective?

RQ3: Which combination of objectives in conjunction with the quality objective work best together?

The following null hypotheses have also been constructed to measure success in the first part of the experimentation (for part 2, the objective scores in the different permutations will be compared to see which combinations are most successful for each supplementary objective):

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

$H2_0$: The many-objective solution does not give higher values for the priority, refactoring coverage and element recentness objectives than the corresponding mono-objective solution.

Overall, the paper tests these hypotheses via the MultiRefactor tool, which is proposed for fully automated maintenance of Java software using mono-objective, multi-objective and many-objective search techniques. The main contribution in this is to demonstrate a *many-objective* search technique to include an objective for *quality*, an objective measuring the *priority of the classes* refactored, an objective for *code coverage* and an objective for *recentness of the code elements* refactored. The findings are that it is possible to improve quality objective value while also improving all three of the other objectives. This work advances related work in [16] which uses three objectives, namely minimising the number of bad-smells, maximising the use of development history and maximising semantic coherence, making use of NSGA-II. In the MultiRefactor approach presented here, we demonstrate that four objectives can be incorporated in a many-objective approach making use of NSGA-III.

In the remainder of this paper is organised as follows. Section 2 discusses related work. Section 3 describes the MultiRefactor tool used to conduct the experimentation along with the searches, refactorings and metrics available along with the modifications made to the tool to implement the objectives and the many-objective search used. Section 4 describes the setup for the experimentation. Section 5 outlines the results while Section 6 analyses and discusses these. /Section 7 concludes the paper with some discussion of the significance of the work.

2. Related Work

The term SBSE was first coined by Harman and Jones in 2001 [17]. Further research in the area was identified in addition to open problems in 2007 [18]. Clarke et al. [19] discussed ways to apply metaheuristic search techniques to software engineering problems and proposed other aspects of software engineering for which they might apply in 2003. Another review, this time concerning SBSE papers in Brazil, gives useful statistics of the impact of researchers from the country on the area [20]. There are literature reviews on the subject [21]–[23], as well as more specific literature reviews focusing on project management [24], [25], software design [26], testing [27], [28] and maintenance [1]. Numerous tools have been proposed that can automate the maintenance process of software refactoring to some extent [29]–[37], although many are limited, and not all are fully automated. Many of the proposed tools isolate design smells in the code using detection rules [29]–[34]. Other tools use metrics to determine ideal refactorings to make to the code that will improve the quality and remove design smells as a by-product of the process [35]–[37].

Using a SBSE approach, refactorings are applied stochastically to the original software solution and then the software is measured using a fitness function consisting of one or more software metrics. There are various metric suites available to measure characteristics like cohesion and coupling, but different metrics measure the software in different ways and thus how they are used will have a different effect on the outcome. The CK[14] and QMOOD [13] metric suites have been designed to represent object-oriented properties of a system as well as more abstract concepts such as flexibility. Metrics can be used to measure single aspects of quality in a program or multiple metrics can be combined to form an aggregate function. The common approach uses metric weights to denote which heuristics are more important so they can be combined into a weighted sum (although this weighting process is often subjective). The weighting process may be appropriate since there is a possibility of metrics conflicting with each other. For instance, one metric may cause inheritance depth to be improved but may increase coupling between the objects. Another method is to use Pareto fronts[4] to measure and compare solutions and have the developer choose which solution is most desirable, depending on the trade-offs allowed. A Pareto front will indicate a set of optimal solutions among the available group and will allow the developer to compare the different solutions in the subset according to each individual objective (i.e. metric) used.

In the solution, refactorings are applied and then the program is measured to compare the quality with the previously measured value. If the new solution is improved according to the software metrics used, this becomes the new solution to compare against. This approach is followed over a number of iterations, causing the software solution to gradually increase in quality until an end point is reached and an optimal (or near optimal) solution is generated. The end point can be triggered by various conditions such as the number of iterations executed or the amount of time passed. The particular approach used by the search technique may vary depending on the type of search-based approach chosen, but the general method consists of iteratively making changes to the solution, measuring the quality of the new solution, and comparing the solutions to progress towards an optimal result.

Several studies in SBSM have used GAs. Van Belle and Ackley [38] introduced an experiment to test the adaptability of a genetic program. Vivanco and Pizza [39] used a parallel GA to select the most suitable maintainability metrics from a group. Fatiregun et al. [35] explored program transformations by experimenting with and comparing different search techniques. The results showed in these cases that the GA performed significantly better than the other searches. Seng et al. [40] introduced an evolutionary algorithm to apply refactorings to a program model and tested it on an open source Java program. Lange and Mancoridis [41] used a GA to perform author identification in a software project. O'Keeffe and Ó Cinnéide [42], [43] conducted an empirical comparison of a GA with two other metaheuristic techniques, hill climbing and simulated annealing. Jensen and Cheng [44] adapted the work of Ó Cinnéide on design patterns [45] by creating a system to optimise the design of a program represented by UML diagrams, and to introduce design patterns into the system design. The tool used genetic programming to find the best design options with the QMOOD suite. Kessentini et al. [46], [47] used bad design examples to produce rules to aid in design defect detection, and then used the rules in a GA to help propose sequences of refactorings to remove the detected defects. This approach was then extended [48] by measuring the similarity between good design examples and the subject code with the GA.

More recent research has explored the use of multi-objective techniques. White et al. [49] used a multi-objective approach to attempt to find a trade-off between the functionality of a pseudorandom number generator and the power consumption necessary to use it. De Souza et al. [50] investigated the human competitiveness of SBSE techniques in four areas of software engineering, and used mono-objective and multi-objective GAs in the study. Ouni et al. [51] created an approach to measure semantics preservation in a software program when searching for refactoring options to improve the structure, by using the NSGA-II search. Ouni et al. [5], [16], [52] then explored the potential of using development refactoring history to aid in refactoring a software project by using NSGA-II. In that paper a multi-objective approach is described where a measure is made of 'co-change', being how often two objects in a project were refactored together at the same time. They also measure the number of changes applied in the past to the objects and explore the effect of using refactoring history on semantics preservation. Their experimentation showed a slight improvement in quality values and semantics preservation with these additional considerations. In another study [53], Ouni et al. combined this approach in a four objective configuration that also aimed to minimise the number of code changes necessary to fix defects. Ouni et al. [54] also expanded upon the code smells correction approach of Kessentini et al. [47] by replacing the GA used with NSGA-II. Wang et al. also expanded on the approach of Kessentini et al. by combining the detection and removal of software defects with an estimation of the number of future code smells generated in the software by the refactorings. Ouni et al. [55] investigated the use of a chemical reaction optimization algorithm to explore the benefits of this approach for SBSM. They compared the algorithm against 3 other search algorithms including a GA. Mkaouer et al. [56], [57] experimented with combining quality measurement with robustness using NSGA-II to create solutions that could withstand volatile software environments. Mkaouer et al. [3], [58], [59] also used the successor algorithm to NSGA-II, NSGA-III, to experiment with automated maintenance. These studies only suggest refactoring sequences to be applied, and do not check the applicability of the refactorings.

3. Refactoring Tool Support

3.1 Supplementary Objectives

The first of the supplementary objectives proposed is a **priority** objective. We propose that it would be helpful to classify classes into a list of "priority" classes and "non-priority" classes in order to focus on the refactoring solutions that have refactored the priority classes and give less attention to the non-priority. There are a number of scenarios that justify this approach. Each member of a development team may be concerned with certain aspects of functionality and has interest in a subset of classes in the program. Further, there may be certain classes which are considered more 'risky' or in some other way more worthy of attention. Certain parts of the code considered having been less well attended to or rushed in implementation. Conversely, there may be some classes considered less suitable for refactoring. A developer may be unsure about some areas of the code and so prefer not to modify those parts. Older more established code might be considered already very stable, possibly having been refactored

extensively in the past, where refactoring might be considered an unnecessary risk. Changing code also necessitates redoing integration and unit tests and this could be another reason for leaving certain parts of the code as they were. There may also be cases where 'poor quality' has been accepted as a necessary evil. For example, while there are design patterns and approaches available for handling crosscutting concerns such as logging and security, these are often met via scattered code, creating overall low cohesion and high coupling. Refactoring these elements may be deemed undesirable since the situation is largely understood and accepted.

However, it is not desirable to exclude less favoured classes from the refactoring process completely, since an overall higher quality code base may be achieved if some of those are included in the refactorings. The priority objective takes count of the classes used in the refactorings of a solution and uses that measurement to derive how successful the solution is at focusing on priority classes and avoiding non priority classes. The refactorings themselves are not restricted so during the refactoring process the search is free to apply any refactoring available, regardless of the class being refactored.

The second of the supplemental objectives is a **refactoring coverage** objective. The justification of including this as a secondary objective is to reflect a developer preference for more diverse refactoring. Since refactoring is about improving code, it makes sense that as many in the development team benefit from this and if the team are split by area of code, the coverage objective should ensure that more diverse areas are affected and so more developers benefit from the improvement in quality. Refactoring coverage favours a refactoring solution to be formed following an inspection of as many areas of the code as possible and avoiding redundant refactorings or solutions that focus too heavily on a single area. The objective has been constructed to inspect the refactoring solutions generated as part of the genetic search, and rank their fitness by analysing the refactorings applied and calculating a coverage score. The coverage score will be determined by two factors. The more elements inspected within a refactoring solution, the better the score will be. These elements include classes, methods and fields/variables. For each refactoring, a single element will be chosen to correspond to it, where class level refactorings will choose the relevant class, and likewise, method and field level refactorings will choose the relevant method or field. The number of distinct elements corresponding to the refactorings in a solution can be calculated this way and therefore it can be determined which solution looks at more. The second factor inspected will be the number of times each element is refactored. The smaller the average number of refactorings for each element in a solution, the better the score will be. This way, the score will minimise the effect of solutions with a larger number of refactorings and encourage the solution to focus less on a specific element or group of elements. This will also minimise the occurrence of redundant refactorings in a solution and allow each refactoring to have meaning in the solution by dispersing the refactorings across the different elements.

The third supplemental objective measures **element recentness**. The justification for including a recentness aspect is that, whereas older elements have been given the chance to be tested more and are more likely to have been updated, newer elements will not have had as much attention. Additionally, newer elements may be more likely to cause issues, especially if a software project has been established and the new functionality has had to be fitted into the current design (as is usually the case). Generally, a programmer may be more interested in testing the code that they have added to a project to ensure there are no unexpected issues caused by its presence. Thus, it can be argued intuitively that the more recent aspects of the code are more suitable candidates for refactoring than older aspects. The element recentness objective uses previous versions of the target software to help discern between old and new areas of code. In order to calculate the 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 1. 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 recentness values.

3.2 MultiRefactor

The MultiRefactor approach[1], in common with those of Moghadam and O' Cinnéide [36] and Trifu et al. [30], uses the RECODER framework[2] to modify source code in Java programs. RECODER extracts a model of the code (represented using an Abstract Syntax Tree) that can be used to analyse and modify the code before the changes are applied and written to file. MultiRefactor combines mono-objective search optimization algorithms like simulated annealing with a multi-objective GA to make 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 necessary library files as compressed jar files. This is a requirement in order to use the RECODER framework and to ensure that any applied refactorings do not break the semantics of the code, for example, if the code is already broken. 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.

A previous study [60] used the A-CMA [37] tool to experiment with different metric functions but that work was not extended to produce output source code (likewise, TrueRefactor [34] only modifies UML and Ouni et al.'s [54] approach only generates proposed lists of refactorings). MultiRefactor was developed in order to be a fully-automated search-based refactoring tool that produces compilable, usable code. As well as the Java code artefacts, 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 (i.e. whether an increase or a decrease denotes an improvement in a metric) can be changed depending on the desired outcome. Figure 1 overviews the MultiRefactor process. Briefly, different configurations can be created and specified for different search techniques. A configuration is made up of the refactorings and details of the metrics chosen. The output from the process is the refactored java code and also details of the search parameters used, the refactorings applied and the fitness scores achieved. There are a few ways that the metric functions can be calculated. An overall metric value can be found using a weighted metric sum or Pareto dominance can be used to compare individual metrics within the functions.

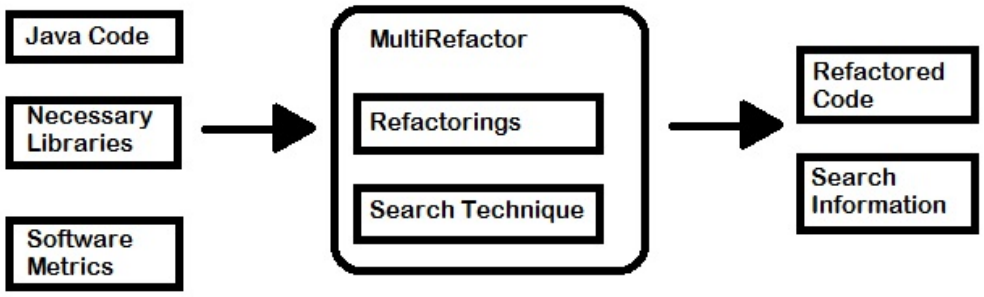

Figure 1 – Overview of the MultiRefactor Process

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 types of search available are detailed in Table 2. The refactorings used in the tool are mostly based on Fowler's list [61], consisting of 26 field-level, method-level and class-level refactorings, as listed in Table 3. In MultiRefactor, each refactoring has a similar structure. Each will relate to a specific program element type (e.g. most field level refactorings will be concerned with global field declarations in a class). A method is used to find the number of elements in a source code file that are applicable for that type of refactoring. This will use a method, '*mayRefactor*', to deduce whether a program element can be refactored. It will make all the relevant semantic checks, tailored for each refactoring. For example, in the "Make Field Final" refactoring, the method checks whether the field is already final or whether it is part of an enum. It also checks whether the field is initialised more than once in the program. If any of these conditions are true, then the method will return false and the refactoring is inapplicable. For "Make Method Static", the method checks whether the method is native, static, abstract, a constructor or part of an interface. It also checks whether there are any methods that redefine or implement the given method, and whether the method contains a reference to any other fields or methods within

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

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

the class that are not static. Thus, the refactorings to be used will be checked for semantic coherence as part of the search, and only if applicable will they be applied automatically. The metrics in the tool measure the current state of a program and are used to assess whether an applied refactoring has had a positive or negative impact. Due to the multi-objective capabilities of MultiRefactor, the metrics (Table 1) can be measured as separate objectives in order to be more precise in measuring their effect on a program.

