Automated Software Maintenance Using Search-Based Refactoring

Michael Mohan (MEng)

School of Electronics, Electrical Engineering and Computer Science

Queen's University Belfast

This dissertation is submitted for the degree of

Doctor of Philosophy

February 2018

Abstract

Search-based software maintenance (SBSM) is an area of research that uses refactorings, software metrics and search-based optimisation algorithms to automate aspects of the software maintenance process. Refactorings are used to improve the structure of software without affecting its functionality. Search-based optimisation algorithms can be adapted to use refactorings to modify software, relying on metrics to deduce how successful the refactorings have been along the way. The research conducted in this thesis aims to explore the research area of SBSM and experiment with methods to automate software refactoring using optimisation algorithms.

The current state of the art in the area is inspected and gaps are identified in the current literature. In particular, the need for further investigation of multi-objective and many-objective optimisation techniques, as well as experimentation with the metrics used to measure the software, is present. In order to experiment with different ways to optimise software for quality an automated refactoring tool is developed. Using this tool, novel aspects of the software are investigated and used as measures to assess and then improve the quality of the software. A multi-objective optimisation algorithm is used so that in addition to quality other, more complex properties are also improved. Using the automated maintenance tool and the underlying approaches, a methodology is presented to automate the refactoring process. Four different areas of importance are investigated as objectives for automated refactoring. The main contributions of the research work are the developed automated refactoring tool, the 4 objectives constructed to measure different aspects of the software code and the methodology developed to maintain the code using the 4 separate measures with a many-objective optimisation algorithm.

Contents

List of Tables

List of Figures

List of Algorithms

List of Equations

Abbreviations

ABC	Artificial Bee Colony
ACO	Ant Colony Optimization
CK	Chidamber & Kemerer
CODe-Imp	Combinatorial Optimisation for Design Improvement
CRO	Chemical Reaction Optimization
DPT	Design Pattern Tool
EA	Evolutionary Algorithm
GA	Genetic Algorithm
GEA	General Evolutionary Algorithms
GP	Genetic Programming
HC	Hill Climbing
JSON	JavaScript Object Notation
LSCC	Low-level Similarity-based Class Cohesion
MOEA	Multi-Objective Evolutionary Algorithm
MOGA	Multi-Objective Genetic Algorithm
MOOSE	Metrics for Object-Oriented Software Engineering
NSGA	Nondominated Sorting Genetic Algorithm
PSO	Particle Swarm Optimization
QMOOD	Quality Model for Object-Oriented Design
ReCon	Refactoring approach based on task Context
SA	Simulated Annealing
SBSE	Search-Based Software Engineering
SBSM	Search-Based Software Maintenance
SIA	Swarm Intelligence Algorithm
VNS	Variable Neighborhood Search
WSL	Wide-Spectrum Language
XOM	XML Object Model

Related Publications

The following papers have been published, or are under consideration for publication, using research from this thesis.

Journal Articles

Michael Mohan, Des Greer and Paul McMullan, "Technical debt reduction using search based automated refactoring", in Journal of Systems and Software (JSS), volume 120, pp. 183-194, October 2016.

Michael Mohan and Des Greer, "A Survey of Search-Based Software Maintenance", in Journal of Software Engineering Research and Development (JSERD), volume 120, issue 3, 2018.

Michael Mohan and Des Greer, "Maximising Code Coverage in an Automated Maintenance Approach using Multi-Objective Optimisation", in IET Software: Special Issue on Search-based Software Engineering, August 2018 (under revision).

Conference/Workshop Articles

Michael Mohan and Des Greer, "MultiRefactor: Automated Refactoring to Improve Software Quality", in Proc. 18th International Conference on Product-Focused Software Process Improvement (PROFES), November 2017, Innsbruck, Austria.

Michael Mohan and Des Greer, "An Approach to Prioritize Classes in a Multi-Objective Software Maintenance Framework", in the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), March 2018, Funchal, Madeira, Portugal.

Michael Mohan and Des Greer, "Automated Refactoring of Software using Version History and a Code Element Recentness Measure", at the 1st

International Workshop on Software Engineering Aspects of Optimizing Systems (SEAOS), March 2018, Funchal, Madeira, Portugal (under review).

Presentations

Michael Mohan, Des Greer and Paul McMullan, "Technical Debt Reduction using Search Based Automated Refactoring", Journal First Presentation, at the Symposium on Search-Based Software Engineering (SSBSE), September 2017, Paderborn, Germany.

Michael Mohan and Des Greer, "MultiRefactor: Automated Refactoring To Improve Software Quality", at the 1st International Workshop on Managing Quality in Agile and Rapid Software Development Processes (QuASD), November 2017, Innsbruck, Austria.

Chapter 1

Introduction & Background

Search-based software engineering (SBSE) concerns itself with the resolution of software engineering problems by restructuring them as combinatorial optimisation problems. The topic has been addressed and researched for a number of areas of the software development life cycle, including software code maintenance, requirements optimisation, debugging and (most frequently) test case optimisation. Figure 1.1 shows the dispersion of SBSE papers across the different areas of software engineering. The specific area focused on for this PhD project, software maintenance, only makes up a small quantity of the overall SBSE research. While the research area has existed since the early 1990s and the term "search-based software engineering" was originally coined by Harman and Jones in 2001 [1], most work in this area has been recent with the number of published papers on the topic exploding in recent years (as seen in Figure 1.2). Many of the proposed SBSE papers use an automated approach to increase the efficiency of the area of the software process inspected.

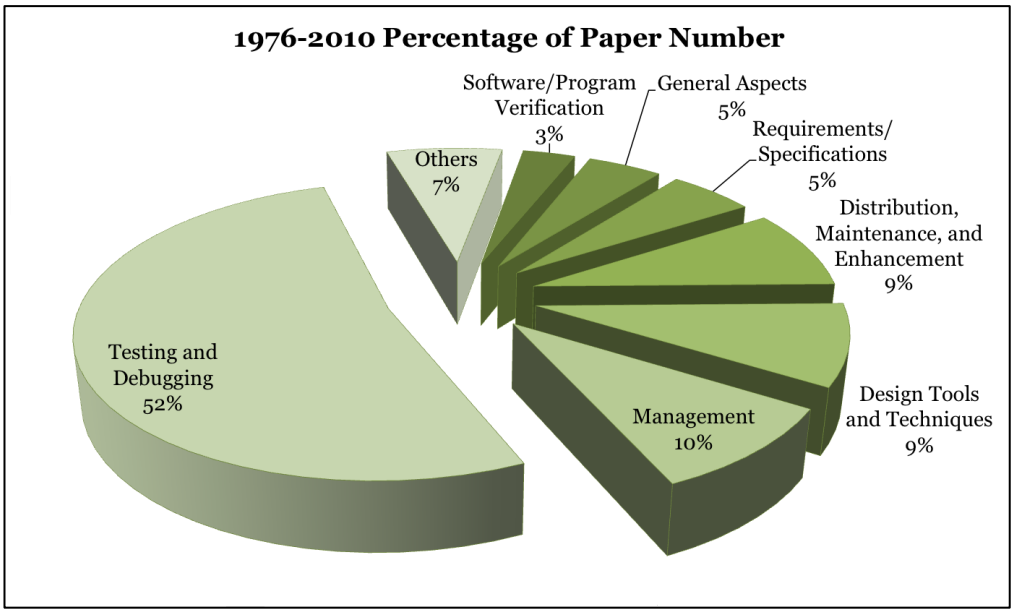

Figure 1.1 – Ratio of Research Fields Studied Involving Search-Based Software Engineering [2]

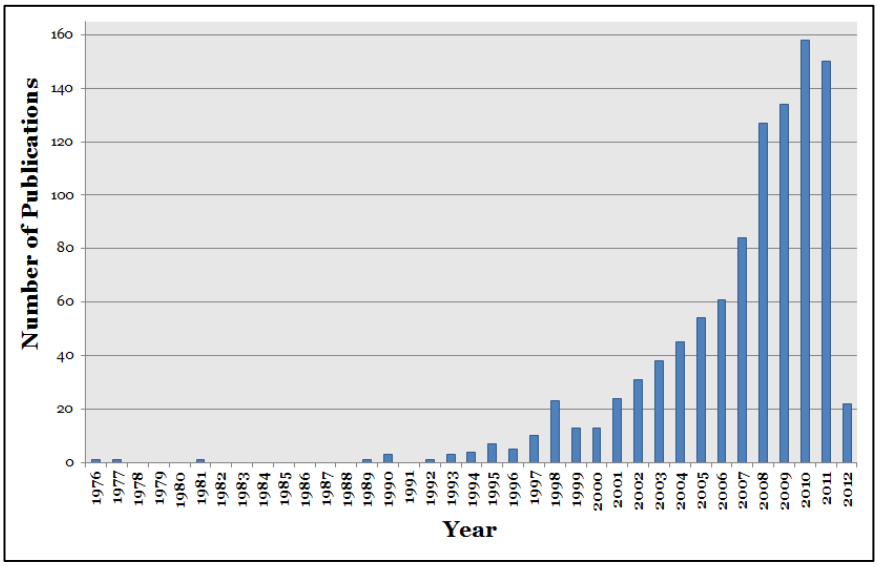

Figure 1.2 – Number of Publications Released by Year (Up to 2012) [2]

There are 3 main aspects to take into consideration when using search-based optimisation on a software problem:

1. *Representation* – To apply search-based algorithms to a search problem and optimise the solution the problem needs to be represented as a search space that can be explored and measured to improve its quality.

2. *Fitness function* – There needs to be a method for measuring the quality of the problem. This concerns which aspects of the software need to be optimised and which metrics are available to measure improvement.
3. *Search technique* – The optimisation technique itself needs to be chosen. The best optimisation technique may depend on the type of problem being addressed. Some may work better than others in certain situations. Other factors to consider are the amount of time and resources available. If a faster solution is more important than an optimal one, then this will inform the choice of the most suitable algorithm.

This chapter gives an introduction to the relevant areas of SBSE and outlines the methodology for the research in the thesis. Section 1.1 discusses how SBSE is applied to software maintenance and gives a brief outline of how SBSM has been used to improve the structure of software in previous research. Section 1.2 outlines the research aim and also formulates research questions to contextualise the scope of the thesis. Section 1.3 lists the contributions of the research within the thesis. Then, Section 1.4 gives an outline of each subsequent chapter in the thesis, briefly detailing the content in each.

1.1 Search-Based Software Maintenance

Software code can fall victim to what is known as *technical debt*. For a software project, especially large legacy systems, the structure of the software can be degraded over time as new requirements are added or removed. This increasing *software entropy* implies that over time, the quality of the software tends towards untidiness and clutter. This degradation leads to negative consequences such as extra coupling between objects and increased difficulty in adding new features. As a result of this issue, the developer often has to restructure the program before new functionality can be added or just to make the code more understandable or easier to amend.

SBSE has been used to automate this process, thus decreasing the time taken to restructure a program. Using a search-based algorithm, the developer starts with the original program as a baseline from which to improve. The measure of

improvement for the program can be subjective and accordingly can be done in a variety of different ways. The developer needs to devise a heuristic, or more likely a set of heuristics to inform how the structure of the program should be improved. Often these improvements are based on the basic tenets of object-oriented design where the software has been written in an object-oriented language (these tenets relate, among other things, to cohesion, coupling, inheritance depth, use of polymorphism and adherence to encapsulation and information hiding). Additionally, there are other sources of heuristics such as the SOLID principles introduced by Robert C. Martin [3]. The developer then needs to devise a set of changes that can be made to the software in order to enforce the heuristics. *Refactoring* is used to restructure existing software code without modifying the external functionality of the program. Different refactorings can be composed to change the software structure, and provide the changes needed to address technical debt. When the refactorings are applied to the software they may improve or degrade the quality, but regardless, they act as tools to modify the solution.

Using a SBSE approach, the refactorings are applied stochastically to the original software solution and then the software is measured using a *fitness function* consisting of 1 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 [4] and QMOOD [5] 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 1 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 [6] 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 at random 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.

Researchers have used SBSM to modify code structure in various ways. Many use tools to detect and list defects in the code. In other cases, the software will be refactored automatically in order to resolve them. Other approaches attempt to suggest sequences of refactorings for the developer to apply manually that can resolve the design defects. As an example, Figure 1.3 gives the design representing by 2 classes. To resolve potential issues with the design, a researcher may have a set of constraints to identify design defects in the code. An example of this would be if a class in the design contains more than a threshold amount of methods (a value supplied by the researcher to define the defect). In this case the class may be seen as too big, and this may be classified as a defect in the code. If this was applied in a SBSM approach and Class 1 in Figure 1.3 was determined to be too large, a refactoring could be applied to the code to move a method from Class 1 to Class 2. If enough refactorings were applied to move methods between classes, resulting in the second design shown in Figure 1.3, this defect could be resolved. This could then be repeated for other design defects to resolve as many as possible with the generated refactorings.

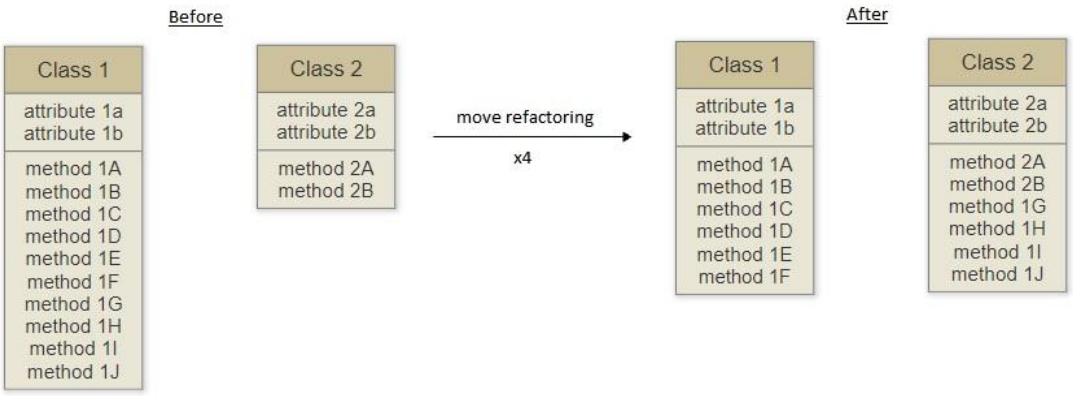

Figure 1.3– Resolving a Design Defect through the Application of Refactorings

Additionally, SBSM has been applied to different models of the software to inform the refactorings to apply. Refactorings can then be generated that can bridge that gap between the current model and an improved one. Another common technique is to determine a measure for quality in the software and use that to determine if a refactoring is good or bad. The software is refactored with no direct concern for design defects, and aims to improve the software quality itself. This can be used to resolve design defects in the code by improving its structure through useful refactorings.

1.2 Research Aim

The general aim of this work is to investigate current techniques in the area of SBSM and to improve upon the techniques available in order to make the process of software refactoring more effective for a realistic software development situation. The available refactoring tools as well as tools proposed from the literature on SBSM are inspected to devise where the current limitations lie. In response to the gaps found, an automated refactoring tool has been created providing a platform for further research. This platform is used to conduct experimentation for the remaining research aims of the thesis. The tool is used to compare different automated approaches.

The more traditional mono-objective approach to improving software with search-based techniques is compared against a multi-objective approach to derive the advantages and disadvantages of applying this approach within the context of a fully automated refactoring platform. The multi-objective approach is then used to test various different objective functions for measuring aspects of the software, and to derive whether the objective functions and tool improvements are effective in improving the specified property. An overall many-objective setup is then used to apply the research on the tool into an approach that can maintain and improve a software input in a practical way across various properties with minimal effort. These research objectives are addressed via the following research questions:

RQ1: What current refactoring and search-based software engineering tools are available?

RQ2: Can a fully automated, practical refactoring tool be developed using techniques from previous literature to improve the maintenance of software?

RQ3: How useful is a multi-objective search-based software maintenance approach in comparison with a mono-objective search-based approach?

RQ4: Can individual, novel objectives be measured and refactored in a software program to maintain the code while also improving the individual properties inspected?

RQ5: Can numerous individual objectives be combined into a fully automated, many-objective approach in order to improve a software program across multiple different properties in an additive fashion, without losing the improvement effect of any individual property?

The experimentation conducted is restricted to Java programs, in order to compare with other research that has mostly focused on this programming language. The scope of this research thesis is restricted to looking specifically at the maintenance process of software development with SBSE. The remainder of this chapter details the methods used to answer the research questions defined above.

1.3 Contributions

The primary contributions of the thesis that result from the research are outlined below:

1. A comparison has been documented on different search-based optimisation techniques used in SBSM along with an analysis of the advantages and disadvantages of the different approaches.
2. A new tool is developed and proposed for fully automated maintenance of Java software using mono-objective, multi-objective and many-objective search techniques.
3. A novel objective that takes in a range of software metrics to measure various structural aspects of the software is proposed and tested to represent quality in a software program.
4. An objective is proposed and tested to measure the priority of the classes refactored in a refactored solution. The objective will guide the refactorings in the search with respect to the relevant classes.
5. An objective is proposed and tested to measure and then take into consideration the code coverage of refactoring solutions generated.
6. An objective is proposed and tested to measure the recentness of the code elements refactored in a refactoring solution, in relation to a set of previous versions of the code.
7. The objectives proposed are combined into an overall framework to use with software in conjunction with the many-objective functionality in order to improve the software across various different properties.
8. The tasks constructed for all of the experimentation are implemented into the tool for use by others in the research community. The data gathered from the experimentation in the thesis is also included in an online repository hosting the tool.

1.4 Thesis Outline

Chapter 2 gives an overview of the search-based optimisation techniques used in the experimentation. The random search is discussed, as well as the local hill climbing (HC) search and the simulated annealing (SA) metaheuristic search. The general outline of a genetic algorithm (GA) is described, along with swarm intelligence algorithms (SIAs). Then, multi-objective and many-objective evolutionary algorithms (EAs) are described. In particular, the NSGA-II and NSGA-III algorithms are discussed. After the EAs are discussed, the chapter contains a detailed literature review that goes over the area of SBSE with respect to refactoring for software maintenance. Patterns and trends in the research are analysed, and gaps are outlined as well as the methods used to address them in the thesis. Chapter 3 describes the refactoring tool developed for the research. The components of the tool as well as the configuration and functionality are discussed. The advantages of the tool in relation to the alternatives are outlined and the online location of the source code is given. Chapter 3 also gives a review of preliminary experimentation conducted using the already existing A-CMA refactoring tool. The tool was used to explore the effectiveness of available refactoring tools for experimentation and research. Chapter 4 assesses the capabilities of the refactoring tool. The various configuration settings of the GA are tested, as well as the metrics available in the tool. The GA is compared against the multi-objective genetic algorithm (MOGA) to inform on the success of the multi-objective approach, and the metrics are ranked in order to construct a quality objective for future experimentation.

Chapter 5 details the construction of a priority objective for use in the MOGA. The objective takes as input a list of classes to favour in the refactoring solution and, optionally, a list of classes to disfavour. This objective is used in conjunction with the quality objective to test whether a refactoring solution can improve the quality of the software and also prioritise the specified classes in the solution with the refactorings. The chapter proposes another objective, to investigate the amount of code coverage given in the refactorings of a refactoring solution. This refactoring coverage objective is also tested by comparing a multi-objective approach with a mono-objective one (using only the quality objective).

The refactoring coverage objective aims to maximise the number of code elements in a software project that are refactored and decrease the number of refactorings applied to each individual element. This allows the refactoring solution to inspect as many areas of the code as possible and avoid redundant refactorings or solutions that focus too heavily on a single area. The final objective proposed in Chapter 5 is the element recentness objective. Like the priority objective, this takes an external input to help maximise the accuracy of the objective. Previous versions of the code are read in to help investigate how long the refactored elements of a solution have been present in the software. The objective aims to direct the search towards the more recently added areas of the code, in order to remove issues that may be present with the newly added code.

Chapter 6 combines the outcomes of the previous research in the thesis to construct an overall framework to address the research questions outlined. The 4 objectives are combined into a many-objective approach in order to test how successful the objectives can be when used together. Different permutations of the objectives are also tested to work out how well the different objectives can work with each other. Chapter 7 inspects the outcomes of the research and compares them with the other related work in the area of SBSM. It also goes over the limitations of the current research and possibilities for future work in the area.

Chapter 2

Optimisation Algorithms & Literature Review

There are numerous different candidate metaheuristic algorithms applicable in the SBSE field. These methods generally automate search-based problems through gradual quality increases. Local search algorithms move from solution to solution in the candidate solutions space by applying local changes, until either an acceptable solution is found or a restriction (e.g. time limit) is reached. In judging a solution, it is often compared against a random search. Additionally, solutions must be assessed for validity and a fitness function is used to evaluate whether the search should continue from that point or backtrack. Various metaheuristic algorithms are used to modify local search algorithms and improve their quality. Figure 2.1 displays a taxonomy of metaheuristic techniques available and their classifications.

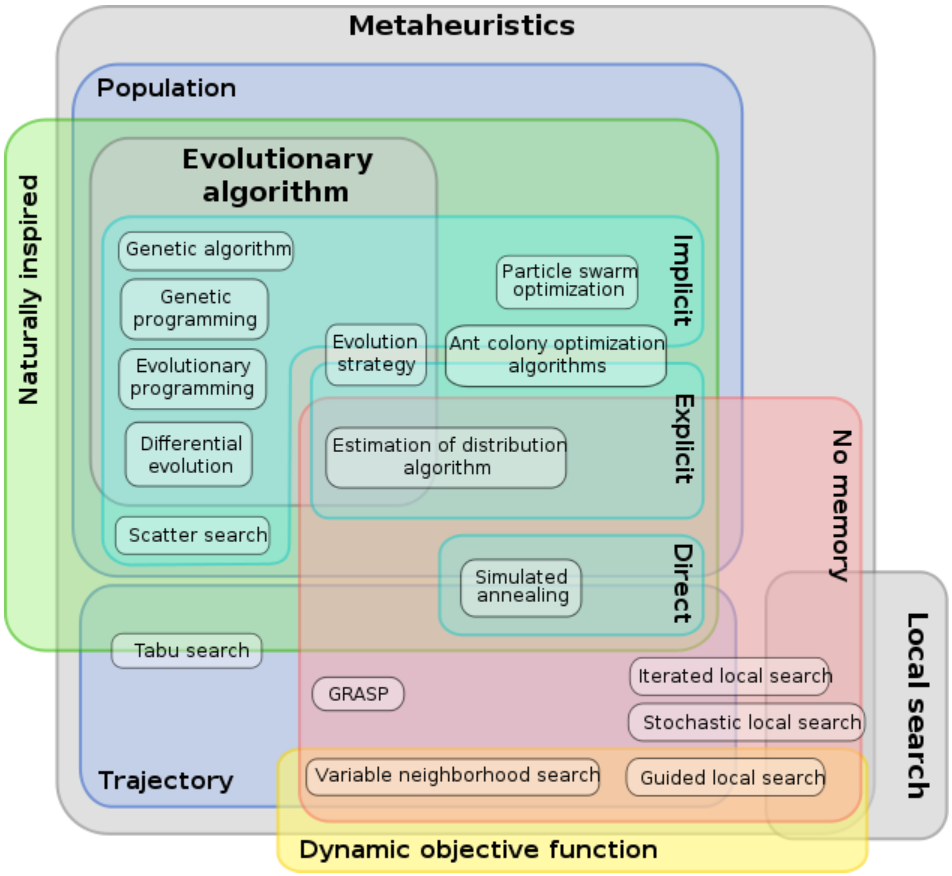

Figure 2.1 – Different Classifications of Metaheuristics[1]

This chapter details the search algorithms used in the experimentation chapters later in the thesis, before reviewing SBSM literature. The chapter is structured as follows. Section 2.1 looks at the random search, Section 2.2 looks at HC and Section 2.3 looks at SA. Section 2.4 details the GA, while Section 2.5 investigates SIAs. Section 2.6 explores multi-objective evolutionary algorithms (MOEAs), and in particular, NSGA-II. Section 2.7 discusses many-objective EAs and gives a description of its successor, NSGA-III. Section 2.8 covers the review of the captured literature of SBSM, as well as discussing related SBSE papers. The review is split into numerous subsections to capture commonly recurring areas, although there may be some overlap between a paper in one section with another section. Section 2.9 gives a meta-analysis of the papers reviewed, and finally, Section 2.10 outlines the gaps in the literature that have been derived through the analysis.

[1]Taken from http://nojhan.free.fr/metah/images/metaheuristics_classification.jpeg

2.1 Random Search

The random search is used as a benchmark for most search-based metaheuristic algorithms to compare against. If the proposed algorithm is not better on average than a random search, then it is not acceptable as a solution. A random search is conducted in a similar manner to a metaheuristic search although over each iteration the choices made are random. Although most metaheuristics also use a nondeterministic approach to making choices, in those cases the choice must be assessed for validity and a fitness function is used to evaluate whether the search should continue from that point or backtrack.

2.2 Hill Climbing

HC is a type of local search algorithm. With the HC approach, a random starting point is chosen in the solution, and the algorithm begins from that point. With software maintenance, this is a random point in the original project. From this point, a change is made, e.g. a refactoring is applied in the code, and the fitness function is used to compare the 2 solutions. The one with the highest perceived "quality" becomes the new optimum solution and the algorithm continues in this way. Over time, the quality of the solution is improved as less optimal changes are discarded and better solutions are chosen. Eventually, an optimal or sub-optimal solution is reached. With software refactoring, this means a modified program with the same functionality but a better structure.

There are 2 main types of HC search algorithm that differ in 1 aspect. First-ascent HC is the simpler version of the algorithm and works as a greedy algorithm. In this version, as the algorithm measures the quality of other variations of the solution adjacent to the current point, the first variation found with a better quality is used. This means that each time a change is made to the solution or a different permutation is inspected and this change is measured as an improvement, it is immediately incorporated and the algorithm is reiterated

with this as the current solution. The risk of being trapped into a local optimum is increased with this version of the algorithm as the solution isn't given as much freedom to explore different options and areas of change.

With steepest-ascent HC, all the available changes are made to the solution first and measured. Once all the available local changes in the current area have been inspected, the option with the biggest improvement in quality is chosen to move forward. Not only does this result in better quality choices in the short term, but it increases the chances of the search being able to escape the local optimum of the neighbourhood and explore better quality solutions globally. This is a superior choice for quality, but it takes more time and computation power to inspect every choice compared to first-ascent climbing which reaches an improved solution at a quicker pace. Other variations are stochastic HC, where neighbours are chosen at random and compared, or random-restart HC, where the algorithm is restarted at different points to explore the search space and improve the local optimum. The pseudocode for the HC search is shown in Algorithm 2.1.

Hill Climbing Algorithm

```
currentNode = startNode;
do
  L = NEIGHBORS(currentNode);
  nextEval = -INF;
  nextNode = NULL;
  for all x in L
    if (EVAL(x) > nextEval)
      nextNode = x;
      nextEval = EVAL(x);
  if nextEval ≤ EVAL(currentNode)
    // Return current node since no better neighbours exist.
    return currentNode;
currentNode = nextNode;
```

Algorithm 2.1 – Pseudocode for the Hill Climbing Algorithm[2]

[2]Taken from Wikipedia: http://en.wikipedia.org/wiki/Hill_climbing

2.3 Simulated Annealing

SA is a modification of the local search algorithm, used to address the problem of being trapped with a locally optimum solution. In SA, the basic method is the same as the HC algorithm. The metaheuristic checks stochastically between different variations of a solution and decides between them with a fitness function until it reaches a higher quality. The variation is that it simulates metallurgical annealing by introducing a *cooling factor* to overcome the disadvantage of local optima in the HC approach. The cooling factor adds an extra heuristic by stating the probability that the algorithm will choose a solution that is less optimal than the current iteration. While this may seem unintuitive, it allows the process to explore different areas of the search space, giving extra options for optimisation that would otherwise be unavailable. This probability is initially high, giving the search the ability to experiment with different options and choose the most desirable neighbourhood in which to optimise. This is then generally decreased gradually until it is negligible. This allows the algorithm to avoid making negative choices as it begins to reach an optimal solution. The probability given by the cooling factor is normally linked to a *temperature* value that is used to simulate the speed in which the algorithm *cools*.

The effectiveness of the process relies heavily on the input parameters used. The algorithm may need to be run numerous times to find the combination of inputs that result in the most effective permutation of the process. Also, the cooling process may allow the SA algorithm to search for maximum improvement within the search space, but its volatile nature means that the process may take longer to find the optimum solution, or may run indefinitely. To tackle this, a time limit or other restriction may be implemented to ensure the search doesn't take up too much time. The pseudocode for this search is shown in Algorithm 2.2.

Simulated Annealing Algorithm

```
// Initial state, energy.
s ← s0;
e ← E(s);
  // Energy evaluation count.
  k ← 0;
// While time left  not good enough:
while k < kmax and e > emax
  // Temperature calculation.
  T ← temperature(k/kmax);
    // Pick some neighbour.
    snew ← neighbour(s);
    // Compute its energy.
    enew ← E(snew);
    // Should we move it?
    if P(e, enew, T) > random() then
      // Yes, change state.
      s ← snew;
      e ← enew;
    // One more evaluation done.
    k ← k + 1;
```

Algorithm 2.2 – Pseudocode for the Simulated Annealing Algorithm[3]

2.4 Genetic Algorithms

GAs are a subset of EAs and in common with those mimic biological processes. GA processes imitate the combination of chromosomes in genetics via a crossover operator that effectively mixes 2 solutions in some way. GAs also introduce mutations to allow for inventive options (the objective of this being to free the search from the confines of one area by allowing it to explore and analyse different options). As shown in Figure 2.2 (adapted from an example in Vivanco and Pizzi's paper [7]) GAs, like other metaheuristic search techniques, use a fitness function to measure the quality among a number of different solutions (known here as *genes*) and prioritise them. At each generation (i.e. each iteration of the search), the genes are measured to determine *fitness*. At every generation, in order to introduce variation into the gene pool, a proportion of the population is selected and used to *breed* the new generation of solutions. The fitter solutions are, the more likely they are to be selected. This is done

[3]Taken from Wikipedia: http://en.wikipedia.org/wiki/Simulated_annealing

using a technique such as tournament selection. Tournament selection involves running several *tournaments* to compare solutions, where the fitter solutions will prevail.

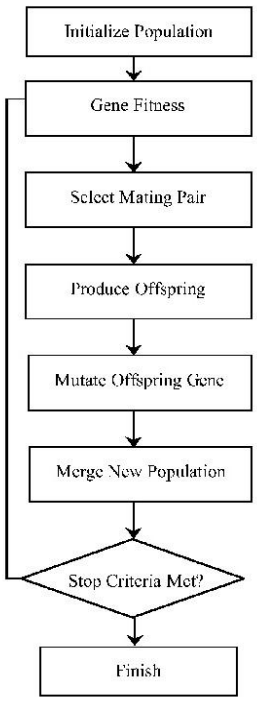

Figure 2.2 – Flow Chart of the Genetic Process

Two steps are used to create the new generation. First, a crossover operator is used to create the child solutions from the parents selected. The algorithm itself determines exactly how the crossover operator works, but generally, selections are taken from each parent and spliced together to form a child. Once the child solutions have been created, the second step is mutation. Again, the mutation implementation depends on the GA adaptation, but an example would be that a bit or number of bits is inverted in the solution. The mutation is used to provide a random change in the solutions to maintain variation in the selection of solutions and prevent early convergence to the optimal solutions. A percentage of child solutions are selected for mutation, and after this occurs they are inserted back into the gene pool. At this point the algorithm calculates the fitness of any new solutions and reorders them in relation to the overall set. Generally, a population size is specified, and the weakest solutions are culled

each generation. This process is repeated until a termination condition is reached. Algorithm 2.3 shows the GA pseudocode taken from Räihä's survey of search-based software design [8].

```
Genetic Algorithm
Input: formalization of solution, initialSolution
 chromosomes ← createPopulation(initialSolution);
 while NOT terminationCondition do
  foreach chromosome in chromosomes
   p ← randomProbability;
   if p > mutationProbability then
    mutate(chromosome) ;
   end if
  end for
  foreach chromosomePair in chromosomes
   cp ← randomProbability;
   if cp > crossoverProbability then
    crossover(chromosomePair);
    addOffspringToPopulation();
   end if
  end for
  foreach chromosome in chromosomes
   calculatefitness(chromosome) ;
  end for
  selectNextPopulation();
 end while
```

Algorithm 2.3 – Pseudocode for the Genetic Algorithm

2.5 Swarm Intelligence Algorithms

Ant colony optimization (ACO), particle swarm optimization (PSO) and artificial bee colony (ABC) are similar techniques used to find the shortest path to a solution using metaheuristic techniques and swarm intelligence. ACO simulates the behaviour of a swarm of ants when they look for a source of food. The ants will initially look randomly for food, and when found, exude pheromones for other ants to follow as they return to the colony. Over time, the pheromones will fade, meaning that longer paths will lose the pheromone trails quicker, whereas shorter paths will maintain pheromone trails longer. This causes the ants to follow the shorter trails and, over time, these trails will become more

highlighted as more ants follow them, whereas the less efficient trails will fade away. The positive feedback will eventually cause all the ants to follow the single, more efficient path. The algorithm simulates the behaviour of the ants in a colony by initially exploring the solution space randomly with multiple agents. The better a path is the higher the probability that the path will be chosen by an agent and this is used to allow the "ants" to converge on an optimal solution. Trails will be updated with *pheromones* and a pheromone evaporation coefficient will be used to simulate the dissipation of the pheromone trails over time.

Similarly, PSO is used to simulate social behaviour in a group. The method is based on behaviours such as how birds flock together or how fish swim together in groups. Again, this technique uses multiple agents to explore a search space to find a better solution. Each agent will be affected by what they know to be the best local solution, but also by the best known global position. With PSO, these agents are known as *particles* and they will explore the search space according to their *position* and *velocity*. The entire swarm of particles will have a best position and it will be used each iteration to guide the particles to the optimal global position. The process is repeated until the swarm converges to the same solution. To avoid the convergence happening too early and the swarm being trapped in a local optimum, the information given to each particle can be limited to the best known positions of *sub-swarms* around the particle. This will give the local best position and the global solution can be found comparing these local optimum solutions.

The ABC algorithm works by simulating the behaviour of honey bees foraging nectar from food sources. In this algorithm, the food sources represent potential solutions and their nectar content represents the fitness. There are 3 groups of bees used to find the food sources. *Employed* bees each correspond to a food source, and can memorise 1 food source position at a time. The employed bee will go to its corresponding food source, evaluate its nectar amount, and go back to the hive. *Onlooker* bees compare the amount of nectar in the food sources that correspond to the employed bee in the hive. Using this information, the onlooker will become an employed bee and choose a food source to go to. Then, it will search for a nearby food source and evaluates its nectar amount. After comparing the 2, if the new source has more nectar, its source will be memorised and the other food source will be abandoned. Employed bees whose food source

has been abandoned become *scout* bees and search for a new food source to replace those that have been abandoned. This process is iterated with scout bees producing new random solutions, employer bees finding and comparing neighbouring solutions and onlooker bees evaluating the best solution in the current population.

2.6 Multi-Objective Evolutionary Algorithms

Multi-objective algorithms are used to tackle problems that have multiple constraints or objectives, and involve more than 1 objective function to be optimised simultaneously. EAs are a suitable choice to apply to multi-objective problems due to their ability to generate multiple possible solutions to a problem instead of only 1. This way multiple conflicting objectives can be addressed with various possible solutions without the need to assign priorities to any individual objective in order to decide on a single, globally optimal solution. Multi-objective algorithms have been applied sparsely to SBSE problems [9]–[24] and only recently have been used to address issues in SBSM (possibly because of the difficulty involved in implementing a multi-objective approach for automated software maintenance, as suggested by Mkaouer *et al.* [25]). Regardless, looking at SBSM with a multi-objective perspective is fitting. When maintaining a software project, there are likely numerous conflicting objectives. A multi-objective algorithm can be used to consider the objectives independently instead of having to combine them into 1 overarching property to improve. The downside of using multi-objective algorithms for software maintenance over a mono-objective metaheuristic algorithm is that the extra processing needed to consider the various objectives can cause an increase in the time needed to generate a set of solutions. Another issue is that when a MOEA generates a population of solutions, the *best* solution is up to the interpretation of the user, depending on which objective fitness functions are considered most important. On the other hand, this gives the user multiple options depending on their desire or the situation.

Most MOEAs use Pareto dominance [26] in order to restrict the population of solutions generated. If, for a solution, at least 1 objective of that solution has a

better fitness value than in another solution and none of the objectives are worse, that solution is said to dominate the other solution. Therefore, a solution is nondominated if none of the other solutions in the population dominate it. Table 2.1 lists MOEAs that use Pareto dominance to choose solutions and a survey of MOEAs is given by Coello Coello [26]. The most popular MOEA available and the one that has been used for SBSM is NSGA-II.

Table 2.1 – Multi-Objective Evolutionary Algorithms That Use Pareto Dominance

MOEA	Full Name	Developers
DMOEA	Dynamic Multiobjective Evolutionary Algorithm	Yen and Lu [27]
M-PAES	Memetic-Pareto Archive Evolutionary Strategy	Knowles and Corne [28]
NGPA	Niched Pareto Genetic Algorithm	Horn *et al.* [29]
NSGA-II	Nondominated Sorting Genetic Algorithm II	Deb *et al.* [30]
PAES	Pareto Archive Evolutionary Strategy	Knowles and Corne [31]
PDE	Pareto-frontier Differential Evolution	Abbass *et al.* [32]
PESA	Pareto Envelope-based Selection Algorithm	Corne *et al.* [33]
SPEA	Strength Pareto Evolutionary Algorithm	Zitzler and Thiele [34]
SPEA2	Strength Pareto Evolutionary Algorithm 2	Zitzler *et al.* [35]

2.6.1 NSGA-II

NSGA-II, proposed by Deb *et al.* [30], was created to improve on the original Nondominated Sorting Genetic Algorithm (NSGA) [36]. As with GAs and their status as a subset of EAs, MOGAs like NSGA are a particular type of MOEA. Like other MOEAs, it uses Pareto dominance to choose the desirable population of solutions. NSGA-II organises the possible solutions into different nondomination levels and further discerns between them by finding the objective distances between them in Euclidean space. The original NSGA approach has been criticised for being computationally expensive and for not using elitism, as other MOEAs do (elitism ensures that the best solutions are preserved and carried into the next generation in an EA). It also uses a sharing parameter in order to ensure diversity in the population of solutions generated, whereas a parameter-less diversity-preservation mechanism is desirable. NSGA-II addresses these points to propose an improved MOEA. In their proposal, Deb *et al.* test the diversity levels of the algorithm against contemporary elitist MOEAs and find that it outperforms the PAES and SPEA algorithms.

NSGA-II proposes a fast nondominated sorting approach (due to fewer comparisons between solutions), improving upon the complexity of the original approach used by NSGA. It looks at the current population of solutions to find the set of solutions that is nondominated. For each possible solution, Pareto dominance is used to compare it against each other solution. After inspecting each, the set of solutions that are not dominated by any other in any of the objectives are given a rank of 1. The remaining solutions are then re-inspected, excluding the set of nondominated solutions. The set of solutions here that are now nondominated within the remaining group are given a rank of 2. This continues until all the solutions are assigned a rank, with a lower rank meaning that fewer solutions dominate that solution.