Table 2 – Available Search Techniques in MultiRefactor Tool

Search Type	Search Variation
Random Search	N/A
Hill Climbing Search	First-Ascent
	Steepest-Ascent
Simulated Annealing	N/A
Genetic Algorithm	Simple Mono-Objective GA
	NSGA-II
	NSGA-III

Table 3 – Available Refactorings in MultiRefactor Tool

Field Level	Method Level	Class Level
Increase Field Visibility	Increase Method Visibility	Make Class Final
Decrease Field Visibility	Decrease Method Visibility	Make Class Non Final
Make Field Final	Make Method Final	Make Class Abstract
Make Field Non Final	Make Method Non Final	Make Class Concrete
Make Field Static	Make Method Static	Extract Subclass
Make Field Non Static	Make Method Non Static	Collapse Hierarchy
Move Field Down	Move Method Down	Remove Class
Move Field Up	Move Method Up	Remove Interface
Remove Field	Remove Method	

MultiRefactor makes use of a *Refactoring Sequence Object* to store the refactorings as they are applied. When the GA is run, the number of available elements in a file is first found by applying the mayRefactor method for every applicable element in the file. Then one of those refactorable elements is chosen at random. The RECODER framework is used to apply the changes to the chosen element. The change applied may be a single change or for more complex refactorings could be a number of changes. Those changes can be categorised as adding an element to a parent element, removing an element from a parent element, or replacing one element with another in the model. Additionally, new elements can be created e.g. new imports may need to be added when moving an element to a new class. As an aside, refactorings can also be reversed in the case where this is needed such as where hill climbing and simulated annealing are being used.

For some refactorings a decision must be taken on how the refactoring is applied. The Move Field Down and Move Method Down are examples of this, since here the subclass to move to has to be selected before the refactoring is applied. In cases where the mayRefactor method finds only one applicable subclass the choice is straightforward, otherwise one is selected and stored randomly. The Increase/Decrease Visibility refactorings are used to change a global field declaration or method declaration to public, protected, package or private visibility. Increase Visibility moves the visibility from public down towards private and Decrease Visibility moves the visibility from private towards public. Each application of the refactoring will move the visibility of the element up or down by one level. The Make Final/Non Final refactorings will either apply or remove the final keyword from a local/global field declaration, method declaration or class declaration. For each of the elements the keyword has a different meaning. For a field it means that the field can't be given a different value after it has been instantiated. For a method it means the method can't be redefined elsewhere, which therefore forbids a final method from also being abstract. For a class, it means the class can't have any subclasses. Likewise, the Make Static/Non Static refactorings are concerned with adding or removing the static keyword from a global field declaration or method declaration. In both cases a static element will be an element that can be called outside of a class without an instance of that class needing to be created. Also, Make Class Abstract/Concrete will add or remove the abstract keyword from a class declaration, allowing or forbidding it from containing abstract methods and forbidding or allowing it to be instantiated as its own class (instead of a subclass needing to be instantiated in its stead). The Move Down/Up refactorings are applied to global field declarations or method declarations and will either move the element to its superclass or to one of its available subclasses. Collapse Hierarchy is applied by taking all the elements of a class (except any existing constructors for that class) and moving them up into the

superclass. It will then remove the class from the hierarchy. The Remove refactorings will remove the element related to that type of refactoring.

The GA used is similar to that in [42]. First, an initial population of solutions will be created from the input by applying numerous refactorings at random to create divergent models. In order to avoid issues with memory storage, multiple different copies of the model are not stored in this process. Instead, a Refactoring Sequence Object will store all the information necessary to reconstruct a model from scratch with the initial program input being used as a starting point. The refactoring sequence for a solution will store the necessary information for each applied refactoring in the solution. As long as they are reapplied successively from the same starting point, the solution can be reconstructed at any later point in the search. The fitness value of each genome in the population will be calculated and stored.

Once refactoring sequences have been constructed for the initial population of genomes in the search, the crossover and mutation processes can be applied to create different offspring solutions and the search can begin. Crossover is represented in the search by combining genomes in the population to create new offspring with different sequences of applied refactorings. Each time the crossover process is run, two parent genomes are passed in and two children are produced from their refactoring sequences. In order to choose the parent genomes from the population, a selection operator is necessary, the method used here being that of is rank selection. Rank selection gives the genomes with better fitness a larger chance of being chosen, resulting in offspring that are more likely to have better fitness values. A cut and splice method, as illustrated in Figure 2, is used to combine the parents to generate different solutions.

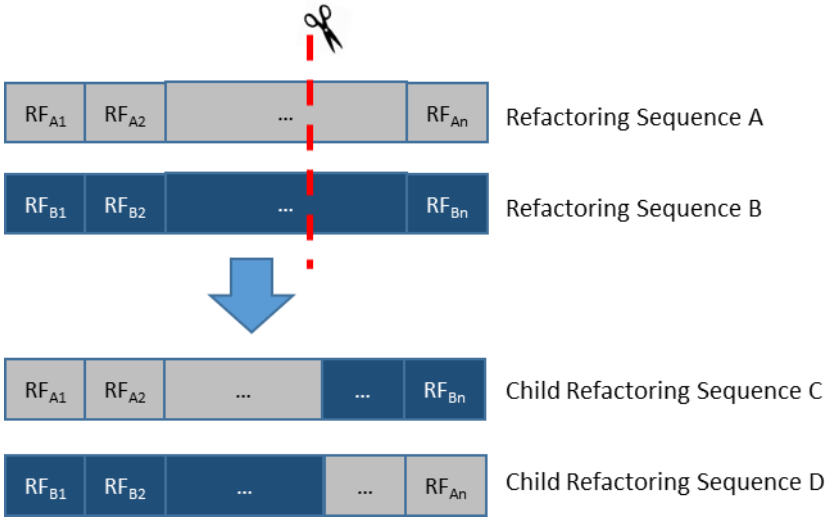

Figure 2– Crossover in MultiReFactor

Once crossover is complete, the mutation operator can be applied. It will be applied a random number of times similarly to the crossover operator (also depending on a probability input), except this time it isn't guaranteed to happen at least once. The operator will be applied to one of the newly generated solutions, chosen at random. The mutation process will first consist of reconstructing the model for the chosen genome. Then, a single random refactoring is applied at the end of the refactoring sequence for that genome, and the fitness level is measured in order to rank the new genomes in the population at the end of the generation. Once the mutation process is complete for a generation, the new offspring are added to the current population and the solutions are ordered according to fitness. As the fitness value of each solution is calculated when it is constructed/mutated, the genomes do not need to be reconstructed and measured at this point. The list of solutions will be sorted with the fittest solutions at the beginning of the list. During the sorting process, only the desired number of solutions will be added to the list truncating it to the desired population size and eliminating the weakest solutions. The updated, sorted population will then be passed on to the next generation and the GA will continue until the desired number of generations is executed. Figure 3 shows a sample output from the program along with sample Beaver v0.9.11 code followed by its refactored version with changes highlighted.

```
======== Search Information ========
Search: Multi-Objective Genetic Algorithm
Generations: 100
Population Size: 50
Crossover Probability: 0.200000
Mutation Probability: 0.800000

======== Initial Metric Info ========
Fitness function 1 score: 0.000000
Fitness function 2 score: 0.000000
Fitness function 3 score: 0.000000

======== Applied Refactorings ========
Iteration 1: "Decrease Method Security" applied at class GrammarSymbol to method GrammarSymbol from package to protected
Iteration 2: "Move Method Down" applied to method toString from GrammarSymbol to Terminal
Iteration 3: "Move Field Up" applied to field found from null to RuleWalker
Iteration 4: "Make Method Static" applied at class GrammarBuilder to method visit
Iteration 5: "Remove Method" applied at class IntArray to method compact

etc...

Iteration 51: "Decrease Method Security" applied at class Terminal to method Terminal from package to protected

Time taken to refactor: 225.34s

======== Final Metric Info ========
Fitness function 1 score: 0.082432
Fitness function 2 score: 0.080304
Fitness function 3 score: 0.075050

This solution has the highest score for fitness function 3 in the final population
```

Original Beaver 0.9.11 code:

```java
public abstract class GrammarSymbol{
    /** This symbol's ID */
    public short id;

    /** Name of the symbol */
    public final String name;

    /** The type of data held by this symbol. */
    public String type;

    /** Number of times this symbol is referenced in productions */
    public int nrefs;

    GrammarSymbol(String name){
        this.name = name;
    }

    GrammarSymbol(String name, String type){
        this.name = name;
        this.type = type;
    }

    public String toString(){
        return name;
    }
}
```

Sample Refactored Beaver 0.9.11 code:

```java
public abstract class GrammarSymbol {
    /** This symbol's ID */
    public short id;

    /** Name of the symbol */
    public final String name;

    /** The type of data held by this symbol. */
    public String type;

    /** Number of times this symbol is referenced in productions */
    public int nrefs;

    protected GrammarSymbol(String name){
        this.name = name;
    }

    GrammarSymbol(String name, String type){
        this.name = name;
        this.type = type;
    }

    public static final Comparator NUMBER_OF_REFERENCES_COMPARATOR = new Comparator(){
        public int compare(Object sym1, Object sym2)        {
            return ((GrammarSymbol) sym2).nrefs - ((GrammarSymbol) sym1).nrefs;
        }
    };
}
```

```
public class Terminal extends GrammarSymbol {
    static public final class Associativity {
        static public final Associativity LEFT = new Associativity("LEFT");
        static public final Associativity RIGHT = new Associativity("RIGHT");
        static public final Associativity NONE = new Associativity("NONE");

        private final String name; Associativity(String name){
            this.name = name;
        }

        public String toString(){
            return name;
        }
    }
```

Figure 3– Sample output from MultiRefactor

3.3 Implementing Many-Objective

In order to implement the refactoring coverage objective, extra information about the refactorings is stored in the *refactoring sequence object* used to represent a complete refactoring solution. The refactoring sequence object at its simplest contains a list of refactorings to be applied. To implement secondary objectives further information needs to be added to this. In the case of the coverage objective, a list of affected elements is also stored along with the number of times that particular element is refactored in the solution. In implementing the element recentness objective, the previous versions of the software are specified in a text file in order from oldest to most recent. Each version of the software is supplied with a specification of where the code is in relation to the home directory. The versions used can be picked out non-successively among a set of available versions in a repository as long as they are ordered. The projects themselves, like the current version, need to include the java code, any necessary jar files and be compilable in order to be read into the tool successfully. Otherwise, obsolete code is more likely to be considered.

In order to conduct the experimentation and test the four objectives together, a many-objective search algorithm has been implemented. The many-objective algorithm adapts the NSGA-III [10] upgrade of NSGA-II [15]. As with the multi-objective NSGA-II adaptation, the many-objective algorithm is built using the GA and differs mainly in the fitness process. The NSGA-III adaptation uses the same configuration parameters as the multi-objective algorithm and the original GA, along with the initial refactoring number. The fitness process replaces the crowding distance computations with an approach that uses reference points to choose between solutions in the same rank and maintain diversity in a population. Therefore, the crowding distance of each solution in a population no longer needs to be calculated, but the reference points on a normalised hyperplane need to be computed and the solutions themselves need to be normalised each generation in relation to the current population. All solutions already added to the population as well as the final rank of solutions in which to select the remaining set is used in the normalisation of the solutions (by finding the ideal point and the extreme points). The number of reference points used depends on the specified population size. As shown by Deb and Jain [10], in lieu of reference points being supplied preferentially by the user, the adaptation applied the systematic approach used by Das and Dennis [62].

4. Experimental Design

In part 1 of the experimentation, the mono-objective approach is compared with the many-objective search using all four objectives. Part 2 tests each different combination of objectives with the quality objective. Table 4 shows the six different permutations that are tested. The metrics used to construct the quality function and the configuration parameters used in the GAs are taken from previous experimentation on software quality [12]. 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 5. 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 testing and are outlined in Table 6. The hardware used to run the experiment is outlined in Table 7.