Depending on the desired population size in the algorithm, some solutions with a higher rank may need to be excluded from the population. In order to find the subset of least desirable solutions within a set considered equally fit, a crowding distance value is calculated for the solutions in that rank. This supplementary measurement estimates the density of solutions surrounding a solution in the objective space. To calculate this, the 2 closest solutions on either side for each objective are taken and the distance between them is used. In order to choose the adjacent solutions, the population is sorted for the corresponding objective in ascending order of magnitude. The solutions with the smallest and largest values for that objective are assigned an infinite distance value. All other intermediate solutions are assigned a distance value equal to the absolute normalised differences in the function values of the 2 adjacent solutions. The overall crowding distance value is then calculated as the sum of the distances for each objective. Equation 2.1 shows the crowding distance calculation for a single solution, where i represents the position of the solution in the sorted population for the relevant objective, m represents the current objective and $f_m i$ represents the fitness of the solution for the relevant objective. f_m^{max} and f_m^{min} are the largest and smallest fitness values among the population of that nondominated rank for that objective. n represents the number of objectives used in the search.

$$\sum_{m=1}^{n} \left(\frac{f_m(i+1) - f_m(i-1)}{f_m^{max} - f_m^{min}} \right) \tag{2.1}$$

The crowding distance measurement with 2 objectives is shown in Figure 2.3 with the crowding distance calculated for one solution by comparing it with adjacent solutions within the same rank. When the crowding distances are calculated for each solution in the set, the solutions with the smaller crowding distances (i.e. the solutions located in more densely crowded regions) are left out of the population. The solutions with the higher crowding distances (i.e. the solutions located in less crowded regions) are considered fitter as they contribute to a more uniformly spread out Pareto front, allowing for a more diverse population. The crowding distance values are used to replace the sharing parameter used in the NSGA approach, allowing for a more dynamic method of ensuring diversity. Each generation, the fitness process is repeated for the current population along with any newly created solutions, allowing for the highest ranked nondominated solutions to be kept and allowing elitism to be introduced in the algorithm. By the end of the algorithm, the fittest solutions will be included in the final population and the less fit solutions will have been culled. Algorithm 2.4 gives the pseudocode of the NSGA-II procedure.

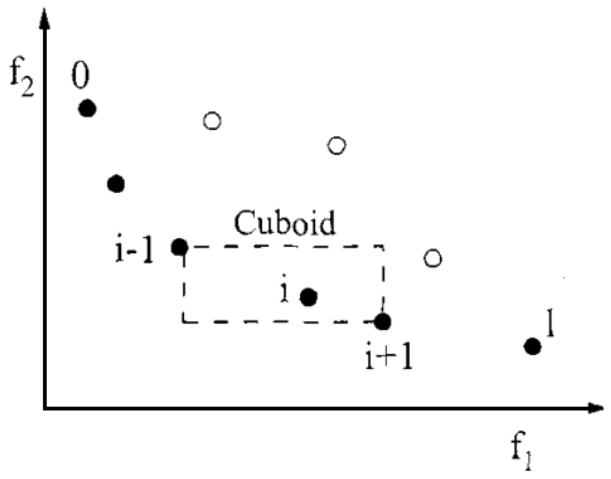

Figure 2.3 – Crowding Distance Calculation Showing Solutions from Two Different Ranks [30]

NSGA-II

Create an initial population P_0
Generate an offspring population Q_0
$t = 0$;
while *stopping criteria not reached* **do**
  $R_t \leftarrow P_t \cup Q_t$;
  $F \leftarrow$ fast-non-dominated-sort (R_t);
  $P_{t+1} \leftarrow \emptyset$;
  i $\leftarrow$ 1;
  while $|P_{t+1}| + |F_i| \leq N$ **do**
    Apply crowding-distance-assignment (F_i);
    $P_{t+1} \leftarrow P_{t+1} \cup F_i$;
    $i \leftarrow i + 1$;
  end while
  Sort (Fi ‹ n);
  $P_{t+1} \leftarrow P_{t+1} \cup F_i[1 : N - |P_{t+1}|]$;
  $Q_{t+1} \leftarrow$ create-new-population (P_{t+1});
end while

Algorithm 2.4 – Pseudocode for NSGA-II [37]

2.7 Many-Objective Evolutionary Algorithms

Many-objective algorithms are multi-objective algorithms that are designed to handle more than 3 objectives (with most practitioners agreeing to a maximum of 10 to 15 objectives). The consensus [20], [32], [33] is that multi-objective algorithms like NSGA-II cannot adequately handle problems involving more than 3 objectives. There are numerous reasons that MOEAs using Pareto dominance can have difficulty handling more than 3 objectives. When the dimensionality of the problem increases, an increasingly larger fraction of the population becomes nondominated. This makes it more difficult to sort the solutions in a population in any useful way (with a decreased number of ranks to compare), and decreases the chances of creating new solutions in a generation. The increased number of objectives also means that the fitness calculation and diversity measure becomes more computationally expensive. The crossover process may also become inefficient as the parent genomes will be more likely to be widely distant from each other. With the parents being less likely to be among the fittest of the population, the offspring generated will also be less likely to be useful, meaning that the genetic process will have less diversity. The Pareto front itself will need to include more solutions to represent the increased

number of objectives and address higher dimensional trade-offs. This can make it more difficult to choose a preferred solution as there will be a larger set of solutions to choose between. Another issue is that the Pareto front is more difficult to visualise in more than 3 dimensions, adding to the difficulty in choosing a solution among the Pareto optimal set.

Numerous different approaches have been used to address the issues with MOEAs and develop many-objective alternatives. Table 2.2 lists some many-objective EAs. So far, for SBSE, the use of these algorithms has been scarce. Salam *et al.* [34], [35] have explored the use of IBEA with software product lines and Mkaouer *et al.* [20], [34] have used NSGA-III to optimise up to 15 objectives for software maintenance. The following subsection details the approach used by NSGA-III to tackle more than 3 objectives.

Table 2.2 – Many-Objective Evolutionary Algorithms That Use Pareto Dominance

Algorithm	**Full Name**	**Developers**
GrEA	Grid-Based Evolutionary Algorithm	Yang *et al.* [43]
HypE	Hypervolume Estimation Algorithm For Multiobjective Optimization	Bader and Zitzler [44]
IBEA	General Indicator-Based Evolutionary Algorithm	Zitzler and Künzli [45]
MOEA/D	Multiobjective Evolutionary Algorithm Based On Decomposition	Zhang and Li [46]
MSOPS	Multiple Single Objective Pareto Sampling	Hughes [47]
N/A	Ranking Dominance-Based Algorithm	Kukkonen and Lampinen [48]
NSGA-III	Nondominated Sorting Genetic Algorithm III	Deb and Jain [49]
PBEA	Preference-Based Evolutionary Algorithm	Thiele *et al.* [50]
PCA-NSGA-II	Principal Component Analysis NSGA-II	Deb and Saxena [38]
PCSEA	Pareto Corner Search Evolutionary Algorithm	Singh *et al.* [51]
PICEA	Preference-Inspired Co-Evolutionary Algorithm	Wang *et al.* [52]
POGA	Preference Order-Ranking Based Algorithm	Di Pierro *et al.* [53]
r-NSGA-II	Reference Solution-Based NSGA-II	Said *et al.* [54]
R-NSGA-II	Reference Point-Based NSGA-II	Deb and Sundar [55]

2.7.1 NSGA-III

The approach used by NSGA-III to improve the process for many-objective problems incorporates a combination of approaches used by other many-objective algorithms. NSGA-III uses multiple predefined targets in the objective space to guide the search. Using multiple different targets allows the population

of solutions to retain their diversity. Furthermore, in order to avoid issues with mating solutions that are too diverse from each other, solutions from neighbouring targets can be used to develop new offspring. NSGA-III uses predefined reference points to improve upon the crowding distance calculations used in NSGA-II.

NSGA-III

Input: H structured reference points Z^s, parent population P_t
Output: P_{t+1}
Begin
$S_t \leftarrow \emptyset$
$i \leftarrow 1$;
$Q_t \leftarrow$ Variation (P_t);
$R_t \leftarrow P_t \cup Q_t$;
$(F_1, F_2, \ldots) \leftarrow$ Non-dominated_Sort (R_t);
Repeat
  $S_t \leftarrow S_t \cup F_i$;
  $i \leftarrow i + 1$;
Until $|S_t| \geq N$;
// *Last front to be included.*
$F_l \leftarrow F_i$; $\cup F_i$;
If $|S_t| = N$ **then**
  $P_{t+1} \leftarrow S_t$;
Else
  $P_{t+1} \leftarrow \cup_{j=1}^{l-1} F_j$;
  // *Number of points to be chosen from* F_l.
  $K \leftarrow N - |P_{t+1}|$;
  // *Normalize objectives and create reference set* Z^r.
  Normalize $(F^m; S_t; Z^r; Z^r)$;
  // *Associate each member s of* S_t *with a reference point.*
  // $\pi(s)$*: closest reference point.*
  // *d(s): distance between s and* $\pi(s)$.
  $[\pi(s), d(s)] \leftarrow$ Associate (S_t, Z^r);
  // *Compute niche count of reference point* $j \in Z^r$.
  $\rho_j \leftarrow \sum_{s \in St/Fl} ((\pi(s) = j) \ ? \ 1 : 0)$;
  // *Choose K members one at a time from* F_l *to construct* P_{t+1}.
  Niching $(K, \rho_j, \pi(s), d(s), Z^r, F_l, P_{t+1})$;
End if
End

Algorithm 2.5 – Pseudocode for NSGA-III [25]

NSGA-III still uses the nondominated functionality of its precursor algorithms. Therefore, Pareto optimality is still used to choose the best solutions in a

population from its top ranks, but the new functionality will help to choose which solutions from the remaining applicable rank are kept, as the crowding distance functionality did in NSGA-II. The improvements to the algorithm are made to replace the crowding distance calculations and maintain the diversity of the solutions generated. In place of the crowding distance calculations, the algorithm contains a number of stages. These stages will be used to locate the relevant reference points and choose a set of solutions that maintain a nice overall spread of Pareto optimal solutions in each generation. Algorithm 2.5 gives the pseudocode for the main functionality of NSGA-III along with an overview of the stages used to select the remaining solutions for a generation. These stages are detailed below.

The reference points can either be predefined in a structured manner or supplied preferentially by the user. In the case that no preferential information is given, Deb and Jain advise that any structured placement of the reference points can be used, but they adopt the systematic approach proposed by Das and Dennis [56] by placing points on a normalised hyperplane in objective space. A hyperplane is a subspace of 1 dimension less than its ambient space, where here the ambient space makes up the dimensions representing each objective in a many-objective problem. The hyperplane will be a simplex (n-dimensional representation of a triangle/tetrahedron) that is equally inclined on all axes. Figure 2.4 visualises this for a 3 objective problem with a 2 dimensional hyperplane. The reference points distributed across the normalised hyperplane will assist in choosing the solutions to keep at the end of each generation.

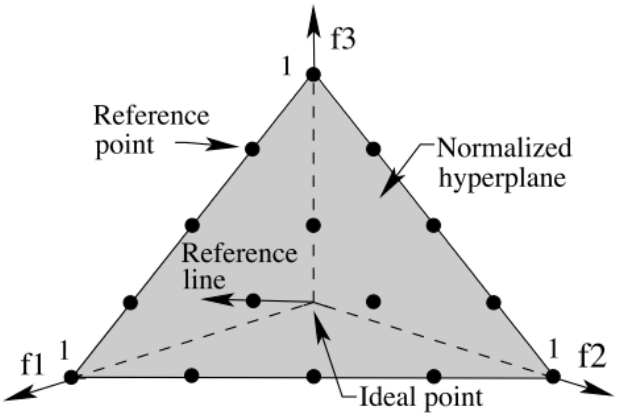

Figure 2.4 – Determination of Points on a Normalised Reference Plane in a Three Objective Case [49]

The normalisation process allows for objective values that are differently scaled in a Pareto optimal front. In order to normalise the hyperplane, the available solutions need to be adaptively normalised each generation. For each objective, the best value that has been reached for it so far is calculated. This allows for an ideal point to be found in the objective space that is mapped from the ideal values of each objective. This becomes a zero vector as it is used to find the translated objective values for each solution. Translated objective values are calculated by subtracting the corresponding objective values of the ideal point to denote how close to the ideal point a solution is. The extreme point for each objective is also found and these vectors are used to mark the boundaries of the hyperplane. The hyperplane can then be normalised by finding the intercepts of the extreme points with the axes. This also allows the solution vectors to be further normalised using the distance between the ideal and worst points for each objective. The construction of the hyperplane is shown in Figure 2.5. The reference points calculated using Dan and Dennis's approach will lie on this normalised hyperplane. As reference points are to be widely distributed on the hyperplane, solutions associated with the reference points are also likely to be widely distributed on or near the Pareto optimal front.

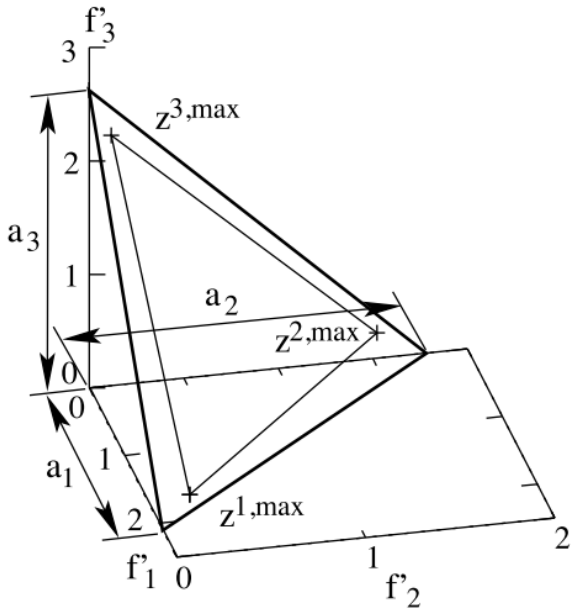

Figure 2.5 – Hyperplane Formed from Extreme Points in a Three Objective Case [49]

Once the hyperplane is constructed and the reference points are chosen, each solution needs to be associated with a reference point. In order to do this, a reference line is defined for each reference point that connects the reference point to the ideal point. For each solution in the population, the perpendicular distance is calculated between the solution and the reference line of each of the reference points. The smallest distance represents the reference point that is closest to that solution and the solution will be associated to the corresponding reference point. Figure 2.6 visualises the reference line calculation and association of solutions for a 3 objective problem. Once this is complete, the number of solutions associated with each reference point is counted. Only the solutions that have been chosen for the next population are included in this count. This is known as the niche count of the reference point. Using these, the solutions to keep from the final front are chosen. In order to improve the diversity of the solutions chosen, the reference points with the smallest niche count are inspected. These will represent the least dense areas of the Pareto optimal front. First the reference point with niche count of 0 will be chosen, if available. If there is more than 1 reference point with this count, one is chosen at random. From the remaining front, if there are any solutions associated with this reference point, the one with the shortest perpendicular distance is chosen

for the next population. The niche count of the reference point is then incremented. Otherwise, if there are no solutions from the remaining front associated with this reference point, it is excluded from consideration. If the niche count for the reference point chosen is more than 0 (and there are multiple possible associated solutions to choose from), the associated solution is picked randomly for the reference point. This process is repeated until the required number of solutions is chosen for the next population.

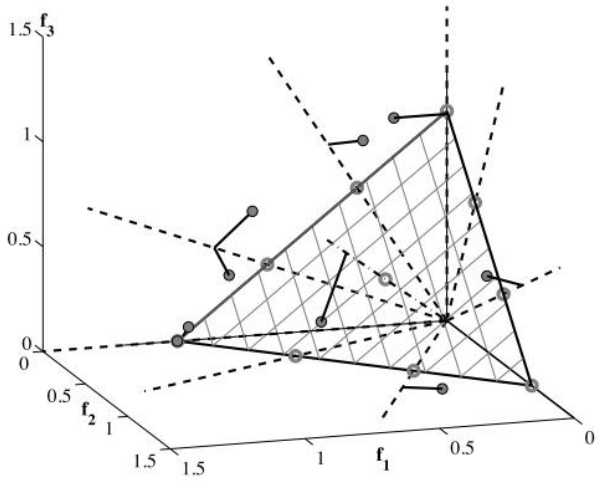

Figure 2.6 – Association of Solutions with Reference Points in a Three Objective Case [49]

Deb and Jain do not employ any explicit selection operator in the algorithm (as the use of reference points already allows for a careful elitist selection of solutions and a way to maintain diversity among the population) and apply the usual crossover and mutation operators from NSGA-II. As such, they have set the population size to be almost equal to the number of reference points in order to give equal importance to each population member. If desired, the algorithm can be run without any additional parameters, although the location of the reference points can be influenced by preference information and the number of reference points can be defined. Jain and Deb later detail an adaptive approach [57] to the algorithm that can add and remove reference points throughout the search, in order to relocate the useless reference points. The useless points are the ones that have no associated solutions and are thus excluded. If there is more than 1 solution associated with a reference point, more reference points

are added around this point in order for each solution to be associated with a separate point, and be more likely to be chosen. A simplex of points is added around the initial reference point, and any that already exist as a reference point are excluded. After this is done for all the relevant reference points the niche counts are updated. For any of the newly added reference points that still have a niche count of 0, they are removed. The original reference points are kept, even if their niche count is 0. At this point there should be a number of reference points with a niche count of 1 that corresponds with the number of solutions. Figure 2.7 gives an example of how the solutions may be dispersed across the reference points and shows how new reference points are added to address this.

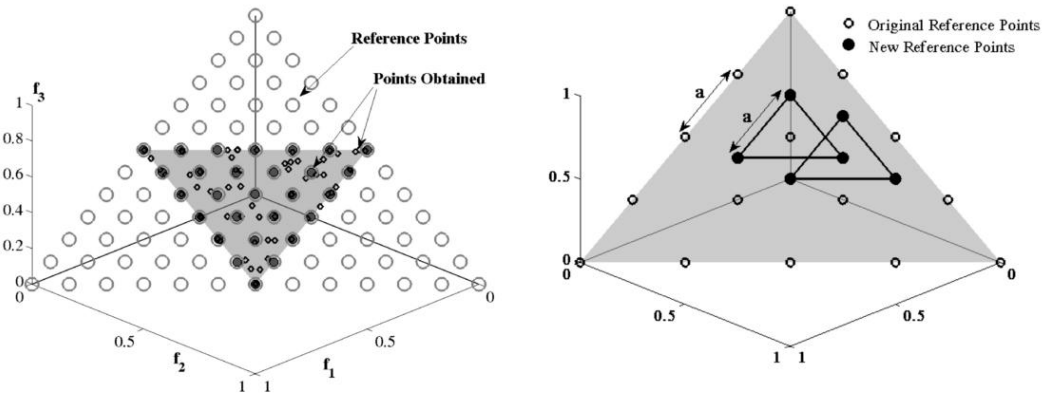

Figure 2.7 – Dispersion of Solutions on the Reference Plane and Addition of Reference Points in a Three Objective Case [57]

2.8 Search-Based Software Maintenance

For the literature review, Google Scholar, IEEE Xplore, ScienceDirect, Springer and Scopus were used to find relevant papers by using the search string "search AND based AND software AND engineering AND maintenance AND refactoring AND metaheuristic". We used AND to connect the keywords as using OR or a combination of the 2 would have been too general, giving hundreds of thousands of results in Google Scholar. The search was conducted by looking for the words anywhere in the article, rather than looking only within the article title or

elsewhere. The amount of papers found in each search repository is given in Table 2.3. Of the 293 papers found with the search (last searched in September 2016), the results were analysed and reduced to only include papers relevant to SBSM (specifically, using refactoring for software maintenance) and involved 1 or more of the following:

- Refactoring with search-based techniques.
- Automated refactoring.
- Investigation of maintenance metrics with search-based techniques.
- Investigation of the search-based optimisation process.

Likewise, the following papers were excluded:

- Papers that involved defect detection but not resolution.
- Papers that were written in a language other than English

Table 2.3 – Amount of Results in Each Repository

Search Repository	Number Of Papers
Google Scholar	293
IEEE Xplore	21
ScienceDirect	24
Springer	27
Scopus	43

A number of other areas of research were captured in the search and removed from the final count. Many papers were found that concerned a similar area of research. The other recurring areas are shown in Table 2.4, and related papers from these and other areas are mentioned below. The papers from similar areas were analysed manually to ensure that relevant papers were not lost from the review. More general papers related to SBSE are also briefly discussed below. Of all the papers analysed, 52 were found to be relevant. On top of this, a number of relevant papers were found on Google Scholar and the IEEE Xplore database by analysing references, researcher profiles and by discussion with other researchers, as well as conducting similar searches. Overall, the number of papers reviewed came to 99. The tables in Appendix D give a list of the papers, as well as the authors and year published.

Table 2.4 – Other Areas of Research Captured in Literature Search

Similar Areas	Others
defect detection	testing
modelling	software product lines
software architecture	class responsibility assignment
clustering	
code clone detection	
formal concept analysis	
relational concept analysis	

2.8.1 General Search-Based Software Engineering

Harman and Jones [1] wrote a paper about SBSE when the area of research was in its infancy. They argued that the metaheuristic algorithms commonly being used in other areas of science should be applied to computer science, and explained how to reformulate software engineering as a search problem using a representation, a fitness function and operators. Clarke *et al.* [58] explained how metaheuristic search techniques could be used in various areas of software engineering to solve problems that could not be attacked exhaustively. An overview is given of the local search techniques of HC, SA and tabu search, as well as GAs and GP. For each technique the key ingredients to define for it were also given. Harman and Clark [59] discussed the use of metrics in SBSE and their utility as fitness functions. They also discussed possible methods for the representation of the fitness landscape and the difficulty of mapping the landscape to a visual representation. Harman [60] wrote a paper to highlight the current progress made in the area of SBSE and topics of future interest that he thought would be important to research.

Harman also discussed the area of program comprehension [61], detailing the work already done in the area and outlining further options for research to improve program comprehension. Harman also discussed the virtual nature of software [62] and explained why this gives it an advantage when it comes to search-based optimisation. Harman [63] wrote an article about the effect that evolutionary computation has had on software engineering research in the past decade. In it, he gives a brief review of research in SBSE, with a focus on the role of testing. He also highlights open problems and challenges in the area for researchers to address in the future. Barros and Dias Neto [64] evaluated the

assessment of threats to validity in SBSE papers from the first 2 editions of the International Symposium on Search-Based Software Engineering. They outlined different possible threats to internal, external, construct and conclusion validity in SBSE experiments. Then they applied a questionnaire to 23 different SBSE papers to find them. DeFreitas and DeSouza [65] performed a bibliometric analysis of SBSE papers published in the years 2001-2010. They analysed various aspects of the papers published, across the 4 categories of publications, sources, authorship and collaboration.

Colanzi *et al.* [66] gave a review of the growth of SBSE in Brazil in 2011 and updated it in 2012 [67]. A summary was given of the work done in each area of SBSE as well as a description of the different algorithms used and the research impact made. Brown *et al.* [68] discussed technical debt as used as a metaphor for software systems and Allman wrote an article [69] discussing technical debt in software and how to handle it. Chatzigeorgiou *et al.* [70] proposed a method for calculating when the technical debt build up in a software project has exceeded the initial savings from ignoring maintenance. The initial savings, referred to as the *principle* need to be paid back as technical debt, and eventually as the project progresses, the technical debt will become greater than this initial amount if it isn't resolved. Morgenthaler *et al.* [71] discussed technical debt in relation to software code at Google. They outlined the various types of build debt that accumulate on the code and how they attempt to address it.

Fatiregun *et al.* [72] wrote a short paper, which was later extended [73], on search-based transformations. In it, they described what program transformations are and how they can be applied using SBSE. Jiang [74] briefly discussed the effectiveness of using GAs in SBSE. He argued that, while experimentation has shown that GAs can improve the solutions of software engineering problems, there is no rigorous proof that GAs can find the optimal or sub-optimal solutions in the software engineering problem domains. This was extended when Jiang *et al.* explored a theoretical description of the properties of GAs [75]. De Souza *et al.* [76] investigated the human competitiveness of SBSE techniques in 4 areas of software engineering. Across the board, the SBSE techniques outperformed the human competitors in terms of quality, speed and

lack of deviation, suggesting that SBSE techniques are indeed human competitive.

2.8.2 Related Areas

One of the areas that are closely related to SBSM is that of search-based approaches as applied to module clustering. Module clustering has a number of benefits including program comprehension, promotion of cohesion and reduction of the search space for search-based algorithms [77]–[94]. Defect detection concerns itself with finding code smells and design defects in the software code. Although some automated refactoring research contains a step for defect detection as part of the refactoring process, there has been work published that is concerned only with the detection process. Several papers look specifically at detecting design defects and code smells [95]–[101]. In particular, Dudziak and Wloka created the J/Art tool [102] to detect structural weaknesses in Java code. It can also perform limited restructuring capabilities for the design defects that are found using refactorings, although this is limited in comparison.

Error detection and resolution is an area of search-based optimisation that is closely related to maintenance. The first step of SBSM is to look for issues in the code and then apply refactorings to resolve them while error removal follows a similar process to find problems in the code and then try to solve them. The following papers are concerned with fault detection and program repair [103]–[106]. Editorials and workshop reports have been written related to SBSE and SBSM, and introductions have been written for various tutorials and talks based on SBSE. These are listed in Table D.3. Additionally, there has already been a number of literature reviews relating to the field of SBSE, or a specific area in SBSE, and they are specified in Table D.4.

2.8.3 Refactoring to Improve Software Quality

Ó Cinnéide and Nixon [107] developed a methodology to refactor software programs in order to apply design patterns to legacy code. They created a tool to convert the design pattern transformations into explicit refactoring techniques that can then be automatically applied to the code. The tool, called DPT (Design Pattern Tool), was implemented in Java and applied the transformations first to an abstract syntax tree that was used to represent the code, before changes were applied to the code itself. The tool would first work out the transformations

needed to convert the current solution to apply the desired pattern (in the example study a plausible precursor was chosen first). It then converted the pattern transformations into a set of minipatterns. These minipatterns would then be further decomposed, if needed, into a set of primitive refactorings. The minipatterns would be reused if applicable for other pattern transformations.

The authors analysed the Gamma *et al.* [108] patterns to determine whether a suitable transformation could be built with the applicable minitransformations. They found that while the tool generally worked well for the creational patterns, applying structural patterns and behavioural patterns caused problems. In a different paper [109], more detail was given on the tool and how it is used to apply the Factory Method pattern, and in another subsequent paper [110], Ó Cinnéide defined further steps of work to test the applicability of the tool.

O'Keeffe and Ó Cinnéide [111] continued to research the area of SBSM by developing a tool called Dearthóir. They introduce Dearthóir as a prototype tool used to refactor Java code designs automatically using SA. The tool used 2 refactorings to modify the hierarchical structure of the target program design. Again, the refactorings must preserve the behaviour of the program in order for them to be applicable. They must also be reversible in order to use the SA method. To measure the quality of the solution, the authors employed a small metric suite to analyse the object-oriented structure of the program. The metrics were measured for each class in the program and a weighted sum was used to give an overall fitness value for the solution. A case study was employed to test the effectiveness of the tool. A simple 6-class hierarchy was used for the experiment. The tool was shown to restructure the class design to improve cohesion and minimise code duplication.

Further work [112] introduced more refactorings and different metrics to the Dearthóir tool. Due to the possibility of the metrics conflicting with each other they were then given dependencies and weighted according to the authors' judgement. Another case study was used to detail the actions of the tool and the outcome was evaluated using the value of the metrics before and after the tool was applied. Every metric used either improved or was unchanged after the tool had been applied, indicating that the tool had been successful in improving the structure of the solution design.

O'Keeffe and Ó Cinnéide developed the Dearthóir prototype into the CODe-Imp platform (Combinatorial Optimisation for Design Improvement). They introduced it initially as a prototype automated design improvement tool [113] using Java 1.4 source code as input. CODe-Imp uses abstract syntax trees to apply refactorings to a previously designed solution, and has been given the ability to implement first-ascent or steepest-ascent HC as well as SA. They based the set of metrics used in the tool on the QMOOD (Quality Model for Object-Oriented Design) model of software quality [5]. Six refactorings were available initially, and 11 different metrics are used to capture flexibility, reusability and understandability, in accordance to the QMOOD model. Each evaluation function is based on a weighted sum of quotients on the set of metrics.

The authors then conducted a case study to test how effective each function and search technique is at refactoring software. The reusability function was found to not be suitable to the requirements of SBSM due to the introduction of a large number of featureless classes. The other 2 evaluation functions were found to be suitable with the understandability function being most effective. All search techniques were found to produce quality improvements with manageable run-times, with steepest-ascent HC providing the most consistent improvements. They further expanded on this work [114] to include a fourth search technique (multiple-restart HC) and larger case studies. The functionality of the CODe-Imp tool was also expanded to include 6 additional refactorings.

They subsequently [115] used the CODe-Imp platform to conduct an empirical comparison of 3 methods of metaheuristic search in search-based refactoring; multiple/steepest-ascent HC, SA and a GA. To conduct the comparison, the mean quality change was measured for each of the 3 metaheuristic techniques. The results were then normalised and compared. They concluded that multiple-ascent HC was the most suitable method for search-based refactoring due to the speed and consistency of the results compared to the other techniques. This work was also expanded [116] with a larger set of input programs, greater number of data points in each experiment and a more detailed discussion of results and conclusions.

At a later point, Koc *et al.* [117] also compared metaheuristic search techniques using a tool called A-CMA. They compared 5 different search techniques; HC

(steepest descent, multiple steepest descent and multiple first descent), SA and ABC, as well as a random search for comparison. The results suggest that the ABC and multiple steepest descent HC algorithms are the most effective techniques of the group, with both techniques being competitive with each other.

O'Keeffe and Ó Cinnéide used steepest-ascent HC with CODe-Imp to attempt to refactor software programs to have a more similar design to other programs based on their metric values [118]. The QMOOD metrics suite was used to compare against previous results, and an overall fitness value was derived. A dissimilarity function was evaluated to measure the absolute differences between the metric values of the programs tested. CODe-Imp was then used to refactor the input programs to reduce their dissimilarity values to the target program. This was tested with 3 open source Java programs. Two of the programs were refactored to be more similar to the targets, but for the third, the dissimilarity was unchanged in both cases. The authors speculated that this was due to the limited number of refactorings available for the program as well as the low dissimilarity to begin with. They further speculated that the reason for the limited available refactorings was due to the flat hierarchical structure in the program.

Moghadam and Ó Cinnéide used CODe-Imp along with JDEvAn [119] to attempt to refactor code towards a desired design using design differencing [120]. The JDEvAn tool is used to extract the UML models of 2 solutions of code, and detect the differences between them. An updated version of the code is created by a maintenance programmer to reflect the desired design in the code and the tool uses this along with the original design to find the applicable changes needed to refactor the code. The CODe-Imp platform then uses the detected differences to implement refactorings to modify the solution towards the desired model.

Seng *et al.* [121] introduced an EA to apply possible refactorings to a program phenotype (an abstract code model), using mutation and crossover operators to provide a population of options. The output of the algorithm is a list of refactorings that the software engineer can apply to improve a set of metrics. They used class level refactorings, noting the difficulty of providing refactorings of this type that preserved behaviour. They tested their technique on the open source Java program JHotDraw, using a combination of coupling and cohesion

metrics to measure the quality gain in the class structure of the program. For the purposes of the case study, they focused on the *move method* refactoring. The algorithm successfully used the technique to improve the metrics. They also tested the ability of the algorithm to reorganise manually misplaced methods, and it was successfully able to suggest that the methods are moved back to their original position.

Harman and Tratt [6] argued how Pareto optimality can be used to improve search-based refactoring by combining different metrics in a useful way. As an alternative to combining different metrics using weights to create complex fitness functions, a Pareto front can be used to visualise the effect of each individual metric on the solution. Where the quality of one solution may have a better effect on one metric, another solution may have an increased value for another. This allows the developer to make an informed decision on which solution to use, depending on what measure of quality is more important for the project in that instant. Pareto fronts can also be used to compare different combinations of metrics against each other. An example was given with the metrics CBO (Coupling Between Objects) and SDMPC (Standard Deviation of Methods Per Class) on several open source Java applications.

2.8.4 Refactoring for Testability

Harman [122] proposed a new category of testability transformation (used to produce a version of a program more amenable to test data generation) called testability refactoring. The aim of this subcategory is to create a program that is both more suited to test data generation and improves program comprehension for the programmer, combining 2 areas of SBSE (testing and maintenance). As testability transformation uses refactorings to modify the structure of a program the same technique can be used for program maintenance, although the 2 aims may be conflicting. Here a testability refactoring refers to a process that satisfies both objectives. Harman mentioned that these 2 possibly conflicting objectives form a multi-objective scenario. He explained that the problem would be well suited to Pareto optimal search-based refactoring.

Morales *et al.* [123] investigated the use of a multi-objective approach that takes into consideration the testing effort on a system. They used their approach to minimise the occurrence of 5 well-known anti-patterns (i.e. types of design

defect), while also attempting to reduce the testing effort. 3 different multi-objective algorithms were tested and compared; NSGA-II, SPEA2 and MOCell. MOCell was found to be the metaheuristic that provided the best performance.

Ó Cinnéide *et al.* [124] used the LSCC (Low-level Similarity-based Class Cohesion) metric with the CODe-Imp platform to test whether automated refactoring with the aid of cohesion metrics can be used to improve the testability of a program. Ten volunteers with varying years of industrial experience constructed test cases for the test program before and after refactoring, and were then surveyed on certain areas of the program to discern whether it had become easier or harder to implement test cases for them after refactoring. The results were ambivalent but generally there was little difference reported in the difficulty of producing test cases in the initial and final program. The authors suggested that these unexpected results may stem from the size of the program being used. They predicted that if a larger, more appropriate application was being used, then the refactored program may produce easier test cases.

2.8.5 Testing Metric Effectiveness with Refactoring

Ghaith and Ó Cinnéide [125] investigated a set of security metrics to determine how successful they could be for improving a security sensitive application using automated refactoring. They used the CODe-Imp platform to test the metrics by using them separately at first. After determining that only 4 of the metrics were affected with the refactoring selection available, the metrics were combined to form a fitness function representing security. The function was then tested using first-ascent HC, steepest-ascent HC and SA. The results for the searches were mostly identical except that SA caused a higher improvement in 1 of the metrics. Conversely, the SA solution entailed a far larger number of refactorings than the other options. The effectiveness of these metrics was also analysed and it was discovered that of the 27% average metric improvement in the program, only 15.7% of that improvement indicated a real improvement in its security. This was determined to be due to the security metrics being poorly formed.

Ó Cinnéide *et al.* [126] conducted an investigation to measure and compare different cohesion metrics with the help of the CODe-Imp platform. It was found that the 5 metrics that aimed to measure the same property disagreed with each

other in 55% of the applied refactorings, and in 38% of the cases metrics were in direct conflict with each other. Two of the metrics were then studied in more detail to determine where the contradictions were in the code that caused the conflicts. Variations of the metrics were used to compare them in 2 different ways. This study was extended [127] to use 2 new techniques to compare the metrics and also to increase the number of metric pairs compared. Among the compared metrics, LSCC was found to be the most representative, while SCOM (Sensitive Class Cohesion) was found to be the least.

Veerappa and Harrison [128] expanded upon this work by using CODe-Imp to inspect the differences between coupling metrics. A similar approach was used to measure the effects of automated refactoring on the coupling metrics and to compare them. This experiment resulted in less divergence between metrics, with only 7.28% of changes directly conflicting. However, in 55.23% of cases the changes were dissonant, meaning that there was a larger chance that a refactoring that caused a change in one metric had no effect on another. They also measured the effect of refactoring with the RFC (Response For Class) metric on a cohesion metric and found that after a certain number of iterations, the coupling will continue to improve as cohesion degrades, minimising the effectiveness of the changes.

Simons *et al.* [129] compared metric values with professional opinions to deduce whether metrics alone are enough to helpfully refactor a program. A survey was conducted, asking experienced software engineers their opinion of the quality of a set of software examples. The metric values for the solutions were corresponded to the quality attributes specified in the survey and correlation plots were produced to measure whether there was any correlation between the engineer's opinions and the metric values. There was found to be almost no correlation between the 2, leading the authors to suggest that metrics alone are insufficient to optimise software quality as they do not fully capture the judgements of human engineers when refactoring software.

Vivanco and Pizzi [7] used search-based techniques to select the most suitable maintainability metrics from a group. They presented a parallel GA to choose between 64 different object-oriented source code metrics. Firstly, they asked an experienced software architect to rank the components of a software system in difficulty. The GA was then run for the set of metrics in sequential and parallel.

Metrics found to be more efficient included coupling metrics, understandability metrics and complexity metrics. Furthermore, the parallel program ran substantially faster than the sequential version.

Bakar *et al.* [130] attempted to outline a set of guidelines to select the best metrics for measuring maintainability in open source software. An EA was used to optimise and rank the metrics, which were listed in previous work [131]. An analysis was conducted to validate the quality model using the CK (Chidamber & Kemerer) metric suite [4] of object-oriented metrics (also known as MOOSE – Metrics for Object-Oriented Software Engineering). The CK metric values were then used in the EA as ranking criteria in selecting the best metrics to measure maintainability in the software product. The proposed approach had not yet been empirically validated, and had presented the outcome of ongoing research.

Harman *et al.* [132] wrote about the need for surrogate metrics that approximate the quality of a system to speed up the search. If non-functional properties of the system mean that there is limited time or power (e.g. if an older device is used with a less efficient CPU), then it may be more important for the fitness function to be calculated quickly or with little computational effort, in which case approximate metrics will be more useful than precise ones. The trade-off here is that the metrics will guide the search in the direction of optimality while improving the performance of the search. This ability would be useful in dynamic adaptive SBSE, where self-adaptive systems may take into account functional as well as non-functional properties. Harman *et al.* had also discussed dynamic adaptive SBSE elsewhere [133].

2.8.6 Refactoring to Correct Design Defects

Kessentini *et al.* [134] used examples of bad design to produce rules to aid in design defect detection with genetic programming (GP), and then used these rules in a GA to help propose sequences of refactorings to remove the detected defects. The rules are made up of a combination of design metrics to detect instances of 3 different design defects. Before the GA was used, a GP approach experimented with different rules to reproduce the example set of design defects, with the most accurate rules being returned. Once a set of rules were derived, they could be used to detect the number of defects in the correction approach. The GA could then be used to find sequences of refactorings to reduce the

number of design defects in a program. The approach was compared against a different rules-based approach to defects detection and was found to be more precise with the design defects found.

Further work with this approach to design smell (defect) correction was investigated in [135]–[137]. In [135], Kessentini *et al.* extended the experimental code base, with the results further supporting the approach. Ouni *et al.* [136] replaced the GA used in the code smell correction approach with a MOGA (NSGA-II). They used the previous objective function to minimise design defects as 1 of 2 separate objectives to drive the search. The second objective used a measure of the effort needed to apply the refactoring sequence, with each refactoring type given an effort value by the authors. Kessentini *et al.* [137] extended the original approach by using examples of good code design to help propose refactoring sequences for improving the structure of code. Instead of generating refactoring rules to detect design defects and then using them to generate refactoring sequences with a GA, they used a GA directly to measure the similarity between the subject code and the well-designed code. The fitness function was used to increase the similarity between the 2 sets of code, allowing the derived refactoring sequences to remove code smells.

Ouni *et al.* [138] created an approach to measure semantics preservation in a software program when searching for refactoring options to improve the structure. They used a multi-objective approach with NSGA-II to combine the previous approach for resolving design defects with the new approach to ensure that the resolutions retained semantic similarity between code elements in the program. The solutions generated with the approach were analysed manually to derive the percentage of meaningful refactorings suggested. The results were then compared against a previous mono-objective and previous multi-objective approach. While the number of defects resolved was moderately smaller, the meaningful refactorings were increased.

Ouni *et al.* [139] then explored the potential of using development refactoring history to aid in refactoring the current version of a software project. They used a multi-objective approach with NSGA-II to combine 3 separate objectives in proposing refactoring sequences to improve the product. Two of the objectives, improving design quality and semantics preservation, were taken from previous work. The third objective used a repository of previous refactorings to encourage

the use of refactorings similar to those applied to the same code fragments in the past. The approach was tested and compared against a random search and a mono-objective approach. The multi-objective algorithm had better quality values and semantics preservation than the alternatives, although this approach did not apply the proposed refactorings to the code, leaving the refactoring sequences to be applied manually by the developer. Similarly in another study [140], they used NSGA-II to test 6 open source projects, this time with 4 objectives. Along with measuring refactoring similarity and the other 2 objectives, this study also aimed to minimise the number of code changes necessary to fix the defects.

They further explored this approach using refactoring history [141] by analysing *co-change* that identified how often 2 objects in a project were refactored together at the same time and also by analysing the number of changes applied in the past to the objects. They also explored the effect of using refactoring history on semantics preservation. Further experimentation showed a slight improvement in quality values and semantics preservation with these additional considerations. Another study [37] investigated the use of past refactorings borrowed from different software projects for when the change history of the applicable project is not available or does not exist. The improvements made in these cases were as good as the improvements made when previous refactorings for the relevant project were available.

Wang *et al.* [142] combined the previous approach by Kessentini *et al.* [134] to remove software defects with time series in a multi-objective approach using NSGA-II. The time series was used to predict how many potential code smells would appear in future versions of the software with the selected solution applied. One of the objectives was then measured by minimising the number of code smells in the current version of the software and estimated code smells in future versions of the software. The other objective aimed to minimise the number of refactorings necessary to improve the software. The experimental results were compared against previous mono-objective and multi-objective approaches and were found to have better results with fewer refactorings, but also took longer to run.

Pérez *et al.* [143] presented a short position paper to propose an approach to resolving design smells in software. They proposed using version control

repositories to find and use previously effective refactorings in code and apply them to the current design as *refactoring strategies*. Refactoring strategies are defined as heuristic-based, automation-suitable specifications of complex behaviour-preserving software transformations aimed at a certain goal e.g. removing design smells. They described an approach to build a catalogue of executable refactoring strategies to handle design smells by combining refactorings that have been performed previously. The authors claimed that, on the basis of their previous work and other available tools, it would be a feasible approach.

Mkaouer *et al.* experimented with combining quality measurement with robustness [144] to yield refactored solutions that could withstand volatile software environments where importance of code smells or areas of code may change. They used NSGA-II to create a population of solutions that used robustness as well as software quality in the fitness measurement. They also used a number of multi-objective performance measurements (hypervolume, inverse generational distance and contribution) to compare against other multi-objective algorithms. To analyse the effectiveness of the approach and the trade-offs involved in ensuring robustness, the NSGA-II approach was compared against a set of other techniques. For performance, it was compared to a multi-objective particle swarm algorithm (as well as a random search to establish a baseline), and was found to outperform or have no significant difference in performance in all but 1 project. It is suggested that since this was the smaller project, the particle swarm algorithm may be more suited to smaller, more restrictive projects. It was also compared to a mono-objective GA and 2 mono-objective approaches that use a weighted combination of metrics. It was found that although the technique only outperformed the mono-objective approaches in 11% of the cases, it outperformed them on the robustness metrics in every case, showing that while it sacrificed some quality, the NSGA-II approach arrived at more robust solutions that would be more resilient in a more unstable, realistic environment. This study was extended [145] by testing 8 open source systems and 1 industrial project, and by increasing the number of code smell types analysed to 7.

They also experimented with the newly proposed evolutionary optimisation method NSGA-III [25]. They tested the algorithm using different amounts of

objectives (3, 5, 8, 10 and 15) to measure the scalability of the approach to a multi-objective and many-objective problem set These results were then compared against other EAs to see how they scaled compared to NSGA-III. The NSGA-III approach improved as the amount of objectives used was increased, whereas the other algorithms did not scale as well. The other EAs were comparable when the amount of objectives used in the search was smaller, but as the amount of objectives used was increased, the results became less competitive with NSGA-III. The search technique was also compared against 2 other techniques that used a weighted sum of metrics to measure the software. These techniques performed significantly worse than the NSGA-III approach. They extended the study [146] by also experimenting on an industrial project and increasing the number of many-objective techniques compared against from 2 to 4. The number of objectives was reduced to 8 and changed to represent the quality attributes of the QMOOD suite as well as other aggregate metric functions. They also looked at many-objective refactoring with NSGA-III for re-modularisation [42]. They compared the technique against other approaches by looking at up to 7 objectives, using objectives from previous work to look at the semantic coherence of the code and the development history along with structural objectives. Again, the approach outperformed the other techniques and more than 92% of code smells were fixed on each of the applications.

More recently, Ouni *et al.* [147] adapted the chemical reaction optimization (CRO) algorithm to the SBSM perspective and explored the benefits of this approach. They compared this search technique against more standard optimisation techniques used in SBSE; a GA, SA and PSO. They combined 4 different prioritisation measures to make up a fitness function that aimed to reduce 7 different types of code smells. The approach was compared against a previous study and a variation of the approach that didn't use prioritisation. The approach was superior using the relevant measures to the other 2 solutions compared against it. It was also shown to give better solutions in larger systems than the other optimisation algorithms tested.

2.8.7 Refactoring Tools

Fatiregun *et al.* [73] explored program transformations by experimenting with a GA and HC approach and comparing the results against each other as well as a random search as a baseline. They used the FermaT transformation tool (and 20

transformations from the tool) to optimise the length of a program by comparing the number of lines of code before and after. The average fitness for the GA was shown to be consistently better than the random search and the HC search, while the HC technique was, for the most part, significantly better than the random search.

Trifu *et al.* [148] proposed an automated design flaw correction approach that uses *correction strategies* to plan for the safe removal of detected design flaws, where a correction strategy is defined as a "structured description that maps a given flaw to a set of possible solutions". They aimed to bridge the gap between design flaw detection and refactoring to remove said flaws. For each stage of the approach, they used a tool: jGoose Echidna for problem detection; Costrat for solution analysis; and Inject/J for refactoring. The Advanced Refactoring Wizard served as an integration platform. The tool had complete support for Java (support for other languages was under development). A case study was presented with a Java program to illustrate the actions of the tool.

DiPenta [149] proposed another refactoring framework, Evolution Doctor, to handle clones and unused objects, remove circular dependencies and reorganise source code files. Afterwards, a hybridisation of HC and GAs is used to reorganise libraries. The fitness function of the algorithm was created to balance 4 factors; the number of inter-library dependencies, the number of objects linked to each application, the size of the new libraries and the feedback given by developers. The framework was applied to 3 open source applications to demonstrate its effectiveness in each of the areas of design flaw detection and removal.

Tsantalis *et al.* [150] wrote about the ability of the JDeodorant tool to resolve the type-checking design smells by replacing them with polymorphism. Type-checking smells attempt to deduce the type of class that will be used at run time and 5 different variations (using 2 approaches) of the smell were outlined in the paper. The first approach uses conditional statements to check the attribute in a class that represents the type. The second uses run time type identification to identify the derived subclass that relates to an abstract superclass at run time. The JDeodorant tool contains 2 different refactorings to replace these smells depending on which of the approaches are used. The refactorings will replace any conditional statements with a reference to an abstract method and will

make sure that each subclass has a concrete instantiation of the method, with the specific code for that class outlined in the body of the method. This way polymorphism can be used to decide which method to use at run time instead of simulating the behaviour in the code using conditional statements.

The Wrangler tool was introduced by Li and Thompson [151] to improve the modularity of programs written in Erlang by suggesting refactoring steps. The tool looks for code smells, but instead of using search-based techniques the tool inspects a module graph and a function call graph that it generates for the program. There are 4 modularity smells that the tool attempts to locate. As a case study to test the effectiveness of the tool, the authors used it on their own source code for the tool itself. They found a number of valid modularity smells in the code of each type and used the analysis to improve the structure of the code for an updated version of the tool.

Moghadam and Ó Cinnéide [152] rewrote the CODe-Imp platform to support Java 6 input and to provide a more flexible platform. It now supported 14 different design-level refactorings across 3 categories; method-level, field-level and class-level. The number of metrics had also been expanded to 24, measuring mainly aspects of cohesion or coupling. The platform was also given the option of choosing between using Pareto optimality or weighted sums to combine the metrics and derive fitness values.

Griffith *et al.* [153] introduced the TrueRefactor tool to find and remove a set of code smells from a program using a GA in order to increase comprehensibility. To detect code smells in a program, each source file is parsed and then used to create a control flow graph to represent the structure of the software. This graph can be used to detect the code smells present. For each code smell type, a set of metrics are used to deduce whether a section of the code is an instance of that code smell type. The tool contains a set of 12 refactorings (at class level, method level or field level) that are used to remove any code smells found. A set of pre conditions and post conditions are generated for each code smell to ensure that they can be resolved beforehand. The paper used an example program with code smells inserted to analyse the effectiveness of the tool. The number of code smells of each type detected over the set of iterations was measured along with the measure of a set of quality metrics. In both cases, the values improved initially before staying relatively stable throughout the process. Comparison of

initial and final code smells showed that the tool removes a proportion of them and also metric values show that the surrogate metrics are improved. The tool is only able to generate improved UML representations of the code and not refactor the source code itself, and this restriction was identified as an aim for future work.

Morales [154] aimed to compare different metaheuristic approaches and use a metaheuristic search to detect anti-patterns in source code. The tool, an Eclipse plugin would then use automated refactoring to help remove the anti-patterns and improve the design of the code. Morales *et al.* [155] addressed this aim with the ReCon approach (Refactoring approach based on task Context). The approach leverages information about a developer's task, as well as 1 of 3 metaheuristics, to suggest a set of refactorings that affect only the entities of the project in the developer's context. The metaheuristics supported are SA, a GA and variable neighborhood search (VNS). They adapted the approach to look for refactorings that can reduce 4 types of anti-pattern. They also aimed to improve 5 of the quality attributes defined in the QMOOD model. The results showed that ReCon can successfully correct more than 50% of the anti-patterns in a project using fewer resources than the traditional approaches from the literature. It can also achieve a significant quality improvement in terms of reusability, extendibility and to some extent flexibility, while effectiveness reports a negligible increment.

2.8.8 Testing Other Aspects of the Search Process

Van Belle and Ackley [156] introduced an experiment to test the adaptability of a genetic program to analyse the evolution of evolvability. The experiment aims to analyse how adaptable the program is to changes in the fitness function over time. They compared the results of a generic *monolithic* genetic program against a variant known as an *automatically defined function*. The automatically defined function was found to be adaptable with change as long as the function wasn't too volatile.

Harman *et al.* [157] proposed a new representation and crossover operator for GAs in SBSM. The representation, which is used to reduce the size of the search space to improve results, represents the solution as a set of numbered modules consisting of components (where the components are be assumed to be a set of

procedures, functions and variables). The newly defined crossover technique worked to preserve the module retention during crossover and promote the formation of good building blocks. An experiment was performed to compare this crossover operator with a standard single point crossover operator. When the appropriate target granularity (the desired number of identified modules) was used, the novel crossover technique outperformed the standard approach, but quickly became trapped in local optima. When the target granularity was deliberately set to a misleading value, the novel operator performed worse. It was suggested that the results showed the novel approach as being more sensitive to inappropriate choices of target granularity.

White *et al.* [158] used a multi-objective approach with a GA to find a trade-off between the functionality of a pseudorandom number generator and the power consumption necessary to use it. They were able to successfully generate Pareto fronts using the GA to show a set of nondominated solutions that balanced the functional objective against the non-functional objective.

Qayum and Heckel [159] used graph transformation techniques to identify dependencies between refactoring steps. They expressed the problem using ACO, where each node in a graph represents a proposed refactoring, and the edges represent the dependencies between them (such as precedence and conflicts). A graph was created to specify and map the available refactorings and their dependencies and the ACO technique was used to produce an optimal order of proposed refactorings to produce an improved structure.

Amal *et al.* [160] used an *artificial neural network* to help their approach choose between refactoring solutions. They applied a GA with a list of 11 possible refactorings to generate refactoring solutions consisting of lists of suggested refactorings to restructure the program design. They then utilised the opinion of 16 different software engineers, with programming experiences ranging from 2-15 years, to manually evaluate the refactoring solutions generated for the first few iterations by marking each refactoring as good or bad. The artificial neural network used these examples as a training set in order to develop a predictive model to evaluate the refactoring solutions for the remaining iterations. Due to this, the artificial neural network worked to replace the definition of a fitness function. The approach was tested on 6 open source programs and compared against existing mono-objective and multi-objective approaches, as well as a

manual refactoring approach. The majority of the suggested refactorings were considered by the users to be feasible, efficient in terms of improving quality of the design and to make sense. In comparison with the other mono-objective and multi-objective approaches, the refactoring suggestions gave similar scores but required less effort and less interactions with the designer to evaluate the solutions. The approach outperformed the manual refactoring approach.

2.9 Gap Analysis

From 1999 to 2010, the largest amount of SBSM papers published in a year was 4. There was a dip in the amount published in 2009 and 2010, but from 2011 there has been an increased amount of SBSM research. From 2011 to 2016, there were at least 4 papers published a year. The most prolific year for SBSM research was 2012 with 8 papers published that year. Overall, from 1999 to 2016, there has been an average of 3 papers published per year. Likewise, when inspecting all 99 of the papers, there is an increase in the amount published after 2009, and the most prolific year was 2012 with 13 papers. The average number of papers published per year was 6.

The majority of the papers were published in journals (28 papers) or featured in conferences (63 papers). Of the remaining papers, 3 were included as book sections [117], [161], [162], 3 are technical reports [8], [64], [163] and 2 were published in magazines [63], [69]. The majority of authors have only published 1 paper. Of the remaining authors, 18 have published 2 papers and 10 have published 3 papers. Only 13 of the 144 authors in the literature have published more than 3 papers. Most of the studies in the main SBSM papers were quantitative. 4 were qualitative in comparison to the 42 quantitative studies. A further 11 were discussion based papers with no experimental portions.

Of the quantitative papers, most of the studies tested different refactoring approaches, but a number of papers [6], [7], [130], [156]–[158], [160] investigated other factors. Various studies examined the setup of the search approach. A few [157], [160] investigated the fitness function or crossover technique used in a GA to choose solutions. Van Belle and Ackley [156] tested the applicability of ADFs

in handling changes in the fitness function of a genetic search. White et al. [158] used a GA to find a trade-off between the functionality of a pseudorandom number generator and the power consumption necessary to use it. Harman and Tratt [6] tested the Pareto optimal approach to combine software metrics in a search. Vivanco and Pizzi [7] used a GA to test metrics and choose the most suitable ones to use. Bakar et al. [130] also proposed a method to do this.

A number of studies were used to detect issues in the code, but not to resolve them [95]–[101]. In these cases, no fitness function was needed in the technique because once the issues were detected the algorithm would be finished. In many cases [73], [112]–[118], [120], [125]–[128], [148]–[151], [153], the studies used tools to detect issues in the code and of these tools some [148]–[151], [153] were used to find specific issues, like god classes or data classes in the program. One of these studies [120] was used to resolve the issues via refactoring, but used a different method to determine the steps needed to resolve them. Two UML models were generated; 1 to represent the current solution and 1 to represent the desired solution. This was created with the assistance of the programmer. Using these 2 models the refactorings needed to improve the program were then calculated and could be applied. In this case the technique was concerned less with code smells detected in the software and more with the desired structure of the solutions in the eyes of the programmers themselves.

This seems to isolate 3 main methods of automated maintenance from the analysed literature. There is the above method of working towards a desired structure. There is the method where problems are first detected in the code and then either refactoring options are generated in order to be applied manually [37], [42], [134]–[139], [141], [142], [144], [145], [147], [153], [155], or the problems are addressed automatically [148], [149], [151]. Finally, there is the method of using quality metrics to refactor the program stochastically and work towards a better solution [73], [112]–[117], [125]–[128] or again, using this approach to suggest refactorings to apply [6], [25], [121], [146].

The types of search technique used in the main SBSM papers of the literature were HC, SA, GAs, GP, general evolutionary algorithms (GEA), PSO, ABC, ACO, CRO and VNS. Among the algorithms, the EAs (GA, GP and GEA) were used the most, at 30 studies (with the majority of EAs being GAs). EAs became more prominent in the research after 2010, with 3 to 4 papers per year involving

them, whereas there had been 9 studies involving EAs altogether between 1999 and 2010. Of the studies containing EAs, 14 used MOEAs. This indicates a promising evolution of SBSM to generate more sophisticated solutions to the problem area. The next most common technique, HC, was used in 14 studies, with SA being used in 10. There has been a fairly consistent presence of HC and SA over the years, with the largest number of studies looking at HC or SA in a single year being 4 in 2007. In comparison, there were 4 studies involving EAs in 2014, 2015 and 2016. The SIAs (PSO, ABC and ACO) were used in only 5 studies [117], [144], [145], [147], [159]. SIAs have been more frequently investigated in recent years as well, with a paper involving 1 of them in 2012, 2014, 2015 and 2016. CRO and VNS were used in 1 study each. Each of these studies [147], [155] were recent (2015 and 2016), suggesting a possibility for CRO and VNS to be explored more in future research. Figures A.6-A.8 in Appendix A display the dispersion of algorithms used in the research.

Of the SBSM papers, 14 didn't inspect or use any search techniques. Of the ones that did, the majority were only concerned with 1, at 28 papers, although 15 other papers involved more than 1. Of these, 12 papers [73], [113]–[117], [125], [144], [145], [147], [155], [157] directly compared the different search techniques against each other to speculate on the most applicable, with the earliest paper to compare search techniques [113] being published in 2006. Four of the papers [113], [115]–[117] focused mainly on comparing search techniques. These studies compared HC with GAs, HC with SA or all 3 with each other. One [117] also involved ABC by comparing it with HC and SA. In the studies, HC seemed to outperform the other techniques. Although it had the possibility of being trapped in local optima, the technique gave consistent results and was faster than other techniques that would take time to gain traction. SA and GAs could give high quality results in certain cases but for both techniques, the results depended highly upon the configuration of the search beforehand.

Most of the programs used in the studies are open source, with 41 different programs being used across 34 studies. As the vast majority of the frameworks used dealt with Java code, the open source programs used are in Java. The remaining programs used consist of test programs developed for the studies [73], [111], [112], [124], [129], [153], [159], in-house programs [7], [117], [148] and industrial programs [42], [142], [145], [146], [151]. Five of the 6 studies to use

industrial programs [42], [142], [145], [146] used a program by the Ford Motor Company referred to as JDI-Ford. The project sizes are generally adequate for the experiments as they are large enough to justify representation of a real project. The sizes generally tend to be tens of thousands of lines of code with hundreds of classes.

A number of maintenance tools were proposed in the literature [73], [88], [102], [107], [111], [117], [148]–[153]. They used various different approaches to maintaining software and some even applied to different types of code. While most of the 12 tools were applied to Java code, a few were used with other programming languages. Bunch [88] is a tool used for software clustering, and has been used with C and C++ code. The Wrangler tool [151] is used to maintain and improve the modularity of programs written in the functional programming language Erlang. Finally, FermaT [73] is used to provide more low-level changes using wide-spectrum language (WSL) transformations. Also of note is the GenProg [104] tool which looks at bug fixing. It has been applied to C code in order to repair defects in an automated manner using GP.

A number of the proposed tools identify design defects first before attempting to resolve them. DPT [107] was proposed to apply design patterns to the code in an automated manner. It uses minitransformations built from refactorings to apply the patterns. The Advanced Refactoring Wizard [148] is itself an integration platform combining 3 other tools that, together, detect problems, analyse solutions and refactor to remove the problems. Similarly, Evolution Doctor [149] is used to diagnose issues with the software files first, before restructuring the source files to ameliorate those issues. Wrangler [151] finds instances of different issues in Erlang code and removes them via refactoring. Likewise, TrueRefactor [153] finds instances of 5 different types of code smells before finding refactorings to resolve them, and JDeodorant [150] identifies and removes 4 different types of design smell. The J/Art tool [102] can detect issues in the code but can only suggest restructuring actions for a selection of the issues. Other tools [73], [111], [117], [152] use refactorings to improve the code according to metric functions. Instead of analysing the code for issues beforehand, they refactor the code up front in order to resolve issues as they go along. Of the available tools, the CODe-Imp tool was used in a myriad of studies

[113]–[116], [118], [120], [124]–[128], [152]. A precursor to CODe-Imp, DPT, was also present in 3 different papers [107], [109], [110].

Of the papers, there have been a number that have investigated [7], [113], [118], [125]–[128], [130] or discussed [129], [132], [133] the metrics used in search-based approaches. Many of the programs analysed in the experiments and case studies conducted have been using Java (1 used C++ [7]) and likewise, many of the metrics investigated have been related to object-oriented behaviours. The most commonly used metrics were ones that measured cohesion or coupling. Numerous different metrics are available to measure these qualities and 1 study [126] compared different cohesion metrics to determine how similar they are, finding conflicting behaviours. Another study [128] did a different comparison with coupling metrics. Some studies also used metrics to represent the class structure of the program or for inheritance based observations. A study was conducted [7] to compare 64 different metrics in an attempt to determine the most effective ones for search-based optimisation. The metrics included in this study measured cohesion, coupling, size (number of methods, classes, lines of code etc.), average size, ratios, complexity, depth of inheritance, comments, code reuse, naming properties and more. The study found the cohesion metrics to be relevant, along with the mean number of lines per method and method name length metrics, suggesting that method names can affect the understanding of the code and that the size of the methods can affect the maintenance of the code. One study [113] used the QMOOD model to represent the properties flexibility, reusability and understandability with various weighted combinations of metrics to analyse which ones were most useful. Appendix A gives more detailed information about the SBSM papers analysed with a set of tables and figures.

2.10 Conclusion

Although significant work has been done to test various aspects of search-based maintenance, there are numerous areas in which ongoing research is important in order to uncover further innovations in the field. The analysis of the literature conducted in the preceding section has uncovered some of these areas. There are a number of automated refactoring tools uncovered in the literature,

though they have limitations. Few of the tools apply actual refactorings to the code itself, therefore limiting how automated they are through the need for refactoring solutions to be applied manually. Also, although the selection of tools as a whole contains a myriad of possible options for refactoring and numerous refactorings, metrics and search techniques to use, the options within many of the individual tools themselves are limited. Likewise, although more recent research has been concerned with using MOGAs to aid in refactoring, none of the presented tools are outfitted with the option to use a MOGA.

A major component of search-based maintenance and SBSE as a whole is the metrics used to measure the quality of a program. Due to the highly subjective nature of the quality of a software system, the metrics can have a huge impact on the usefulness of the metaheuristic optimisation technique, depending on how accurately they portray quality in the eyes of the user. Explicit metrics are needed to guide the optimisation of a solution, but one developer's view of quality may be different to another's.

Most previous research has been applied to object-oriented programs and as such most fitness functions aim to improve object-oriented behaviours like cohesion or flexibility. Even defining these aspects has proven to be difficult. Experimentation has been carried out to combine different software metrics together to create more useful measures of quality, typically using either weighted sums or Pareto fronts. There has also been some research into the applicability of certain metrics. There is an opportunity for research into using different combinations to improve the software in different ways, similar to how a human assisted tool can guide the improvement of the software design to a suitable solution for the user.

Of the different search techniques used to address software maintenance, a large proportion of the analysed literature used EAs. Among these studies, a lot of recent work has looked at multi-objective approaches. This indicates a promising evolution of SBSM to generate more sophisticated solutions to the problem area. The methods address the issue by allowing multiple aspects to be taken into consideration. Further inspection of these techniques is required to discover the potential of their use and derive ways to make the approach more practical for use in a software development environment. In the following experimental chapters, research is conducted and detailed to address the gaps

uncovered from the analysis and to answer the research questions outlined in Chapter 1. The gaps to be addressed are listed below:

1. Limited options for research and experimentation with automated refactoring tools.
2. Insufficient investigation and experimentation with different metrics to measure software quality.
3. Insufficient investigation and experimentation with different fitness functions to measure certain behaviours and properties of the software.
4. Insufficient investigation and experimentation with multi-objective search techniques.

Chapter 3

Approach & Tool Support

3.1 Introduction

In order to answer the research questions, repeated below, controlled experiments have been designed.

RQ1: What current refactoring and search-based software engineering tools are available?

RQ2: Can a fully automated, practical refactoring tool be developed using techniques from previous literature to improve the maintenance of software?

RQ3: How useful is a multi-objective search-based software maintenance approach in comparison with a mono-objective search-based approach?

RQ4: Can individual, novel objectives be measured and refactored in a software program to maintain the code while also improving the individual properties inspected?

RQ5: Can numerous individual objectives be combined into a fully automated, many-objective approach in order to improve a software program across multiple different properties in an additive fashion, without losing the improvement effect of any individual property?

A controlled experiment is defined as "an investigation of a testable hypothesis where 1 or more independent variables are manipulated to measure their effect on 1 or more dependent variables" [164]. Controlled experiments are helpful as they can outline and isolate properties to measure and compare, a task that is

essential to answer **RQ**s 3-5. Although **RQ**s 1 and 2 investigate the availability of automated refactoring tools for SBSM research and whether a fully automated refactoring tool can be developed, **RQ**s 3-5 require experimentation. To address **RQ3**, an experiment is set up comparing 2 variations of a GA using mono-objective and multi-objective techniques to execute. Comparing the multi-objective approach with the mono-objective approach gives us a measure of success for the multi-objective approach. **RQ4** can be addressed with the 3 different objectives being created, corresponding to new ways to measure the software. In order to examine their effectiveness, experiments are constructed using an automated refactoring approach. They are each paired with an objective that provides a measure of software quality and compared against a mono-objective approach that only measures the quality.

To test **RQ5**, a many-objective setup is created using the 3 novel objectives along with the quality objective. Like before, they are compared with alternate setups that use a smaller selection of the objectives together. The different permutations are compared using the objective scores and the success of the many-objective approach is considered using the measurements and comparisons. In controlling the construction of the approaches being compared they are similarly set up with respect to variables and environmental factors that aren't being measured.

In order to experiment with different techniques using SBSE a tool is needed that can run mono-objective, multi-objective and many-objective refactoring tasks and make different sets of measurements that can be used to compare the approaches. To this end, a refactoring tool was constructed (named MultiRefactor) that combines the approaches of other known automated refactoring tools in order to overcome their individual weaknesses. The tool has similarities to the CODe-Imp tool in terms of the underlying framework used and the refactoring approach adopted, but MultiRefactor also contains a MOGA with which to apply multiple different metric configurations for accumulated fitness calculations. This tool is also open source and can be run as an executable in order for there to be minimal confusion for users. The approach of Ouni *et al.* uses multi-objective algorithms to refactor Java programs, but their approach isn't fully automated. They only produce a list of possible refactorings to apply to the programs, which then need to be applied manually. Furthermore,

those refactorings aren't checked for semantic conformity, resulting in analysis being necessary to measure the number of refactorings that can actually be applied. Not only is MultiRefactor fully automated, with the GAs giving a population of output solutions with fully compilable code, but the refactorings applied will preserve the semantics of the program.

In contrast to Ouni *et al.* as well as some other tools, MultiRefactor doesn't use its metric configurations to isolate and remove design flaws. It uses the alternative approach of applying the changes and analysing the effect they will have on the program, allowing for more novel solutions. The tool contains a wide selection of metrics, searches and refactorings making it useful for research purposes on top of its suitability for practical use. Although the tool doesn't contain all of the refactorings used across the various other tools, it does contain more refactorings in a single tool than any of the known alternatives. Also, MultiRefactor gives usable source code as an output of the process along with information on the refactoring process, whereas various other tools produce less useful artefacts. One of the more promising tools proposed in the literature, A-CMA [117], was experimented with to decide whether it could be used for the research. Unfortunately, the tool only uses bytecode as an input, and doesn't produce any program output. It also doesn't contain any multi-objective or many-objective capabilities. Nonetheless, as a prelude to this thesis experimentation was conducted using A-CMA [165], producing useful results and providing a learning aid for the work ahead.

This chapter is structured as follows. Section 3.2 details the preliminary experimentation conducted using the A-CMA tool to construct a measure for technical debt in the tool. Section 3.3 discusses the construction and capabilities of the MultiRefactor tool. Section 3.4 details and discusses the relevant search techniques available in the tool. It defines the implementation choices made as well as configuration settings implemented when incorporating the searches into the tool. Section 3.5 provides an overview and description of the refactorings available in the tool as well as the choices made when implementing the more ambiguous refactorings. Section 3.6 also provides an overview and description of the metrics available, and outlines the metric suites used to adapt a number of the metrics.

3.2 Preliminary Work

As preliminary work for this thesis technical debt was chosen as an interesting way to combine metrics into an objective function and use this as a means to investigate automated refactoring with the aim of increasing quality. Technical debt as described earlier, is "a situation in which long-term code quality is traded for short-term gain" [68]. It accumulates interest and becomes more expensive to repay with time. Over time it becomes harder to add functionality due to structural issues becoming more critical and the occurrence of defects becomes more likely. To improve the long term efficiency of a project and to lower its operational risk, the technical debt can be kept to a minimum by making regular repayments, i.e. refactorings. The negative side of this is that time spent on refactoring will in turn decrease the amount of time used to add functionality to software. Therefore, any approach that makes this easier or that can automate it is likely to be financially beneficial.

There has been little research done to investigate technical debt specifically. A review of the impact of technical debt on software systems as well as methods to handle it and the cost from different perspectives is given in an article by Allman [69]. The properties of technical debt have also been discussed elsewhere [68], where a particular connection has been noted between technical debt and maintenance activities. Developers at Google have given their experience of attempts to pay off technical debt in the form of build debt [71]. They use various attempts to uncover and remove the debt in Google code, which consists of millions of lines of code, much of which is monolithic. No previous work is known to attempt to create a metric function to tackle technical debt. As preparation for the main work in this thesis, an experiment to investigate the effectiveness of using technical debt to direct automatic refactoring was designed. The aim is to know whether technical debt can be used effectively as a fitness function for search-based automatic refactoring.

To consider this, a technical debt measure is established and compared against measures based on levels of abstraction, coupling and inheritance, all of which are well established as design quality factors [5]. These properties have been chosen to represent individual quality indicators as they can represent a range

of different aspects of software measurement. Inheritance will be a good indication of whether the design is badly organised or whether the classes are related and extended properly. Inheritance is concerned with measuring how the objects in a project are organised hierarchically, so class level metrics are used as a measure. The metrics used incorporate interface implementation and use of abstract classes, and hence a high measure is considered desirable. Coupling can be used to derive the extent of which the objects in a software system depend on each other, generally preferred to be as low as possible. Abstraction will indicate the number of changes needed between specific objects in order to implement new additions to the system. Again, a high value here is considered better. As previous work in the area has investigated abstraction [89], [111] coupling [128], [166] and inheritance [115], there is support for the position that these are useful properties to use for a comparative study against an approach for tackling technical debt.

An experiment has been conducted using the refactoring tool A-CMA [117] to assess the effectiveness of 3 sets of metrics that measure these object-oriented properties and compare them against a proposed set of metrics to measure technical debt. A weighted sum is used to combine the metrics into an overall score to improve. A further question investigated is whether a SA search can perform well compared to HC and a random search in a search-based automated refactoring approach to address technical debt. The same measures can be used i.e. technical debt reduction, abstraction gain, coupling reduction, inheritance gain but also execution time.

3.2.1 A-CMA Tool

A-CMA is an automated refactoring tool developed by Koc et al. [117] that refactors Java programs using Java bytecode as input. An advantage of this tool over many others is that it has many options for refactoring as well as metrics available. Additionally, A-CMA is highly configurable. The tool allows the user to construct different metric combinations that can be used on a task. An overall metric score is derived using a weighted sum of each enabled metric. A weighted sum allows some metrics to be given more influence than others. The metrics can be specified as *maximised* or *minimised*. Maximised metrics are metrics where an increase in value causes an improvement and minimised metrics are metrics where a decrease in value causes an improvement. The overall quality

gain of a task in A-CMA can be derived by finding out how much the overall score has reduced. Before the experiment was conducted, some changes were made to the existing tool for the purposes of the study[4].

The tool has the ability to run 5 different searches with 10 different variations, but for the purposes of the study only 3 are used. Initially a random search is run to provide a benchmark against which the other searches can be compared. The 2 heuristic searches, HC and SA, were chosen as they are used commonly in the research and therefore can be compared against other work in the area (e.g. O'Keeffe and M. Ó Cinnéide [114]), and because they are relatively easy to implement and modify for the purposes of the experiment. The A-CMA tool contains 20 available refactoring options to apply on the field, method and class level of a Java program. The available refactorings are listed and described in Table 3.1. Many of these refactorings implement refactoring options proposed by Fowler in his book [167] and on his website [168]. There are 24 metrics available in the A-CMA tool but in the experiment only 17 are used. The metrics used along with descriptions for each one are given in Table 3.2.

Table 3.1 - Refactorings Available in the A-CMA Tool

Field Level	**Method Level**	**Class Level**
Increase Field Security	Increase Method Security	Introduce Factory
Decrease Field Security	Decrease Method Security	Make Class Abstract
Move Down Field	Move Down Method	Make Class Final
Move Up Field	Move Up Method	Make Class Non-Final
Remove Field	Move Method	Remove Class
	Instantiate Method	Remove Interface
	Freeze Method	
	Remove Method	
	Inline Method	

[4] The original tool can be found at https://github.com/eknkc/a-cma and the updated version at https://github.com/mmohan01/a-cma

Table 3.2 – Software Metrics Used in Experiment

Metric Identifier	Description
numField	Number of fields per class
numOps	Number of methods per class
numCls	Number of classes in a package
numInterf	Number of interfaces in a package
iFImpl	Number of interfaces implemented by a class
abstractness	Ratio of abstract class to classes in a package
avrgField Visibility	Average amount of field visibility per class (where field visibility is represented by Private:0, Package:1, Protected:2, Public:3)
nesting	Nesting level per class
NOC	Number of children per class
numDesc	Number of descendants per class
numAnc	Number of ancestors per class
iC_Attr	Number of attributes in a class using another class or interface as type
eC_Attr	Number of external uses of a class as attribute type
iC_Par	Number of parameters in class methods using another class or interface as type
eC_Par	Number of external uses of class as parameter type in method
Dep_In	Number of elements that depend on a class
Dep_Out	Number of elements depended on by a class

3.2.2 Experimental Design

The experiment aims to compare 4 different fitness functions that each uses a combination of available metrics to represent some measurable property of software design. In order to compare these fitness functions, each function is given weights for each metric that must add to 1. This way the functions will be normalised for comparison against each other. The direction of improvement of each software metric must be taken into consideration (whether a metric is maximised or minimised). Of the 17 metrics used, 10 have been determined to be minimised metrics and the other 7 have been determined to be maximised metrics. The positive/negative aspect of the metrics did not need to be taken into consideration when aggregating the weights to 1. Using the A-CMA tool, the goal is to minimise the value of the metric function being inspected in order to improve the property being represented.

Three fitness functions are created from the metrics to represent important quality properties of object-oriented programs (abstraction, coupling and inheritance), and a fourth is created to represent technical debt in the system. The technical debt score is based on the SOLID principles of object-oriented design [3], as well as the QMOOD metrics suite [5]. All available refactoring

actions are enabled for the 4 fitness functions to give the maximum potential for change. Table 3.3 gives details about each fitness function compared along with weights used and whether the metrics are maximised or minimised (denoted by '+' and '-' respectively).

Table 3.3 – Metric Details for Each Fitness Function

Software Property	**Metric Components And Weights**
Technical Debt	-0.1*numFields - 0.1*avrgFieldVisibility - 0.1*numOps - 0.06*nesting + 0.1*abstractness + 0.1*numCls + 0.1*numInterf + 0.1*iFImpl + 0.06*NOC + 0.06*numDesc - 0.06*Dep_In - 0.06*Dep_Out
Coupling	-0.125*iC_Attr - 0.125*eC_Attr - 0.125*iC_Par - 0.125*eC_Par - 0.25*Dep_In - 0.25*Dep_Out
Inheritance	0.25*iFImpl + 0.25*NOC + 0.25*numDesc + 0.25*numAnc
Abstraction	0.33*abstractness + 0.33*numInterf + 0.33*iFImpl

Of the available software metrics, the most applicable are chosen to represent components of the 3 software properties. Metrics were already grouped together as coupling and inheritance metrics in the A-CMA tool, so these are the metrics used to represent the coupling and inheritance properties. The abstraction property is made up of the 3 metrics determined to be related to abstraction due to them measuring properties of the interfaces present in the software. In most cases, the weights are kept level between the metrics used in each fitness function. For the coupling function, the Dep_In and Dep_Out metrics are given priority over the others as they contain aspects of the other coupling metrics used as part of their calculations.

For the technical debt function, the 12 metrics intuitively considered to be most relevant are chosen. Initially the metrics are prioritised into 4 different groups. In order to normalise the weights and allow the metrics to accumulate to 1, these are reduced to 2 different weights; 0.06 to represent the bottom 2 categories and 0.1 to represent the top 2. The nesting, NOC and numDesc metrics are given less priority due to their more descriptive nature compared to the other metrics. In a software system, more nesting, more descendants and less classes in a package may not particularly be a bad thing, whereas less classes overall may result in classes with too many responsibilities. The Dep_In/Dep_Out metrics are deemed less important as, while dependencies

should be minimised between classes, they may be required in certain cases. In all cases metrics and weights chosen are speculative and based on intuition. In some cases directions of improvement also had to be chosen.

Each fitness function is compared using 3 different searches. The random search is used as a benchmark with 5,000 iterations. Steepest-ascent HC is chosen for the experiment with 30 restarts at a depth of 5 neighbours (chosen based on published comparisons between different HC parameters [117]). The third search used is low temperature SA (as low temperatures have been found to be more effective by O'Keeffe and Ó Cinnéide [114]) with 5,000 iterations and with the starting temperature set to 1.5. Each search is conducted 10 times using the 4 fitness functions with average values calculated. The input programs for the experiment consist of 6 open source Java projects. These programs were chosen as they have all been used in previous SBSM studies and so there is an increased ability to compare the results and also because they promote different software structures. Details about the programs are given in Table 3.4. The total number of runs of the experiment comes to 10*3(searches)*4(functions)* 6(benchmarks), giving an overall amount of 720 runs.

Table 3.4 – Java Programs Used in Experiment

Name	LOC	Classes	Initial Refactorings Available
JSON 1.1	2,196	12	167
Mango	3,470	78	598
Apache XML-RPC 3.1.1	6,532	100	712
Beaver 0.9.8	7,851	81	801
JFlex 1.4.1	15,094	56	1,094
JHotDraw 5.3	27,824	241	3,297

3.2.3 Results

The time taken to complete the tasks for each program is given in Table 3.5. Clearly here the JHotDraw program caused a bottleneck in execution time and this is most likely due to its size compared to the other projects (containing more than double the number of classes than the other projects). For instance, JHotDraw contains 27,824 lines of code compared against 15,094 for JFlex, the program with the next longest execution time. It is reasonable to assume that as

the project size increases, the search space for the refactoring process will increase also giving a large upswing in time taken even with the metaheuristic searches used. This can lead to an increase in time of order n^2. These large execution times for certain tasks suggest that a more efficient method is needed to refactor larger programs.

Table 3.5 – Java Program Execution Times

Name	Time Taken
JSON	0h 3m 13s
JFlex	2h 6m 38s
Apache XML-RPC	1h 23m 43s
Mango	1h 1m 29s
Beaver	1h 25m 4s
JHotDraw	49h 28m 4s

Figure 3.1 shows the average quality gain across the 6 programs for each fitness function using each of the 3 searches. For the abstraction and inheritance functions where the scores are more difficult to see, data labels have also been given to display the scores (unless they are 0). For all of the following figures in the chapter, data labels are also given where the scores are not 0 but are difficult to see. The results show that SA gives the highest relative quality improvement, but they also show that the random search outperforms HC. The technical debt quality gain values for each pair of searches were compared using a two-tailed Wilcoxon rank-sum test (for unpaired data sets) with a 95% confidence level (α = 5%). The SA results were analysed to be statistically different when compared against the random search and the HC search across every technical debt result. The random search results were also found to be significantly different to the HC search. The random search understandably has a larger range of values but the better outcome it gives implies that the HC search was inefficient for the set of tasks. Perhaps the input parameters were not optimal for that search. The SA and HC searches failed to create any quality gain using the inheritance function whereas the random search yielded a small increase in quality. It is assumed this is due to the freedom and volatility of the random search to find different solutions, but not necessarily to find optimal solutions.