Table 4 – Different Combinations of Objectives Tested in Experiment 2

Q-P	Quality	Priority	-
Q-C	Quality	Refactoring Coverage	-
Q-R	Quality	Element Recentness	-
Q-P-C	Quality	Priority	Refactoring Coverage
Q-P-R	Quality	Priority	Element Recentness

Q-C-R Quality Refactoring Coverage Element Recentness

Table 5 – Metrics Used in 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 6 – GA Configuration Settings

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

Table 7 – Hardware Details for Experimentation

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

For the tasks, six different open source programs are used as inputs to ensure a variety of different domains and sizes are tested. These programs were chosen as they have all been used in previous SBSM studies and so there is an increased ability to understand the results and also because they promote different software structures and sizes. Beaver is a parser generator. Apache XML-RPC is a Java implementation of XML-RPC that uses XML to implement remote procedure calls. JRDF is a Java library for parsing, storing and manipulating RDF (Resource Description Framework). GanttProject is a tool for project scheduling and management. JHotDraw is a two dimensional graphics framework for structured drawing editors. Finally, XOM is a tree based API for processing XML. The source code and necessary libraries for all of the programs are available to download in the GitHub repository for the MultiRefactor tool. The inputs used in the experiment as well as the amount of classes and lines of code they contain are given in Table 8.

Table 8 – Java Programs used in Experimentation

Name	Domain	LOC	Classes
Beaver 0.9.11	Parser generator	6,493	70
Apache XML-RPC 3.1.1	B2B Communications	14,241	185
JRDF 0.3.4.3	Semantic Web (Resource management)	18,786	116
GanttProject 1.11.1	Project Management	39,527	437
JHotDraw 6.0b1	Graphics Tool	41,278	349
XOM 1.2.1	XML Tool	5,136	224

Table 9 gives the previous versions of code used for each input, in conjunction with the element recentness objective, 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. For part 1 of the experimentation each of the six inputs is run ten times for the mono-objective approach and ten times for the many-objective approach. In part 2, the tasks are run five times for each input in each of the six approaches. This results in there being 120 tasks for part 1 and 180 tasks for part 2..

Table 9 – Previous Versions of Java Programs used in Experimentation

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

In order to implement the priority objective, the important classes need to be specified in the refactoring tool. Nested classes can be included and read in by discerning them the same way packages are supplied. Using the '\' character as a delimiter, the set of outer classes can be supplied in order to inform what classes a nested class is within in a file, or what package a class is within in a project. With the list of priority classes and, optionally, non priority classes and the list of affected classes in each refactoring solution, the priority objective score can be calculated for each solution. To calculate the score, the list of affected classes for each refactoring is inspected, and each time a priority class is affected, the score increases by 1. This is done for every refactoring in the solution. Then, if a list of non-priority classes is also included, the affected classes are inspected again. This time, if a non priority class is affected, the score decreases by 1. The higher the overall score for a solution, the more successful it is at refactoring priority classes and avoiding non priority classes. It is important to note that, for the non priority classes, they are not necessarily excluded completely, although the solutions that do not involve those classes will be given priority.

For the tasks in the experiment that use the priority objective, both priority classes and non-priority classes are specified for the relevant inputs. The number of classes in the input program is used to identify the number of priority and non-priority classes to specify, so that 5% of the overall number of classes in the input are specified as priority classes and 5% are specified as non-priority classes. In order to choose which classes to specify, the number of methods in each class of the input was found and ranked. The top 5% of classes that contain the most methods are the priority classes and the bottom 5% that contain the least methods are the non-priority classes for that input. Using the top and bottom 5% of classes means that the same proportion of classes will be used in the priority objective for each input program, minimising the effect of the number of classes chosen in the experiment. In lieu of a way to determine the priority of the classes their complexity, as derived from the number of methods present, is taken to represent priority.

For the refactoring coverage objective, the number of elements refactored is counted and then divided by the average number of times each element is refactored in order to get an overall score. This allows the refactoring coverage objective to take into account both the number of elements refactored and the number of times an element is refactored. The score will prioritise solutions that have refactored as many elements as possible. Equation 1 gives the formula used to calculate the coverage score in a refactoring solution, where m represents the current element, A_m represents the number of times the element has been refactored in the solution and n represents the number of elements refactored in the refactoring solution.

$$n^2/(\sum_{m=1}^{n} A_m) \quad (1)$$

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 individual values will encourage the solution to refactor as many recent elements as possible, and it will prioritise 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 where m represents the current element, A_m represents the number of times the element has been refactored in the solution, R_m represents the recentness value for the element and n represents the number of elements refactored in the refactoring solution.

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

For the quality objective the metric changes are calculated using a normalisation function. This function causes any greater influence of an individual metric in the objective to be minimised, 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 and the weights given to provide an overall value for the metric function. A negative change in the metric will be reflected by a decrease in the overall function 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 quality objective are given in Table 5. 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 normalisation function can still be used on the metric and its value still starts off low. Equation 3 defines the normalisation function, where C_m is the current metric value and I_m is the initial metric value. W_m is the applied weighting for the metric and D is a binary constant that represents the direction of improvement of the metric. n represents the number of metrics used in the function.

$$\sum_{m=o}^{n} D . W_m \left(\frac{C_m}{I_m} - 1\right) \quad (3)$$

In order to give a better balance between the supplemental objectives, each of them have been normalised as well. The scores given with the refactoring coverage objective and the element recentness objective will be between 0 and 1. The priority objective, if using non priority classes (which in this experimentation it is), will give a score between -1 and 1 (otherwise it would also be between 0 and 1). The priority score becomes a ratio over the number of classes refactored in a solution i.e. that maximum possible priority score for that refactoring solution. Similarly, the coverage score is given as a ratio over the maximum score it could be for the respective refactoring solution. In this case, the maximum value is the number of distinct refactored elements divided by 1 (if each element was refactored only once). The calculation of this ratio is streamlined to be more efficient. For the element recentness objective, the average recentness value per element is calculated by dividing the original score by the number of elements refactored. This is then divided by the maximum possible element recentness value for an element to give a ratio between 0 and 1.

In order to find the other objective scores with the mono-objective approach to compare it against the other approaches and see how the other objectives fare when they are not being used in the search, the mono-objective GA has been modified to output the other scores 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 scores for all three of the other objectives are calculated for the top solution in the final population. Then, before the search terminates, these scores are output at the end of the results file for that solution. This way the scores don't need to be calculated manually for the mono-objective approach. With the multi and many-objective approaches, the solution chosen to compare against the mono-objective approach and with each other is taken from the top rank of solutions in the final population. The solution used is the one with the highest quality improvement value among those in the top rank. This is done in order to avoid the possibility that the search will minimise the number of refactorings to increase the ratio values for these objectives, to the detriment of the actual software quality improvement. The

solution is marked as the ideal solution when the population is written to file. On top of this, the results file for the corresponding solution is also updated and marked as the top ranked solution.

5. Results

Figure 4 gives the average quality gain values for each input program used in the experimentation with the mono-objective and many-objective approaches. The vertical 'error bars' show the maximum and minimum quality gains achieved in each case. In all of the inputs, the mono-objective approach gives a better quality improvement than the many-objective approach. This is largely to be expected, but what is interesting is that for many of the programs the quality gain is still very healthy, regardless of the competing objectives in the many objective approach. For the many-objective approach all the runs of each input were able to give an improvement for the quality objective as well as address the other three objectives. For both approaches, the smallest improvement was given with Apache XML-RPC, closely followed by GanttProject. The input with the largest improvement in both cases was XOM. This corresponds with this being the largest project in terms of LOC and of number of classes (Table 8), possibly providing more opportunity for refactoring. This view is somewhat countered in that Beaver has the next highest gain but the smallest number of classes. The simplest explanation is probably that the quality gains possible are due to the structure of the code in the first place.

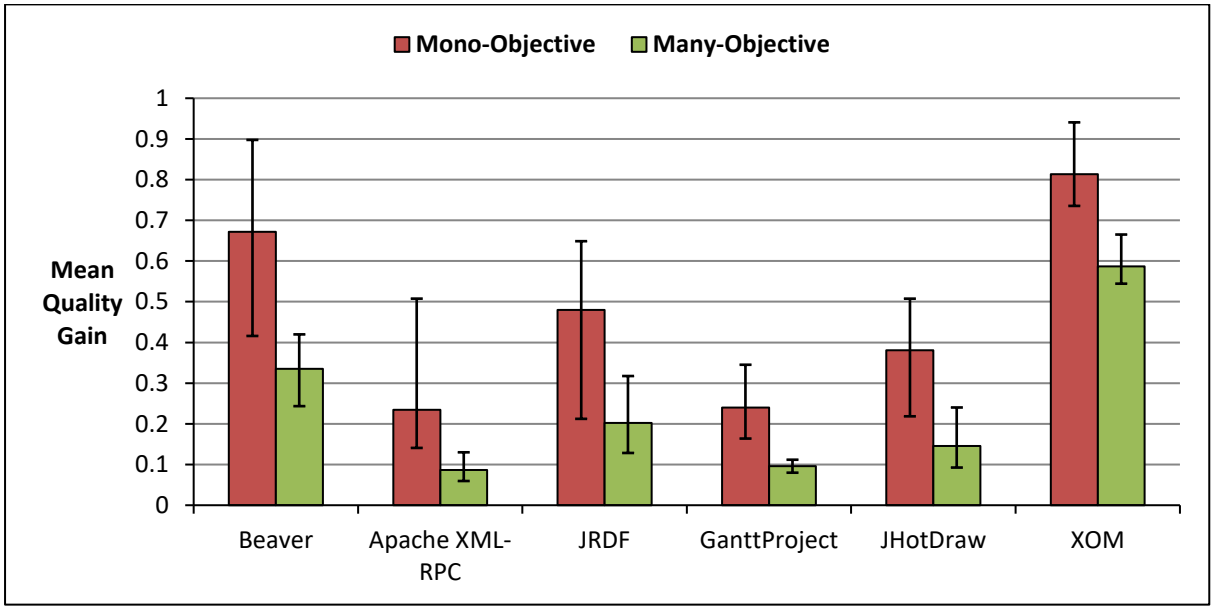

Figure 4 – Mean Quality Gain Values for each Input

Figure 5 shows the average priority scores for each input with the mono-objective and many-objective approaches. The vertical 'error bars' show the maximum and minimum priority scores achieved in each case. For all but one of the inputs, the many-objective approach was able to yield better scores coupled with priority objective. For JRDF though, the mono-objective approach was better. This is a surprising result. Indeed a few of the inputs had scores below zero. The mono-objective scores for JRDF, GanttProject and JHotDraw went below zero as well as the many-objective scores for JRDF. Here, the lowest scores for both approaches were found in JRDF, whereas the highest were yielded with XOM. Although most of the inputs yielded improved scores for the many-objective approach, it seems the priority scores are restricted for each input. To try to explain this we need to remember that the intention of priority is to allow a choice to the developer on which classes to prioritise and which to disfavour. Our experiment simulated this by choosing the top 5% of classes with the most methods as priority classes and the bottom 5% as non-priority classes. Thus, while we have demonstrated that it is possible to improve the priority objective using a many objective approach, the explanation of why some programs e.g. XOM have high gains while others e.g. JRDF have only low gains remains part of future work.

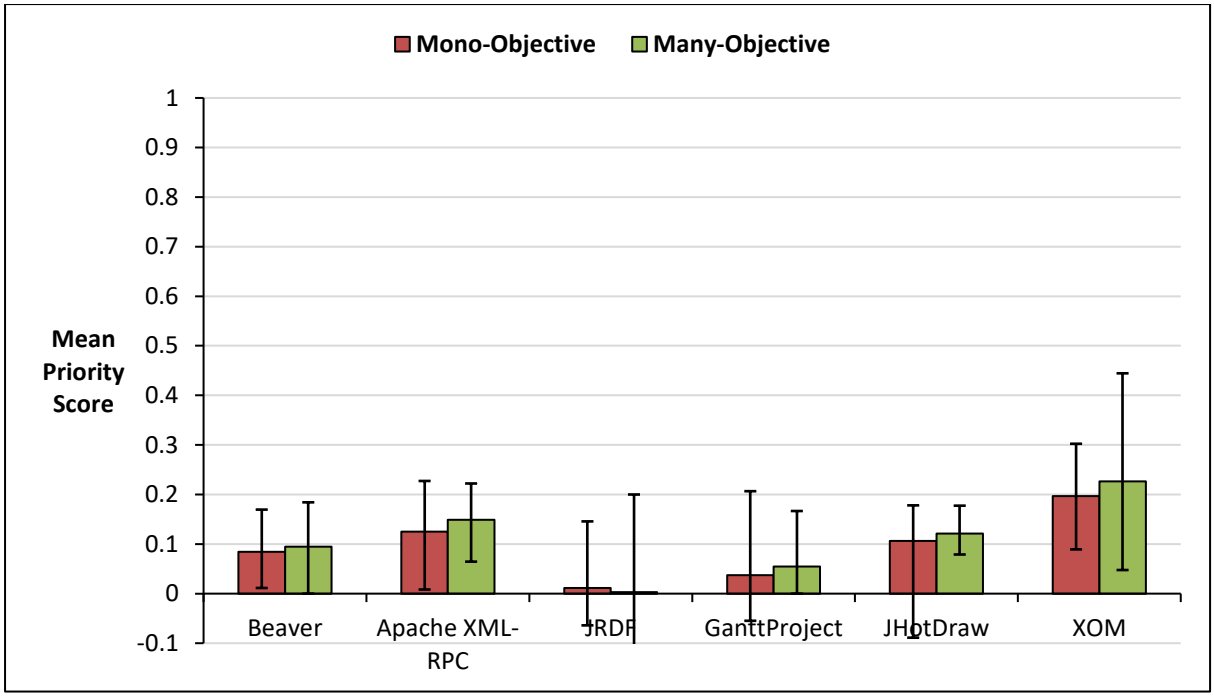

Figure 5– Mean Priority Scores for each Input

Figure 6 shows the average coverage scores for each input with the mono-objective and many-objective approaches. Here the vertical 'error bars' show the maximum and minimum coverage scores achieved in each case. For all of the inputs, the many-objective approach was able to yield better scores coupled with the refactoring coverage objective. The scores seemed to vary slightly less with the multi-objective approach compared to the mono-objective counterparts. This is likely due to the mono-objective coverage scores being more likely to be achieved as a by-product of the quality objective, leading to more fluctuating sets of scores among the tasks. On the other hand, the refactoring coverage objective used in the multi-objective approach will drive the solutions towards more diverse sets of refactorings, pushing the coverage scores towards the maximum possible value. When in comparison with the coverage scores given in the many-objective approach, there was also a lot more variability among the mono-objective scores. For the mono-objective approach, the lowest score was with JRDF whereas the highest was with JHotDraw. With the many-objective approach, Apache XML-RPC was lowest and GanttProject was highest, although four of the input programs contained runs where the score was the maximum value of 1.

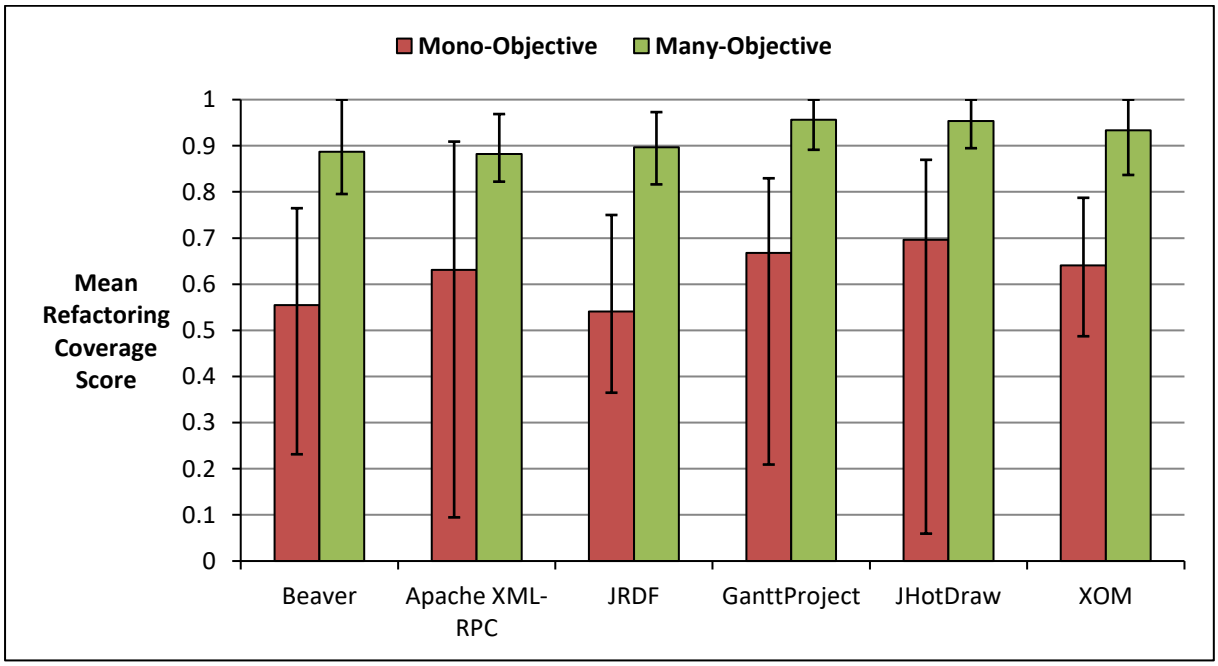

Figure 6 – Mean Refactoring Coverage Scores for each Input

Figure 7 shows the mean element recentness scores for each input with the mono-objective and many-objective approaches. The vertical 'error bars' here show the maximum and minimum element recentness scores achieved in each case. For each input program, the scores between each approach were closely tied. For all but one of the inputs, the many-objective approach was able to yield better scores coupled with the recentness objective, although for JRDF, it did not. For both approaches, the highest values were given with the Beaver input and the lowest were given with XOM. There was generally a higher range of values with the mono-objective approach, particularly with Beaver, GanttProject and JHotDraw.

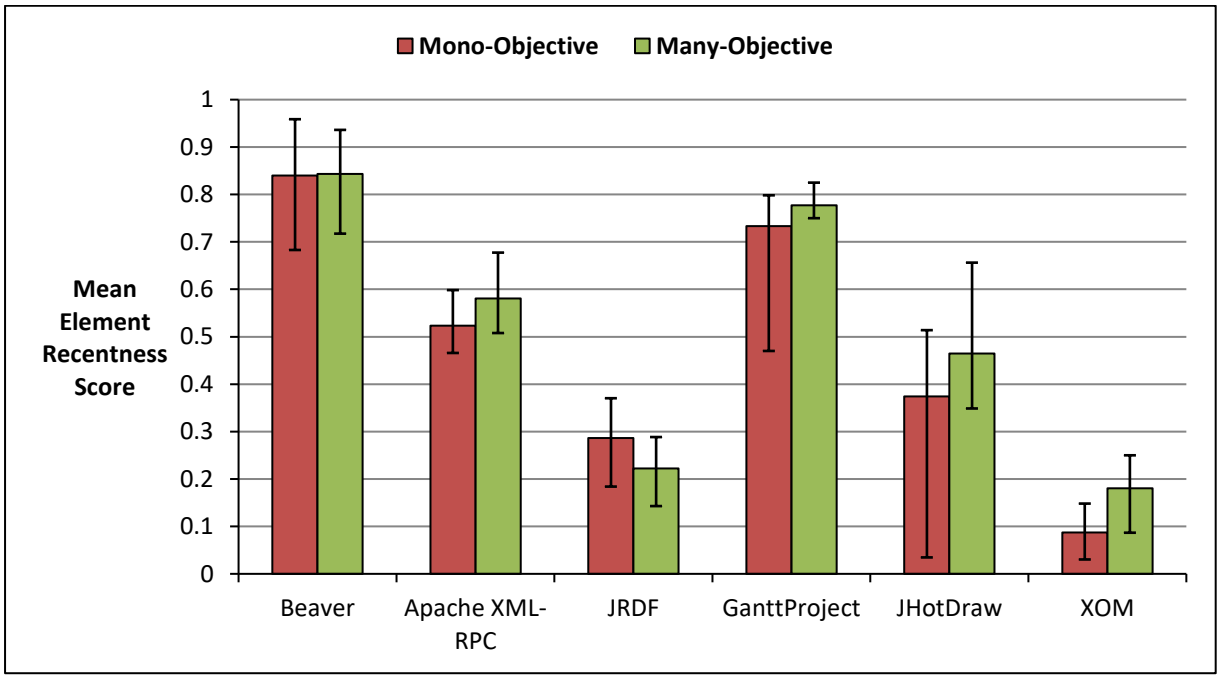

Figure 7– Mean Element Recentness Scores for each Input

Figure 8 gives the average execution times for each input with the mono-objective and many-objective searches with the vertical 'error bars' showing the maximum and minimum times in each case. The times for the mono-objective and many-objective tasks are similar, but in all cases, the many-objective approach was faster on average. The times generally increased as the number of classes in the project increased. Therefore, the times for GanttProject were longest and the tasks for JHotDraw took longer than XOM despite XOM being the largest program in terms of lines of code. The input program with the smallest number of classes, Beaver, took the shortest amount of time to run the tasks for both approaches. The longest time taken by a task was 44 minutes and 53 seconds by the mono-objective approach with GanttProject. On the other hand, the shortest task, with the many-objective approach using the Beaver input, was 2 minutes and 29 seconds. The larger programs had a greater variability in times than the smaller ones, probably because there is more opportunity for variation in refactorings applied.

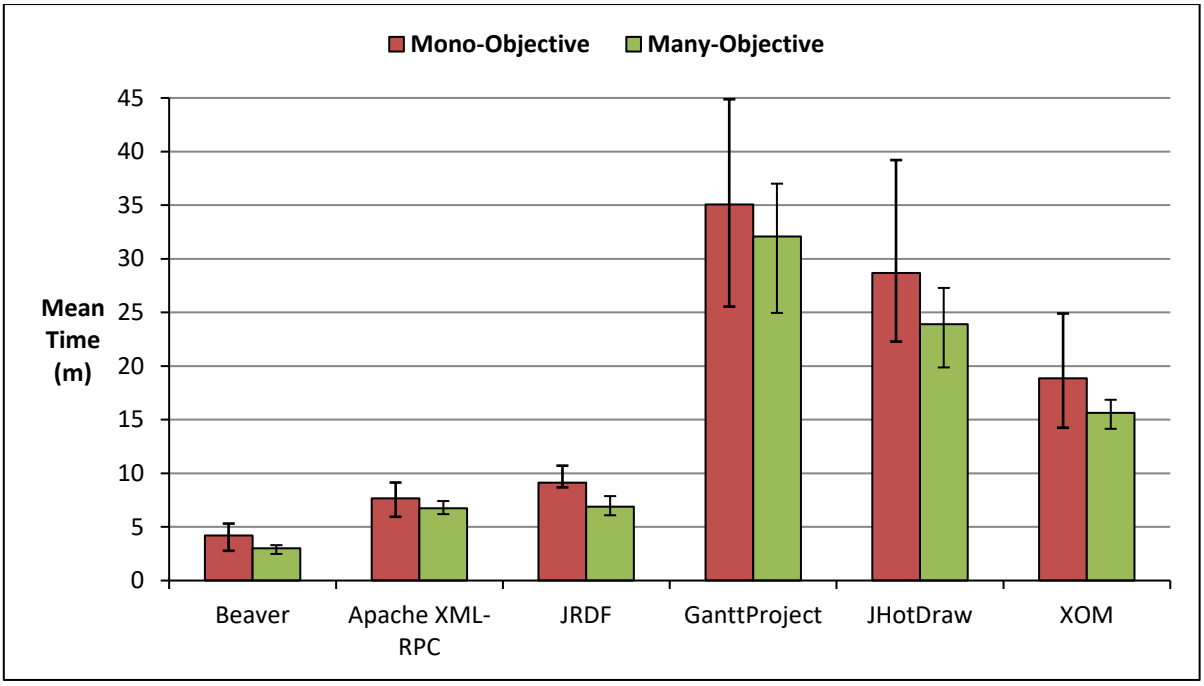

Figure 8 – Mean Times Taken for each Input

Figure 9 gives the average quality gain values for each input program used across every approach while the vertical 'error bars' show the maximum and minimum quality gains achieved. All of the scores across every input gave an improvement in the quality objective. For all but one of the inputs, the mono-objective approach gives a better quality improvement than the multi/many-objective approaches. With the GanttProject input, the second multi-objective permutation gave a higher average score than the mono-objective approach. This can likely be attributed to the run that generated a score of 0.42. None of the other tasks in any of the permutations yielded that high of a score at that level with GanttProject. The smallest improvement was given with the many-objective approach with Apache XML-RPC, whereas the largest was found with the mono-objective approach with XOM. For all of the inputs, the many-objective approach gave the smallest improvement scores, with each of the multi-objective approaches giving a better improvement.

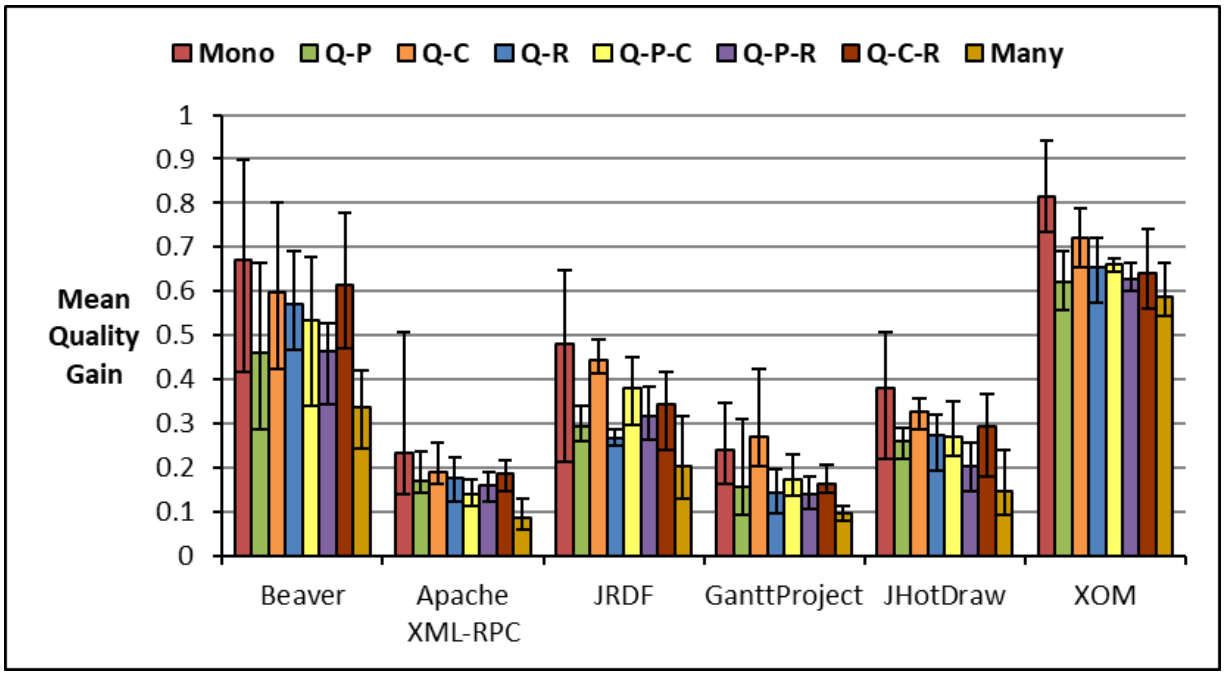

Figure 9 – Mean Quality Gain Values for each Input across each GA Approach

Figure 10 gives the averages of the quality gain values from Figure 9 across all the inputs. The vertical error bars give the highest and lowest of the average values from Figure 9. As observed, the many-objective approach gave the smallest improvements. The Q-C permutation of the multi-objective approaches gave the closest score to the mono-objective approach. The Q-R, Q-P-C and Q-C-R multi-objective approaches all generated similar scores. The two objective solutions that used the refactoring coverage objective along with one of either priority or element recentness gave better scores than those that used priority and element recentness together.