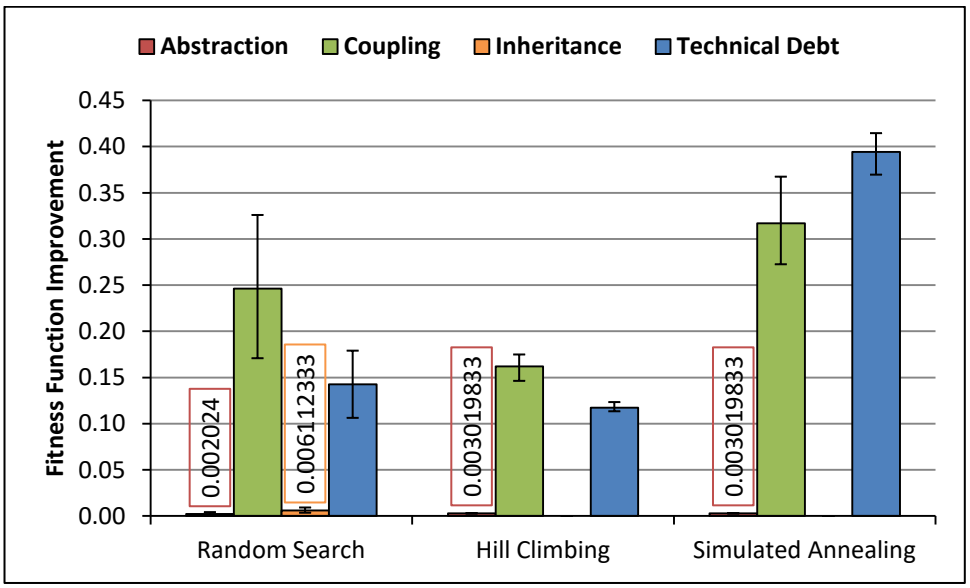

Figure 3.1 – Overall Mean Quality Gain for Each Fitness Function per Search Type

Figure 3.2 inspects the SA results, showing the average quality gain for each of the fitness functions across each of the 6 benchmark programs (this is the initial overall metric score minus the final score, averaged over the 10 runs). Of the 3 individual property fitness functions, coupling seems to be the only one that had shown any significant improvement. The abstraction tasks show minimal improvement and the inheritance tasks had no change at all. In fact, the only case where the inheritance function had any change was in the random search as shown in Figure 3.1. The technical debt function was more effective in showing an improvement. The initial and final metric scores for the technical debt function were assessed using a two-tailed Wilcoxon signed-rank test (for paired data sets) with a 95% confidence level (α = 5%). The obtained results were statistically significant when comparing every run of the technical debt function, meaning that the quality gains for the technical debt function were significant. The lack of improvement in the abstraction and inheritance functions implies that there is a lack of volatility in the metrics used to compose these functions.

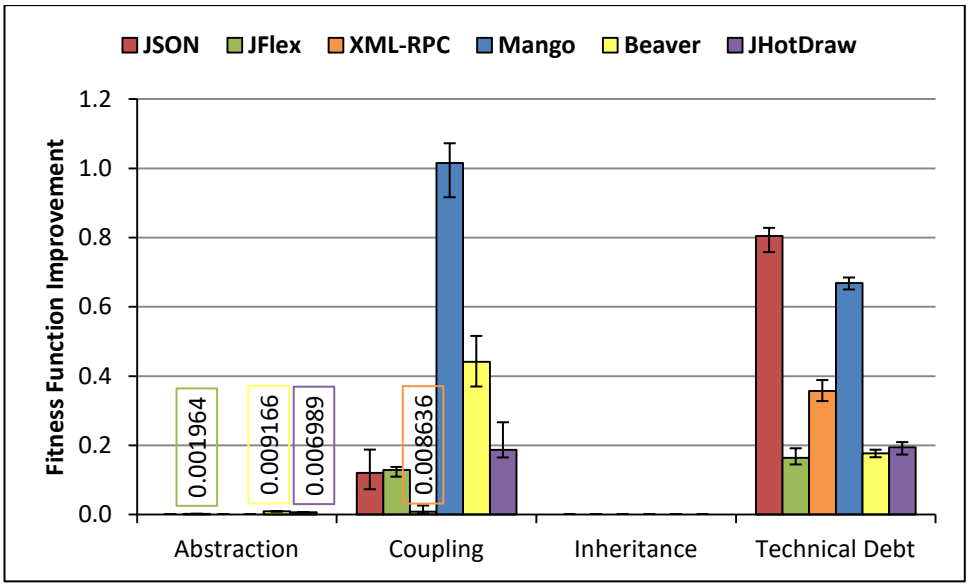

Figure 3.2 – Mean Quality Gain of Each Fitness Function Using Simulated Annealing

Figure 3.3 shows the average number of applied actions for each of the SA tasks. These results show a similar trend to the quality gain results and the abstraction and inheritance tasks are similarly devoid of applied refactoring actions. This implies that the reason for the poor quality gain results for those functions stems from the lack of available actions whereas the other metrics are more volatile and have more refactoring actions available to improve them. Figure 3.4 gives the overall average applied actions for each fitness function. This continues to show a relationship between the number of actions available for each fitness function and the quality gain values for the functions shown in Figure 3.2. It seems that the volatility of the metrics that make up each function is important to allowing the program to be refactored in any way. The harder the metrics are to improve, the less chance the program will be refactored.

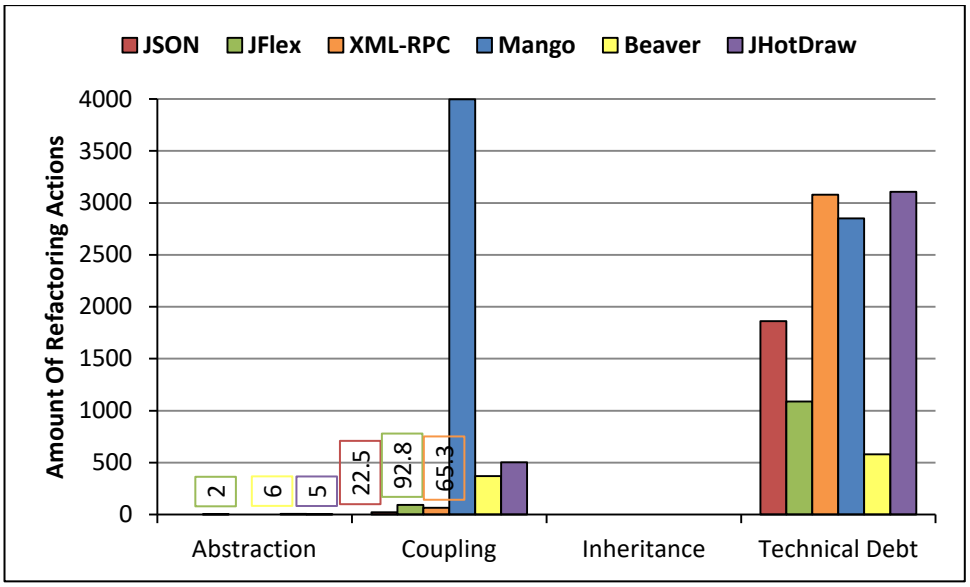

Figure 3.3 – Mean Number of Actions Applied to Each Fitness Function Using Simulated Annealing

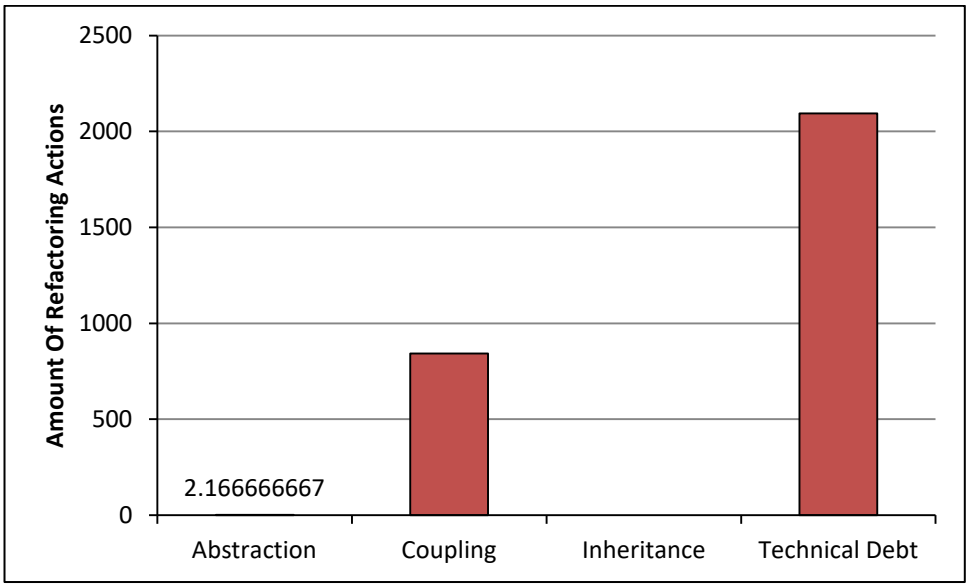

Figure 3.4 – Overall Mean Applied Actions Using Simulated Annealing

Figure 3.5 gives another view of the quality gain results, this time highlighting the results for each individual program and allowing a better comparison of the coupling and technical debt values. Most of the results favour the technical debt function over the others, but in 2 cases, Mango and Beaver, the coupling function shows higher quality gains than the technical debt function by a notable amount. This could suggest that for these 2 programs coupling was high

and so amenable to improvement therefore contributing less to the technical debt calculation. The 2 programs that show the most noteworthy improvement of the technical debt function over the coupling function are JSON and Apache XML-RPC. JSON is the smallest program used so perhaps the minimal amount of classes makes it harder to reduce the coupling between them as there is minimal coupling in the first place. Likewise, Apache XML-RPC contains almost no improvement in coupling implying it too contains little coupling between the classes. The largest quality gain among all the programs was in Mango. Figure 3.6 gives the overall average quality gain for each fitness function. It confirms that the technical debt function had a more significant improvement among the programs than the other 3 fitness functions that represented specific properties. Figure 3.3 also shows that the technical debt function involved more refactorings than the other 3 functions.

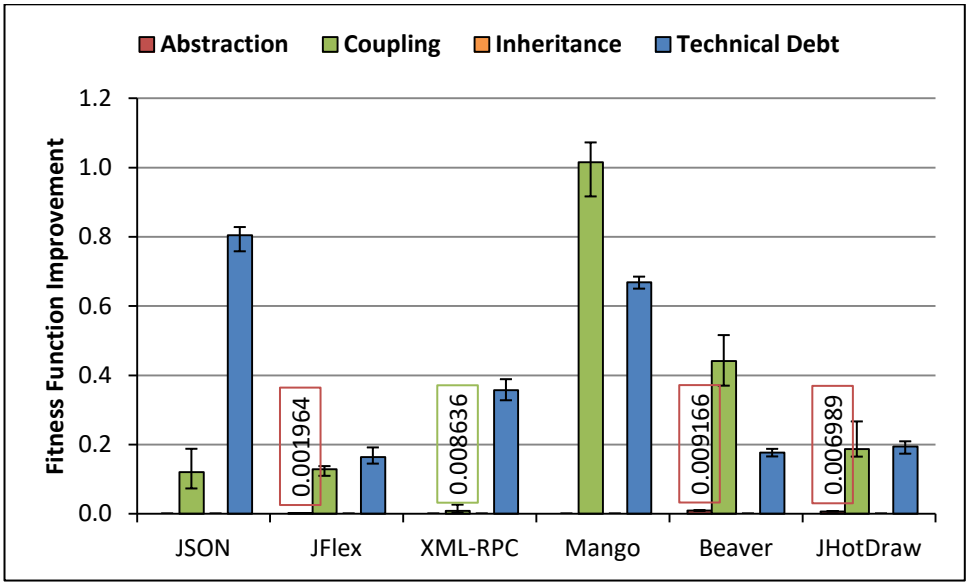

Figure 3.5 – Mean Quality Gain of Each Program Using Simulated Annealing

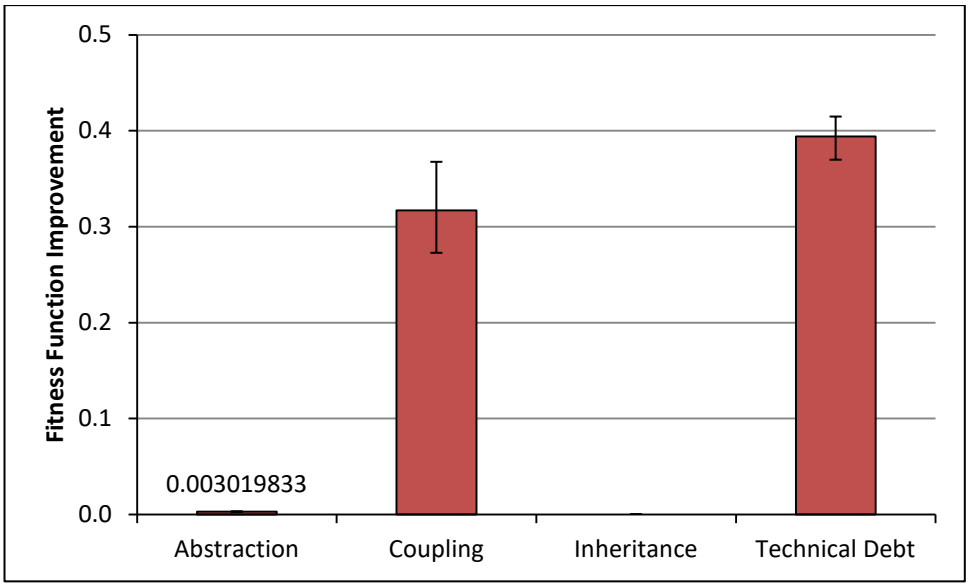

Figure 3.6 – Overall Mean Quality Gain for Each Fitness Function Using Simulated Annealing

Figures 3.7 and 3.8 show the average quality gain values for each individual metric in the technical debt and coupling fitness functions (across all 6 benchmarks), giving an idea of the volatility of each metric and their influence on the overall metric scores. In the technical debt function, only 5 of the 12 metrics show notable quality gain values with the most influential being the numOps metric. The numInterf, NOC and numDesc metrics had no quality gain, and the numCls metric decreased in quality. Amongst the other metrics in the technical debt function, Dep_In gave a decrease in quality for the JFlex program and avrgFieldVisibility gave a decrease for the Mango program. The quality gain values for the coupling functions were smaller in comparison to the technical debt function, although they were more consistent across the metrics. While this function contained only 6 metrics, 4 out of the 6 contained notable improvements (a larger proportion compared to the technical debt fitness function).

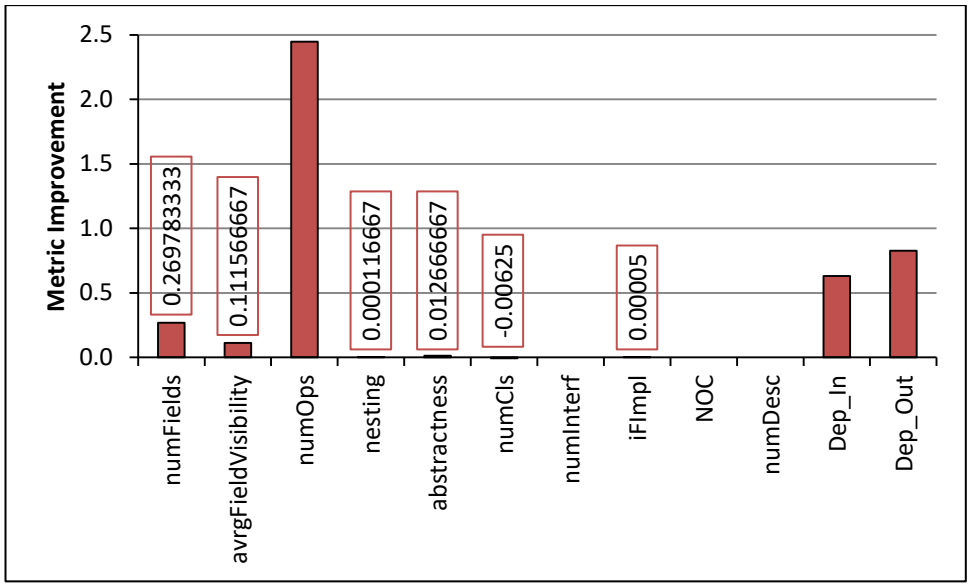

Figure 3.7 – Mean Quality Gain for Each Metric of the Technical Debt Function Using Simulated Annealing

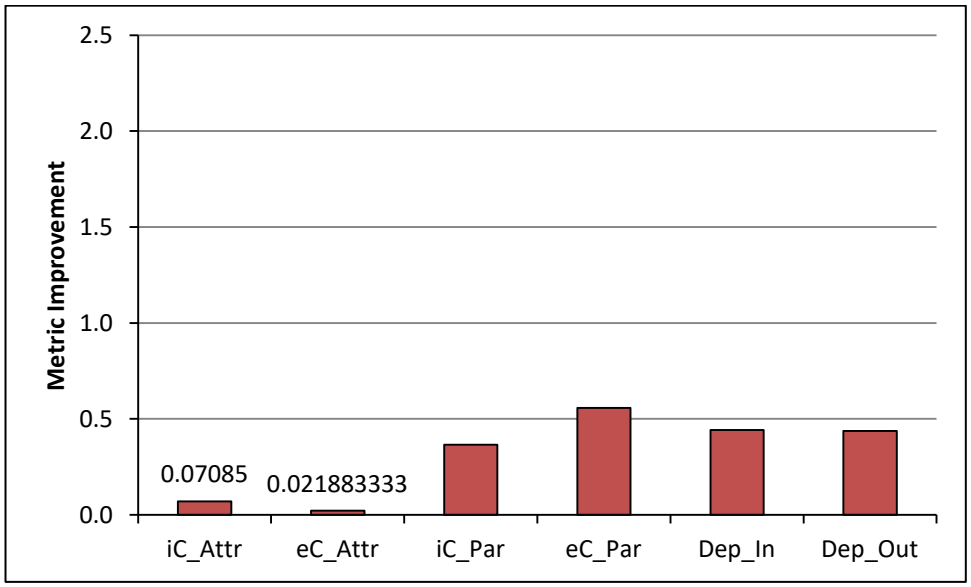

Figure 3.8 – Mean Quality Gain for Each Metric of the Coupling Function Using Simulated Annealing

The Dep_In and Dep_Out metrics were amongst the most improved (which was to be expected due to them containing aspects of the other coupling metrics used), although the parameter metrics (iC_Par and eC_Par) were also influential. The eC_Par metric showed the largest overall quality gain of the

coupling metrics, improving more than even the Dep_In and Dep_Out metrics. Of the 6 metrics, the attribute metrics (iC_Attr and eC_Attr) were affected the least, although none of the metrics showed an average decrease in quality (where the average represents the mean across 10 runs of each task) across any of the benchmarks as some technical debt metrics did. The inheritance function showed no improvement with any of the metrics used across any of the benchmark programs. The abstraction function, while only using 3 metrics, showed quality improvements with just 1 of those metrics. The abstractness metric showed a small increase in quality whereas the iFImpl and numInterf metrics showed no change across any of the programs tested. The iFImpl metric similarly showed no change when used in the inheritance function and was the smallest of the improved metrics in the technical debt function. The numIterf metric showed no change in the technical debt function either. The changes shown by the individual metrics may provide a good basis to influence how the weights should be distributed among the fitness functions. The values shown in Figures 3.7 and 3.8 are not affected by metric weights (this is only applied when the metrics are combined to derive the overall metric score). The results from the experiment [165] can be found on GitHub along with the updated version of the A-CMA tool used in the experiment.

3.3 The MultiRefactor Tool

The MultiRefactor[5] tool has been developed in support of the thesis, in Java. MultiRefactor integrates 25 different refactorings, 23 metrics and 6 different search techniques and is a fully automated framework for improving the quality of Java programs. The design of the tool, in common with the approaches of Moghadam and Ó Cinnéide [152] and Trifu *et al.* [148] uses the RECODER framework[6] to modify source code in Java programs. RECODER was detailed in a paper by Dirk Heuzeroth and Uwe Aßmann [169] in 2005, and has been supported up until 2014, with the last release (version 0.86) being in June 2008.

[5] https://github.com/mmohan01/MultiRefactor
[6] http://sourceforge.net/projects/recoder

RECODER extracts a model of the code by creating an abstract syntax tree to represent it. This model can then be used to analyse and modify the code before the changes are applied and printed out to an output folder. Each of the search techniques use these models to modify and update the code, and use the available refactorings and metrics to assist with their execution. The searches include metaheuristic searches as well as a genetic search. The tool also contains a multi-objective genetic search adapted from NSGA-II [30], and a many-objective genetic search adapted from NSGA-III [49]. The algorithm applies the approach of Ouni *et al.* [138] in using MOGAs for automated software maintenance, but instead of using the search to minimise design flaws, it applies Moghadam and Ó Cinnéide's approach to improve the code quality.

The tool has been uploaded to GitHub including any code bases used and results generated. It can be used either as an executable Java program or from the command line with the .bat file included in its repository. The tool can run with numerous different configurations and search settings and they are currently specified using the *Tasks* file and subsequent inheriting classes in the program. If desired, multiple searches can be applied consecutively with this file. The tool uses Java source code as an input and the code needs to be fully compilable. The RECODER framework used to read the java code has not been outfitted to parse the features introduced with Java 8 (such as Lambda expressions and default methods), so Java 8 code is unsupported. If a new project is to be specified for automated refactoring, the source code needs to be supplied along with any necessary library files (as .jar files). As long as the directory of the folder containing these files is supplied in the Tasks file, the program will be able to read the files and generate a model of the input for modification. The refactored program will then be output to the specified output folder and can then be run as source code. Text files will also be output containing details of the search and specifying the refactorings that have been applied to the program along with the final metric values associated with that search. The input folder of the project to be refactored can either be specified directly in the Tasks file, passed into the command line or be included in a file that can then be read from the command line. Figure 3.9 gives a brief overview of the process used by MultiRefactor to generate refactored Java code.

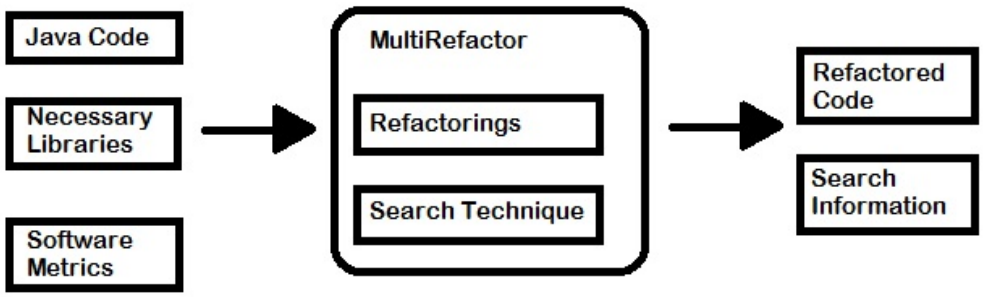

Figure 3.9 – Overview of the MultiRefactor Process

Configurations can be set up with the tool. As the tool is built to be extendable, different configurations can be created and specified for different searches, and in the case of the MOGA, 1 or more configurations can be passed into the search. A configuration is made of the refactorings and metrics chosen from the available sets, as well as extra information for the metrics to specify the weighting of a metric and whether the metric makes a negative or positive change to the program. There are numerous different ways to create a configuration in order to allow for various different methods of reading in the metric details. The information can be passed in directly in the program as the list of available refactorings is, but it can also be read in from a file. Both .txt files and .xml files are readable. The online repository contains examples of both with the desired formatting. The files contain a list of the desired metrics and values to represent the desired weighting of the metrics (with 1 equating to no weighting) and with true or false values to represent the direction of improvement of the metric (where true equates to a metric that improves with an increase of value and false equates to a metrics that improves with a decrease in value). With the tools capability of setting up different configurations, different fitness functions can be used in the tool to create different results in the program, allowing for experimentation with different objectives. The program also has the ability to compare different program states using a Pareto approach [6] instead of by combining the individual metric measurements into an overall value (with the Pareto approach the metric weights become irrelevant), although this functionality hasn't been used for any of the currently implemented search techniques.

A third possible method available for combining metrics uses a normalisation function to minimise any greater influence any individual metric may have. 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 or objective. A negative change in the metric will be reflected by a decrease in the overall function/objective value. In the case that an increase in the metric denotes a negative change, the overall value will still decrease, ensuring that a larger value represents a better metric value regardless of the direction of improvement. 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. Equation 3.1 defines the normalisation function, where m represents the selected metric, C_m is the current metric value and I_m is the initial metric value. W_m is the applied weighting for the metric (where 1 represents no weighting) and D is a binary constant (-1 or 1) that represents the direction of improvement of the metric. n represents the number of metrics used in the function.

$$\sum_{m=1}^{n} D.W_m\left(\frac{C_m}{I_m} - 1\right) \tag{3.1}$$

The results output the program produces for each search contains information about the search type and its settings, initial and final metric values for the configuration(s) used and the list of refactorings applied in order of application for that program. The command line on the program will also output when it is run including the overall improvement in the metrics and the time taken to run the search (if multiple searches are run, it will also give the overall time taken). The metric output gives the individual values for each metric in the configuration (without weights being applied) as well as the overall score (if the multi/many-objective GA is begin used, it will give the value for each objective). The refactoring outputs give the refactoring name, the name of the relevant element(s) the refactoring is applied to as well as any relevant classes in the program and, if applicable, the change made to the element as a result of the refactoring. For the GAs, an output file is created for each solution in the final population.

3.4 Available Search Techniques

Table 3.6 gives the search techniques available in MultiRefactor for assistance in choosing the right refactorings to apply to improve a program. For each search type there are a selection of configurable properties to signify how the search will run (and in some cases, other properties that also effect the execution of the program but are not included as configurable parameters). The searches that have been adapted and implemented in the tool have been chosen because of familiarity through previous related research as well as previous experimentation and validation of the searches by other researchers in the area of SBSM. Furthermore, the HC and SA searches have been tested and compared in the preceding experimentation and the GA is tested against the MOGA in Chapter 4, to further influence the searches chosen in the subsequent experimentation. Below there is a description of how each of the GAs are implemented in the MultiRefactor tool, focusing on the specific choices made with these search adaptations and the influences they may have on the behaviour of the search.

Table 3.6 – Available Searches in the MultiRefactor Tool

Search	Configurable Properties
Random	2
Hill Climbing	4
Simulated Annealing	4
Genetic Algorithm	5
Multi-Objective Genetic Algorithm	5
Many-Objective Genetic Algorithm	5

3.4.1 Genetic Algorithm

In this tool, the GA is designed to be similar to the implementation used by O'Keeffe and Ó Cinnéide [115] (which itself was adapted from the adaptation used be Seng *et al.* [121]) so that it can be applied to the problem of automated software refactoring for program maintenance. The algorithm mimics the natural process of genetic replication by merging solutions in a population to

create new offspring. The adaptation considers a model representing a set of refactorings applied to a code base to represent an individual genome in a population. The GA contains a number of stages, with numerous decisions being made as to how to represent and execute these stages.

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 *RefactoringSequence* object will store all the information necessary to reconstruct a model from scratch with the initial program input being used as a starting point. This way, a single model can be used in the program, sacrificing some measure of speed (through the necessary reconstruction of solutions in the mutation and printing stages of the search), but saving memory. 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 a later point in the search. In order to accommodate this, the *resetModel* method will revert the model back to the initial state read in from the input program by repeating the process (and resetting the refactoring objects so that they don't save a copy of the previous model instance). The fitness value of each genome in the population will be calculated before the model is reset, to eliminate the need to reconstruct the models during the crossover process.

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, 2 parent genomes are passed in and 2 children are produced from their refactoring sequences. In order to choose the parent genomes from the population, a selection operator is necessary, and the method used is rank-based selection. Rank-based 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. As the method only uses the ranks to select the genomes, it only needs to know the population size. However, in order to pick the relevant solutions, the population will need to be ordered by fitness

with the better solutions at the beginning of the list in order for the chosen genomes to be relevant to the rank proportions. A balanced set of proportions are produced for the different ranks, meaning that the increase in likeliness with selecting a rank is linearly calculated in proportion to the number of ranks. Equation 3.2 details the calculation [170] used with rank-based selection to generate the proportion for each rank, with a higher proportion representing a higher likelihood of being chosen (lR is the last rank, cR is the rank being calculated and sp is a selective pressure constant that can be in the range $1 < sp \leq 2$, with 2 being used as a default. In the search the ranks range from 0 to the population size minus 1). Once the rank proportions are calculated for each genome a random value is chosen within the range of the rank proportions and the respective rank is chosen. This will be repeated to find the second parent, making sure to choose a different rank.

$$(2 - sp) + 2(sp - 1)\left(\frac{lR - cR}{lR}\right) \tag{3.2}$$

Once the selection operator is executed and the parent genomes are chosen for crossover, the offspring can be generated. Like before, the process used to generate child solutions is based off O'Keeffe and Ó Cinnéide's approach. A cut and splice method is used to combine the parents to generate different solutions. A single, separate point is chosen for each parent in order to facilitate the technique. The point is chosen at random along the refactoring sequence in each of the parent solutions, with at least 1 refactoring present on each side. For each child, the 2 sets of refactorings are then mixed together. The first set of refactorings in 1 parent will be applied first and then the second set of refactorings from the other parent will be applied. For the second set, each refactoring needs to be checked to ensure that it is applicable in the child sequence, due to the differing model states. If a refactoring is not applicable, it will simply be left out of the sequence and the next refactoring will be checked. When the refactoring sequences of the offspring are generated, they may contain differing numbers of refactorings depending on where the cut points were chosen and if any of the refactorings were found to be inapplicable. The fitness's for each child sequence will be calculated immediately after they are constructed, in order to rank the population at the end of that generation of the search. For each generation of the search, the crossover operator will be applied once and

then applied again a random number of times depending on the crossover probability input to create a set of newly generated solutions.

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. A solution can be chosen more than once. 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. This is the same approach O'Keeffe/Ó Cinnéide and Seng *et al.* use. This mutated solution will replace the pre mutated version in the set of new solutions from that generation. If a random refactoring cannot be found for the solution, the original solution will be returned. Ouni [171] also developed a GA adaptation in order to implement a multi-objective refactoring approach [141]. The selection and crossover operators used for the original GA are similar to the operators used in MultiRefactor, although the mutation operator is different. This approach chooses 1 or more of the refactorings at random points in a refactoring sequence and replaces them with different refactorings. The MOGA adapted in MultiRefactor will employ a separate selection operator, but the crossover and mutation operators will stay the same.

Once the mutation process is complete for a generation, the new offspring is added to the current population and the solutions are ordered according to fitness. As the fitness values 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. When the search is complete, the final population of solutions will be generated and stored in the output folder. When the printing process is performed to apply the model

refactorings to the code, the model will have to be reconstructed to represent each genome in the population before it is printed, using the information in the stored refactoring sequences.

The numerous available configuration parameters will determine the behaviour of the search. The print all parameter will determine whether the program should store the whole population of solutions resulting from the process, or only the most fit solution. If only the top solution is desired, then the model will only need to be constructed for this genome and printed. Also, the data output associated with the fittest solution will be the only one needed. The number of generations for the search to run for as well as the desired size of the population will be specified as search parameters. The population size parameter will determine how many solutions are generated during the initialisation process and also how many of the genomes will survive to the next generation after combining the population with the new offspring.

The crossover probability and mutation probability parameters determine the likeliness of these processes being executed during the search. Each generation, the crossover process will be executed once. After this, a random value between 0 and 1 will be generated. If this is less than the crossover probability, the process will execute again. This will continue until the value generated is greater. This allows the probability value to determine whether the process will be executed more times or less in the search. The mutation method uses the same decision, except it is not guaranteed to execute at all. These parameters must be between 0 and 1 and the higher they are, the more likely the process will be run. Another important value that is not included as an input parameter is the initial refactoring range. This value will determine the initial number of refactorings applied to the solutions during the initialisation process. For each solution, a random number of refactorings between 1 and the preset refactoring range will be chosen to apply. If the solution runs out of available refactorings before the desired number, the initialisation for that solution will finish at that point. The initial refactoring range determines the size of the genomes in the initial population and will also influence the sizes of the offspring generated.

3.4.2 Multi-Objective Algorithm

The multi-objective algorithm is built using the GA and mainly differs in how the fitness of the solutions is processed. As with the multi-objective adaptation setup by Ouni *et al.* [141], the algorithm is an adaptation of the multi-objective NSGA-II [30]. Due to the minimal number of modifications needed on the GA, it is possible to upgrade the adaptation to represent a different multi-objective genetic implementation at a future point. In contrast to the initial GA, the multi-objective algorithm can use 1 or more different fitness objectives as inputs to measure the solutions of a population. The configuration parameters used with the multi-objective algorithm are identical to those used with the original GA (as well as the initial refactoring number).

The fitness process uses the nondominated sorting algorithm to sort the population into different ranks. Within the ranks, the genomes are given crowding distance values to help the selection operator choose between possible parent solutions and to further sort when choosing the solutions of a rank to leave out of the population. When the fitness method is run and the population is sorted there may only be a selection of genomes from a rank needed in the population before the population size is reached. In this case the crowding distance values are used to decide which solutions to keep and which to cull from the population in that rank. The population is sorted after initialisation and then after the crossover and mutation operators are applied each generation and the new offspring is added, the population is sorted again. During the fast nondominated sort, the efficiency upgrade suggested by Liu and Zeng [172] is implemented to improve performance. The improved algorithm is able to avoid generating unnecessary nondominated fronts, reducing the run time complexity of the algorithm from $O(N^2)$ to $O(kN + NlogN)$ (with k representing the number of nondominated fronts) in bi-objective optimisation problems.

To reflect the difference in fitness calculation, a different selection operator is used to the rank-based selection method used in the GA. Binary tournament selection allows the better of 2 solutions to be picked based on their rank and crowding distance values. The solution with the better rank is returned. If the ranks are equal, the crowding distance values of the solutions will be compared instead. If these values are equal as well, one will be selected at random. Each time crossover is executed, 2 separate solutions are chosen from the population

at random and the tournament selection method is used to select the best one. This is done twice to find the 2 parent solutions for the crossover operation.

In order to choose a solution to use from the final population, certain decisions are made in the algorithm. First, the solutions are limited to the ones that are in the top rank. Of these, a process is performed that is partially inspired by the NSGA-III paper by Deb and Jain [49]. In it, an ideal solution is composed, containing the best values across all objectives for the solutions in a population. In order to make a choice between the top ranked solutions in the final population, a similar technique is used. Like in the paper, the ideal point of the set of solutions is composed, that contains the best objective values, and the translation vector between the solutions and the ideal point is found (by finding the distance of each solution vector from the ideal point). Then, for each solution, the worst of the objective distances is used to represent that solution in order to indicate the maximum distance from the ideal point. Using these values, a solution can be chosen by selecting the one with the minimum worst objective distance. The refactoring tool stores the top ranked solutions in a separate sub folder and indicates the ideal solution from that rank that has been determined using the above method. The results file for the applicable solution will also add a note that says "This solution has the closest maximum distance from the ideal point in the top rank of solutions".

3.4.3 Many-Objective Algorithm

The many-objective algorithm adapts the NSGA-III upgrade of NSGA-II. As with the multi-objective NSGA-II adaptation, the many-objective algorithm is built off 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 [49], in lieu of reference points being supplied preferentially by the user, the adaptation applied the systematic approach used by Das and Dennis [56]. The number of reference points is decided by finding the closest number greater than or equal to the specified population size with the relevant number of objectives. This allows the number to be roughly equal to the number of solutions observed each generation (where the slight remainder will represent the other solutions from the remaining rank, making the overall number greater than the specified population size). Once the number of axis divisions is found, the reference points can be predefined initially and used throughout the search.

As mentioned by Deb and Jain, the NSGA-III process performs a careful elitist selection of solutions in an attempt to maintain diversity. For this reason, and also because the number of reference points is almost equal to the number of solutions, each population member is given equal importance, therefore no selection operator is used in order to perform crossover. The parent solutions are chosen at random and the only check is to ensure that they are distinct from one another. Also mentioned by Deb and Jain is how, in a many-objective approach, genetic offspring that are closer to their parent solutions are more desirable. In order to address this Mkaouer *et al.* [25], in their approach, restricted the cutting point of the crossover process to belong to either the first tier or the last tier of the refactoring sequence. Similarly, this adaptation restricts the cutting point of the crossover process to either be at the first refactoring or the last refactoring of the sequence. Likewise, the cutting point for both solutions is kept the same (both at the first refactoring or both at the last refactoring), in order to preserve the sequence size of the parent solutions in their offspring. The many-objective adaptation does not include functionality to generate multiple layers of reference points [49] or to add and delete reference points to provide better niche values [57].

3.5 Available Refactorings

The refactorings used in the tool are generally based on the list by Fowler [167], and consist of field-level, method-level and class-level refactorings. Table 3.7 lists these refactorings. In the tool, 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 another method, *mayRefactor*, to deduce whether a program element can be refactored. It will make all the relevant semantic checks and return true or false to reflect whether the element is applicable. The checks applied in this method will depend on the refactoring, and are important in order to exclude elements that are not applicable for that refactoring. The search algorithm will use the method to find the number of applicable elements in the file and will choose a number within that size to pick the refactoring element. The semantic checks that have been incorporated into the mayRefactor method for each refactoring are numerous. They have been tested with the inputs used in the experimentation over the course of the research, to ensure that the refactorings that are applied are valid and to bypass any potential issues reading the abstract syntax trees of the modified code with the underlying RECODER framework. Any subsequent testing beyond this has been deemed out of scope in relation to the research project.

Table 3.7 – Available Refactorings in the 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	Collapse Hierarchy
Make Field Non Static	Make Method Non Static	Remove Class
Move Field Down	Move Method Down	Remove Interface
Move Field Up	Move Method Up	
Remove Field	Remove Method	

The *analyze* method is used to apply the refactoring itself. It takes as an input an integer to represent the file being inspected and another to represent which element among the applicable ones will be used. The program will iterate through the elements of the applicable type in the file, using the mayRefactor method to exclude inapplicable elements from the selection, until it finds the relevant element. The RECODER framework allows the tool to apply the changes to the element in the model. This may consist of a single change or, as in the case of the more complex refactorings, may include a number of individual changes to the model. Specific changes applied with the RECODER framework consist of either adding an element to a parent element, removing an element from a parent element, or replacing one element with another in the model. The refactoring itself is constructed using these specific model changes. In some cases new elements will be created for use in the refactoring (for instance, new imports may need to be created when moving an element to a new class), and where possible, these will be constructed from existing elements to minimise the potential for issues. There is also an *analyzeReverse* method for each refactoring. This allows the program to undo the changes made in the last instance of that refactoring. This method is used with the HC and SA searches to check neighbouring refactorings from the current state and measure their impact on the program.

For some refactorings, choices have to be made in relation to how specifically the refactoring is applied. The *Move Field Down* and *Move Method Down* refactorings involve moving program elements down to a subclass. Here, the subclass to be used needs to be chosen before the refactoring is applied. In these refactorings, the choices are made during the mayRefactor checks. This allows the program to find a permutation of the refactoring that is applicable if, for instance, only 1 subclass from a set will allow the refactoring to be applied. The alternative is to choose one permutation and only check for it, meaning applicable refactorings may be returned as inapplicable depending on which settings are chosen whenever it is checked. With this approach, if there are no permutations of the refactoring with which it will be applicable, false will be returned, but if there is, one of those permutations will be used. As the specific choice needs to be known in order to check whether it is applicable, this will be handled in the mayRefactor method for these refactorings and the choice will be saved in that class for the analyze method to use later. This will require more

processing in the mayRefactor method for these refactorings, although it will save processing in the analyze method when the refactoring itself is applied. This will also allow for more available refactorings to be found during the search. Descriptions of the available refactorings are given below.

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 1 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 added 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. They are based off the Fowler refactorings. *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. It is based off the Fowler refactoring. The *Remove* refactorings will remove the element related to that type of refactoring.