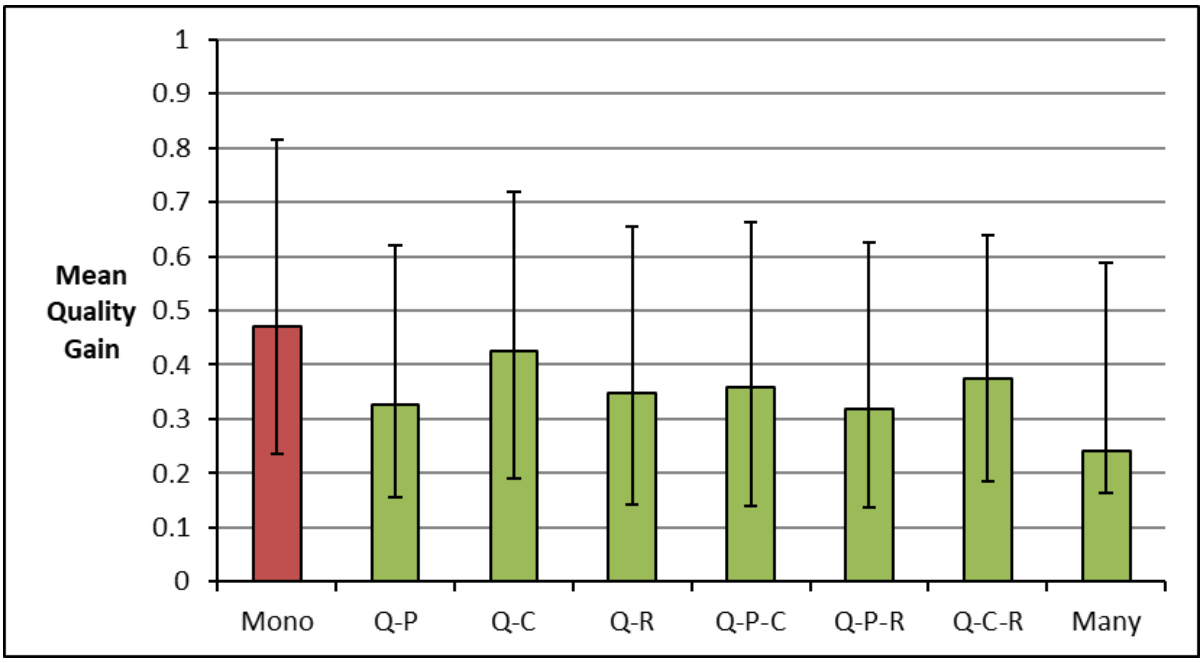

Figure 10 – Mean Quality Gain Values across each GA Approach

Figure 11 shows the average priority scores for each input program used across every relevant approach. Although the many-objective approach was not able to give better priority scores in all cases, each of the three multi-objective permutations that used the priority objective were able to outperform the mono-objective approach for all inputs. Likewise, they outperformed the many-objective approach in all cases. Although there were negative scores given (indicating that non priority classes were among the list of classes refactored in the solution) for some inputs with the mono-objective approach and with one input using the many-objective approach, no such score was generated for any of the multi-objective approaches. The largest score was given with the first permutation of the multi-objective approach, with the XOM input, whereas the smallest was for JRDF with the many-objective approach.

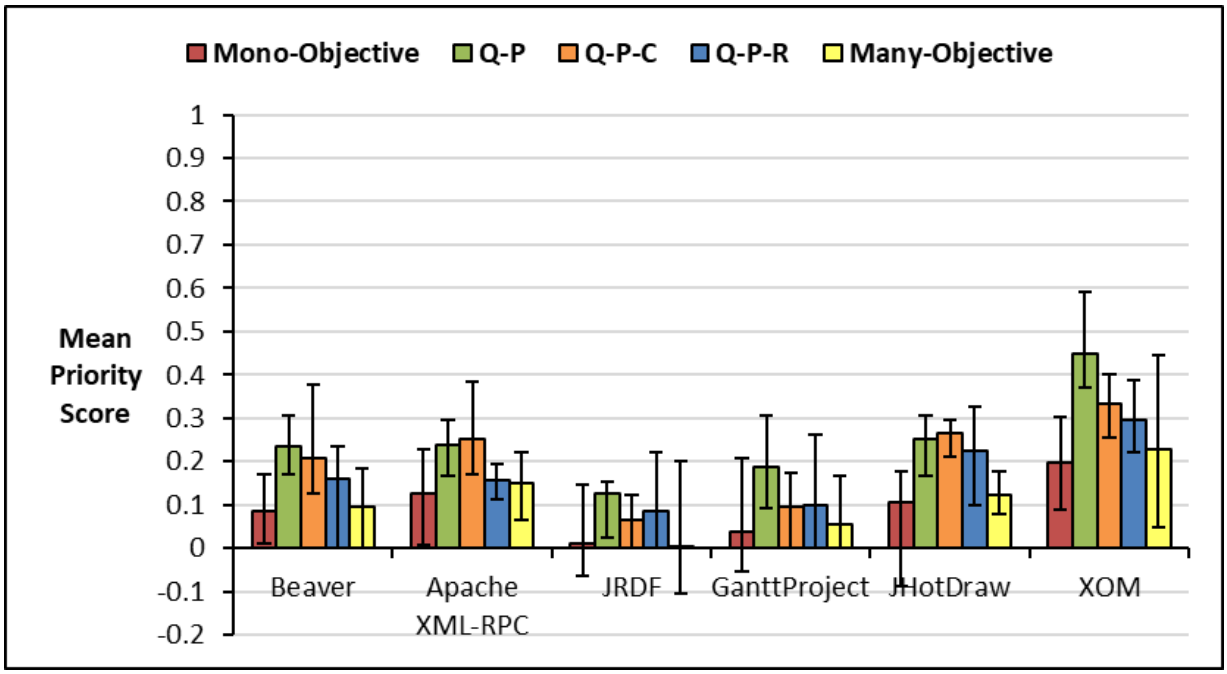

Figure 11 – Mean Priority Scores for each Input across each Relevant GA Approach

Figure 12 gives the averages of the priority score gains from Figure 11 across all the inputs. The error bars give the highest and lowest of the average values in Figure 11. The mono-objective score was the lowest, while the first multi-objective permutation gave the highest score. The two permutations where the priority objective was used in conjunction with the element-recentness objective gave lower improvements among the multi-objective approaches, whereas when the priority objective was used on its own with the quality objective or in conjunction with the refactoring coverage objective, there was a greater improvement in the priority score.

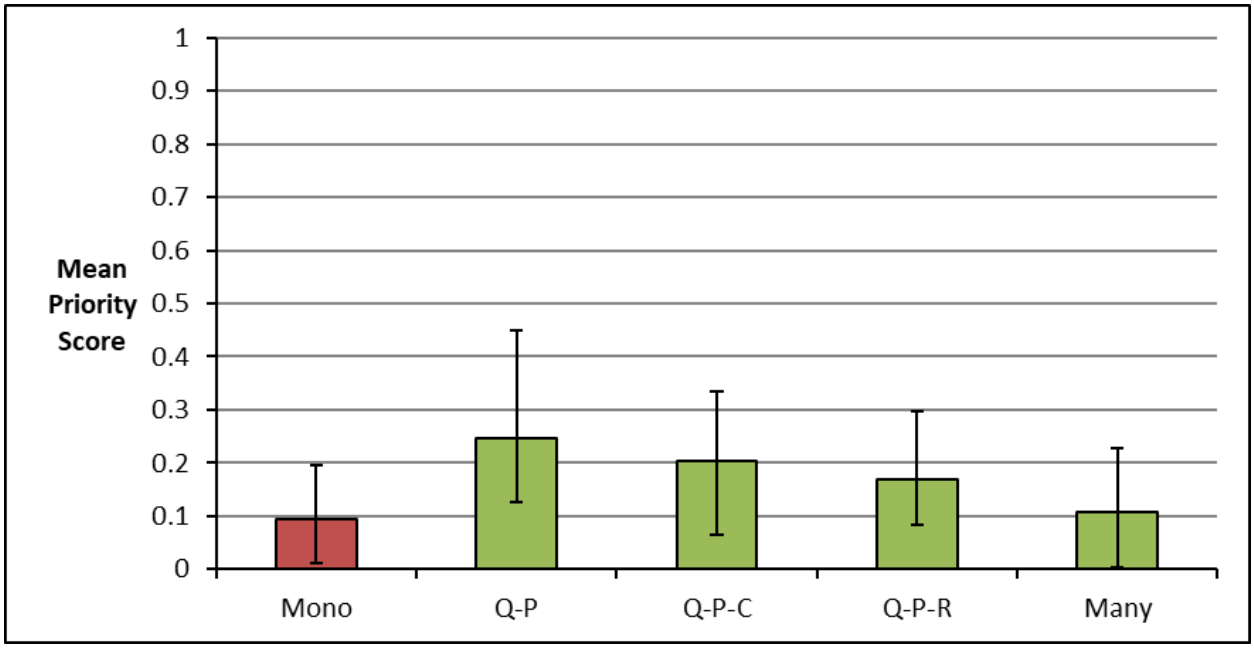

Figure 12 – Mean Priority Scores across each Relevant GA Approach

Figure 13 shows the average coverage score gains for each input program used across every relevant approach. For all of the inputs, all of the multi/many-objective approaches were able to yield better scores coupled with the refactoring coverage objective. The multi/many-objective scores seemed to be similar for each input, with the

many-objective approach generally yielding the highest coverage scores among each approach. The exception to this is with the Apache XML-RPC input, where the fourth multi-objective permutation had the highest coverage score. The input that gave the highest average score was GanttProject input, although maximum coverage scores of 1 were given across almost all of the inputs. The mono-objective scores had a far larger amount of variation than any of the other approaches, with scores as low as 0.06, whereas the lowest coverage score among any of the other approaches was 0.65.

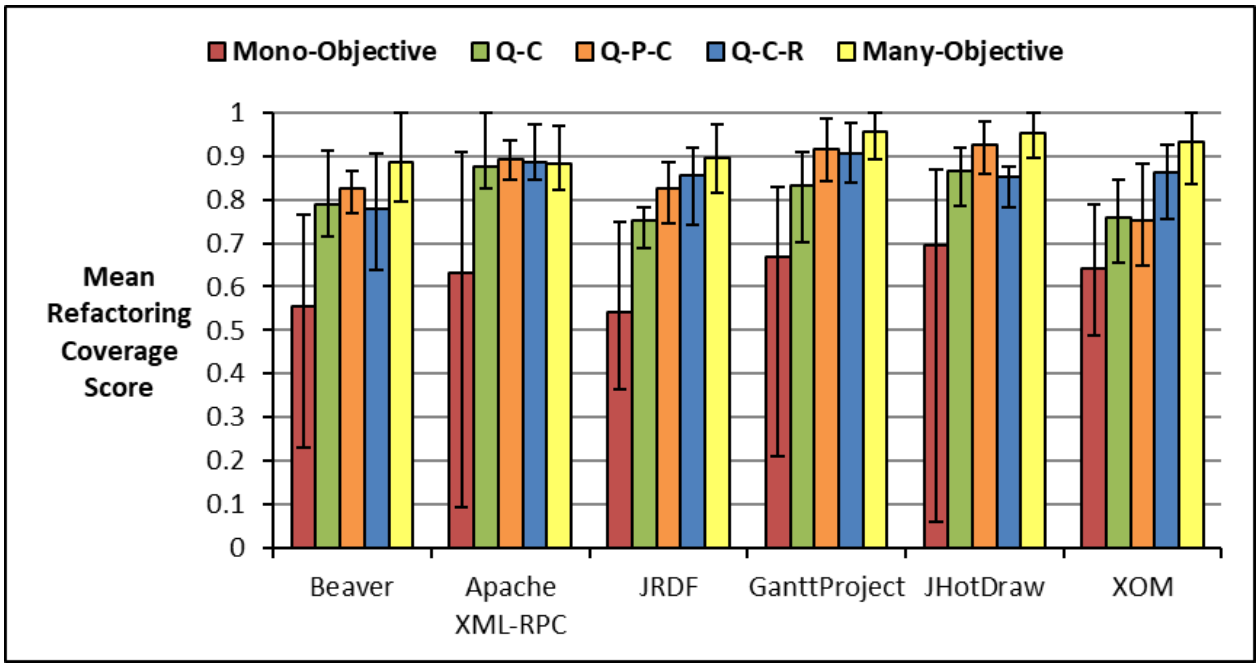

Figure 13 – Mean Refactoring Coverage Scores for each Input across each Relevant GA Approach

Figure 14 gives the averages of the refactoring coverage value gains from Figure 13 across all the inputs. The error bars give the highest and lowest of the average values from Figure 13. As observed, the mono-objective approach gave the smallest score. The highest score was given with the many-objective approach. The refactoring coverage objective actually works better along with either priority or element recentness than it does alone. Likewise, the objective works successfully when it is part of a many-objective approach with both priority and element recentness.