3.6 Available Metrics

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 can be measured as separate objectives to be more precise in measuring their effect on a program. A number of the metrics available in the tool are adapted from the list of metrics in the QMOOD [5] and CK/MOOSE [4] metrics suites. Table 3.8 lists the available metrics in the tool, and descriptions are given below.

Table 3.8 – 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		

Class Design Size counts the number of classes in a project (ordinary classes and interfaces). *Number Of Hierarchies* counts the number of distinct class hierarchies in a project (excluding interfaces and also excluding hierarchies made up of single classes). *Average Number Of Ancestors* measures the average count of each class declaration (not including interfaces) away from its root class (where the root class is restricted to the class at the highest level within the project). *Data Access Metric* measures the average ratio of non public fields to public fields per class (only including global fields in a class). If a class has no fields, the value isn't calculated for that class (as it would be 0) but the class is still included in the average calculation. *Direct Class Coupling* finds the distinct number of other classes that each class depends on then calculates the average

number per class. The classes counted are ones used in field declarations and included as method parameters. Of these, primitive types are not included.

Cohesion Among Methods finds the average cohesion among methods per class. The metric is calculated for each relevant type (ordinary classes and interfaces) by getting the accumulation of the number of distinct parameter types for each method over the maximum possible number of distinct parameter types across all the methods. The maximum number of distinct parameters across all the methods of a class is calculated by multiplying the number of methods by the number of distinct parameter types in all of the methods of the class. The average is then found by calculating the accumulative cohesion among methods for all the classes over the class number. If there are no methods or dependant classes being counted in a class, the value is not calculated for that class but the class is still included in the average calculation. A value closer to 1 relates to better cohesion. *Aggregation* finds the average number of user defined attributes (global fields) per class, where a user defined attribute is of a type defined within the project.

Functional Abstraction finds the average functional abstraction per class. In each class (ordinary classes and interfaces), the metric measures the number of methods in that class and the number of methods that can be inherited in the class from its superclasses. Functional abstraction is the number of inherited methods over the number of methods in the class. If a class has no methods, the value isn't calculated for that class but the class is still included in the average calculation. *Number Of Polymorphic Methods* finds the average number of methods per class that are redefined elsewhere in the project (i.e. have inherited methods). *Class Interface Size* finds the average number of public methods per class. *Number Of Methods* counts the average number of methods per class.

Weighted Methods Per Class measures the average method complexity per class. To do this, it counts the number of lines of code each method in a class contains and accumulates them to get the value for each class. *Number Of Children* measures the average number of immediate subclasses per class (excluding interfaces). *Abstractness* measures the ratio of interfaces in a project over the overall number of class declarations. *Abstract Ratio* gives the average ratio of abstract methods (as well as the class itself if it is abstract) per class. If there are no abstract elements in a class, the value isn't calculated for that class but

the class is still included in the average calculation. *Static Ratio* and *Final Ratio* give the average ratios of static and final elements per class (*Static Ratio* looks at classes and methods, whereas *Final Ratio* looks at classes, methods and local or global field declarations), and *Constant Ratio* calculates the average ratio of elements (classes, methods and global fields) that are both static and final per class. *Inner Class Ratio* calculates the ratio of the number of inner classes (ordinary classes or interfaces) over the number of classes in a project.

Referenced Methods Ratio finds the average ratio of inherited methods referenced per class. In each class (ordinary classes and interfaces), the metric measures the number of distinct external methods (methods defined outside the current class) referenced among the methods of the class. For each class, the ratio of the number of these methods that are inherited by the class over the number referenced is calculated. *Visibility Ratio* measures how secure all the elements of a project are by calculating an average visibility ratio per class. In a class, each method and global field declaration (as well as the class itself) is given a visibility value, where a private member has a value of 0 and a public member has a value of 1 (and other visibilities have values in between). The visibility ratio for that class will calculate the accumulated visibility values over the number of elements. The smaller this ratio, the more secure the elements of the project are. Finally, *Lines Of Code* calculates the overall number of lines of code in a project and *Number Of Files* counts the number of Java files in a project.

Chapter 4

Quality Objective

4.1 Introduction

In order to assess the capabilities of the MultiRefactor approach, a set of experiments have been set up to compare different procedures available within the tool. Experiments by others have been conducted comparing the other metaheuristic searches [113], [115], [117], so the experimentation focuses on the use of the GAs in the tool and aims to find out 4 things:

1. The first 2 aims of the experimentation focus on the configuration settings for the GAs available. The first part of the experimentation tests numerous different permutations of the crossover and mutation probabilities in a mono-objective GA.
2. Using these results to derive desirable values for the probabilities, the second part of the experimentation looks at the other GA settings available. Different generation numbers are used along with different population sizes and refactoring ranges to analyse how successful the different permutations of these settings can be with a baseline setup.
3. Once the preferred GA configuration settings are established, the third aim is to test the available software metrics within the tool and discover which are more successful. Some metrics may be more useful than others in measuring the changes made by the available refactorings. These will be more helpful when trying to analyse the changes made to a solution

and as such, a metric function made from these metrics may assist in creating a more prosperous solution.

4. The final aim is to compare the mono-objective approach with the multi-objective search available and see whether using a multi-objective algorithm to automate maintenance of a software solution is as practical as using a mono-objective algorithm. A multi-objective algorithm involves more processing and as such may take more time to complete. On the other hand, a multi-objective algorithm can be used to improve multiple objectives concurrently, and within the same solution.

The experimentation in this chapter aims to test whether, in a fully automated solution, a multi-objective algorithm using similar settings can yield comparable results across all the objectives used, and whether it is worth the time taken to do so. The following research questions have been formed:

RQ4.1: Which set of software metrics are most volatile when used with a mono-objective genetic algorithm to refactor software?

RQ4.2: Does a multi-objective refactoring approach give comparable results on all objectives to corresponding mono-objective refactoring runs?

To answer **RQ4.1**, each available metric is used individually as the objective within a mono-objective GA. Once the ideal settings for the GA are decided, each metric is run separately with a number of open source Java programs. Then an average improvement value is formed for each metric.

For **RQ4.2**, 2 factors are investigated. The first factor compared is the set of objective improvement values yielded by the 2 approaches. Three separate objectives are constructed, influenced by the results of the previous experiment in the chapter. For each, a mono-objective approach is used with the same parameters as before, and the top objective improvement values are acquired. Then, using the same setup parameters, a multi-objective approach is run using all 3 objectives. Hence, the top objective improvement value is derived among the solutions for each objective, and these are compared against the mono-objective approaches. The second factor compared is the time taken to run the different approaches. The overall times taken to run the 3 mono-objective solutions are compared against the overall time taken to run the multi-objective solution in which all 3 objectives can be taken into account and solutions can be used for

each one. A separate set of hypotheses and alternative hypotheses have been derived for each factor of this experiment:

H4.1: The overall objective improvements in the multi-objective search are not significantly worse than the overall objective improvements in the mono-objective search.

H4.1A: The overall objective improvements in the multi-objective search are significantly worse than the overall objective improvements in the mono-objective search.

H4.2: The overall time taken to run the multi-objective search is not significantly higher than the time taken to run any of the 3 mono-objective searches.

H4.2A: The overall time taken to run the multi-objective search is significantly higher than time taken to run any of the 3 mono-objective searches.

The remainder of this chapter is organised as follows. Section 4.2 explains the details of the experiments conducted. Section 4.3 discusses the results of the experiments, by looking at the metric improvement values and the times taken to run the tasks. Section 4.4 inspects the threats to validity of the experiments and Section 4.5 discusses the outcome of the experiments and addresses the research questions.

4.2 Experimental Design

Five open source programs are used in the experimentation. The programs range in size from relatively small to medium sized, as shown in Table 4.1. These programs were chosen as they have all been used in previous SBSM studies and so comparison of results is possible (and also because they promote different software structures and sizes). JSON (JavaScript Object Notation) is a lightweight data interchange format. Mango is a Java library, loosely inspired by the C++ standard template library. Beaver is a parser generator. Apache XML-RPC is a Java implementation of XML-RPC that uses XML to implement remote

procedure calls. Finally, JHotDraw is a 2-dimensional graphics framework for structured drawing editors. The source code and necessary libraries for all of the programs are available to download in the GitHub repository for the MultiRefactor tool.

Table 4.1 – Java Programs Used in Experimentation

Name	LOC	Classes
JSON 1.1	2,196	12
Mango	3,470	78
Beaver 0.9.11	6,493	70
Apache XML-RPC 2.0	11,616	79
JHotDraw 5.3	27,824	241

The experimentation is split into 4 parts. In order to choose configuration parameters for the searches used, trial and error is used to derive the most effective settings. The first experiment compares different combinations of crossover probabilities and mutation probabilities to test their effect on the search algorithm. The probability values are compared using a baseline metric and input. The largest input, JHotDraw, is used with the *Visibility Ratio* metric. This metric is assumed to be volatile since it is directly related to the increase/decrease visibility refactorings. Nine different tasks are used to compare crossover and mutation probabilities of 0.3, 0.5 and 0.8. Each task is run 5 times to get an average value.

The second experiment compares the other configuration parameters available in the GA to find the best trade-off between software improvement and time taken. The same setup is used for the GA with the JHotDraw input and the *Visibility Ratio* metric, and the ideal crossover and mutation values are used. There are 27 different tasks set up to compare different combinations of generation numbers, refactoring ranges and population sizes. The generation numbers tested are 50, 100 and 200. The refactoring ranges used are also 50, 100 and 200 and the population sizes used are 10, 50 and 100. For the first experiment used to compare crossover and mutation values, these configurations are set to their lowest value i.e. 50 generations, refactoring range of 50 and population size of 10.

In the third experiment, each metric is run as an individual fitness function with the GA using the configuration parameters derived from the previous 2 experiments. The metrics are run with each of the input programs 5 times. Average values are calculated for each metric with each input program, and then the average value is found across the 5 inputs programs, giving an overall average improvement value for each metric. The average values for each of the metric are then compared to find the most volatile (i.e. the most sensitive) metrics with the available refactorings in the tool. The final experiment compares the more effective metrics in a mono-objective setup against a multi-objective approach. A set of metric functions are constructed using the results from the previous experiment by excluding the metrics that have the least effect. The relevant metrics are split into 3 functions in order to be used as separate objectives in a MOGA. To compare the multi-objective approach with a mono-objective analogue, the 3 objectives are used as separate metric functions in different runs of the mono-objective algorithm. Each objective with the mono-objective search is run 6 times for each of the 5 inputs, giving 30 runs of the search. Likewise, the MOGA with the 3 objectives is run 6 times for each input. Therefore, across all 4 different search approaches, there are 120 tasks run. The results with the mono-objective metric function are compared with the multi-objective approach to derive insight into the practicality of a multi-objective setup.

For each objective, the GA algorithm is run using the configuration parameters chosen from experiments 1 and 2 for each input, and the average metric improvement is calculated for the top solution across the different inputs. The MOGA uses the 3 metric functions as separate objectives in a single approach. The study aims to find out whether each separate objective is comparable. Therefore, the top solutions for each individual objective are found and the average improvements are calculated across the different inputs. In order to aid in finding the top scores for each objective in the final population of the multi-objective tasks, the multi-objective search has been modified in this experiment to update the relevant results files to state that they contain the highest score for the corresponding objective. This tweak circumvents the need to manually check the scores in each solution to find the largest score for each objective.

The metric changes are calculated using the normalisation function described in Chapter 3, as defined in Equation 3.1. For the experiments used in this chapter, no weighting is applied to any of the metrics. The directions of improvement used for each metric is defined in Table 4.4, where a plus indicates a metric that will improve quality on increasing and a minus indicates a metric that will improve quality on decreasing. The hardware used for the experimentation is detailed in Table 4.2.

Table 4.2 – 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

4.3 Results

Figure 4.1 gives the metric improvement values for the *Visibility Ratio* metric in a GA using different variations of the crossover probability and mutation probability settings. In the figure, "MP" represents the mutation probability value used. For each parameter, 3 variations were tested along the range from 0 to 1, resulting in 9 different permutations of the search altogether. It seems from the results that lower crossover values result in greater improvements, although there was a greater range of results apparent in most of the other permutations of the search. The most improved configuration tested has a crossover value of 0.2 and a mutation value of 0.8. Conversely, the least improved configuration was with a crossover value of 0.5 and a mutation value of 0.2. The results seem to imply that the crossover setting has a larger effect on the quality of the solutions derived from the search.

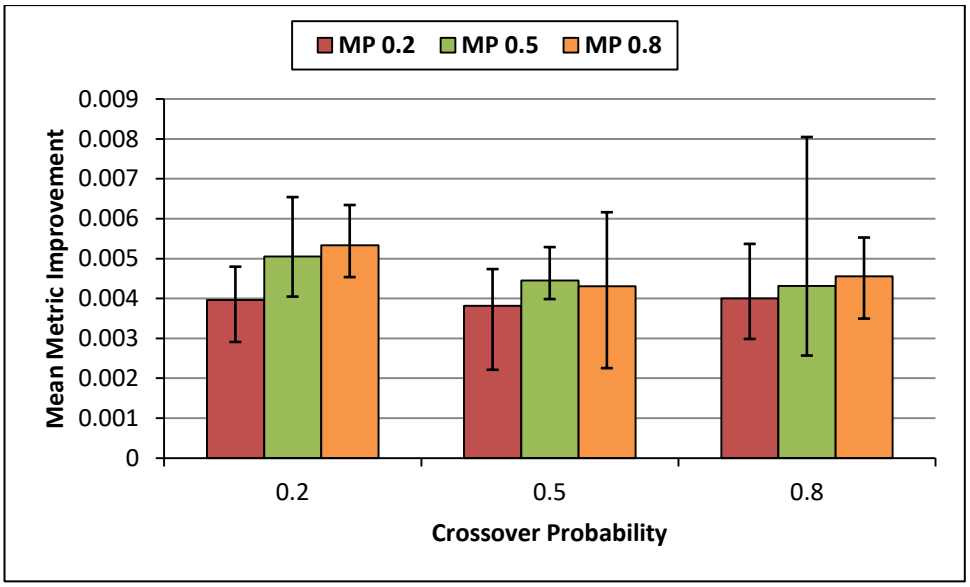

Figure 4.1 – Mean Metric Improvement Values with Different Crossover and Mutation Probabilities.

Figure 4.2 displays the different times taken to run each permutation of the GA, in minutes. As with Figure 4.1, "MP" represents the mutation probability value used. The times ranged from 2 minutes and 22 seconds for crossover and mutation values of 0.2, to 9 minutes and 32 seconds for a crossover value of 0.8 and a mutation value of 0.8. Larger crossover probabilities seem to affect the execution time of the search in a negative way, an expected result given that extra crossovers in the search means extra necessary processing. Although this is also true for the mutation process, it is not as intensive as crossover as it only relates the application of a single refactoring, whereas crossover demands inspection of numerous refactorings in a solution. Alas, while the larger mutation values do result in increased execution times for 2 of the crossover settings, for the crossover value of 0.8, the time actually decreases between the mutation values of 0.2 and 0.5. For the ideal settings of 0.2 for crossover and 0.8 for mutation, the execution time is relatively small at 4 minutes and 23 seconds.

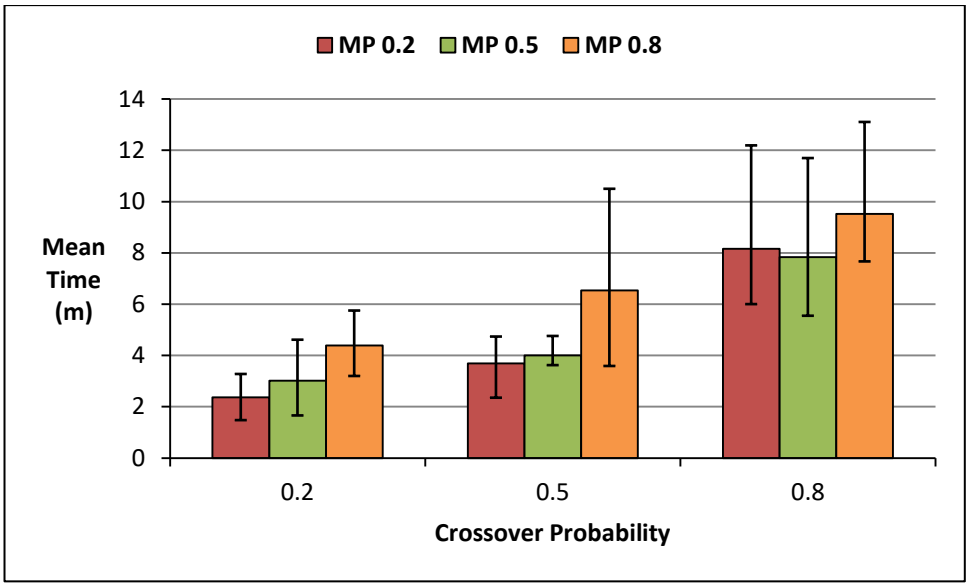

Figure 4.2 – Mean Execution Times for Different Crossover and Mutation Probabilities.

Figure 4.3 shows the metric improvement values for each permutation of the generation, refactoring range and population size GA settings and Figure 4.4 compares them against the time taken to run them. Each setting was tested with 3 different values leading to 27 different permutations overall. As shown in the scatter plot, 1 configuration stands out as having a larger increase in quality without having a similar increase in necessary time. This configuration (using 100 generations, refactoring range of 50 and population size of 50) has an execution time of 12 minutes and 29 seconds in comparison to the 1 other configuration with a better improvement value (200 generations, refactoring range of 200 and population size of 200) that took 43 minutes and 59 seconds.

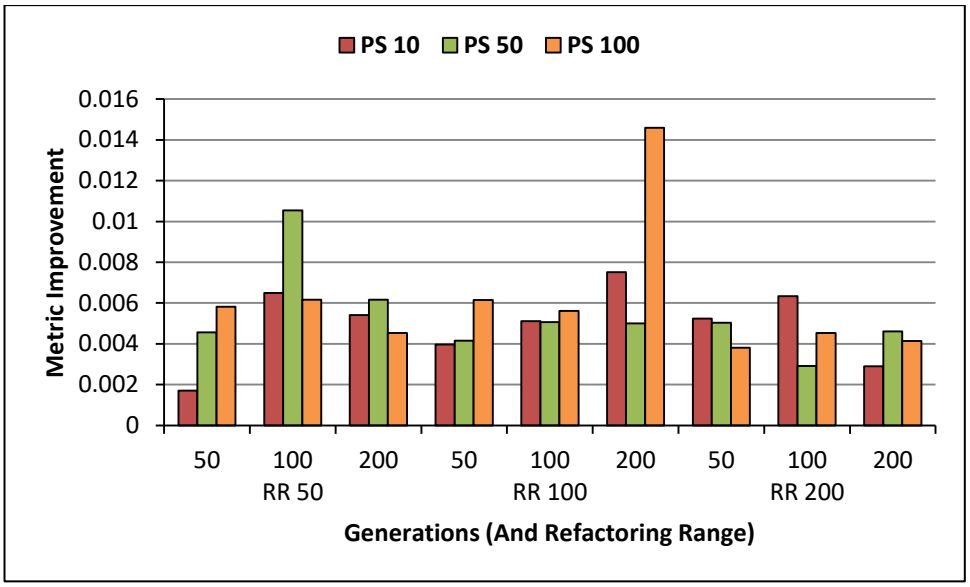

Figure 4.3 – Metric Improvements for Different Configuration Parameters.

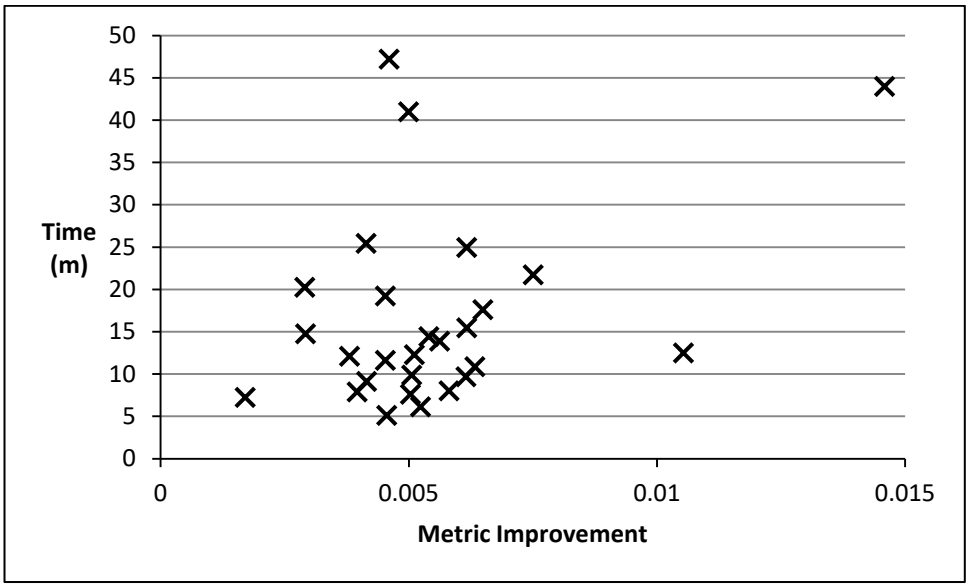

Figure 4.4 – Metric Improvements Mapped Against Time Taken for Different Configuration Parameters.

The configuration settings used in experiments 3 and 4 are derived from the ones determined to be more ideal and are listed in Table 4.3. Figure 4.5 gives the average quality gains conceived by each individual metric across all of the inputs. For many of the metric, where the quality gains are difficult to see, data labels have been given. In Table 4.4, they are grouped into the metrics that have

similar levels of volatility. Table 4.4 also gives abbreviations for each metric which is used in Figure 4.5. Three of the metrics, *Class Design Size*, *Number Of Hierarchies* and *Number Of Files*, showed no improvement at all. These metrics are more abstract, relating to the project design and class measurements as opposed to other metrics measuring more low-level attributes like methods and fields. Although class level refactorings do exist in the MultiRefactor tool, they will be less likely to be applied due to the conditions necessary to apply them without modifying the program functionality. Likewise, the most volatile metrics captured in the bottom group all relate to more low-level aspects of the code. It seems that these types of software metric may be more useful for driving change in an automated refactoring system due to the increased likelihood that the structural refactorings will be able to affect them.

Table 4.3 – Genetic Algorithm Configuration Settings

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

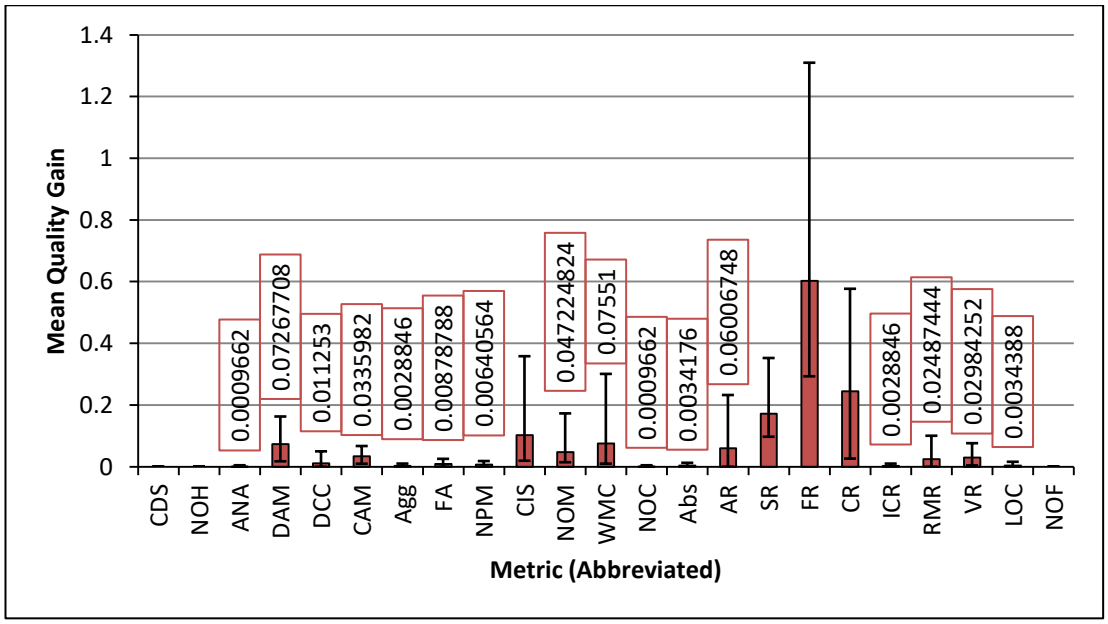

Figure 4.5 – Mean Metrics Gains

Table 4.4 – Mean Metric Gains with Abbreviations and Directions of Improvement

Metric	Abbreviation	Direction	Mean Metric Gain
Class Design Size	CDS	+	0
Number Of Hierarchies	NOH	+	0
Number Of Files	NOF	+	0
Average Number Of Ancestors	ANA	+	0.0009662
Number Of Children	NOC	+	0.0009662
Aggregation	Agg	+	0.0028846
Functional Abstraction	FA	+	0.00878788
Number Of Polymorphic Methods	NPM	+	0.00640564
Abstractness	Abs	+	0.0034176
Inner Class Ratio	ICR	+	0.0028846
Lines Of Code	LOC	-	0.0034388
Data Access Metric	DAM	+	0.07267708
Direct Class Coupling	DCC	-	0.011253
Cohesion Among Methods	CAM	+	0.0335982
Number Of Methods	NOM	-	0.047224824
Weighted Methods Per Class	WMC	-	0.07551
Abstract Ratio	AR	+	0.06006748
Referenced Methods Ratio	RMR	+	0.02487444
Visibility Ratio	VR	=	0.02984252
Class Interface Size	CIS	+	0.10246376
Static Ratio	SR	-	0.17167356
Final Ratio	FR	+	0.60217196
Constant Ratio	CR	+	0.24485396

The metric functions used in experiment 4 were taken from the metric groups derived in Table 4.4. The least volatile metrics that were from the top 2 groups were left out and the remaining metrics were split into 3 individual objectives to be used in a multi-objective setup by using the 3 remaining groupings of metrics to each represent an objective. These particular groupings are informed by the average quality gains, with similarly volatile metrics being grouped together, although these groupings are used more as example objectives for the current experiment. The 3 groups of metrics can (and will) be combined to represent an overall improvement function for a generalised measure of software quality, with the average quality gain values across numerous different input programs informing its composition. Table 4.5 gives the list of metrics associated with each objective.

Table 4.5 – Individual Objectives Derived from Metric Experimentation

Objective 1	Objective 2	Objective 3
Class Interface Size	Data Access Metric	Aggregation
Static Ratio	Direct Class Coupling	Functional Abstraction
Final Ratio	Cohesion Among Methods	Number Of Polymorphic Methods
Constant Ratio	Number Of Methods	Abstractness
	Weighted Methods Per Class	Inner Class Ratio
	Abstract Ratio	Lines Of Code
	Referenced Methods Ratio	
	Visibility Ratio	

Figure 4.6 and Table 4.6 compare the average objective values with the separate mono-objective runs against the values generated with the multi-objective approach. The values for objective 1 had the largest ranges of results. The mono-objective approach for objectives 1 and 2 yielded improvements 1.2 and 1.3 times greater than the multi-objective approach, respectively. The other objective was slightly better with the multi-objective approach, though both improvement values where relatively small. The objective values for the 2 search approaches with the first and second objective were compared using a two-tailed Wilcoxon rank-sum test (for unpaired data sets) with a 95% confidence level. The multi-objective values were found not to be significantly lower than the mono-objective values in either case.

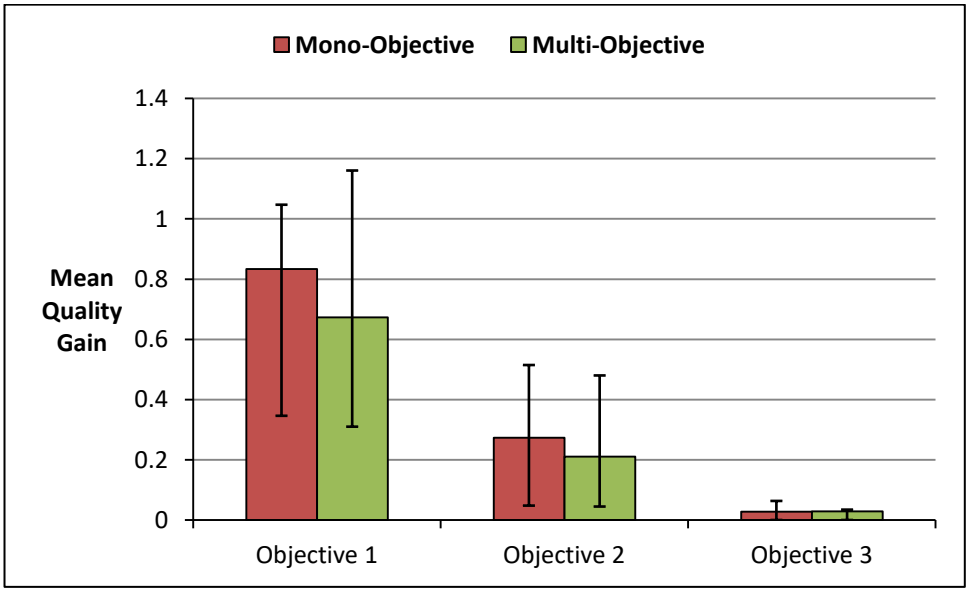

Figure 4.6 – Mean Metric Gains for Each Objective in a Mono-Objective and Multi-Objective Setup

Table 4.6 – Individual Objective Mean Metric Gains for Mono-Objective and Multi-Objective Optimisation

	Objective 1	Objective 2	Objective 3
Mono-Objective	0.8335831	0.2732774	0.028064733
Multi-Objective	0.672707033	0.210753367	0.028501433

The execution times for the 2 approaches were also compared to analyse how much more time is needed in the multi-objective approach to handle the 3 objectives simultaneously. Figure 4.7 compares the average times taken to run each approach, by finding the average values taken to run each input and then averaging them together for each objective to give an overall mean time. The multi-objective time is shorter than the mono-objective times for all 3 of the objectives. The range of times for each approach is quite large due to the disparate average times for each input, from 33 seconds for the JSON input to 28 minutes and 40 seconds for JHotDraw.

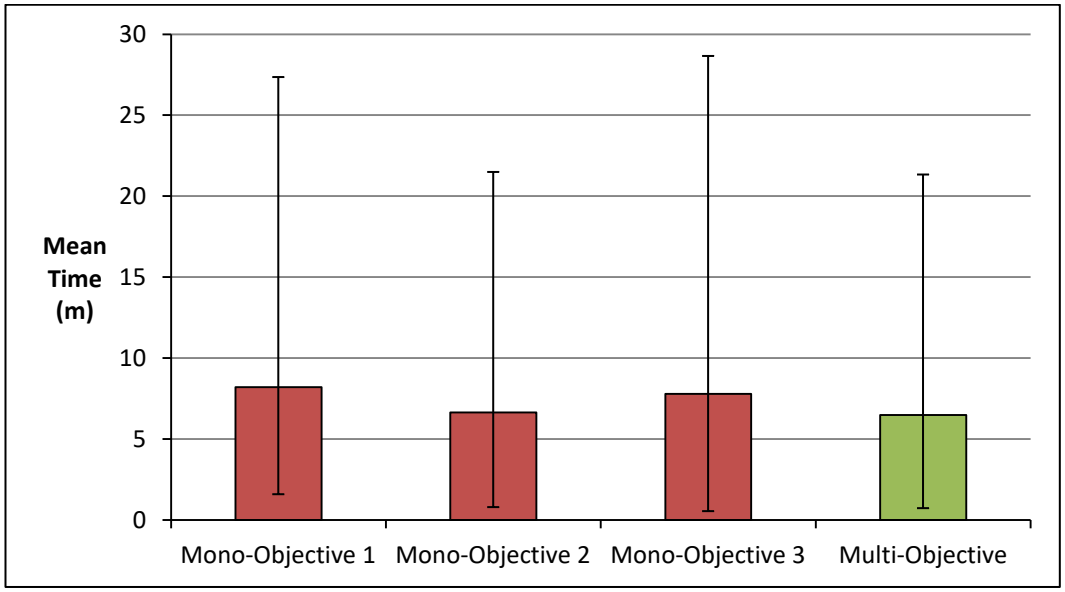

Figure 4.7 – Mean Time Taken to Run Each Objective of the Mono-Objective Approach and the Multi-Objective Approach

Figures 4.8 and 4.9 compare the overall times taken for the mono-objective and multi-objective approaches. In Figure 4.8, the overall times taken for each individual objective of the mono-objective search are compared with the overall time taken to run the 3 objectives in the multi-objective approach. Figure 4.9 compares the overall time taken to run all 3 objectives in the mono-objective approach against the multi-objective counterpart. It stacks the times for each separate objective in the mono-objective search to show the influence of each one on the time. In an unexpected result, the multi-objective approach took less time than each of the mono-objective approaches, despite using the search technique with more complex fitness calculations. The average time taken for the mono-objective algorithm to run for each objective was 3 hours, 46 minutes and 17 seconds. For the multi-objective approach to run for all the inputs it took 3 hours, 14 minutes and 49 seconds, a reduction against the mono-objective average of 31 minutes and 28 seconds. Also, as shown in Figure 4.9, for the mono-objective approach to run the inputs for all 3 objectives would take over 11 hours, meaning 71.3% of time is saved running 1 multi-objective search against running 3 separate mono-objective searches.

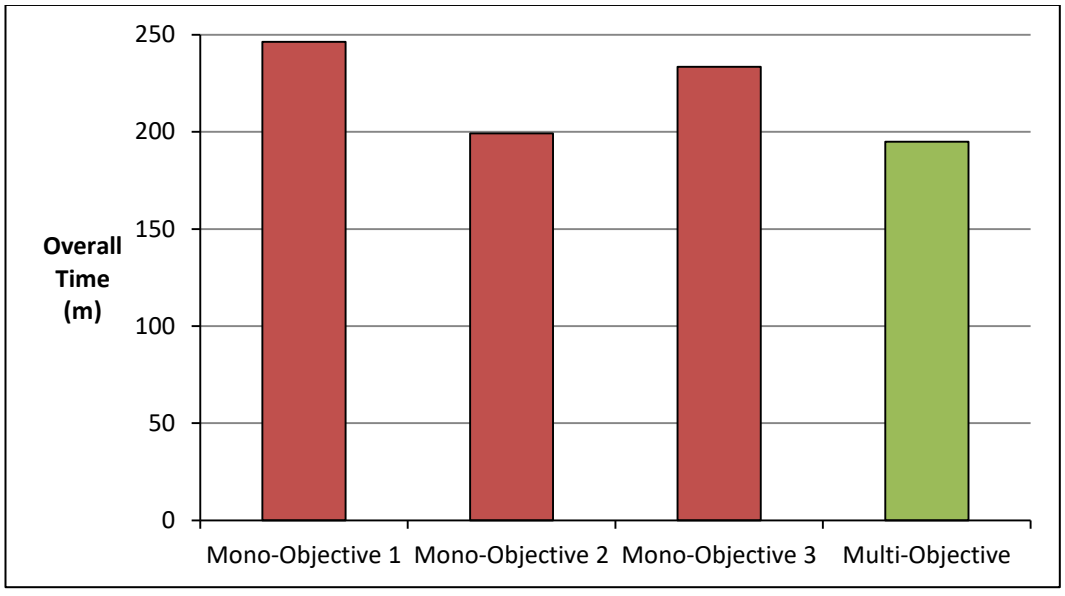

Figure 4.8 – Overall Time Taken to Run Each Objective of the Mono-Objective Approach and to Run the Multi-Objective Approach

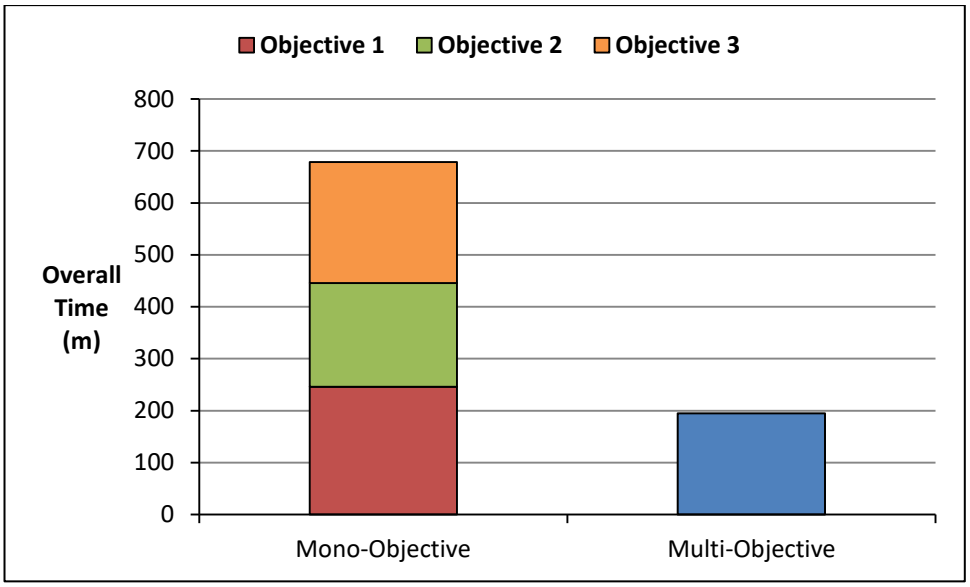

Figure 4.9 – Overall Time Taken for Each Approach, with Each Objective of the Mono-Objective Approach Stacked on Top of Each Other

4.4 Threats to Validity

4.4.1 Internal 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 the tasks across 5 different open source programs and for experiment 3 each metric is run against each program 5 times. Average values are then used to compare against each other. Likewise, for experiment 4, each approach is run 6 times for each input and average values are used. 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 in experiments 1 and 2.

4.4.2 External Validity

External validity is concerned with how well the results and conclusions can be generalised. In this study, the experimentation was performed on 5 different

real world open source systems belonging to different domains and with different sizes and complexities. However, the experiments and the capabilities of the refactoring tool used are restricted to open source Java programs. Therefore, it cannot be asserted that the results can be generalised to other applications or to other programming languages.

4.4.3 Construct Validity

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. 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 2 search approaches was measured using execution time in order to measure and compare cost.

4.4.4 Conclusion Validity

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 experiment. At the outset, 2 research questions have been provided, and for **RQ4.2**, 2 sets of corresponding hypotheses have been constructed in order to aid in drawing a conclusion. To accompany these, non-parametric statistical tests have been used to test the significance of the results generated. These tests make no assumption that the data is normally distributed and are suitable for ordinal data.

4.5 Conclusion

Four experiments were run to test various aspects of the MultiRefactor tool. The configuration parameters of the GA were tested to analyse the effect that they

can have on the refactoring process and to deduce what settings can have a better trade-off between metric improvement and time taken. Each of the available metrics were then tested with the GA across a number of real world, open source Java programs to find the least volatile metrics interacting with the - available refactorings, and address **RQ4.1**. It was found that the more low-level metrics produced greater average improvements compared to the more abstract, class level metrics. The results of this experiment were then used to construct metric functions to compare a mono-objective refactoring approach against a multi-objective approach. The more volatile metrics were split into 3 separate objectives to see if the multi-objective approach could generate comparable results to the mono-objective counterparts. The individual mono-objective approaches gave better results for 2 out of the 3 objectives but the multi-objective approach managed to generate suitable improvements for all of the objectives. The multi-objective approach took less time than each mono-objective approach, with the single multi-objective run taking 71% less time than the 3 combined mono-objective runs.