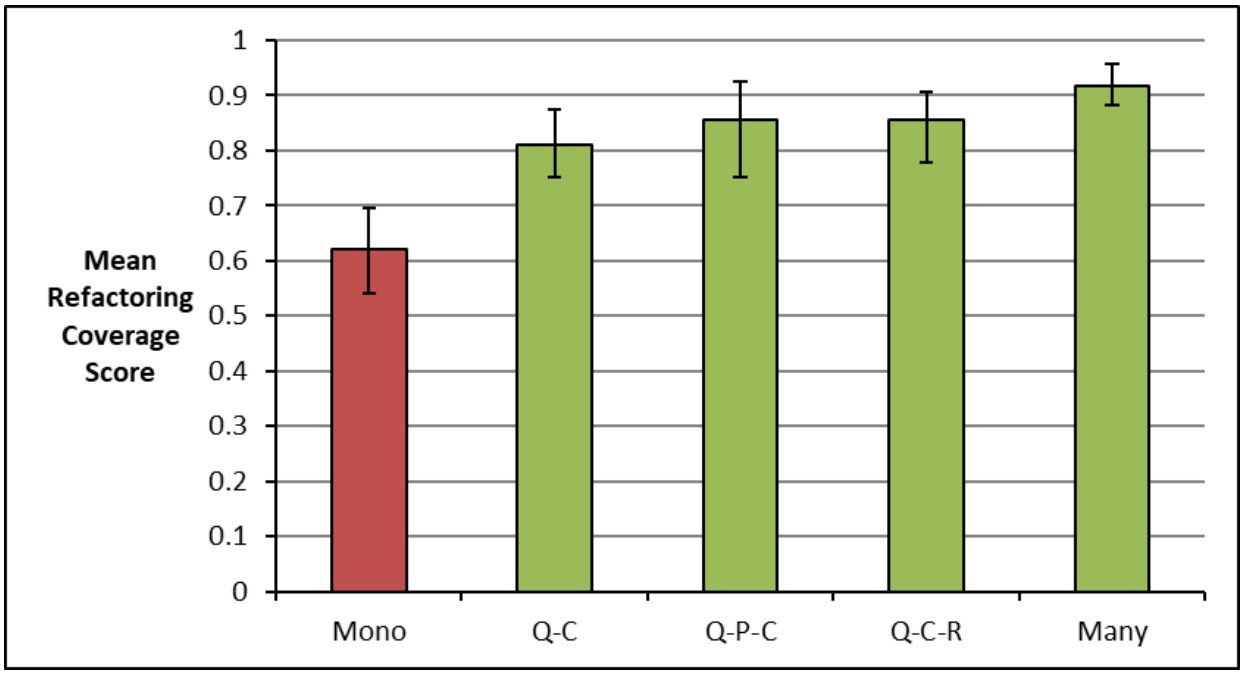

Figure 14 – Mean Refactoring Coverage Scores across each Relevant GA Approach

Figure 15 shows the average element recentness score gains for each input program used across every relevant approach. Of the different input programs, Beaver generated the highest scores. The approach with the top score varied across each input, but the highest average score is given with the sixth multi-objective permutation and the Beaver input. Although the many-objective approach was, in one case, worse than the mono-objective approach, all the other approaches gave better element recentness scores across every input program.

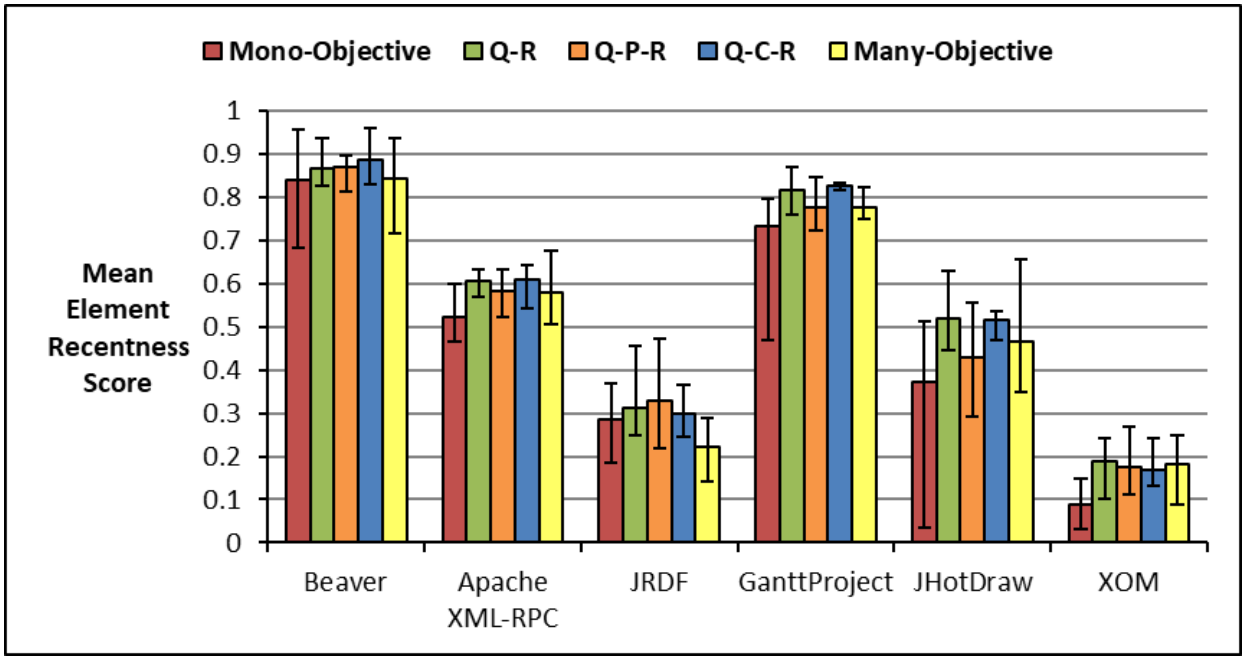

Figure 15 – Mean Element Recentness Scores for Each Input across each Relevant GA Approach

Figure 16 gives the averages of the recentness score gains from Figure 15 across all the inputs. The error bars give the highest and lowest of the average values in Figure 15. The mono-objective approach gave the lowest score, while the third multi-objective approach gave the highest. With this objective, all the relevant multi-objective approaches, along with the many-objective approach, had very similar scores. Therefore, although their average

scores ranged from 0.17 to 0.89, they all managed to average out between 0.51 and 0.56. This objective supports the observations made with the priority objective results in that the two approaches used that pair element recentness with priority gave lower results, whereas when the objective is used only in conjunction with the quality objective or if it is used with the refactoring coverage objective, the scores were slightly better.

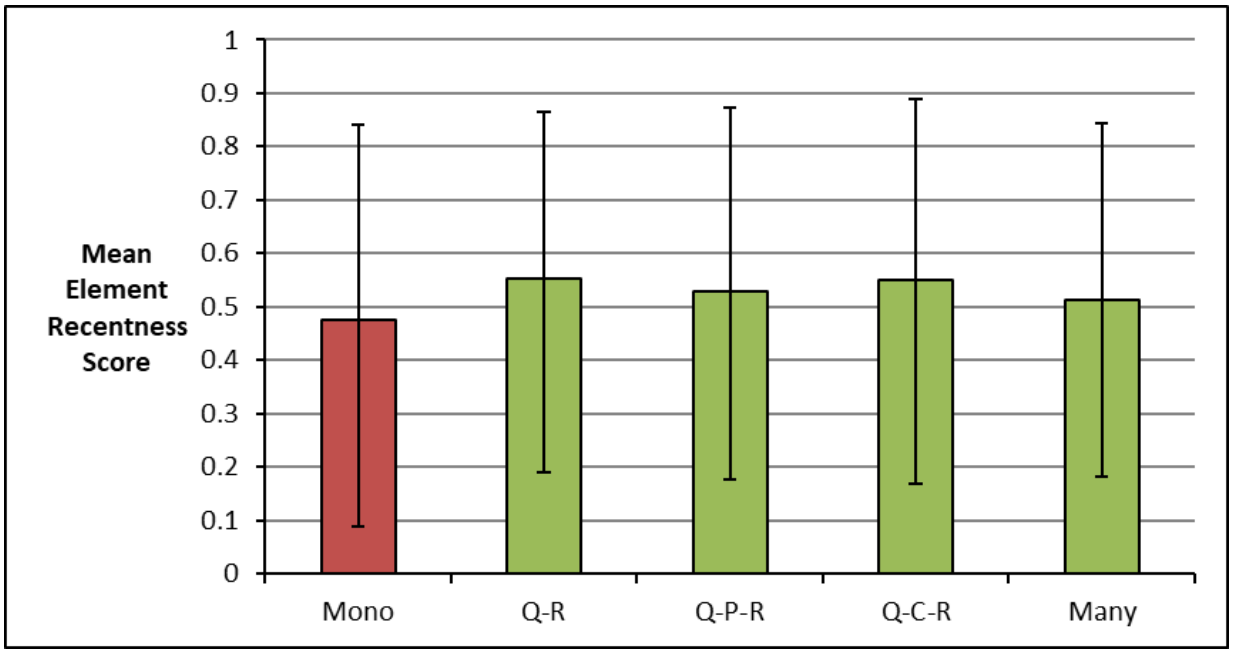

Figure 16 – Mean Element Recentness Scores across each Relevant GA Approach

Regarding execution times, we measured this for every run of the program. Figure 17 gives the average execution times for each input program used across every approach. For most approaches, the times were faster than the mono-objective approach. This is rather unexpected and we put this down to more attention being made to efficiency in the implementation for the multi and many objective approaches. For example, we improved the speed of the multi/many-objective algorithms by adapting the approach of Liu and Zeng[63]. In any case we have presented a comparison of execution times mainly to show that the mono, multi and mono-objective approaches have similar results.

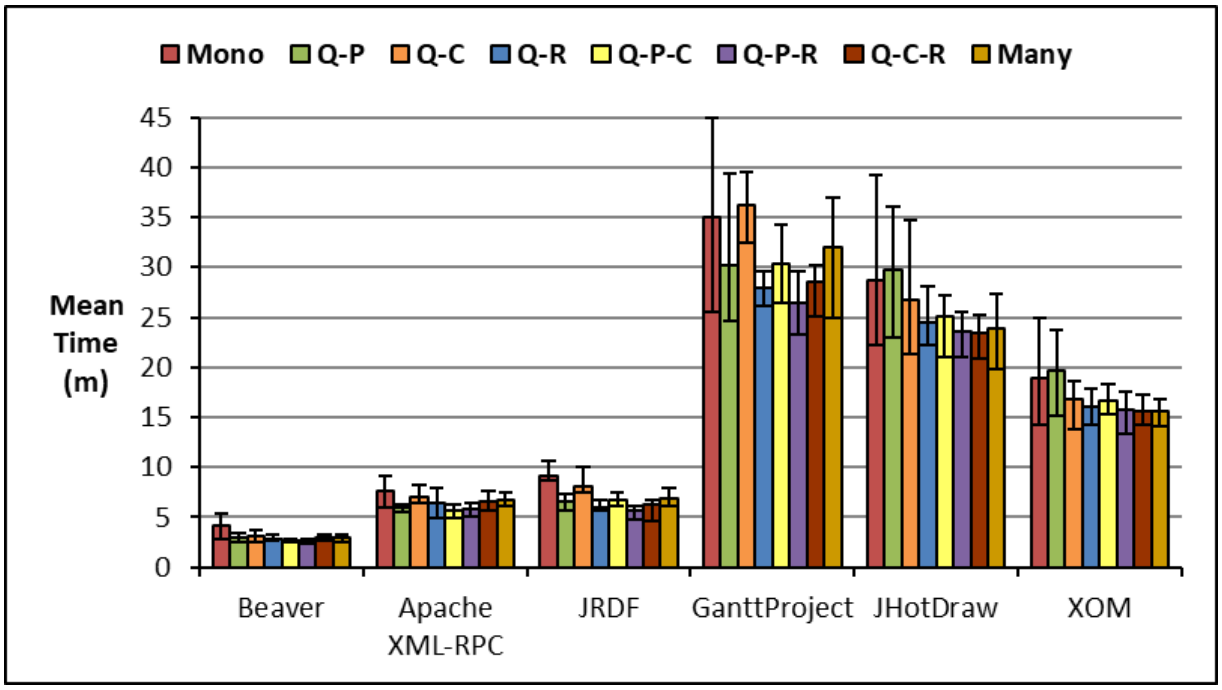

Figure 17 – Mean Times Taken for Each Input across each GA Approach

Figure 18 gives the averages of the execution times from Figure 17 across all the inputs. The error bars give the highest and lowest of the average values in Figure 17. Again, surprisingly, the mono-objective approach mostly took the longest and again probably the explanation is the improvements made for the many objective algorithm. All of the multi/many-objective approaches gave similar times, ranging from 13 minutes and 19 seconds to 16 minutes and 18 seconds.