To address **RQ4.2** and to answer the hypotheses constructed, statistical tests were used to decide whether the data sets were significantly different. While the other objective was better with the multi-objective approach, the statistical test was run for the first and second objectives where the multi-objective approach was worse. The values in the multi-objective approach were not significantly worse than in the mono-objective approach for either objective, thus rejecting the alternative hypothesis **H4.1**$_A$. The execution time taken to run the multi-objective approach was compared against the times for each of the 3 mono-objective approaches. In none of the 3 cases did the multi-objective approach take longer to run than the mono-objective approach, thus rejecting the alternative hypothesis **H4.2**$_A$. No known refactoring tool currently allows the user to use multi-objective techniques to improve the software without having to manually apply the refactorings. The experiments conducted in this chapter suggest that this fully automated approach may be feasible and can allow for multiple separate objectives to be considered in a single run within an acceptable amount of time, although the improvement of a subset of these objectives may take a hit. The next chapter investigates 3 newly proposed objectives; priority, refactoring coverage and element recentness. The tool is

outfitted to incorporate the use of the new objectives, and they are tested in a multi-objective setup against the basic GA using just the quality objective.

Chapter 5

Secondary Objectives

5.1 Introduction

There are 3 secondary objectives proposed and tested in this chapter. The experiments used to test each objective are set up in a similar manner. This chapter is organised as follows. Firstly, the properties that the objectives relate to are detailed, and a justification is given for each. Research questions and hypotheses are proposed for each objective. Then, Section 5.2 discusses the modifications made to the MultiRefactor tool after the previous experiment to incorporate each of the new objectives. Section 5.3 explains the setup of the experimentation used to test the new objectives, as well as the new inputs used in the priority experiment and the element recentness experiment. Section 5.4 analyses the results of the priority experiment by looking at the objective values and the times taken to run the tasks. Section 5.5 discusses the results of the priority experiment. Section 5.6 analyses the results of the refactoring coverage experiment, and Section 5.7 discusses them. Likewise, Section 5.8 analyses the results of the element recentness experiment, and Section 5.9 discusses them. Section 5.10 inspects the threats to validity of the experimentation and Section 5.11 concludes the chapter.

5.1.1 Priority Objective

The first objective to use in conjunction with a quality function is one that incorporates the priority of the classes in the solution. There are a few situations in which this may be useful. Suppose a developer on a project is part of a team,

where each member of the team is concerned with certain aspects of the functionality of the program. This will likely involve looking at a subset of specific classes in the program. The developer may only have involvement in the modification of their selected set of classes. They may not be aware of the functionality of the other classes in the project. Likewise, even if the person is the sole developer of the project, there may be certain classes which are more risky or more recent or in some other way more worthy of attention. Additionally, there may be certain parts of the code considered less well structured and therefore most in need of refactoring. Given this prioritisation of some classes for refactoring, tool support is better employed with refactoring directed towards those classes.

Another situation is that there may be some classes considered less suitable for refactoring. Suppose a developer has only worked on a subset of the classes and is unsure about other areas of the code, they may prefer not to modify that section of the code. Similarly, older 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 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, a project may have a class for logging that is referenced by many other classes. Generally, highly coupled classes are seen as having a negative impact, but for the purposes of the project it may be deemed unavoidable. In cases like this where the more unorthodox structure of the class is desired by the developer, these classes could be specified in order to avoid refactoring them to appease the software metrics used. 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.

For these reasons, 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 classes. The priority objective proposed 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 evading 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 priority objective measures the solutions after the refactorings have been applied to aid in choosing between the options available. This will then allow the objective to discern between the available refactoring solutions. An experiment has been constructed to test a GA that uses it against one that does not. The experiment is derived from the following research questions:

RQ5.1: Does a multi-objective solution using a priority objective and a quality objective give an improvement in quality?

RQ5.2: Does a multi-objective solution using a priority objective and a quality objective prioritise classes better than a solution that does not use the priority objective?

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

H5.1: The multi-objective solution gives an improvement in the quality objective value.

H5.1$_A$: The multi-objective solution does not give an improvement in the quality objective value.

H5.2: The multi-objective solution gives significantly higher priority objective values than the corresponding mono-objective solution.

H5.2$_A$: The multi-objective solution does not give significantly higher priority objective values than the corresponding mono-objective solution.

5.1.2 Refactoring Coverage Objective

The second objective considered measures the amount of coverage that a refactoring solution can give among the elements of the solution. Coverage is important since a developer may not want the solution to focus on only a few parts of the code or get stuck on certain areas. Ensuring good coverage also

avoid the possibility of a solution focusing more on specific areas of a class performing redundant refactorings. As the MultiRefactor tool has many complimentary refactorings available (such as *Make Class Abstract*/*Make Class Concrete*), it is possible that a solution will have a number of refactorings that are applied to the same elements and that reverse the effects of the refactorings that come before it, causing the effect to be meaningless. Having an objective measurement to increase the code coverage of the refactorings reduces the likelihood of these redundant refactorings being performed. Another advantage of this objective is that it could be used in conjunction with the priority objective proposed in the previous chapter to focus a refactoring solution to a certain area of the code by listing a certain selection of classes, but also to increase the coverage of the refactorings within that area to look at as many elements within the selection of classes as possible.

A refactoring coverage objective has been constructed within the tool to assess 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 2 factors. Firstly, the number of elements inspected within a refactoring solution is considered, where more elements will give a better score. 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. This means that for example, in the *Collapse Hierarchy* refactoring, none of the methods or fields being moved up within that refactoring will be noted. Instead the class being collapsed will be considered to be the relevant element corresponding to the refactoring when calculating the coverage score. 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 elements. Secondly, the number of times each element is refactored is considered. 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 and encouraging the dispersion of refactorings across the code.

Note that if a developer disagrees with the choice of how to prioritise either of the 2 factors, it is a trivial matter to tweak the objective to change them. If it is more desirable to minimise the number of elements refactored or to maximise the number of times an element can be refactored, the objective calculation can be flipped to allow that, as the tool has been designed to be configured to a developers needs. In order to test the effectiveness of the objective a choice has been made as to how these factors should be prioritised. To test the effectiveness of the refactoring coverage objective, the experiment conducted tests a GA that uses it against one that does not, like with the priority objective. In order to judge the outcome of the experiment, the following research questions have been derived:

RQ5.3: Does a multi-objective solution using a refactoring coverage objective and a quality objective give an improvement in quality?

RQ5.4: Does a multi-objective solution using a refactoring coverage objective and a quality objective diversify code coverage among refactorings better than a solution that does not use the refactoring coverage objective?

The following hypotheses and alternative hypotheses have been constructed to measure success in the experiment:

H5.3: The multi-objective solution gives an improvement in the quality objective value.

H5.3$_A$: The multi-objective solution does not give an improvement in the quality objective value.

H5.4: The multi-objective solution gives significantly higher refactoring coverage objective values than the corresponding mono-objective solution.

H5.4$_A$: The multi-objective solution does not give significantly higher refactoring coverage values than the corresponding mono-objective solution.

5.1.3 Element Recentness Objective

The final objective proposed for use in a multi-objective solution incorporates the use of numerous previous versions of the software code. It is fairly common for a programming team to develop successive releases of a product in order to add new features over time. It is likely that the team will have a repository with

various compilable versions of the code leading up to the current release. Therefore, it is possible to use these previous versions to gather information of the program and to allow that information to aid in the maintenance approach of the current version. This idea forms the basis of the element recentness objective, by incorporating the use of multiple versions of the code as artefacts to aid the refactoring search. The justification for including a recentness aspect is that, whereas older elements have been given the chance to be tested more and have likely already been updated, newer elements will not have been considered. 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 is 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 among the code versions. 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 values.

The effectiveness of the element recentness objective is tested in the same way as the priority and refactoring coverage objectives, by testing a GA that uses it against one that does not. It may be argued that it is more relevant to refactor the older elements of the code. The more important aspects of the code may be different depending on the circumstances and the developer's opinion. As with the refactoring coverage objective, this objective can be tweaked to focus one way or the other (the older elements can be given higher scores and more recent elements lower scores) depending on the developers needs if this is desired. The choice has been made in this chapter to focus on more recent elements instead of older elements in order to test the effectiveness of the objective itself in doing what it aims. In order to judge the outcome of the experiment, the following research questions have been derived:

RQ5.5: Does a multi-objective solution using an element recentness objective and a quality objective give an improvement in quality?

RQ5.6: Does a multi-objective solution using an element recentness objective and a quality objective refactor more recent code elements than a solution that does not use the element recentness objective?

The following hypotheses and alterative hypotheses have been constructed to measure success in the experiment:

H5.5: The multi-objective solution gives an improvement in the quality objective value.

H5.5$_A$: The multi-objective solution does not give an improvement in the quality objective value.

H5.6: The multi-objective solution gives significantly higher element recentness objective values than the corresponding mono-objective solution.

H5.6$_A$: The multi-objective solution does not give significantly higher element recentness values than the corresponding mono-objective solution.

5.2 Refactoring Tool Evolution

For each of the secondary objective experiments the MultiRefactor tool had to be modified to improve the tool and to outfit it for the new objectives. These modifications are described below.

5.2.1 Priority Objective

For the purposes of experimentation with the tool a decision needed to be made in order for there to be a consistent way to choose solutions from the multi-objective tasks. With the mono-objective solutions, the top solution will always correspond to the best objective score but with the multi-objective population, numerous solutions may be valid depending on which objective has more importance. While this choice is useful for a developer, for experimentation, only 1 solution needs to be chosen in order to compare against the mono-objective counterpart. The tool was updated in order to choose a suitable solution out of the final population to inspect, using the process detailed below.

Firstly, the solutions in the population from the top rank are stored separately. It is from this subset that the best solution will be chosen from when the task is finished. Among these solutions, the tool inspects the individual objective values and for each, the best objective value across the solutions is stored. This set of objective values is the ideal point i.e. the best possible state that a solution in the top rank could have. After this is calculated, each objective score is compared with its corresponding ideal score. The distance of the objective score from its ideal value is found and, for each solution, the largest objective distance (i.e. the distance for the objective that is furthest from its ideal point) is stored. At this point each solution in the top rank has a value to represent the furthest distance among its objectives from the ideal point. The smallest among these is then considered to be the most suitable solution and is marked as such when the population is written to file. On top of this, the results file for the corresponding solution is flagged as the solution with the closest maximum distance from the ideal point in the top rank of solutions.

In order to implement the priority objective the tool needed to be upgraded to keep track of the classes modified in the refactorings. This involves tracking

when a class (or classes) is involved in the refactoring. This way the number of priority and non-priority classes used and the number of times they are used, as well as the overall number of class instances affected can be derived for each solution. In order to inform the tool of the relevant classes to use as priority and non-priority classes, they need to be specified in a text file and used as input in the place of a configuration file. When this happens, the tool will store the list of class names for reference when the fitness is calculated for the objective.

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 as an ordinal value. 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 disfavouring non-priority classes. It is important to note that non-priority classes are not necessarily excluded completely but solutions that do not involve those classes will be given priority. In this way the refactoring solution is still given the ability to apply structural refactorings that have a larger effect on quality even if they are in undesirable classes, whereas the priority objective will favour the solutions that have applied refactorings to the more desirable classes.

5.2.2 Refactoring Coverage Objective

For the mono-objective GA we need only measure the coverage score for the top solution in the population instead of for the whole population as in the priority experiment. Another change made was to reduce memory use when storing refactoring details during crossover. The methods used to check for the applicability of the refactorings during crossover were also updated to check for the *Move Method Down* and *Move Field Down* refactorings that, if these refactorings are reconstructed for crossover they will be executed in an identical way to the original refactoring.

The tool was also updated in order to reduce the necessary processing within the GA search. Originally, the tool was set up to find available elements to refactor

by checking for each relevant element in the file whether it was applicable for the chosen refactoring or not. This way, the tool can calculate the number of available elements that can be refactored in a file and then choose one at random for the search. Each refactoring has its own unique mayRefactor method to find out whether an element can be refactored. This method can be expensive as, depending on the complexity of the refactoring, a lot of work may be needed to ensure that an element can be refactored without negatively affecting the semantics of the program.

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 1 of those refactorable elements is chosen at random. Then, when the refactoring is applied (or when it is being reapplied to reconstruct the model for mutation or the first part of crossover or when printing out final population), the mayRefactor method is used again a number of times to find the relevant element. If the search wants, for instance, the third refactorable field in a file in order to apply a field refactoring, the mayRefactor method will be applied to every field in the class until the third refactorable field is found. This is done for each refactoring in each solution at every iteration of the GA. In order to reduce the use of this method, and improve the efficiency of the GA search, the approach to finding applicable elements was modified. Now, the positions of the relevant refactorable elements are stored in a different way, meaning that the need for the mayRefactor method is no longer needed to find them again. It is now used only when a new refactoring is created (during initialisation or mutation) or, modestly, during crossover.

In order to implement the refactoring coverage objective itself, extra information about the refactorings had to be stored in the refactoring sequence object used to represent a refactoring solution. For each solution, a list of the affected elements and the number of times each element has been refactored is stored in a hash table. For each refactoring, one element, considered to be most relevant to that refactoring, is chosen to be stored. For most refactorings this is straightforward as there is only 1 element being considered (i.e. for *Increase Field Visibility*, the field that is being refactored to increase its visibility is most relevant), but for some refactorings there could be more than 1 element that is considered relevant to the refactoring. For the move refactorings (*Move Field Down*, *Move*

Field Up, *Move Method Down*, *Move Method Up* and *Move Method*), the field or method being moved is considered and not the classes they are being moved to or from. For the *Collapse Hierarchy* and *Extract Subclass* refactorings, the class that the element(s) are being moved from is considered and not the element(s) being moved (after all, these are class level refactorings). After the solution has been created, the hash table will have a list of all the elements affected and the number of times for each. This information can then be used to construct the coverage score for that solution. More information about the coverage score itself is given in Section 5.3.

5.2.3 Element Recentness Objective

The refactoring output has been updated to give more information about the code elements that the refactorings have been applied to. For methods, the method signature (i.e. the set of parameter types in the method, if there are any) is now given as well as the method name. For the refactorings that apply to local fields or local parameter declarations, the method that the field is in will also be supplied. This is a less common possibility that is only applicable for the *Make Field Final* and *Make Field Non Final* refactorings. For these, the refactoring can be applied to global fields, local fields or local field parameters, whereas for any other field refactoring, the operation is only applicable for global fields. The output has also been updated to give more information about nested classes. Instead of just displaying the nested class name, the names for the full set of outer classes within the file will be supplied as well. Whereas before, these extra details were not given for the refactoring output, now they can be used to discern between code elements in cases where there are multiple methods or fields with the same name in a class or whether a class is nested or not.

The priority and refactoring coverage objectives were also updated to use these details to discern between possible duplicate elements. Whereas the priority objective was outfitted to be able to discern between classes with the same name by supplying the packages that the classes are within, it was not equipped to deal with nested classes. Now, nested classes can be included and read in by discerning them the same way packages are supplied. The refactoring coverage objective can now discern better between elements in order to ensure that they are distinct, and that identically named items aren't mistaken for being the same. The extra information supplied in the refactoring output (method

signatures, methods that local fields/parameters are in and nested class information) is now used to check that the refactoring coverage objective is more accurate when counting how many distinct elements are being refactored in a refactoring solution. The objective still can't discern, though, between duplicate elements that have the same name (and, if it's a method, the same signature or if it's a local field/parameter, are within a method with the same name and signature) within a different class. If 2 (non class) elements with the same details from different classes are come across within a refactoring solution, they will be treated as a single element instead of 2 distinct elements. Fortunately, these outlier cases will be less likely as, beyond being present in different classes, the details of the elements would have to be identical for them to be treated as the same element.

Another change to the refactoring output is that the visibility refactoring (*Increase Method Visibility*, *Decrease Method Visibility*, *Increase Field Visibility*, *Decrease Field Visibility*) outputs have been updated to reflect their name, whereas previously the outputs would note the refactorings as security refactorings instead of visibility refactorings (i.e. Increase Method Security applied...). The set of available refactorings have also been updated to include the *Extract Subclass* refactoring to complement the *Collapse Hierarchy* refactoring. *Extract Subclass* is, like many of the other available refactorings, based off the Fowler refactorings. *Extract Subclass* will choose a selection of local field declarations and/or method declarations from a class that are related to each other as a distinct unit, and will move them to a newly created subclass.

In order to inform the tool of the previous versions of the software to use in the element recentness objective, they need to be specified in a text file and used to replace the configuration file normally used as input. Each version of the software is to be supplied with a specification of where the code is in relation to the home directory. Each version needs to be ordered from oldest to most recent, although 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. When this configuration is read in, the tool will store the list of projects to inspect when the fitness is calculated for the objective.

The element recentness objective uses the same list that had been added for the refactoring coverage objective to store the code elements used in a refactoring solution. The recentness scores are calculated and stored as the objective is calculated, for each element it comes across. If the element has already been encountered in the search, its score will not need to be recalculated. This eliminates the need to calculate redundant element recentness values for elements that aren't refactored in the search, or that have already been refactored. The objective has some weaknesses in the accuracy of its measurements. One condition that isn't accommodated is the case where a code element exists in one version of the software, is removed and then is added back again. The element recentness objective will look back from the most recent version of the code and see that the element is not present. It will not continue to look through the older versions to see if the element had been removed and added back in. Instead it will count that as a sign that the element was not present before the applicable version. Also, as the element names are used to check their presence in previous versions, the objective will not be able to accommodate for elements that were present but had different names. As far as the objective is concerned, an element with a different name is a different element and it will not count.

For elements that have the same name in different classes, or classes that have the same name but are in different packages or are nested, the objective will not be able to tell the difference. It will look for that name and, if it is present in that version of the code, it will be counted. This introduces the possibility that code elements are noted as being older than they are, because another element with the same name was present when the relevant element wasn't. The issue with providing the class or package that the element is in to discern it from a possible duplicate is that the element may have been moved between classes from version to version. This would introduce the more likely possibility that an element that is older is not found with the element recentness objective. If the element is in a different class or the class is in a different package to the location in the current version of the code read in, it will be thought of as a distinct element with the same name and the element will be noted as being more recent than it is. The refactoring coverage objective has the same problem. For this reason, the extra information isn't included when calculating the element recentness.

5.3 Experimental Design

The experimental design was common across all 3 objectives. A set of tasks were set up in each experiment that used the objective to be compared against another set of tasks that didn't. The control group is made up of a mono-objective approach that uses a function to represent quality in the software. The corresponding tasks use the multi-objective algorithm and have 2 objectives. The first objective is the same function for software quality as used for the mono-objective tasks. The second objective is the secondary objective being tested, be it the priority, refactoring coverage or element recentness objective. The metrics used to construct the quality function and the configuration parameters used in the GAs are taken from the previous experiment on software quality. The software quality function used combines the metrics of the 3 objectives tested in the previous chapter (outlined in Table 4.5). No weighting is applied for any of the metrics. The metrics used in the quality function are given in Table 5.1. The configuration parameters used for the mono-objective and multi-objective tasks were derived through trial and error in the previous chapter, and were outlined in Table 4.3. Likewise, the hardware used to run the experiment was outlined in Table 4.2.

For the tasks, 6 different open source programs are used as inputs. Each one is run 5 times for the mono-objective approach and 5 times for the multi-objective approach, resulting in 60 different tasks for each objective experiment, and 180 tasks overall. For the priority experiment, 4 of the inputs used are the same as the inputs used in the previous experiment. The JSON program contains only 12 classes, and so was discarded for these experiments, being considered too small. In order to increase the external validity of the experiment, 2 larger Java programs, GanttProject and XOM, were used. GanttProject is a tool for project scheduling and management, whereas XOM (XML Object Model) is a tree based API for processing XML. The inputs used in the experiment as well as the number of classes and lines of code they contain are given in Table 5.2.

Table 5.1 – 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 5.2 – Java Programs Used in Priority Experiment and Refactoring Coverage Experiment

Name	LOC	Classes
Mango	3,470	78
Beaver 0.9.11	6,493	70
Apache XML-RPC 2.0	11,616	79
JHotDraw 5.3	27,824	241
GanttProject 1.11.1	39,527	437
XOM 1.2.1	45,136	224

For the refactoring coverage experiment, the same inputs are used, but in the element recentness experiment, they are changed again. Three of the inputs used are the same as the inputs used in the priority and refactoring coverage experiments. In order to ensure that there were a suitable number of previous versions of the project available for the element recentness objective to use, 3 of the inputs were updated. For 2 of the inputs, Apache XML-RPC and JHotDraw, later versions of the projects were used. As the Mango input doesn't have any different versions, it has been replaced. JRDF was chosen to replace Mango as it, like the others, has been used in previous studies related to SBSE and it is of a similar size to the other projects being used. JRDF is a Java library for parsing, storing and manipulating RDF (Resource Description Framework). The inputs used in the experiment as well as the number of classes and lines of code they contain are given in Table 5.3.

Table 5.3 – Java Programs Used in Element Recentness Experiment

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

Table 5.4 gives the previous versions of code used for each input, in order from the earliest version to the latest version used (excluding the current version being read in for maintenance). For each input, 5 different versions of code were used overall. The limit of 5 was set for pragmatic reasons in that the Beaver input had only 5 versions. For both the Apache XML-RPC and JHotDraw inputs, the versions previously used in experimentation are now included as part of the list of previous versions for the corresponding input. Not all sets of previous versions contain all the releases between the first and last version.

Table 5.4 – Previous Versions of Java Programs Used in Element Recentness Experiment

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 find the relevant secondary objective score for the mono-objective approach to compare against the multi-objective approach, the mono-objective GA has been modified to output the objective score after the task finishes in that experiment. For the quality function the metric changes are calculated using the normalisation function detailed in Chapter 3. The function was defined in Equation 3.1. 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 assessed by how far it is from its initial value. For metrics that start with a value of zero, the initial value used to compare against is changed to 0.01. This way, the normalisation function can still be used on the metric and its value still

starts off as low. For the secondary objectives, this normalisation function is not needed. These objective scores depend on the refactorings applied in a refactoring solution and will reflect the properties being measured.

For the multi-objective tasks used in the priority experiment, 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. In order to choose which classes to specify, the number of methods in each class of the input is 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. Using this process, the configurations of the priority objective for each input were constructed and used in the experiment.

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 maximum code coverage among their refactorings, and that have refactored as many elements as possible. The calculation of the refactoring coverage score has been streamlined in the tool to improve its efficiency. Equation 5.1 gives the formula used to calculate the coverage score in a refactoring solution using the hash table structure. m represents the current element and A_m represents the number of times the element has been refactored in the solution. n represents the number of elements refactored in the refactoring solution. Equation 5.2 gives the simplified version of the equation used in the tool.

$$n/(\frac{\sum_{m=1}^{n} A_m}{n}) \tag{5.1}$$

$$n^2/(\sum_{m=1}^{n} A_m) \tag{5.2}$$

For the element recentness objective, the recentness value of each element refactoring is calculated and then added together to get an overall score. Accumulating the score instead of getting an average recentness value avoids the solution applying a minimal number of refactorings in order to keep a low average and thus possibly yielding inferior quality improvements. Accumulating the individual values will encourage the solution to refactor as many recent elements as possible, and it will prioritise these elements, but it will also allow for older elements to be used if they improve the quality of the solution. Equation 5.3 gives the formula used to calculate the element recentness score in a refactoring solution using the hash table structure. m represents the current element, A_m represents the number of times the element has been refactored in the solution and R_m represents the recentness value for the element. n represents the number of elements refactored in the refactoring solution.

$$\sum_{m=1}^{n} A_m . R_m \tag{5.3}$$

5.4 Priority Results

Figure 5.1 gives the average quality gain values for each input program used in the experiment with the mono-objective and multi-objective approaches. For most of the inputs, the mono-objective approach gives a better quality improvement than the multi-objective approach, although for Mango the multi-objective approach was better. For the multi-objective approach all the runs of each input were able to give an improvement for the quality objective as well as look at the priority objective. For both approaches, the smallest improvement was given with GanttProject. The inputs with the largest improvements were different for each approach. For the mono-objective approach it was Beaver whereas, for the multi-objective approach, it was Apache XML-RPC.

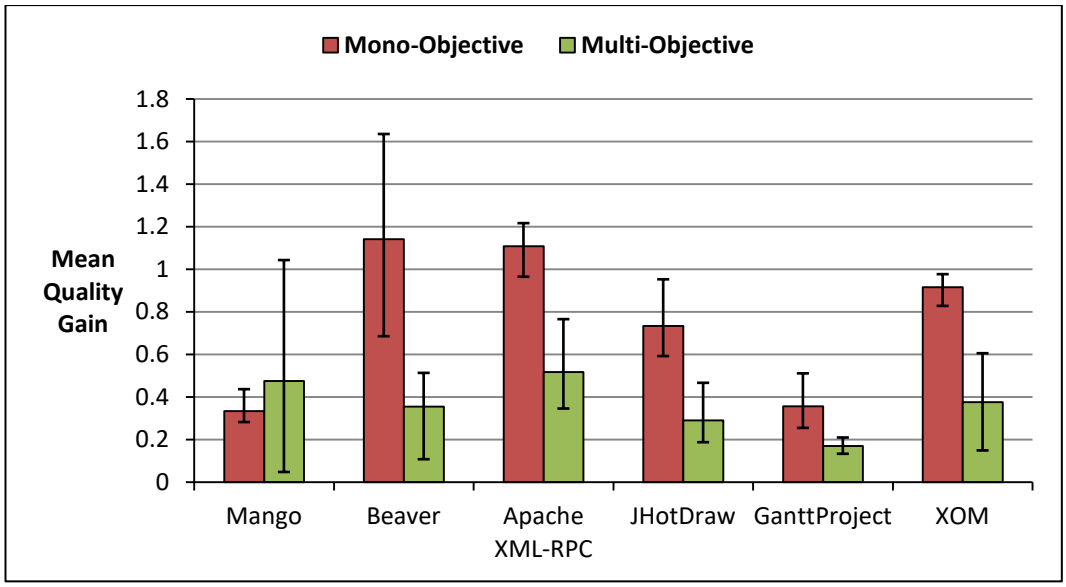

Figure 5.1 – Mean Quality Gain Values for Each Input

Figure 5.2 shows the average priority scores for each input with the mono-objective and multi-objective approaches. For all of the inputs, the multi-objective approach was able to yield better scores coupled with the priority objective. The values were compared for significance using a one-tailed Wilcoxon rank-sum test (for unpaired data sets) with a 95% confidence level (α = 5%). The priority scores for the multi-objective approach were found to be significantly higher than the mono-objective approach. For 2 of the inputs, Beaver and Apache XML-RPC, the mono-objective approach had priority scores that were less than 0. With the Beaver input, 1 of the runs gave a score of -6 and another gave a score of -10. Likewise, 1 run of the Apache XML-RPC input gave a priority score of -37. This implies that, without the priority objective to direct them, the mono-objective runs are less likely to focus on the more important classes (i.e. the classes with more methods), and may significantly alter the classes that should be disfavoured (leading to the minus values for the 3 mono-objective runs across the 2 input programs).

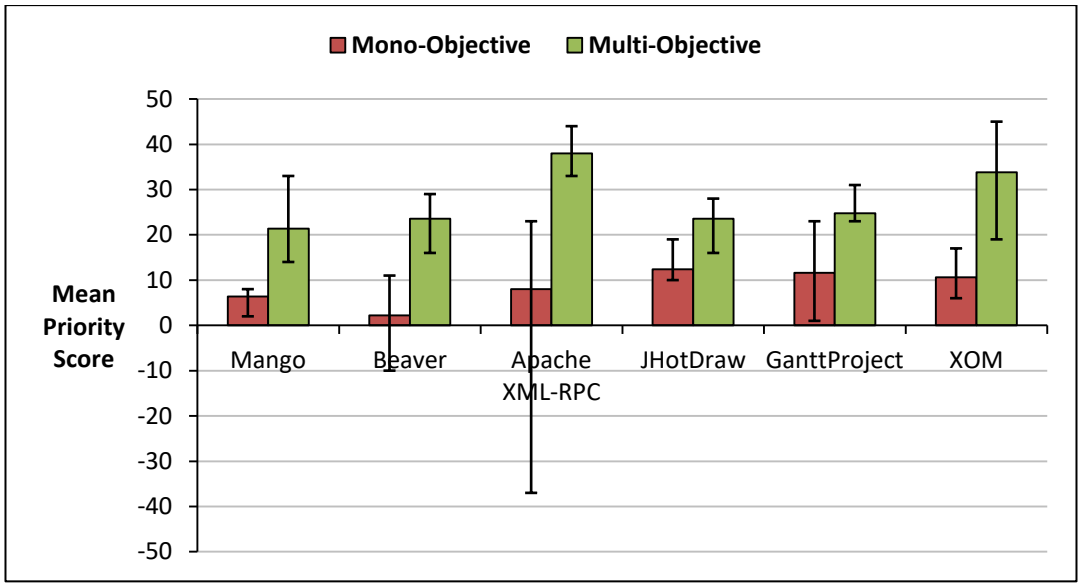

Figure 5.2 – Mean Priority Scores for Each Input

Figure 5.3 gives the average execution times for each input with the mono-objective and multi-objective searches. For most of the input programs, the multi-objective approach took less time than the mono-objective but, for GanttProject, the multi-objective approach took longer. To check that the execution times weren't significantly different, the Wilcoxon rank-sum test (two-tailed) was used again and the values were found to not be significantly different. The times for both approaches understandably increase as the input program sizes get bigger and the GanttProject input stands out as taking longer than the rest, although the largest input, XOM, is unexpectedly quicker. The execution times for the XOM input are smaller than both JHotDraw and GanttProject, despite it having more lines of code. However, both of these inputs do contain more classes. Considering the relevance of the list of classes in an input program to the calculation of the priority score, it makes sense that this would have an effect on the execution times. Indeed, GanttProject has by far the largest number of classes, at 437, which is almost double the amount that XOM contains. Likewise, the execution times for GanttProject are similarly around twice as large as those of XOM for the 2 approaches. The longest task run was for the multi-objective run of the GanttProject input, at over an hour. The average time taken for the multi-objective GanttProject tasks was 53 minutes and 6 seconds.

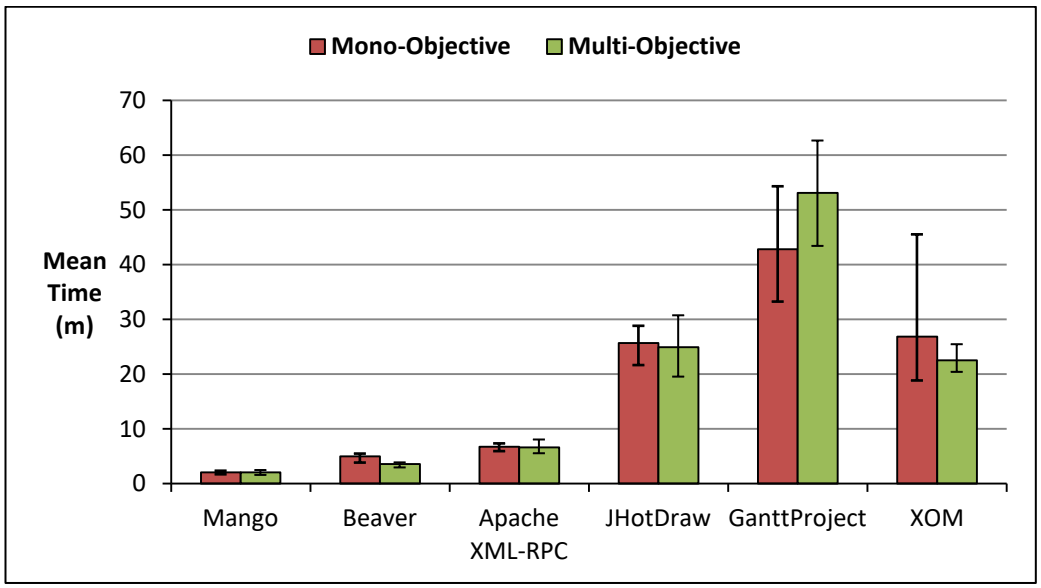

Figure 5.3 – Mean Times Taken for Each Input

5.5 Priority Objective Discussion

The average quality improvement scores were compared across 6 different open source inputs and, for the most part, the mono-objective approach gave better improvements. The likely reason for the better quality score in the mono-objective approach is due to the opportunity for the mono-objective GA to focus on that single objective without having to balance the possibly conflicting aim of favouring priority classes and disfavouring non-priority classes. The multi-objective approach was able to yield improvements in quality across all the inputs. In one case, with the Beaver input, the multi-objective was able to not only yield an improvement in quality, but also generate a better improvement on average than the mono-objective approach. This may be due to the smaller size of the Beaver input, which could mean a restricted number of potential refactorings in the mono-objective approach. It could also be influenced by the larger range of results gained the multi-objective approach for that input. The average priority scores were compared across the 6 inputs and, for the mono-objective approach, were able to give some improvement. However, in some specific runs, the priority scores were negative. This would relate to there being

more non-priority classes being refactored in a solution than priority classes, which, for the mono-objective approach, is unsurprising. The average priority scores for the multi-objective approach were better in each case. It is presumed that, as the mono-objective approach has no measures in place to improve the priority score of its refactorings, the solutions are more likely to contain non-priority classes and less likely to contain priority classes than the solutions generated with the multi-objective approach.

The average execution times for each input were inspected and compared for each approach. For most inputs, the multi-objective approach was slightly quicker than the mono-objective counterpart. The times for each input program increased depending on the size of the program and the number of classes available. The average times ranged from 2 minutes and 2 seconds for the Mango program, to 53 minutes and 6 seconds for GanttProject. While the increased times to complete the tasks for larger programs make sense due to the larger amount of computation required to inspect them, XOM took less time than GanttProject and JHotDraw. Although XOM has more lines of code than these inputs, the reason for this is likely due to the number of classes available in each program, which is more reflective of the time taken to run the tasks for them. Therefore, it seems to be implied that the number of classes available in a project will have a more negative effect on the time taken to execute the refactoring tasks on that project than the amount of code. It was expected that, due to the higher complexity of the MOGA in comparison to the basic GA, the execution times for the multi-objective tasks would be higher also. Although the times taken were similar for each approach, and were more affected by the project used, this wasn't the case for all of the inputs. This may have been due to the stochastic nature of the search. Depending on the iteration of the task run, there may be any number of refactorings applied in a solution. If one solution applied a large number of refactorings, this could likely have a noticeable effect on the time taken to run the task. The counterintuitive execution times between the mono-objective and multi-objective tasks may be a result of this property of the GA.

In order to test the aims of the experiment and derive conclusions from the results a set of research questions were constructed. Each research question and their corresponding set of hypotheses looked at 1 of 2 aspects of the experiment.

RQ5.1 was concerned with the effectiveness of the quality objective in the multi-objective setup. To address it, the quality improvement results were inspected to ensure that each run of the search yielded an improvement in quality. In all 30 of the different runs of the multi-objective approach, there was an improvement in the quality objective score, therefore rejecting the alterative hypothesis **H5.1**$_A$ and supporting **H5.1**. **RQ5.2** looked at the effectiveness of the priority objective in comparison with a setup that did not use a function to measure priority. To address this, a non-parametric statistical test was used to decide whether the mono-objective and multi-objective data sets were significantly different. The priority scores were compared for the multi-objective priority approach against the basic approach and the multi-objective priority scores were found to be significantly higher than the mono-objective scores, supporting the hypothesis **H5.2** and rejecting the alterative hypothesis **H5.2**$_A$. Thus, the research questions addressed for this experiment help to support the validity of the priority objective in helping to improve the focus of a refactoring solution in the MultiRefactor tool while in conjunction with another objective.

5.6 Refactoring Coverage Results

Figure 5.4 gives the average quality gain values for each input program used in the experiment with the mono-objective and multi-objective approaches. In all of the inputs, the mono-objective approach gives a better quality improvement than the multi-objective approach. For the multi-objective approach all the runs of each input were able to give an improvement for the quality objective as well as look at the refactoring coverage objective. For both approaches, the smallest improvement was given with GanttProject. The inputs with the largest improvements were different for each approach. For the mono-objective approach it was Beaver, whereas, for the multi-objective approach, it was XOM. Many observations about the quality gain values mirror those of the values derived from the priority experiment, as expected, although the mono-objective results for Beaver were less disparate and the average was smaller.

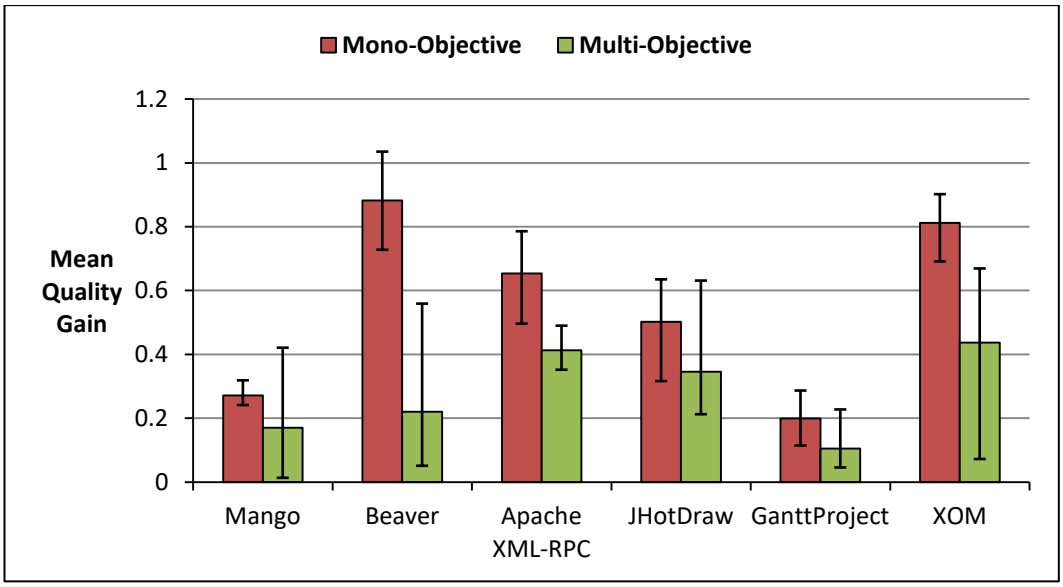

Figure 5.4 – Mean Quality Gain Values for Each Input

Figure 5.5 shows the average coverage scores for each input with the mono-objective and multi-objective approaches. For all of the inputs, the multi-objective approach was able to yield better scores coupled with the refactoring coverage objective. The values were compared for significance using a one-tailed Wilcoxon rank-sum test (for unpaired data sets) with a 95% confidence level. The coverage scores for the multi-objective approach were found to be significantly higher than the mono-objective approach. With the multi-objective approach, the average scores mostly increased as the input program sizes increased. This makes sense as the larger programs will contain more refactorable elements and classes in which to apply the refactorings in a solution. The notable exception to this is XOM, which had an average coverage score that is smaller than JHotDraw. This is likely due to the number of classes in the project being smaller than the GanttProject and JHotDraw class sizes. While the number of lines of code in XOM is greater, it may be that the number of code elements in the program is smaller. The scores seemed to vary slightly less with the multi-objective approach compared to the mono-objective counterparts. Again, this is understandable as the refactoring coverage objective used in the multi-objective approach to improve the program will drive the solutions towards more diverse sets of refactorings, pushing the coverage scores towards a higher peak. On the other hand, the mono-objective coverage scores

are more likely to be achieved as a by-product of the other objective, leading to more fluctuating sets of scores among the tasks.