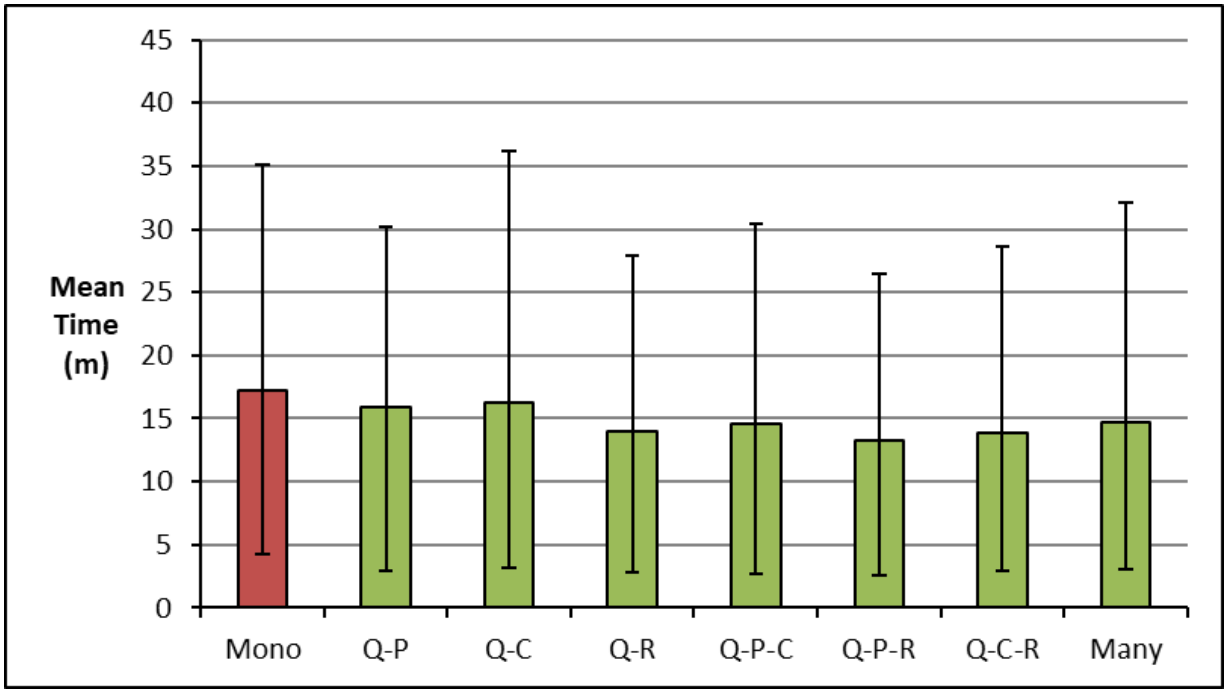

Figure 18 – Mean Times Taken across each GA Approach

6. Discussion

Whereas the refactoring coverage objective gave better scores when combined with all of the other objectives in a many-objective approach, the priority and element recentness objectives were both found to be less successful in multi/many-objective setups when they were used together. Although the solutions still gave better results in most cases (the notable exceptions being those mentioned above) in comparison with a basic mono-objective approach, the objectives weren't able to generate scores as high as those generated in the other permutations. As such, the conclusions derived from experiment 2 are that the objectives don't work as well together and that they may be conflicting with each other when generating refactoring solutions. Another factor to take into consideration is the input in question that produced the less desirable results. It may be the case that for the JRDF input, many of the priority classes listed were also among the oldest classes in relation to the set of program versions available to the search. Therefore, it may have been more difficult to find possible refactorings to apply to the solution that focus on the priority classes and are also able to focus on the more recent elements of the project.

Consequently if a developer has more interest in a refactored solution using the classes specified, or in focusing on the most recent elements of code, it is recommended to use them with only the quality objective or in conjunction with the refactoring coverage objective, and to avoid using them together. On the other hand, the refactoring coverage objective gave better results the more objectives it was used with. So, if the code coverage of the refactorings in the solution is more important, it is recommended that the refactoring coverage objective is used in the many-objective solution with all of the other objectives to get better scores.

To measure success in the many-objective approach, a mono-objective search was conducted using the quality objective, with each of the three supplementary objective scores output at the end of the search for the top solution in each task. The mono-objective and many-objective scores were then compared for each objective, as well as the times taken to run the tasks. A second experiment was conducted to investigate different combinations of the three supplementary objectives, in conjunction with the quality objective. Along with the previous tasks in the mono-objective and many-objective approaches, six multi-objective setups were tested to use each permutation of the three supplementary objectives, along with the quality objective. The scores were compared across all eight different approaches for each of the objectives along with the time taken.

The average many-objective quality improvement scores were compared against the mono-objective scores across six different programs and, for all inputs, the mono-objective approach gave better improvements. The many-objective approach gave improvements in quality across all the inputs. When the priority objective was compared, five of the inputs gave better scores with the many-objective approach, whereas the JRDF program gave better improvements with the mono-objective approach. Likewise, for the element recentness objective, the mono-objective approach gave a better score for the JRDF input but with the refactoring coverage objective all inputs gave better scores with the many-objective approach in comparison to the mono-objective approach. For every input, the many-objective approach took less time. The tasks took longer depending on the class size of the input program in question.

When the average quality improvement scores were compared for each of the seven multi/many-objective approaches against the mono-objective approach, one approach yielded a better score for the GanttProject program than the mono-objective approach. Each approach gave improvements in quality across all inputs. When the priority scores were compared across all the approaches, each of the multi-objective setups gave better values than the mono-objective approach across all the inputs. Similarly, when the scores for the other two objectives, refactoring coverage and element recentness, were compared, the multi-objective scores were higher than their mono-objective counterparts.

In order to test the aims of the experimentation and derive conclusions from the results a set of research questions were constructed. **RQ1** and **RQ2**, each had corresponding hypotheses, led to experiment one and. **RQ1** was concerned with the effectiveness of the quality objective in the many-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 60 of the different runs of the many-objective approach (as well as the 180 runs of the six multi-objective approaches), there was an improvement in the quality objective score, therefore rejecting the null hypothesis $\mathbf{H1_0}$.

RQ2 looked at how effective the other three objectives were in a many-objective setup in comparison with the mono-objective approach. With the refactoring coverage objective, each input gave a better score with the many-objective approach compared with the mono-objective approach although, for the other two objectives this wasn't the case for all inputs. For the JRDF input, both the priority and element recentness scores were smaller with the many-objective approach. Therefore, for these two objectives, the null hypothesis $\mathbf{H2_0}$ cannot be fully rejected. The results generated in experiment 2 and analysis of those results were able to address this observation, and

helped provide an explanation for why the JRDF input didn't yield results that were as successful with the applicable objectives.

To address **RQ3** and derive the most successful combination of objectives to use for each of the three supplemental objectives, the scores have been averaged together for each input program to give overall scores for each permutation with each objective. The priority objective and element recentness objectives are both more successful in a bi-objective setup with the quality objective. This could go some way towards explaining why they were unable to yield better scores in the many-objective setup with the JRDF input. The refactoring coverage objective is more successful in a 4-objective setup with the quality, priority and element recentness objectives.

6.1 Threats to Validity

Internal Validity focuses on the causal effect of the independent variables on the dependent variables. The stochastic nature of the search techniques means that each run will provide different results. This threat to validity has been addressed by running each of the tasks across six different open source programs and running against each program five times for experiment 2 and ten times for experiment 1. Average values are then used to compare against each other. The choice of parameter settings used by the search techniques can also provide a threat to validity due to the option of using poor input settings. This has been addressed by using input parameters deemed to be most effective through trial and error via previous experimentation [12]. Regarding the refactoring process, one weaknesses at present is in relation to the recentness measure. There exists the the possibility that the name of an element has changed in the history. This is not straightforward since id numbers in the Abstract Syntax Tree (AST) will be different across different ASTs. Equally the full file pathway could be used to identify an element, but there is a chance that elements could move between classes between software versions. For the priority objective, the intention was to allow a free choice to the developer. We simulated this by preferring those classes with more methods. The results will be different if a different mechanism is used to choose the priority classes.

External Validity is concerned with how well the results and conclusions can be generalised. In this study, the experimentation was performed on six different real world open source systems belonging to different domains and with different sizes and complexities. However, the experimentation and the capabilities of the refactoring tool used are restricted to Java programs; therefore it cannot be asserted that the results can be generalised to other applications or to other programming languages. Further, the experiments conducted for this paper, although they have been applied to different software projects and repeated numerous times, are still limited in the scale of the tasks conducted. All of the target programs tested were open source. The sizes of the programs ranged from small to medium size (from 2,196 lines of code to 45,136. Further experimentation should be conducted with larger programs and programs of different types. Hence, further replications of this study are necessary to confirm the generalisation of the findings.

Construct Validity refers to how well the concepts and measurements are related to the experimental design. The validity of the experimentation is limited by the metrics used, as they are experimental approximations of software quality, as well as the objectives used to measure various aspects of the refactorings applied. What constitutes a good metric for quality is very subjective. The cost measures used in the experimentation can also indicate a threat to validity. Part of the effectiveness of the search approaches was measured using execution time in order to measure and compare cost.

Conclusion Validity looks at the degree to which a conclusion can reasonably be drawn from the results. A lack of a meaningful comparative baseline can provide a threat by making it harder to produce a conclusion from the results without the relevant context. In order to provide descriptive statistics of the results, tasks have been repeated and average values have been used to compare against. Another possible threat may be provided by the lack of a formal hypothesis in the experimentation. At the outset, three research questions have been provided and for the first two (relating to experiment 1), a set of corresponding hypotheses have been constructed in order to aid in drawing a conclusion. For **RQ3**, the objective values across the different permutations of the search tested in experiment 2 are compared to deduce the most suitable permutation for each secondary objective.

7. Conclusion

It is important to discuss how this work builds on and enhances existing work. The work by Ouni et al [16] has demonstrated optimisation with three objectives. In this paper an experiment has been conducted to test four objectives constructed using the MultiRefactor tool. The extended MultiRefactor tool described in this paper approach advances auto-refactoring from existing tools in [36] and [30], which also modify source code in Java programs, in providing an overall framework with many-objective functionality in order to improve the software across various different properties. To conduct the search, an adaptation of the many-objective GA, NSGA-III,

was used. The NSGA-III search was used to improve the target projects in correspondence to the objectives measuring quality, priority of refactored classes, code coverage of refactored elements, and recentness of refactored elements.

Indeed, while a number of recent studies in SBSM have used multi-objective techniques, not a lot of studies have progressed to using many-objective techniques. Some attention has been given to using many-objective techniques in other areas. For example, there is a recent exploratory study on using several evolutionary many-objective optimisation approaches, including NSGA-III for the Next Release Problem [64]. Another example approach [65] looks at optimal feature selection by first optimising on one objective and then trying to improve others. Looking at these alternative evolutionary optimisation approaches and how they might apply to refactoring is a matter for future work. In regard to many objective techniques in software refactoring, Mkaouer et al. [58] experimented with many-objective techniques using NSGA-III. They tested the algorithm with different numbers of objectives and compared it against other EAs to see how effective it is at handling multiple objectives in comparison [3]. Mkaouer et al. also used the algorithm to combine objectives from previous work (number of classes per package, number of packages, cohesion, coupling, number of code changes, refactoring history and semantic similarity together into an approach to re-modularise software [59]. The many-objective experimentation conducted in this paper combines four different objectives: quality, class priority, refactoring code coverage, and element recentness. The approach used is also different. An adaptation of the NSGA-III algorithm is also used but instead of generating refactoring suggestions for a software system, it actually applies the refactorings to the code. Thus, the solutions generated will be refactored versions of the software code.

In terms of tool support there is much that can still be developed. There are many different refactorings that could be added e.g. “Extract Method”. “Extract Method” is possible with SBSE but difficult because of the stochastic nature of our approach and the likelihood of breaking the code. In fact, the problems in implementing this are similar to the “Extract Subclass” refactoring, already in the tool, in checking semantic coherence after the refactoring. Additionally, there are a large number of ‘newer’ refactorings including those added to Fowler’s catalogue [66] and by other researchers e.g. [67]. There are also other possible recent refactoring ideas that could be exploited such as those for concurrency [68], where metrics related to performance may be more relevant.

Even better, it would be helpful to gain the insight of experienced developers by asking for their opinion of the capabilities of the tool and the usefulness of the separate objectives and the many-objective approach. The combined insight of developer opinion and an industrial target program could be used to gain a more realistic insight into how effective the MultiRefactor tool and the multi/many-objective search-based approach could be in tackling the software maintenance issue. Added to this validation of the refactorings, an assessment of the usability of the approach and tool support is also planned.

There is also room to investigate and build upon the approaches proposed in other SBSM papers. For example, Amal et al.[69], introduced an artificial neural network to choose between solutions in the final population of the search. Another study worthy of consideration is [49] which also investigated a multi-objective approach but incorporated a non-functional objective along with functional ones. Useful aspects to look at might be non-functional measures of aspects like security and performance.

Overall, there are many possibilities for the work both in extending the work to other refactorings, adding metrics, adding secondary objectives, including other programming languages, experimentation with larger programs and of different types, empirical studies of the method and tool support, use of other aspects of a software project such as change logs or unit tests to further influence how the refactorings are chosen or how the fitness for each refactoring solution is calculated and many more. Given the huge possible benefits there are many potentially research and practice developments that can be taken forward in search-based automated refactoring.

Acknowledgements

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

References

[1] M. Mohan and D. Greer, “A Survey of Search-Based Refactoring for Software Maintenance", Journal of Software Engineering Research and Development,” *Softw. Eng. Res. Dev.*, vol. 6, no. 3, p. 52p, 2018.

[2] D. Bell, *Software Engineering: A Programming Approach*. Addison Wesley, 2000.

[3] W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, “High Dimensional Search-Based Software Engineering: Finding Tradeoffs Among 15 Objectives For Automating Software Refactoring

Using NSGA-III," in *Genetic and Evolutionary Computation Conference, GECCO 2014.*, 2014.
[4] M. Harman and L. Tratt, "Pareto Optimal Search Based Refactoring At The Design Level," in *9th Annual Conference on Genetic and Evolutionary Computation, GECCO 2007.*, 2007, pp. 1106–1113.
[5] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving Multi-Objective Code-Smells Correction Using Development History," *J. Syst. Software.*, vol. 105, pp. 18–39, 2015.
[6] K. Deb and D. K. Saxena, "Searching For Pareto-Optimal Solutions Through Dimensionality Reduction For Certain Large-Dimensional Multi-Objective Optimization Problems," in *IEEE Congress On Evolutionary Computation, CEC 2006.*, 2006.
[7] H. K. Singh, A. Isaacs, and T. Ray, "A Pareto Corner Search Evolutionary Algorithm And Dimensionality Reduction In Many-Objective Optimization Problems," *IEEE Trans. Evol. Comput.*, vol. 15, no. 4, pp. 539–556, 2011.
[8] F. Di Pierro, S.-T. Khu, and D. A. Savić, "An Investigation On Preference Order - Ranking Scheme For Multi Objective Evolutionary Optimization," *IEEE Trans. Evol. Comput.*, vol. 11, no. 1, pp. 1–33, 2007.
[9] Q. Zhang and H. Li, "MOEA/D: A Multiobjective Evolutionary Algorithm Based On Decomposition," *IEEE Trans. Evol. Comput.*, vol. 11, no. 6, pp. 712–731, 2007.
[10] K. Deb and H. Jain, "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Non-Dominated Sorting Approach, Part I: Solving Problems With Box Constraints," *IEEE Trans. Evol. Comput.*, vol. 18, no. 4, pp. 1–23, 2013.
[11] L. Thiele, K. Miettinen, P. J. Korhonen, and J. Molina, "A Preference-Based Evolutionary Algorithm For Multi-Objective Optimization," *Evol. Comput.*, vol. 17, no. 3, pp. 411–436, 2009.
[12] M. Mohan and D. Greer, *MultiRefactor: Automated refactoring to improve software quality*, vol. 10611 LNCS. 2017.
[13] J. Bansiya and C. G. Davis, "A Hierarchical Model For Object-Oriented Design Quality Assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, 2002.
[14] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite For Object Oriented Design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.
[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast And Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, 2002.
[16] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "The Use Of Development History In Software Refactoring Using A Multi-Objective Evolutionary Algorithm," in *Genetic and Evolutionary Computation Conference, GECCO 2013.*, 2013, pp. 1461–1468.
[17] M. Harman and B. F. Jones, "Search-Based Software Engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, Dec. 2001.
[18] M. Harman, "The Current State And Future Of Search Based Software Engineering," in *Future Of Software Engineering, FOSE 2007.*, 2007, pp. 342–357.
[19] J. Clarke *et al.*, "Reformulating Software Engineering As A Search Problem," *IEE Proc. - Software.*, vol. 150, no. 3, pp. 1–25, 2003.
[20] T. E. Colanzi, S. R. Vergilio, W. K. G. Assunção, and A. Pozo, "Search Based Software Engineering: Review And Analysis Of The Field In Brazil," *J. Syst. Software.*, vol. 86, no. 4, pp. 970–984, 2013.
[21] M. Harman, S. A. Mansouri, and Y. Zhang, "Search Based Software Engineering: Trends, Techniques And Applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 1–64, 2012.
[22] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search Based Software Engineering: Techniques, Taxonomy, Tutorial," in *Empirical Software Engineering and Verification.*, 2012, pp. 1–59.
[23] A. S. Sayyad and H. Ammar, "Pareto-Optimal Search-Based Software Engineering (POSBSE): A Literature Survey," in *2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE 2013.*, 2013, pp. 21–27.
[24] F. Ferrucci, M. Harman, and F. Sarro, "Search-Based Software Project Management," in *Software Project Management in a Changing World.*, no. 1994, 2014, pp. 373–399.
[25] A. M. Pitangueira, R. S. P. Maciel, M. Barros, and A. S. Andrade, "A Systematic Review Of Software Requirements Selection And Prioritization Using SBSE Approaches," in *5th International Symposium On Search-Based Software Engineering, SSBSE 2013.*, 2013, pp. 188–208.
[26] O. Räihä, "A Survey On Search-Based Software Design," *Comput. Sci. Rev.*, vol. 4, no. 4, pp. 203–249, 2010.
[27] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Softw. Testing, Verif. Reliab.*, vol. 14, no. 2, pp. 1–58, 2004.
[28] M. Harman and P. McMinn, "A Theoretical And Empirical Study Of Search-Based Testing: Local, Global, And Hybrid Search," *IEEE Trans. Softw. Eng.*, vol. 36, no. 2, pp. 226–247, Mar. 2010.
[29] T. Dudziak and J. Wloka, "Tool-Supported Discovery And Refactoring Of Structural Weaknesses In Code," 2002.
[30] A. Trifu, O. Seng, and T. Genssler, "Automated Design Flaw Correction In Object-Oriented Systems," in

8th European Conference on Software Maintenance and Reengineering, CSMR 2004., 2004, pp. 174–183.

[31] M. Di Penta, "Evolution Doctor: A Framework To Control Software System Evolution," in *9th European Conference on Software Maintenance and Reengineering, CSMR 2005.*, 2005, pp. 280–283.

[32] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification And Removal Of Type-Checking Bad Smells," in *12th European Conference on Software Maintenance and Reengineering, CSMR 2008.*, 2008, pp. 329–331.

[33] H. Li and S. Thompson, "Refactoring Support For Modularity Maintenance In Erlang," in *10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM 2010.*, 2010, pp. 157–166.

[34] I. Griffith, S. Wahl, and C. Izurieta, "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.*, 2011.

[35] D. Fatiregun, M. Harman, and R. M. Hierons, "Evolving Transformation Sequences Using Genetic Algorithms," in *4th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2004.*, 2004, pp. 65–74.

[36] I. H. Moghadam and M. Ó Cinnéide, "Code-Imp: A Tool For Automated Search-Based Refactoring," in *4th Workshop on Refactoring Tools, WRT 2011.*, 2011, pp. 41–44.

[37] E. Koc, N. Ersoy, A. Andac, Z. S. Camlidere, I. Cereci, and H. Kilic, "An Empirical Study About Search-Based Refactoring Using Alternative Multiple And Population-Based Search Techniques," in *Computer and Information Sciences II.*, E. Gelenbe, R. Lent, and G. Sakellari, Eds. London: Springer London, 2012, pp. 59–66.

[38] T. Van Belle and D. H. Ackley, "Code Factoring And The Evolution Of Evolvability," in *Genetic and Evolutionary Computation Conference, GECCO 2002.*, 2002, pp. 1383–1390.

[39] R. Vivanco and N. Pizzi, "Finding Effective Software Metrics To Classify Maintainability Using A Parallel Genetic Algorithm," in *6th Annual Conference on Genetic and Evolutionary Computation, GECCO 2004.*, 2004, pp. 1388–1399.

[40] O. Seng, J. Stammel, and D. Burkhart, "Search-Based Determination Of Refactorings For Improving The Class Structure Of Object-Oriented Systems," in *Genetic and Evolutionary Computation Conference, GECCO 2006.*, 2006, pp. 1909–1916.

[41] R. Lange and S. Mancoridis, "Using Code Metric Histograms And Genetic Algorithms To Perform Author Identification For Software Forensics," in *9th Annual Conference on Genetic and Evolutionary Computation, GECCO 2007.*, 2007, pp. 2082–2089.

[42] M. O'Keeffe and M. Ó Cinnéide, "Getting The Most From Search-Based Refactoring," in *9th Annual Conference on Genetic and Evolutionary Computation, GECCO 2007.*, 2007, pp. 1114–1120.

[43] M. O'Keeffe and M. Ó Cinnéide, "Search-Based Refactoring: An Empirical Study," *J. Softw. Maint. Evol. Res. Pract.*, vol. 20, no. 5, pp. 1–23, 2008.

[44] A. C. Jensen and B. H. C. Cheng, "On The Use Of Genetic Programming For Automated Refactoring And The Introduction Of Design Patterns," in *12th Annual Conference on Genetic and Evolutionary Computation, GECCO 2010.*, 2010, pp. 1341–1348.

[45] M. Ó Cinnéide, "Automated Application Of Design Patterns: A Refactoring Approach," 2000.

[46] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design Defects Detection And Correction By Example," in *IEEE International Conference on Software Engineering, ICSM 2011.*, 2011, pp. 81–90.

[47] M. Kessentini, W. Kessentini, and A. Erradi, "Example-Based Design Defects Detection And Correction," in *19th International Conference On Program Comprehension, ICPC 2011.*, 2011, pp. 1–32.

[48] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What You Like In Design Use To Correct Bad-Smells," *Softw. Qual. Journal.*, vol. 21, no. 4, pp. 551–571, Oct. 2012.

[49] D. R. White, J. Clark, J. Jacob, and S. Poulding, "Searching for Resource-Efficient Programs: Low-Power Pseudorandom Number Generators," in *Genetic and Evolutionary Computation Conference, GECCO 2008.*, 2008, pp. 1775–1782.

[50] J. T. De Souza, C. L. Maia, F. G. De Freitas, and D. P. Coutinho, "The Human Competitiveness Of Search Based Software Engineering," in *2nd International Symposium on Search-Based Software Engineering, SSBSE 2010.*, 2010, pp. 143–152.

[51] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-Based Refactoring: Towards Semantics Preservation," in *28th IEEE International Conference on Software Maintenance, ICSM 2012.*, 2012, pp. 347–356.

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

[53] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-Criteria Code Refactoring Using

Search-Based Software Engineering: An Industrial Case Study," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, 2016.

[54] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability Defects Detection And Correction: A Multi-Objective Approach," *Autom. Softw. Eng.*, vol. 20, no. 1, pp. 47–79, 2013.

[55] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing Code-Smells Correction Tasks Using Chemical Reaction Optimization," *Softw. Qual. Journal.*, vol. 23, no. 2, 2015.

[56] W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó Cinnéide, and K. Deb, "Software Refactoring Under Uncertainty: A Robust Multi-Objective Approach," in *Genetic and Evolutionary Computation Conference, GECCO 2014.*, 2014.

[57] M. W. Mkaouer, M. Kessentini, M. Ó Cinnéide, S. Hayashi, and K. Deb, "A Robust Multi-Objective Approach To Balance Severity And Importance Of Refactoring Opportunities," *Empir. Softw. Eng.*, 2016.

[58] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó Cinnéide, and K. Deb, "On The Use Of Many Quality Attributes For Software Refactoring: A Many-Objective Search-Based Software Engineering Approach," *Empir. Softw. Eng.*, 2015.

[59] W. Mkaouer, M. Kessentini, P. Kontchou, K. Deb, S. Bechikh, and A. Ouni, "Many-Objective Software Remodularization Using NSGA-III," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, 2015.

[60] M. Mohan, D. Greer, and P. McMullan, "Technical debt reduction using search based automated refactoring," *J. Syst. Softw.*, vol. 120, 2016.

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

[62] I. Das and J. E. Dennis, "Normal-Boundary Intersection: A New Method For Generating The Pareto Surface In Nonlinear Multicriteria Optimization Problems," *SIAM J. Optim.*, vol. 8, no. 3, pp. 631–657, 1998.

[63] M. Liu and W. Zeng, "Reducing The Run-Time Complexity Of NSGA-II For Bi-Objective Optimization Problem," in *International Conference on Intelligent Computing and Intelligent Systems, ICIS 2010*, 2010, pp. 546–549.

[64] J. Geng, Jiangyi; Ying, Shi ; Jia, Xiangyang; Zhang, Ting; Liu, Xuan; Guo, Lanqing; Xuan, "Supporting Many-Objective Software Requirements Decision: An Exploratory Study on the Next Release Problem," *IEEE Access (Early Access)*, 2018. .

[65] W. Hierons, R.M.;Li, M.; Liu, X.; Segura, S.; Zheng, "SIP: Optimal Product Selection from Feature Models Using Many-Objective Evolutionary Optimization," *ACM Trans. Softw. Eng. Methodol*, vol. 25, no. 2, p. 17.1-17.39, 2016.

[66] M. Fowler, "Catalog of Refactorings," 2019. [Online]. Available: https://refactoring.com/catalog/. [Accessed: 14-Jan-2019].

[67] R. Khatchadourian, "Automated refactoring of legacy Java software to enumerated types," *Autom. Softw. Eng.*, vol. 24, no. 4, pp. 757–787, 2017.

[68] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in *Proc IEEE 31st International Conference on Software Engineering*, 2009, pp. 397–407.

[69] B. Amal, M. Kessentini, S. Bechikh, J. Dea, and L. Ben Said, "On The Use Of Machine Learning And Search-Based Software Engineering For Ill-Defined Fitness Function: A Case Study On Software Refactoring," in *6th International Symposium On Search-Based Software Engineering (SSBSE)*, 2016, p. 31–45.