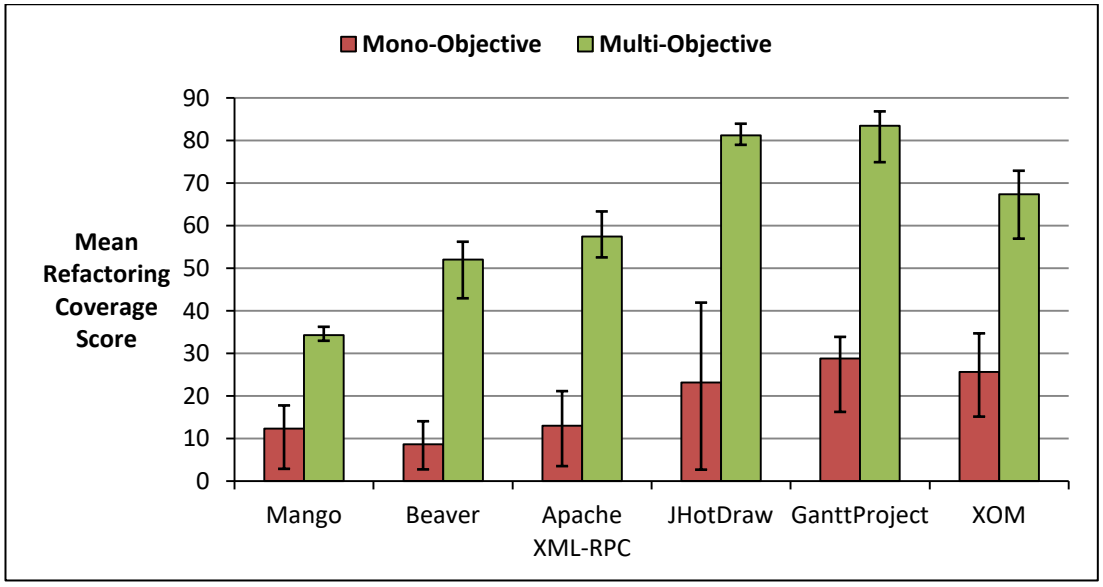

Figure 5.5 – Mean Refactoring Coverage Scores for Each Input

Figure 5.6 gives the average execution times for each input with the mono-objective and multi-objective searches. The times for the mono-objective and multi-objective tasks mirrored each other. For most input programs, the mono-objective approach was faster on average, with the exception to this being Apache XML-RPC. To ensure that the execution times weren't significantly different, the Wilcoxon rank-sum test (two-tailed) was used again and the values were found to not be significantly different. Again, the times generally increased as the project sizes increased, except for XOM, where the times where smaller than both JHotDraw and GanttProject. Once again these programs, while smaller, contain more classes than XOM, which may have contributed to their increased execution times. The GanttProject program stands out as taking the longest, with the longest tasks taking over 45 minutes to run, whereas the longest tasks among the other input programs took little over 30 minutes. Again, the execution times for GanttProject are around twice as large as those of XOM for the 2 approaches, which mirrors GanttProject having almost double the amount of classes as XOM.

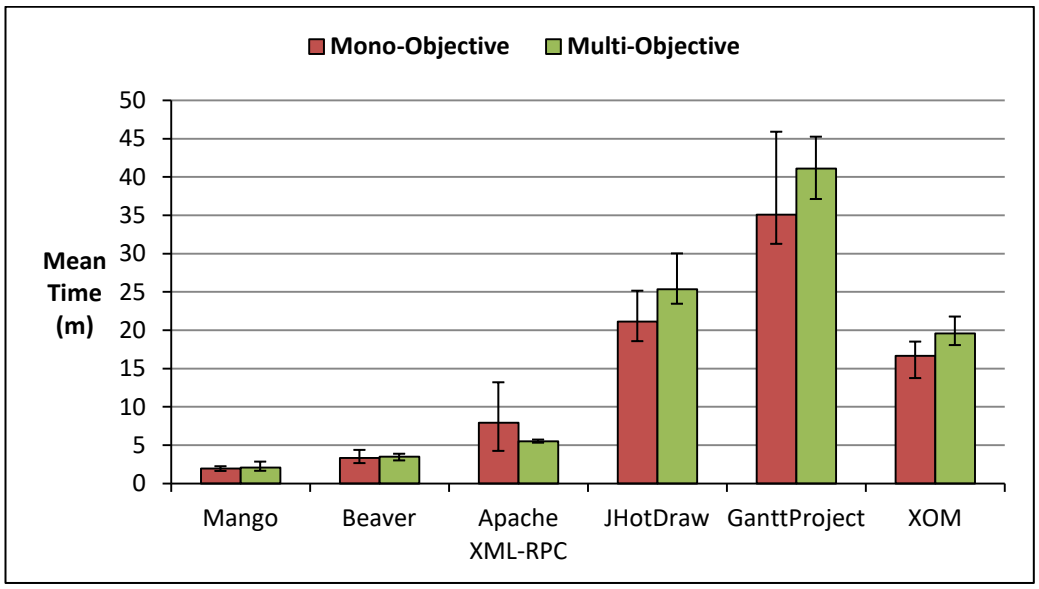

Figure 5.6 – Mean Times Taken for Each Input

When comparing the average execution times of the tasks in this experiment against those in the priority experiment, they have similar trends. What does stand out, though, is that the mono-objective times in the refactoring coverage experiment seem to have improved somewhat for many of the inputs against the priority times. This has caused the most of programs to take less time than their multi-objective counterparts whereas this wasn't the case before. The multi-objective times, on the other hand haven't changed much except for GanttProject, where the average time was 12 minutes shorter. There is a possibility that these slight improvements in the mono-objective times were caused by the modifications made to the MultiRefactor tool in order to reduce the use of the mayRefactor method.

5.7 Refactoring Coverage Objective Discussion

The average quality improvement scores were compared across 6 different open source inputs and, for all input programs, the mono-objective approach gave better improvements. The multi-objective approach gave improvements in

quality across all the inputs. The average coverage scores were compared across the 6 inputs. The scores for the multi-objective approach were better in each case. Finally, the average execution times for each input were inspected and compared for each approach. The times for each approach were similar but, for most inputs, the mono-objective approach was quicker than the multi-objective counterpart. The times for each input program increased depending on the number of classes available in the program. The average times ranged from 1 minute and 56 seconds for the Mango program, to 41 minutes and 7 seconds for GanttProject.

In order to test the aims of the experiment and derive conclusions from the results a set of research questions were constructed. **RQ5.3** was concerned with the effectiveness of the quality objective in the multi-objective setup. To address it, the quality improvement results were inspected to ensure that each run of the search yielded an improvement in quality. In all 30 of the different runs of the multi-objective approach, there was an improvement in the quality objective score, therefore rejecting the alternative hypothesis **H5.3A** and supporting **H5.3**. **RQ5.4** looked at the effectiveness of the refactoring coverage objective in comparison with a setup that did not use a function to measure refactoring coverage. To address this, a non-parametric statistical test was used to decide whether the mono-objective and multi-objective data sets were significantly different. The coverage scores were compared and the multi-objective coverage scores were found to be significantly higher than the mono-objective scores, supporting the hypothesis **H5.4** and rejecting the alternative hypothesis **H5.4A**. Thus, the research questions addressed for this experiment help to support the validity of the refactoring coverage objective in helping to improve the code coverage of a refactoring solution in the MultiRefactor tool while in conjunction with another objective.

5.8 Element Recentness Results

Figure 5.7 gives the average quality gain values for each input program used in the experiment with the mono-objective and multi-objective approaches. In all of the inputs, the mono-objective approach gives a better quality improvement

than the multi-objective approach. For the multi-objective approach all the runs of each input were able to give an improvement for the quality objective as well as look at the element recentness objective. For the mono-objective approach, the smallest improvement was given with GanttProject, and for the multi-objective approach, it was Apache XML-RPC. For both approaches, XOM was the input with the largest improvement. The mono-objective Beaver results were noticeable for having the most disparate range in comparison to the rest, which is somewhat similar to in the priority experiment. The results are similar to those captured in the previous 2 experiments for the 3 inputs that were used across all 3.

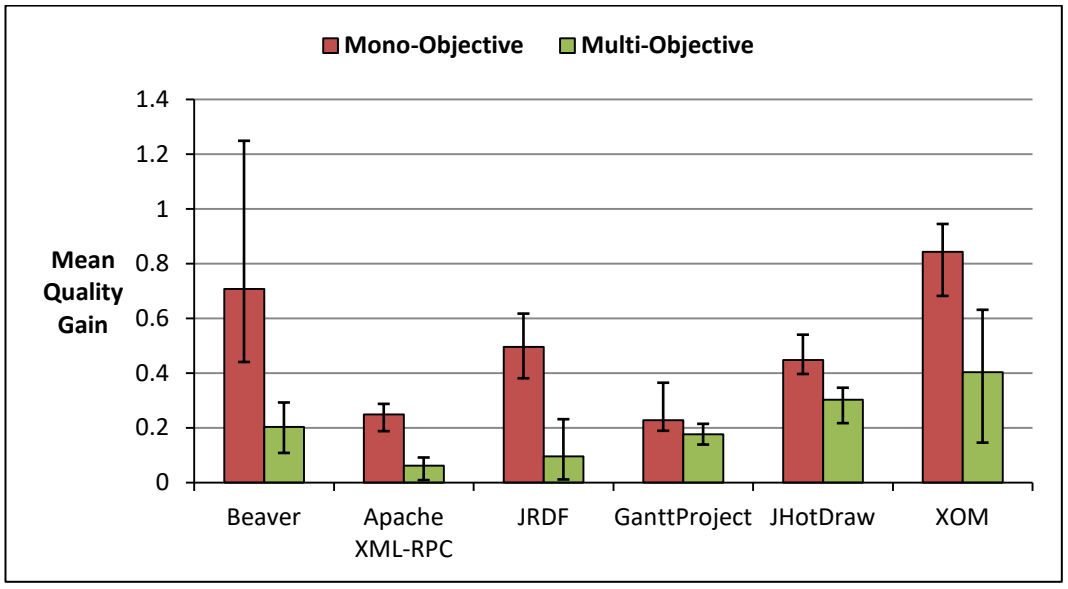

Figure 5.7 – Mean Quality Gain Values for Each Input

Figure 5.8 shows the average element recentness scores for each input with the mono-objective and multi-objective approaches. For all of the inputs, the multi-objective approach was able to yield better scores coupled with the recentness objective. The values were compared for significance using a one-tailed Wilcoxon rank-sum test (for unpaired data sets) with a 95% confidence level. The element recentness scores for the multi-objective approach were found to be significantly higher than the mono-objective approach. The scores tended to vary with both the mono-objective and multi-objective approaches. The exception to this in the

XOM input which had a more refined set of results for both approaches. Also, for this input, in comparison to the others, the multi-objective approach didn't give as much of an improvement in the element recentness score in relation to its mono-objective counterpart. For the mono-objective GanttProject scores, 1 of the tasks gave an anomalous result of 784 (the other values were between 212 and 400) that was greater than even the average multi-objective score for the input, at 764.8.

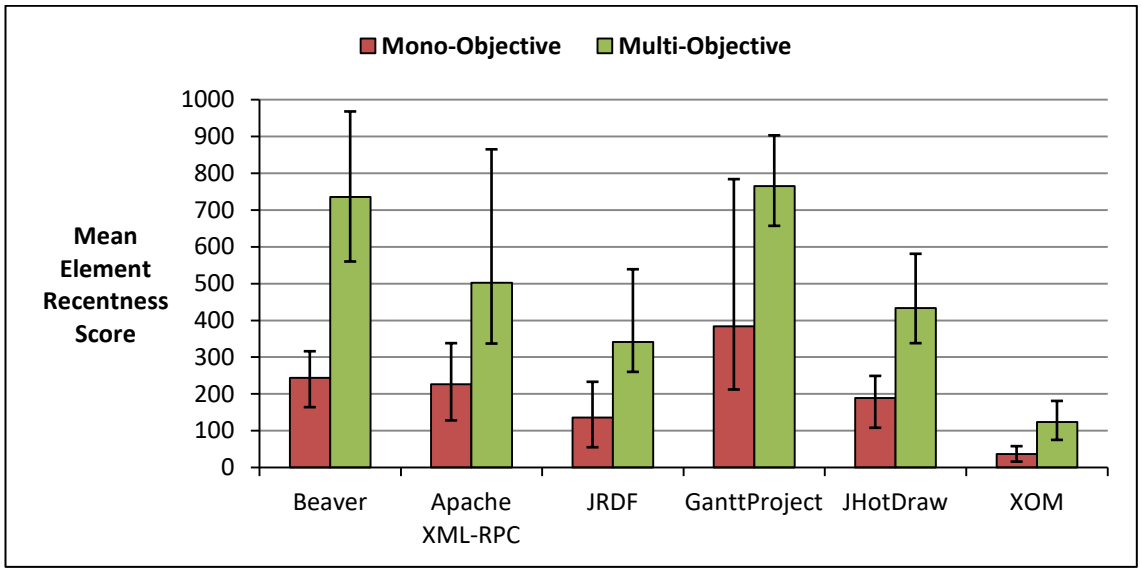

Figure 5.8 – Mean Element Recentness Scores for Each Input

Figure 5.9 gives the average execution times for each input with the mono-objective and multi-objective searches. The times for the mono-objective and multi-objective tasks mostly mirrored each other. For most input programs, the mono-objective approach was faster on average, with the exception being Beaver which is slightly longer. To ensure that the execution times weren't significantly different, the Wilcoxon rank-sum test (two-tailed) was used again and the values were found to not be significantly different. The times seemed to increase in relation to the number of classes in the project, although the mono-objective GanttProject time was slightly smaller than JHotDraw, an input with fewer classes. The multi-objective GanttProject times stand out as taking the longest, with the longest task taking almost 71 minutes to run. The average time for the multi-objective GanttProject tasks was just under 64 minutes, whereas the

average time for the next largest input, JHotDraw, was only 41 minutes and 6 seconds. Whereas the inputs had similar times for the mono-objective and multi-objective approaches, for GanttProject the multi-objective tasks took quite a bit longer (over 28 minutes longer on average).

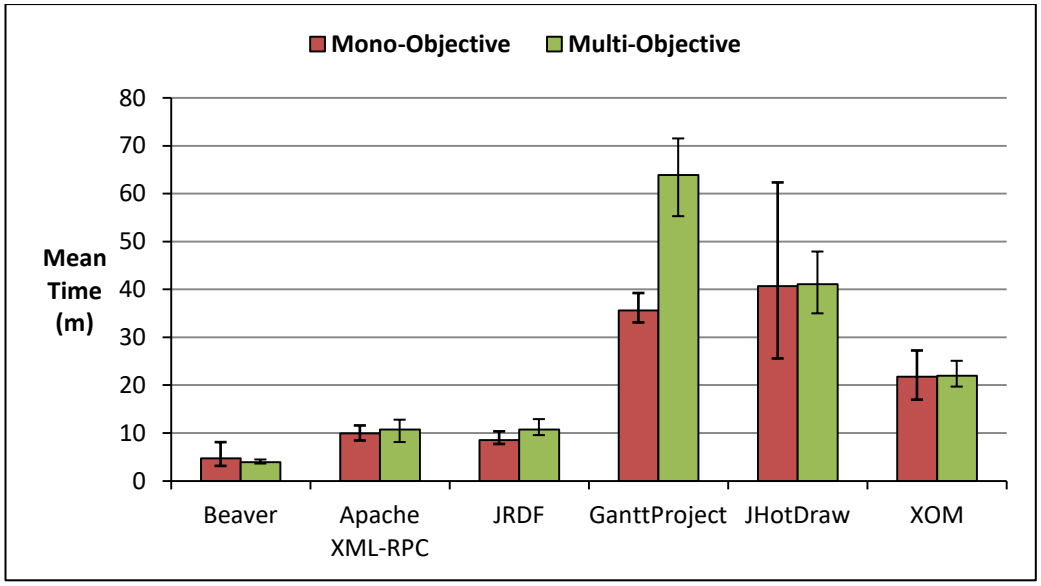

Figure 5.9 – Mean Times Taken for Each Input

5.9 Element Recentness Objective Discussion

The average quality improvement scores were compared across 6 different open source inputs and, for all input programs, the mono-objective approach gave better improvements. The multi-objective approach gave improvements in quality across all the inputs. The average element recentness scores were compared across the 6 inputs. The scores for the multi-objective approach were better in each case. Finally, the average execution times for each input were inspected and compared for each approach. The times for each approach were similar but, for most inputs, the mono-objective approach was quicker than the multi-objective counterpart. The average times ranged from 3 minutes and 57 seconds for Beaver, to 63 minutes and 54 seconds for GanttProject.

In order to test the aims of the experiment and derive conclusions from the results a set of research questions were constructed. **RQ5.5** was concerned with the effectiveness of the quality objective in the multi-objective setup. To address it, the quality improvement results were inspected to ensure that each run of the search yielded an improvement in quality. In all 30 of the different runs of the multi-objective approach, there was an improvement in the quality objective score, therefore rejecting the alternative hypothesis **H5.5A** and supporting **H5.5**. **RQ5.6** looked at the effectiveness of the element recentness objective in comparison with a setup that did not use a function to measure element recentness. To address this, a non-parametric statistical test was used to decide whether the mono-objective and multi-objective data sets were significantly different. The recentness scores were compared for the multi-objective approach against the basic approach and the multi-objective element recentness scores were found to be significantly higher than the mono-objective scores, supporting the hypothesis **H5.6** and rejecting the alternative hypothesis **H5.6A**. Thus, the research questions addressed for this experiment help to support the validity of the element recentness objective in helping to focus refactorings on recent elements in a software program with the MultiRefactor tool, while in conjunction with another objective.

5.10 Threats to Validity

5.10.1 Internal Validity

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 6 different open source programs and running against each program 5 times. 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. In the priority experiment, the classes chosen as priority and non-priority classes for each input may affect the results gained. This has been addressed by selecting the top 5% of classes that have the

most methods as priority classes and the bottom 5% with the least methods as non-priority classes for each input.

5.10.2 External Validity

In this study, the experiment was performed on 6 different real world open source systems belonging to different domains and with different sizes and complexities. However, the experiment and the capabilities of the refactoring tool used are restricted to open source Java programs. Therefore, it cannot be asserted that the results can be generalised to other applications or to other programming languages.

5.10.3 Construct Validity

The validity of the experiment is limited by the metrics used, as they are experimental approximations of software quality, as well as the priority objective used to measure the importance of the classes modified, the refactoring coverage objective used to measure the number of elements refactored, and the element recentness objective used to measure the recentness of the elements refactored. What constitutes a good metric for quality is very subjective. The cost measures used in the experiment can also indicate a threat to validity. Part of the effectiveness of the mono-objective and multi-objective search approaches was measured using execution time in order to measure and compare cost.

5.10.4 Conclusion Validity

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 formal hypotheses in the experimentation. At the outset of each experiment, 2 research questions have been provided and for each, a set of corresponding hypotheses have been constructed in order to aid in drawing a conclusion. To accompany these, non-parametric statistical tests have been used to test the significance of the results gained. These tests make no assumption that the data is normally distributed and are suitable for ordinal data.

5.11 Conclusion

In this chapter a set of experiments were conducted to test 3 new fitness objectives in the MultiRefactor tool. Each one of the objectives was detailed and modifications made to the MultiRefactor tool to incorporate them were discussed. Each newly proposed objective was tested in conjunction with the quality objective tested in Chapter 4 in a multi-objective setup. To measure the effectiveness of the secondary objective, the multi-objective approach was compared with a mono-objective approach using just the quality objective. The quality objective values were inspected to deduce whether improvements in quality can still be derived in this multi-objective approach. Then, the secondary objective scores for that experiment were compared.

The priority scores were compared to measure whether the developed priority function can be successful in improving the focus of the refactoring approach. The coverage scores were compared to measure whether the developed refactoring coverage function can be successful in improving the coverage of the refactoring approach and reducing redundant refactorings. Finally, the element recentness scores were compared to measure whether the developed element recentness function can be successful in focusing refactorings on more recently added elements in a software program. The next chapter tests a many-objective setup that uses all 4 objectives combined together using the many-objective GA. It also tests different variations of the objectives together to find out which objectives work well together and which don't.

Chapter 6

Many-Objective Approach

6.1 Introduction

In the previous chapters, a quality objective was constructed to measure quality in a software program as well as 3 supplementary objectives to assist the quality objective in improving other aspects of the software. In this chapter an experiment is set up to run all 4 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. In order to run the 4 objectives in a single many-objective solution, an adaptation of the many-objective algorithm NSGA-III is used in place of the NSGA-II adaptation.

It has been suggested that the Pareto dominance approach to handling multiple objectives becomes less effective with more than 3 objectives. For example, Deb and Saxena [38] demonstrated that the NSGA-II approach is vulnerable to a large number of objectives. Mkaouer *et al.* [25], [146] also investigated this claim in respect to the area of SBSE. They compared their approach on up to 15 objectives with numerous different EAs, including NSGA-II and NSGA-III. The performance with the NSGA-II approach degraded as the number of objectives increased, whereas the NSGA-III approach continued to be effective and outperformed the other algorithms. They concluded that NSGA-II is not adequate for problems involving more than 3 objectives, whereas NSGA-III is a very good candidate for tackling many-objective SBSE problems. NSGA-III replaces the crowding distance functionality with an alternative approach to

maintain the diversity in the chosen solutions. The algorithm was described in detail in Chapter 2 Section 2.7 along with other many-objective EAs, and the adaptation of the algorithm used in the MultiRefactor tool was discussed in Chapter 3 Section 3.4.3.

The experimentation is split into 2 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 3 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, similar to that for the previous experimental chapters. Then, the different permutations of the objectives are tested with each other and the quality objective in a 3-objective search. Overall, there are 6 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 derived:

RQ6.1: Does a many-objective solution using the priority, refactoring coverage and element recentness objectives with the quality objective give an improvement in quality?

RQ6.2: 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?

RQ6.3: Which combination of objectives in conjunction with the quality objective work best together?

The following hypotheses and alternative 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):

H6.1: The many-objective solution gives an improvement in the quality objective value.

H6.1A: The many-objective solution does not give an improvement in the quality objective value.

H6.2: The many-objective solution gives higher values for the priority, refactoring coverage and element recentness objectives than the corresponding mono-objective solution.

H6.2A: The many-objective solution does not give higher values for the priority, refactoring coverage and element recentness objectives than the corresponding mono-objective solution.

The remainder of this chapter is organised as follows. Section 6.2 explains the setup of the experimentation. Section 6.3 analyses the results of the experimentation, looking at the objective values and the times taken to run the tasks. Section 6.4 discusses these results, analysing the most successful ways to use each of the objectives. Section 6.5 inspects the threats to validity of the experimentation and Section 6.6 concludes the chapter.

6.2 Experimental Design

In part 1 of the experimentation, the mono-objective approach is compared with the many-objective search using all 4 objectives. Part 2 tests each different combination of objectives with the quality objective. Table 6.1 shows the 6 different permutations that are tested (along with the mono-objective and many-objective approaches), with abbreviations given for each permutation for reference. The metrics used to construct the quality function and the configuration parameters used in the GAs are taken from the experiment on software quality. The metrics used in the quality function were given in Table 5.1 and no weighting is applied. The configuration parameters used for the mono-objective and multi-objective tasks were derived through trial and error in the quality experiment, and were outlined in Table 4.3. Likewise, the hardware used to run the experiment was outlined in Table 4.2. For the tasks, 6 different

open source programs are used as inputs. The inputs used in the experimentation are the same as those used in the previous experiment and details of each were given in Table 5.3. For part 1 of the experimentation each of the 6 inputs is run 10 times for the mono-objective approach and 10 times for the many-objective approach. In part 2, the tasks are run 5 times for each input in each of the 6 approaches. This results in there being 120 tasks for part 1 and 180 tasks for part 2, meaning 300 tasks overall.

Table 6.1 – Different Combinations of Objectives Tested in Experimentation

Mono-Objective	Quality			
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	
Many-Objective	Quality	Priority	Refactoring Coverage	Element Recentness

As in the previous experiments, in order to find the other objective scores with the mono-objective approach to compare it against the other approaches, the mono-objective GA has been modified to output the other scores after the task finishes. This time, the scores for all 3 of the other objectives will be given, to see how those objectives fare when they are not being used in the search. This way the scores don't need to be calculated manually for the mono-objective approach. Again, the score will only be output for the top GA solution in the final population. For the quality function the metric changes are calculated using the normalisation function detailed in Chapter 3. The function was defined in Equation 3.1. 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. For metrics that start with a value of zero, the initial value used to compare against is changed to 0.01. This way, the normalisation function can still be used on the metric and its value still starts off as low.

In order to give a better balance between the supplemental objectives, each of them have been normalised as well. Whereas before, the scores given for these objectives were ordinal (or in the case of the element recentness objective, constrained by the number of versions of code available to use), in this case they are changed to 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. The original 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). Like with the original coverage calculation, 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 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 top solutions in the multi and many-objective approaches are chosen differently as well. The solution is still chosen from the top rank of solutions in the final population. Now, the solution used to compare against the other approaches is the one with the highest quality improvement value among those in the top rank. In order to compare the different permutations of objectives using the different input programs and newly normalised objective scores, the mono-objective and 3 bi-objective approaches have been repeated as part of the experimentation.

6.3 Results

Figure 6.1 gives the average quality gain values for each input program used in the experimentation with the mono-objective and many-objective approaches. In all of the inputs, the mono-objective approach gives a better quality

improvement than 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 3 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. The quality gain scores were similar to those derived from the element recentness experiment for each input program.

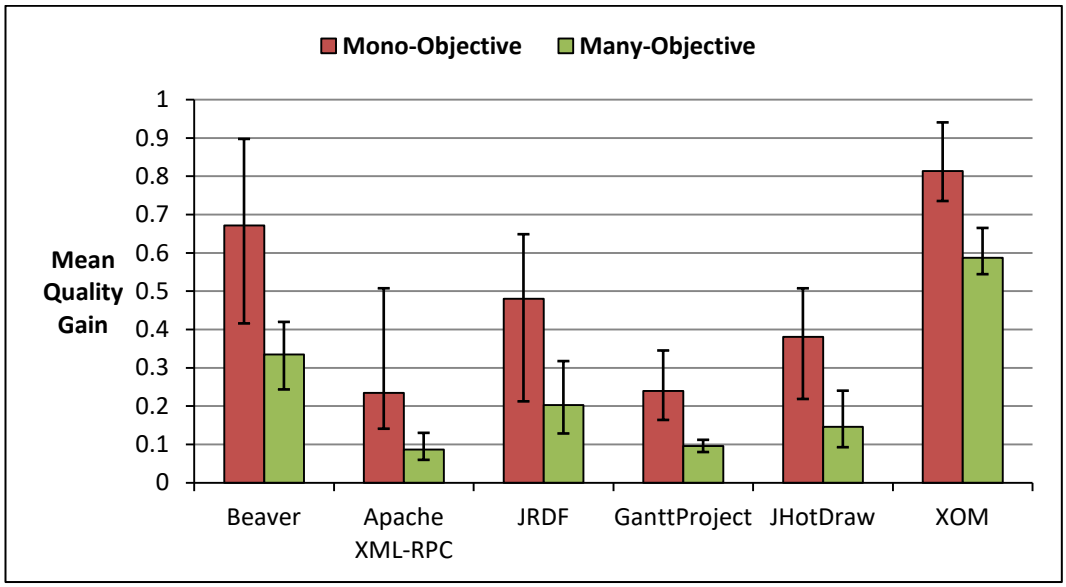

Figure 6.1 – Mean Quality Gain Values for Each Input

Figure 6.2 shows the average priority scores for each input with the mono-objective and many-objective approaches. For the JRDF input, where the scores are difficult to see, data labels have been provided. For all but 1 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. A few of the inputs had scores below 0. The mono-objective scores for JRDF, GanttProject and JHotDraw went below 0 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.

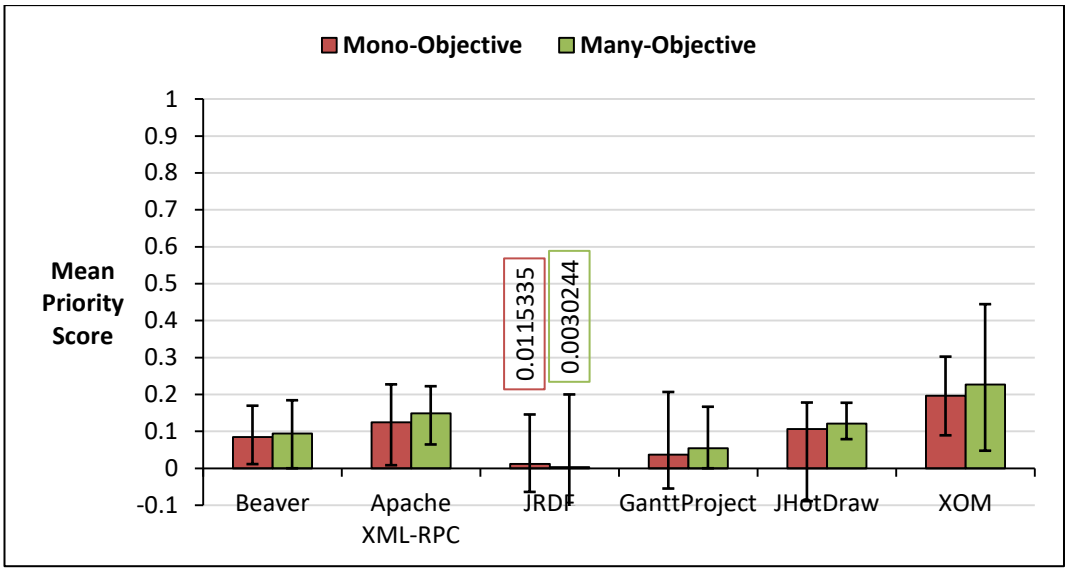

Figure 6.2 – Mean Priority Scores for Each Input

Figure 6.3 shows the average coverage scores for each input with the mono-objective and many-objective approaches. For all of the inputs, the many-objective approach was able to yield better scores coupled with the refactoring coverage objective. As seen in the refactoring coverage experiment, the scores seemed to vary slightly less with the multi-objective approach compared to the mono-objective counterparts. Again, 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 4 of the input programs contained runs where the score was the maximum value of 1.

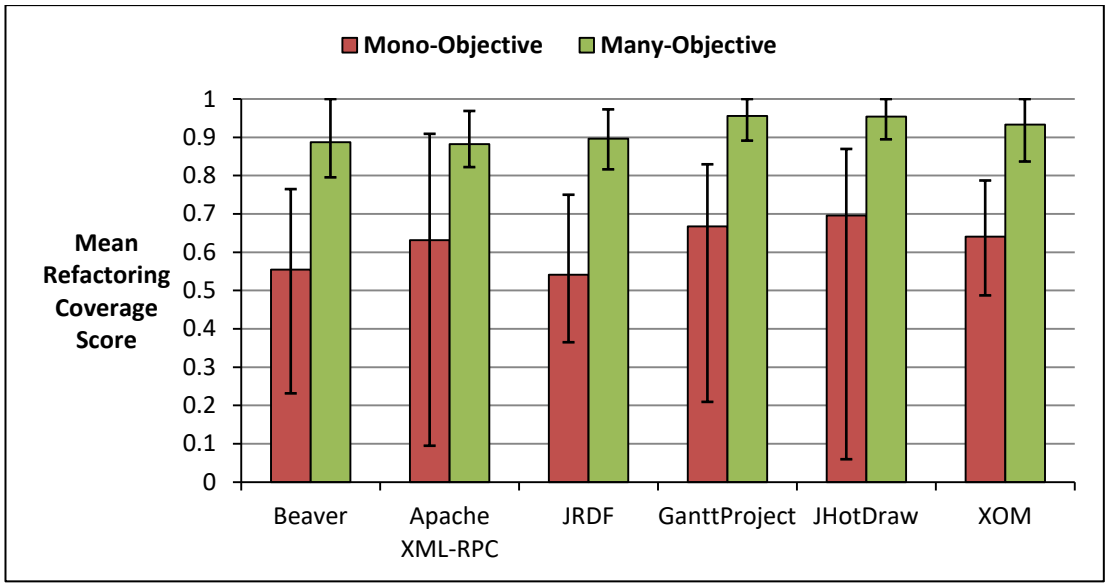

Figure 6.3 – Mean Refactoring Coverage Scores for Each Input

Figure 6.4 shows the average element recentness scores for each input with the mono-objective and many-objective approaches. For each input program, the scores between each approach were closely tied. For all but 1 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.

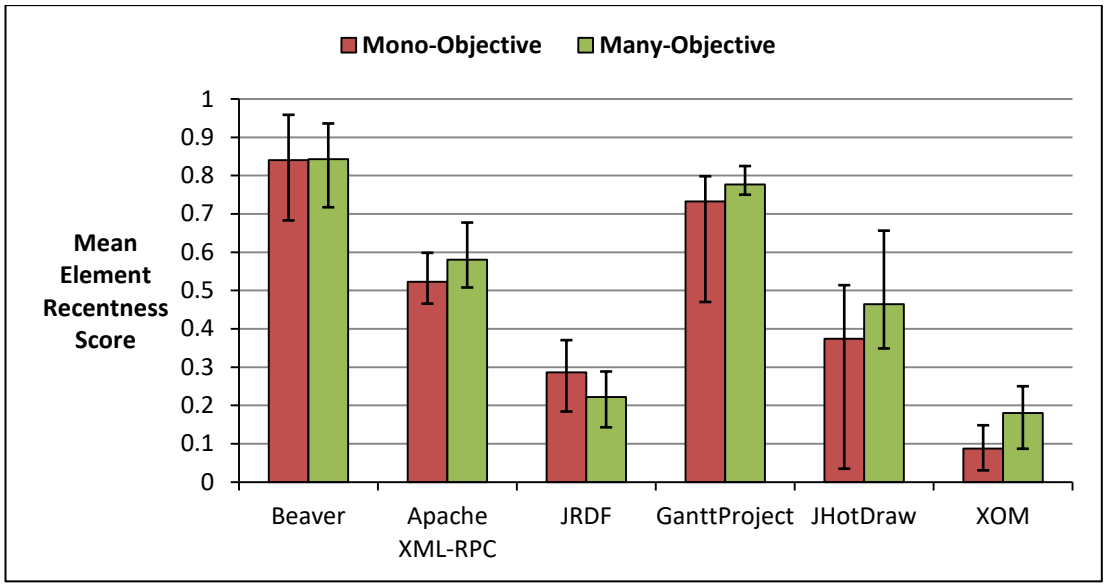

Figure 6.4 – Mean Element Recentness Scores for Each Input

Figure 6.5 gives the average execution times for each input with the mono-objective and many-objective searches. The times for the mono-objective and many-objective tasks are similar, but in all cases, the many-objective approach was faster on average. As observed before, the times generally increased as the number of classes in the project increased, with the trend here being mirrored by the times in the element recentness experiment. 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. In comparison with the times taken during in the element recentness experiment, the times here, especially with the many-objective runs, seem to show an improvement. The many-objective times for the GanttProject input in particular were a lot shorter, with the average time being cut in half. Perhaps the many-objective search is more efficient than the MOGA.

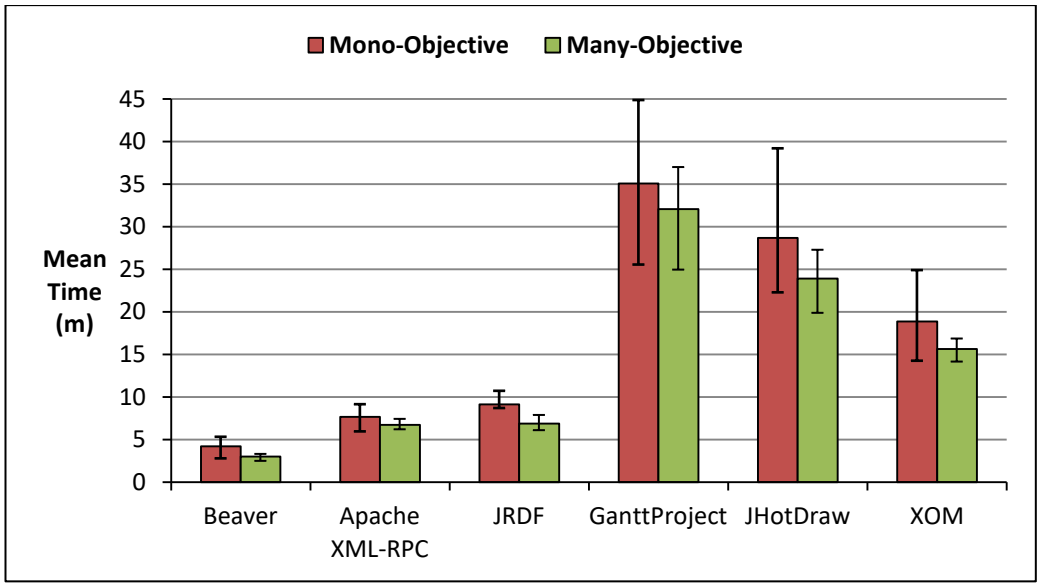

Figure 6.5 – Mean Times Taken for Each Input

Figure 6.6 gives the average quality gain values for each input program used across every approach. All of the scores across every input gave an improvement in the quality objective. For all but 1 of the inputs, the mono-objective approach gives a better quality improvement than the multi/many-objective approaches. With the GanttProject input, the Q-C permutation gave a higher average score than the mono-objective approach. This can likely be attributed to the run that generated a score of 0.422141. None of the other tasks in any of the permutations yielded that high of a score 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. Of the scores, the ones found with the Beaver input seemed to have a larger range than the other projects for all of the approaches.

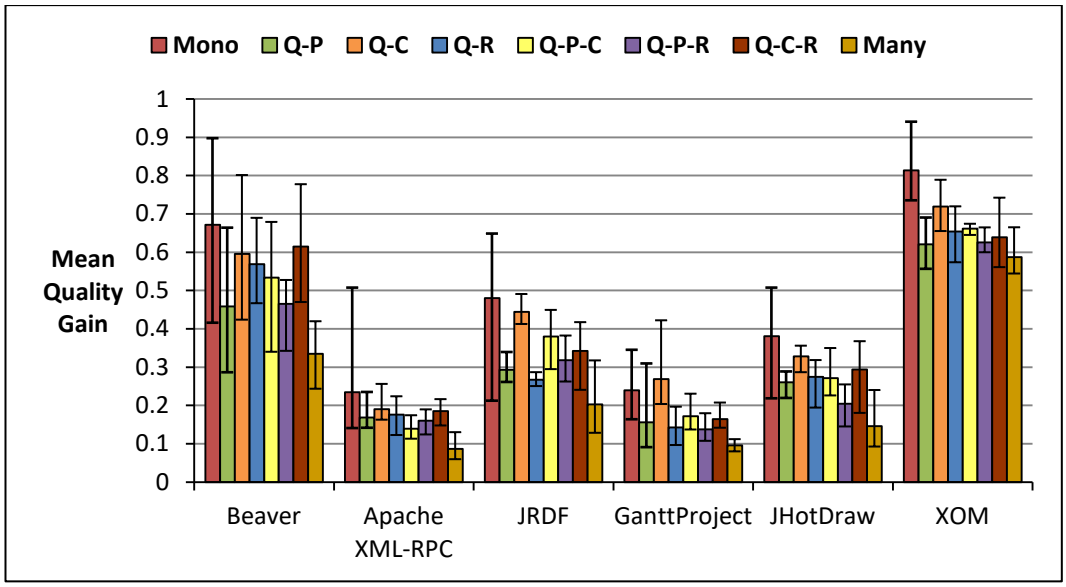

Figure 6.6 – Mean Quality Gain Values for Each Input Across Each Genetic Algorithm Approach

Figure 6.7 gives the averages of the quality gain values from Figure 6.6 across all the inputs. The error bars give the highest and lowest of the average values in Figure 6.6. As observed, the many-objective approach gave the smallest improvements. The Q-C permutation gave the closest score to the mono-objective approach. The Q-R, Q-P-C and Q-C-R permutations all generated similar scores. The 2 objective solutions that used the refactoring coverage objective along with 1 of either priority or element recentness gave better scores than those that used priority and element recentness together.

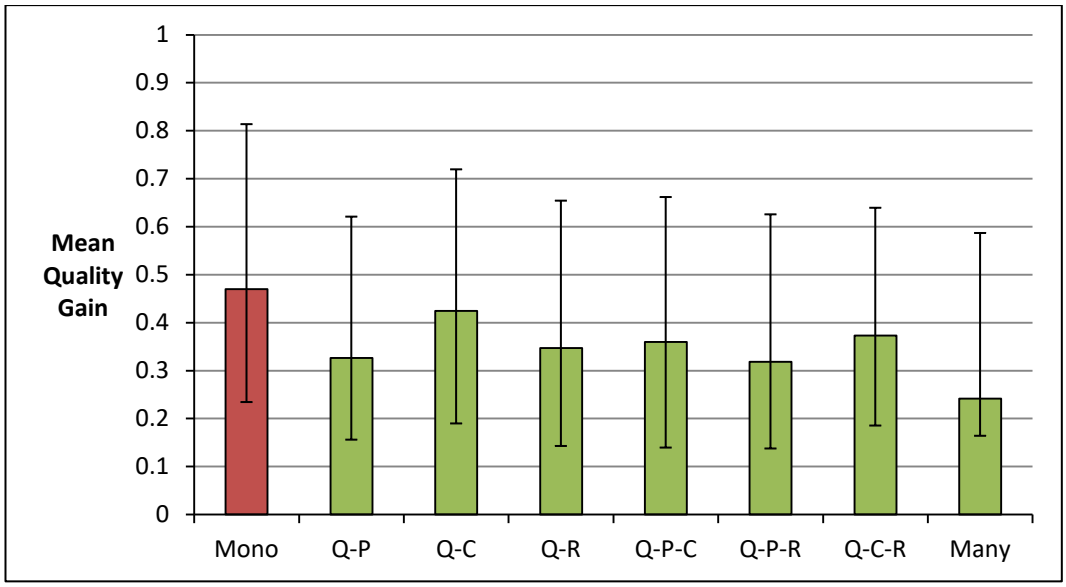

Figure 6.7 – Mean Quality Gain Values Across Each Genetic Algorithm Approach

Figure 6.8 shows the average priority scores for each input program used across every relevant approach. For 2 of the JRDF scores, where they are difficult to see, data labels have been provided. Although the many-objective approach was not able to give better priority scores in all cases, each of the 3 multi-objective permutations to use 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 1 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 Q-P permutation of the multi-objective approach, with the XOM input, whereas the smallest was for JRDF with the many-objective approach.

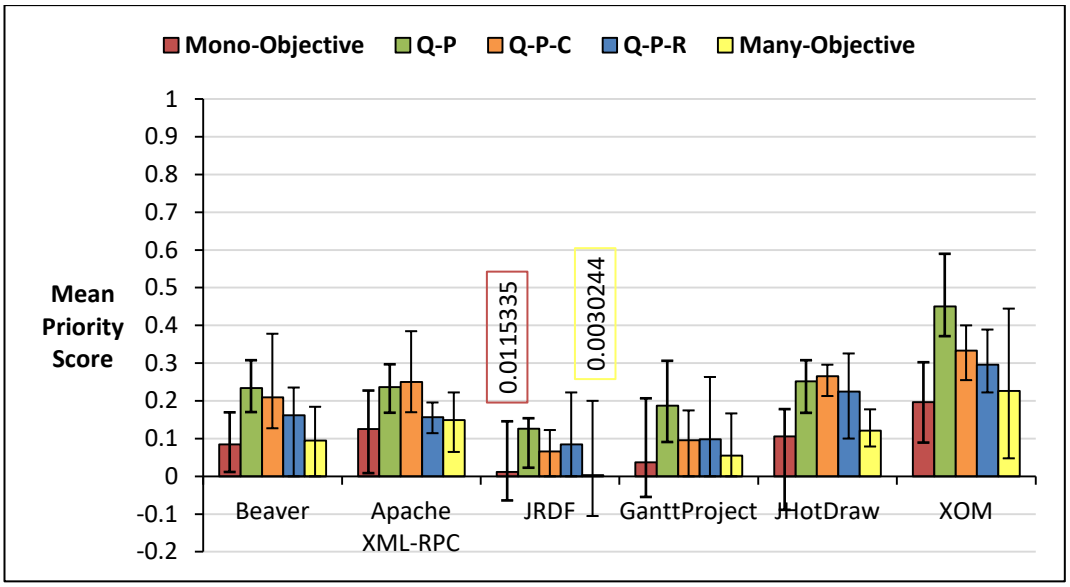

Figure 6.8 – Mean Priority Scores for Each Input Across Each Relevant Genetic Algorithm Approach

Figure 6.9 gives the averages of the priority scores from Figure 6.8 across all the inputs. The error bars give the highest and lowest of the average values in Figure 6.8. The mono-objective score was the lowest, while the Q-P permutation gave the highest score. The 2 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.

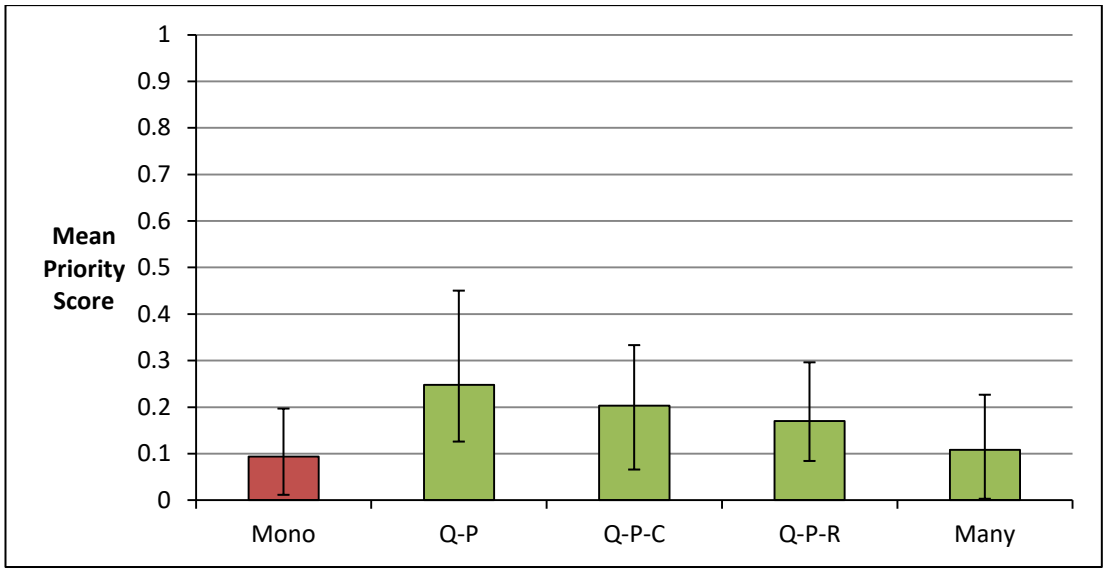

Figure 6.9 – Mean Priority Scores Across Each Relevant Genetic Algorithm Approach

Figure 6.10 shows the average coverage scores 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 Q-P-C 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 0.059364, whereas the lowest coverage score among any of the other approaches was 0.647059.

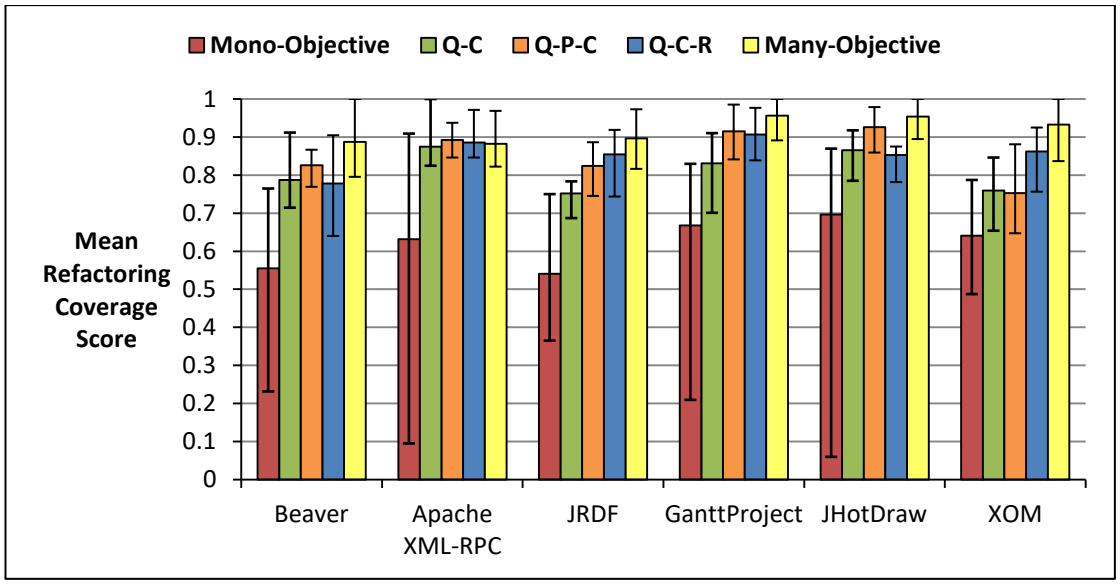

Figure 6.10 – Mean Refactoring Coverage Scores for Each Input Across Each Relevant Genetic Algorithm Approach

Figure 6.11 gives the averages of the quality gain values from Figure 6.10 across all the inputs. The error bars give the highest and lowest of the average values in Figure 6.10. 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, giving better scores. Likewise, the objective works successfully when it is part of a many-objective approach with both priority and element recentness.

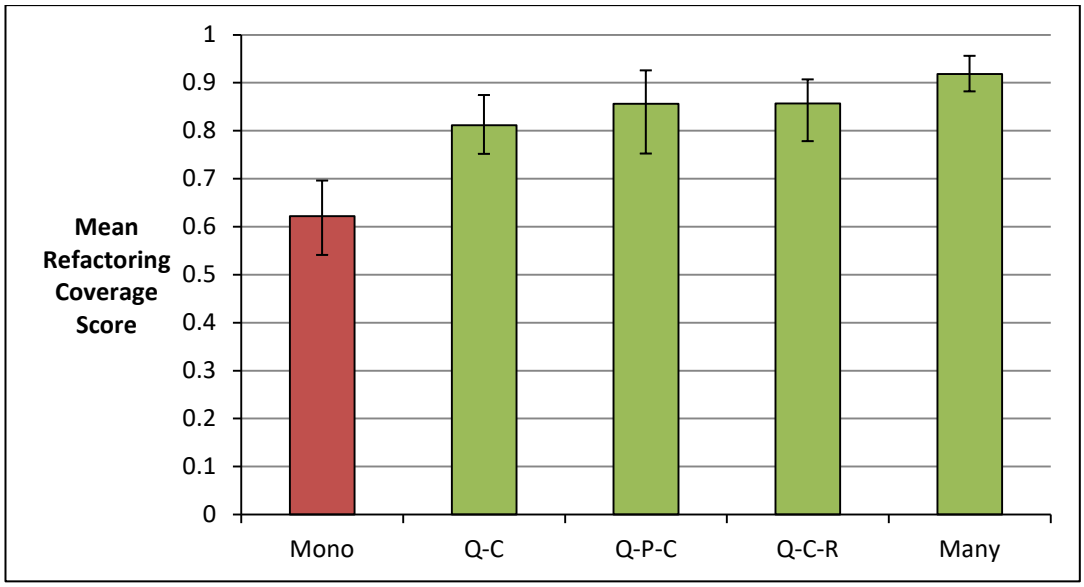

Figure 6.11 – Mean Refactoring Coverage Scores Across Each Relevant Genetic Algorithm Approach

Figure 6.12 shows the average element recentness scores 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 Q-C-R 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.

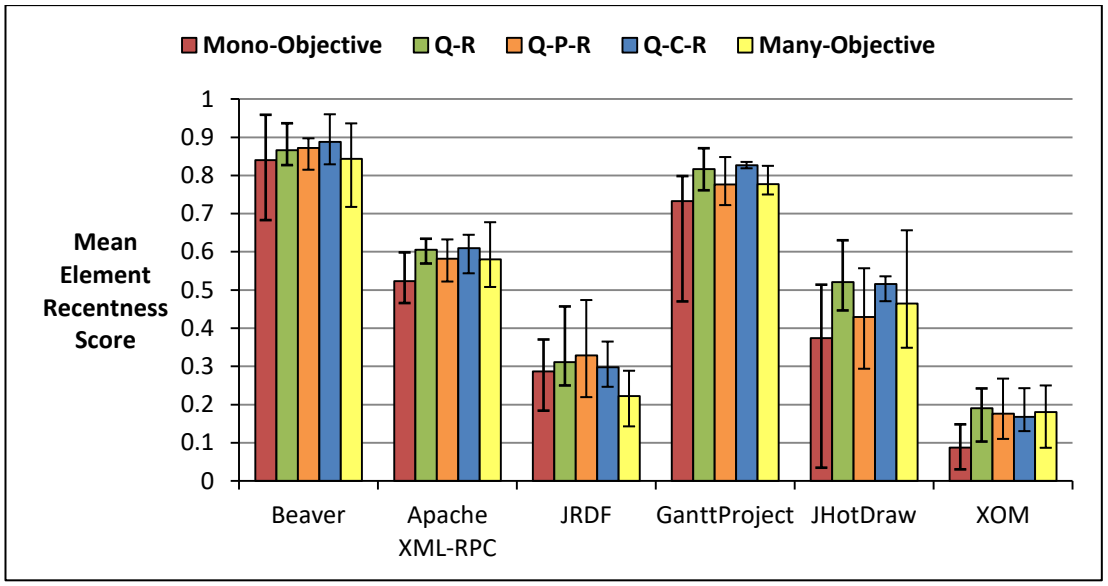

Figure 6.12 – Mean Element Recentness Scores for Each Input Across Each Relevant Genetic Algorithm Approach

Figure 6.13 gives the averages of the quality gain values from Figure 6.12 across all the inputs. The error bars give the highest and lowest of the average values in Figure 6.12. The mono-objective approach gave the lowest score, while the Q-R permutation 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 0.1682482 to 0.8878646, 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 2 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.

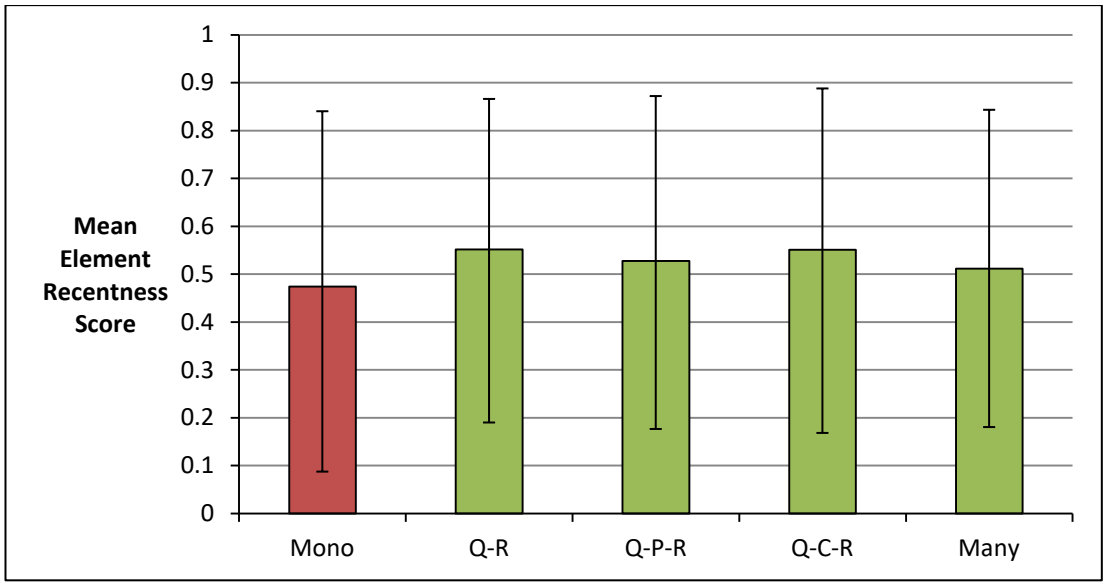

Figure 6.13 – Mean Element Recentness Scores Across Each Relevant Genetic Algorithm Approach

Figure 6.14 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. There were 3 exceptions. The Q-C permutation took longer with the GanttProject input and the Q-P permutation took longer with JHotDraw and XOM. As discussed when inspecting the mono and many-objective times, the tasks took longer depending on how many classes were present in the input project. The longest average time was the aforementioned time given with the Q-C permutation using GanttProject. The shortest time was with the Q-P-R permutation with Beaver.

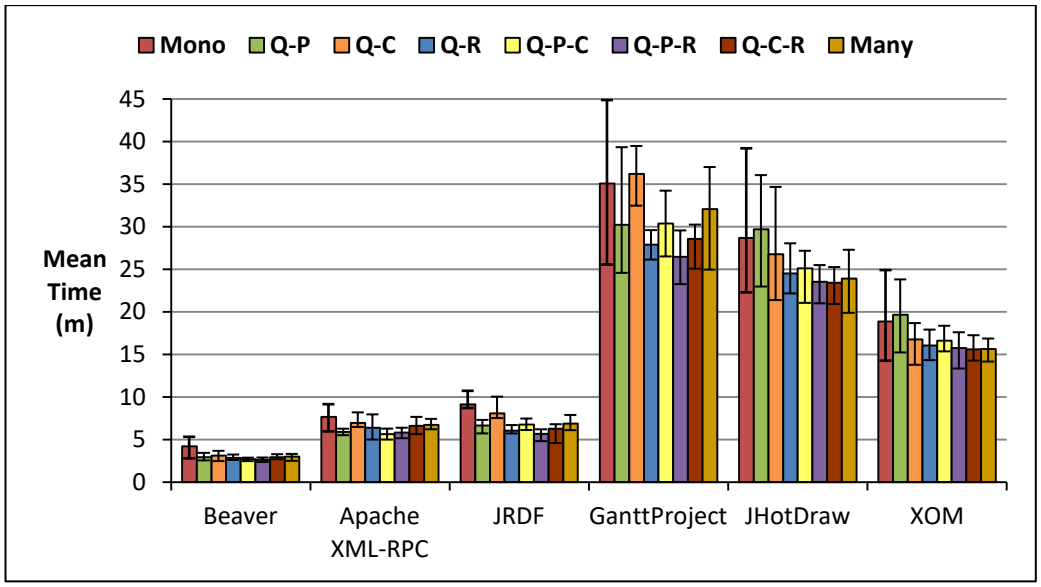

Figure 6.14 – Mean Times Taken for Each Input Across Each Genetic Algorithm Approach

Figure 6.15 gives the averages of the execution times from Figure 6.14 across all the inputs. The error bars give the highest and lowest of the average values in Figure 6.14. The mono-objective approach took the longest while the shortest was the Q-P-R permutation. All of the multi/many-objective approaches gave similar times, ranging from 13 minutes and 19 seconds to 16 minutes and 18 seconds.

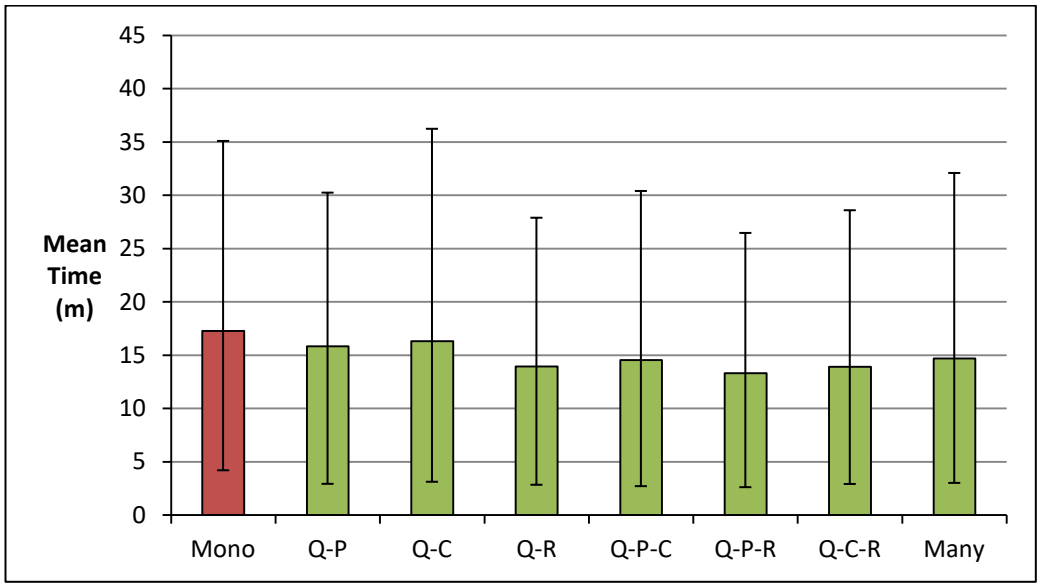

Figure 6.15 – Mean Times Taken Across Each Genetic Algorithm Approach

6.4 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.

Therefore, 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.

6.5 Threats to Validity

6.5.1 Internal Validity

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 6 different open source programs and running against each program 5 times for experiment 2 and 10 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.

6.5.2 External Validity

In this study, the experimentation was performed on 6 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 open source Java programs. Therefore, it cannot be asserted that the results can be generalised to other applications or to other programming languages.

6.5.3 Construct Validity

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.

6.5.4 Conclusion Validity

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, 3 research questions have been provided and for the first 2 (relating to experiment 1), a set of corresponding hypotheses have been constructed in order to aid in drawing a conclusion. For **RQ6.3**, 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.

6.6 Conclusion

In this chapter an experiment was conducted to test the 4 objectives previously constructed with the MultiRefactor tool. They were combined together in a many-objective setup to improve the state of a set of open source Java programs. To conduct this search, an adaptation of the many-objective GA, NSGA-III, was used. The GA used the same configuration parameters as the NSGA-II adaptation used in previous experimentation. 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. To measure success in the many-objective approach, a mono-objective search was conducted using the quality objective, with each of the 3 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 3 supplementary objectives, in conjunction with the quality objective. Along with the previous tasks in the mono-objective and many-objective approaches, 6 multi-objective setups were tested to use each permutation of the 3 supplementary objectives, along with the quality objective. Due to the different input projects used over the course of the experiments in the previous chapters and the normalisations applied to the supplementary objectives in this chapter, each bi-objective setup was tested again. The scores were compared across all 8 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 6 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, 5 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 7 multi/many-objective approaches against the mono-objective approach, 1 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 2 objectives, refactoring coverage and element recentness, were compared, the multi-objective scores were higher than their mono-objective counterparts. When the execution times were compared, all 7 of the multi and many-objective approaches gave faster times than the mono-objective approach being compared against. There was only 1 case where the average execution time was shorter for the mono-objective approach, and that was with the same approach and input that generated the better quality improvement score, the GanttProject program with the refactoring coverage and quality objectives.

In order to test the aims of the experimentation and derive conclusions from the results a set of research questions were constructed. **RQ6.1** and **RQ6.2** were proposed to address experiment 1 and each had corresponding hypotheses. **RQ6.1** 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 6 multi-objective approaches), there was an improvement in the quality objective score, therefore rejecting the alternative hypothesis **H6.1**$_A$ and implying support **H6.1**.

RQ6.2 looked at how effective the other 3 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 2 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 2 objectives, the alternative hypothesis **H6.2**$_A$ 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 **RQ6.3** and derive the most successful combination of objectives to use for each of the 3 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. The next chapter concludes the research by investigating the contributions and outcomes, and outlining limitations and possible areas for future work.

Chapter 7

Conclusions & Future Work

7.1 Summary

Chapter 1 outlined the research area of SBSE, with focus given to the area examined within the thesis, software maintenance. The methodology of the research conducted throughout the thesis was outlined by cataloguing the research questions used to contextualise the scope of the research, as well as the outcomes of the research itself. Chapter 2 examined in detail the different search algorithms used in the experimentation. First, random search was discussed along with HC and SA. Then, the basic GA was described, and SIAs were investigated. Then multi-objective and many-objective EAs were explained, in particular the NSGA-II and NSGA-III algorithms. Chapter 2 also analysed and summarised the literature relating to SBSM. Trends in the literature were isolated and discussed, along with gaps in the research area. Other, more general SBSE papers were also discussed. Chapter 3 detailed the refactoring tool used to conduct the research experimentation, MultiRefactor. The framework of the tool was described and its capabilities were discussed along with the search techniques, refactorings and metrics available to use. Preliminary examinations tested an existing refactoring tool by comparing metaheuristic search approaches and developing and testing a configuration to measure and improve technical debt in software programs.

The experimentation began with Chapter 4. The metrics in the tool were tested to derive the most relevant options to use. They were tested in isolation with the

GA. The configuration settings of the GA were also tested in order to find, through trial and error, the most successful parameter configurations to use. Once the metrics were tested to indicate the ones that are more useful with the setup of the tool, they were used to compare the MOGA with the mono-objective GA. Using the conclusions gained from this experimentation, Chapter 5 experimented with new objectives to use in a multi-objective approach. First, it inspected and tested a priority objective, which used as input a list of classes to focus the search on as well as a list of classes to disfavour. Then, it detailed a refactoring coverage objective, to measure the amount of code coverage of the refactorings applied, and tested that too. Finally, software version history was used to gather information about how recently the code elements that had been refactored were added to the software. This was used to construct a recentness objective which was then tested in the same way as the others.

In order to verify the efficacy of these newly constructed objectives, they were used with the MOGA in a bi-objective setup. The target objective was combined with a quality objective constructed from the experimentation in Chapter 4. This was then compared with a mono-objective search using just the quality objective and then finding a measure of the secondary objective for the top solution. A heuristic was used to define the top solution in the multi-objective search and they were compared using the scores for the 2 objectives and the execution times. Chapter 6 concluded the research by testing all of the constructed objectives together in many-objective approach. The approach was tested to gauge its validity and other permutations were also tested and compared to derive how well the objectives work together in a refactoring approach.

7.2 Experimentation

The experimentation began with Chapter 4. This chapter inspected the capabilities of the MultiRefactor tool and informed the decisions on how to go forward with the options of the tool to further test new objectives. The experimentation in Chapter 4 is split into 4 parts. First, the configuration options of the available GA are inspected to deduce the choice of settings to use in the later experimentation. To test these configurations, a set of tasks are set

up with the JHotDraw input (at the time, this was the largest program available among the set of JSON, Mango, Beaver, Apache XML-RPC and JHotDraw used) and using the *Visibility Ratio* metric. For these example tasks, the crossover and mutation probabilities parameters were tested by having different values in each of the tasks. Nine different permutations were used to test crossover/mutation values of 0.3, 0.5 and 0.8. Among the tasks run, the permutation with a crossover value of 0.2 and a mutation value of 0.8 was most successful and took a relatively small amount of time in comparison with the other options.

The second experiment then used the same setup, with crossover and mutation values of 0.2 and 0.8 respectively, to test the other configuration options. The number of generations, initial refactoring range and population size were tested using trial and error. There were 27 different tasks set up, using different permutations of the 3 parameters. Generation numbers of 50, 100 and 200 were tested along with population sizes of 10, 50 and 100. The refactoring ranges tested were the same as the generations. The metric improvement values were compared against the time taken for each of the options, and the permutation found to have the best trade-off was with 100 generations, a refactoring range of 50 and a population size of 50.

With the configuration parameters tested, experiment 3 then tested the available metrics in the tool. Tasks were set up to test each of the 23 metrics individually. This time, they were run for each of the 5 input programs 5 times for every metric, in order to give a more well-rounded review of the performance of the metrics. The metrics were ranked according to their average performance across the input programs, with the ranking able to answer **RQ4.1** in finding the most volatile metrics used with the GA. In the final experiment, these ranks were used to split the top 18 metrics into 3 separate groups to act as fitness functions. Now that the configuration parameters had been tested for the GA, and the metrics had been inspected to derive the ones that are more useful within the scope of the refactoring tool, experiment 4 tested the effectiveness of the MOGA against the GA.

The GA was tested with 3 different objectives, 1 for each group of metrics derived. For each objective, there were 30 tasks run, 6 tasks for each of the 5 inputs. The MOGA was then set up with the 3 objectives combined into a multi-

objective approach. This, likewise, was run 6 times for each input. The average objective values were compared, along with the time taken to run each approach. To answer **RQ4.2** and deduce whether the MOGA gave comparable results to the GA runs, statistical tests were used to compare the metric function improvement values. For 2 of the 3 objectives, the mono-objective approach gave better improvement values, but they were not significantly better. The times taken to run the single MOGA run were less than the times taken for each of the GA runs for the single objectives. As a result of the experimentation in Chapter 4, an overall quality objective was derived using the set of metrics that made up the 3 objectives in the final experiment. This, along with the configuration parameters derived from the earlier trial and error experiments, was used in the experimentation in the next chapters.

The 3 secondary objectives were tested in Chapter 5. The priority objective took as input a list of classes to favour in the refactoring solution and optionally, a list of classes to disfavour. For this experiment, the smallest input program, JSON, was removed and 2 other programs (GanttProject and XOM) were added to test. Using the quality objective made up of the metrics derived from the previous experimentation in Chapter 4, the mono-objective GA was run 5 times for each of the 6 programs. The search was modified to also output, in the final population of results, the priority objective for those solutions. The MOGA was then run, using both the quality objective and the newly proposed priority objective. In order to choose classes to specify for each input, the number of methods in each class was used. The top 5% of classes for the project with the highest number of methods were used as the priority classes, and the bottom 5% were used as the non-priority classes. After the GA and MOGA runs were completed, using the configuration parameters derived from the previous experimentation, the approaches were then compared.

To answer **RQ5.1**, the multi-objective quality scores were inspected to see whether an improvement in quality can be given. For all the input programs, the quality objective yielded improvements and, for the Mango input, the improvement score was higher than with the mono-objective approach. **RQ5.2** addressed the priority scores in the 2 approaches. The scores were found to be significantly better for the multi-objective approach against the mono-objective approach, using the Wilcoxon rank-sum statistical test. The average times for

each input were then compared. Again, the Wilcoxon rank-sum test was used to derive that the times were not significantly different for each approach and, for 5 of the 6 inputs, the multi-objective approach took less time on average.

The refactoring coverage objective was tested next. Unlike the priority objective, it didn't take any external inputs to function. It captured information about the refactored code elements in a solution to gain a measure of how often the specific code elements were refactored. Like before, each approach was run 5 times for each of the 6 input programs, in order to compare them. This time the multi-objective approach used the quality objective and the new refactoring coverage objective. The same configuration parameters were used for both approaches as before. The multi-objective quality scores were inspected to answer **RQ5.3**, and for all the input programs, the quality objective gave improvements. **RQ5.4** addressed the coverage scores in the 2 approaches. The scores were found to be significantly better for the multi-objective approach against the mono-objective approach, using the Wilcoxon rank-sum statistical test. The average times for each input were compared, and for 5 of the 6 inputs, the mono-objective approach took less time on average. The Wilcoxon rank-sum test was used to confirm that the mono-objective times were not significantly different.

The element recentness objective was proposed as well and was tested last. This objective, like the priority objective before, depended on external input of information into the tool in order to construct the objective measurement. It took as input a set of previous versions of the target software to read in. Also like with the priority experiment, the inputs that were tested were changed. Mango was replaced with JRDF as it didn't have multiple versions to use. Also, the Apache XML-RPC and JHotDraw inputs used different versions of the code, in order to use the other versions as part of the set of previous inputs to inform the element recentness objective. Each approach was run 5 times for each of the 6 input programs, with the same configuration parameters as before. This time the multi-objective approach used the quality objective with the element recentness objective. The multi-objective quality scores were inspected to answer **RQ5.5**, and for all of the input programs, the quality objective gave improvements. **RQ5.6** addressed the coverage scores in the 2 approaches. The scores were found to be significantly better for the multi-objective approach against the mono-objective approach, using the Wilcoxon rank-sum statistical

test. The average times for each input were compared, and for 5 of the 6 inputs, the mono-objective approach took less time on average. The Wilcoxon rank-sum test was used to confirm that the mono-objective times were not significantly different.

The many-objective algorithm was used in Chapter 6 to set up a refactoring approach that combines all 4 objectives proposed in the previous chapters. Like the GA and MOGA before it, the many-objective algorithm used the same configuration parameters. The measurement of the 3 secondary objectives was updated in order to normalise their scores between 0 and 1 (or, in the case of the priority objective, -1 and 1). The mono-objective GA tasks were repeated 10 times for each input program, with the values of all 3 secondary objectives given for the top solution in the final population. The many-objective approach was then run 10 times for each input as well, in order to compare the scores. To address **RQ6.1**, the quality scores were inspected for the many-objective approach. For all the inputs, the many-objective approach generated improved quality scores. To answer **RQ6.2**, the many-objective scores for the priority, refactoring coverage and element recentness objectives were compared against the corresponding mono-objective scores. For the refactoring coverage objective, the many-objective approach yielded better scores for all inputs. However, for both the priority and element recentness objectives, there was 1 input (JRDF) where the mono-objective approach gave a better average score. The many-objective approach took less time for all of the inputs.

To address the behaviour observed for the JRDF input with the priority and element recentness objectives, different permutations of the objectives were tested together will the multi-objective algorithm to analyse whether any of the pairs of objectives were incompatible. In order to address **RQ6.3** and find out which combination of objectives worked best together, there were 6 different permutations of the objectives to test, as well as the mono-objective and many-objective approaches tested already. Each permutation used the quality objective along with 1 or 2 of the other objectives, and was run 5 times for each of the 6 inputs. It was found that the priority and element recentness objectives were less successful when they were used together, which may address why they were less useful when compared with the refactoring coverage objective in the many-objective approach. For both of these objectives, they gave better scores

when used in a bi-objective setup along with the quality objective. On the other hand, the refactoring coverage objective was most successful with the many-objective approach in comparison to the other options, and was less successful when it wasn't used along with the other secondary objectives. For all of the multi-objective approaches, they took less time to run than the mono-objective approach on average.

7.3 Outcomes

Five research questions were laid out in Chapter 1 that drove the research within the thesis. Each question is outlined below along with the outcomes of the research conducted to address them.

RQ1: What current refactoring and search-based software engineering tools are available?

In Chapter 2, the tools proposed in the SBSM literature are discussed, particularly in Section 2.8.7. Appendix B goes into more details about the SBSE tools available by listing them and discussing each one. Appendix C also details other useful tools for the research area. Tables are given to list open source refactoring tools, commercial refactoring tools, open source search-based optimisation tools and open source metrics tools. The A-CMA tool was tested and used in Chapter 3, but wasn't sufficient to experiment with multi-objective and many-objective refactoring techniques.

RQ2: Can a fully automated, practical refactoring tool be developed using techniques from previous literature to improve the maintenance of software?

Chapter 3 outlines the details of the MultiRefactor tool developed to refactor software and improve the maintenance process of software development for Java programs. The tool has been developed to include as many options for refactoring as possible. It is highly configurable and contains 26 different refactorings (with the *Extract Subclass* refactoring being added during the course of the experimentation), 23 metrics and 6 different search techniques. The tool is fully automated and produces information about the search

conducted as well as refactored, fully compilable Java source code. It can be used for both research purposes (as it has within this thesis) or for practical purposes to maintain Java software. Tasks can be set up to run multiple searches in 1 go, so if you want to test different configurations or refactor multiple different Java programs, the tool can be set up to do so without needing to continually rerun the program.

RQ3: How useful is a multi-objective search-based software maintenance approach in comparison with a mono-objective search-based approach?

In Chapter 4, the MultiRefactor tool is used to test and compare the multi-objective and mono-objective optimisation approaches against each other. The multi-objective approach gives comparable results for the objectives tested to each of the mono-objective runs. It is also able to complete the 3-objective approach in less time than any of the mono-objective approaches. Further experimentation in Chapter 5 also compares results from multi-objective runs with the mono-objective GA. The disadvantages are the following. Sometimes, the multi-objective approach can take longer to run than the mono-objective approach, although never significantly longer. Likewise, although the multi-objective approach will give improvements in the objectives, the mono-objective approach will generally give better results for the single objective that it focuses on. For instance, in Chapter 5 when the quality objective is used in both approaches, the mono-objective approach yields better improvements for that single objective while the multi-objective approach works to improve that objective as well as another. Therefore, if there is a single objective that needs to be focused on, it seems the multi-objective approach is no substitute for the simple GA in improving that objective on its own. On the other hand, if there are multiple properties to keep in mind in a refactoring solution, the MOGA will be able to generate suitable improvements for all of them.

RQ4: Can individual, novel objectives be measured and refactored in a software program to maintain the code while also improving the individual properties inspected?

Chapter 5 addresses this research question by proposing and testing 3 new novel objectives in the MultiRefactor tool. The new objectives look at different, more non-functional properties based around the applied refactorings themselves (priority of classes refactored, code coverage of refactorings and

recentness of code elements refactored). These objectives are each tested individually by using them in a multi-objective search along with the quality objective tested in Chapter 4. The experiments conducted confirm that, for each of these objectives, the objective can be used to improve the inspected property while also maintaining the code to improve quality.

RQ5: Can numerous individual objectives be combined into a fully automated, many-objective approach in order to improve a software program across multiple different properties in an additive fashion, without losing the improvement effect of any individual property?

In Chapter 6, a many-objective approach is set up to combine the 4 objectives tested previously (quality, priority, refactoring coverage and element recentness) into an overall framework. The approach is mostly successful in generating refactoring solutions to improve all 4 objectives. However, the priority and element recentness objectives do not seem to be as successful when combined together. Further experimentation compares different permutations of the objectives to see what the best combinations are. This experimentation confirms that the priority and element recentness objectives are less successful when combined together, although the refactoring coverage objective gives better results the more objectives it is combined with. Therefore, although the many-objective approach is effective in improving the 4 objectives, the priority and element recentness objectives can yield better results when kept separate from each other.

7.4 Comparison With Previous Literature

Twelve other approaches with associated tools for software maintenance can be found in the literature. J/Art [102] detects design smells in the code and JDeodorant [150] and TrueRefactor [153] also detect certain design smells and remove them. Evolution Doctor [149] and the Advanced Refactoring Wizard [148] work in a similar manner by finding certain types of issue in the structure of the code and reorganising or refactoring the program to remove them. Wrangler [151] applies elemental structural refactorings to Erlang programs to

resolve different types of modularity smells. Other tools provide modifications to a software system in order to improve it, without finding specific defects to resolve. Bunch [88] applies module clustering to a software system by looking at the different modules and dependencies in the system. FermaT applies low-level WSL-to-WSL transformations to reduce the size of programs. The A-CMA [117] tool uses refactorings to improve the bytecode of a software program, according to various software metrics. Similarly CODe-Imp [152], which was built from the prototype tool Dearthóir [112], uses refactorings to improve the structure of the software with the help of different software metrics. Before the Dearthóir prototype was developed by O'Keeffe and Ó Cinnéide, Ó Cinnéide and Nixon had developed a similar tool, DPT [107], to apply design patterns to code by modifying the structure using different refactorings.

MultiRefactor was developed to address some of the weaknesses present among these maintenance tools in order to be used for research and also to improve the maintenance of actual software programs. Many of the tools modify some artefact of the software in place of the code itself, and so the actual refactorings to apply to improve the code still need to be done manually. Bunch generates clustering solutions using module dependency graphs. J/Art only provides limited refactoring suggestions for the design smells detected, and likewise, Wrangler gives refactoring suggestions for the Erlang code. FermaT, although it applies transformations to the code during the search, generates lists of transformation sequences to be applied as the output. TrueRefactor generates UML diagrams as an output and Dearthóir also applies transformations to a design of the program. JDeodorant resolves the detected design smells in the code itself, but to do so they need to be manually selected through a plugin. Similarly, DPT is used by manually selecting the design patterns to be applied to the code. A-CMA does apply the actual refactorings, but they are applied to Java bytecode instead of to the source code. This leaves 3 tools. Evolution Doctor reorganises the source files of a program library to resolve issues with their organisation. The Advanced Refactoring Wizard and CODe-Imp are the 2 tools that actually produce refactored code as the output of the process.

With MultiRefactor, Java code can be read in (as long as it compiles) and at the end of the refactoring process, Java code will be given as an output, allowing the process to be fully automated and eliminating the need to apply any further

changes to the code afterwards. It is also highly configurable, giving the user the ability to use different combinations of refactorings and metrics to improve the structure of the software depending on their specific needs. The tool has 25 refactorings and 23 metrics to use as well as a number of different search options. For comparison, Wrangler and JDeodorant can each resolve 4 different design smells and TrueRefactor can resolve 5. J/Art can detect 15 types of design smell and the Advanced Refactoring Wizard can detect 19, but can only resolve 4 of them. Evolution Doctor looks at 4 different factors to resolve. Dearthóir has 8 different refactorings and 5 metrics, while FermaT has 20 different low-level refactorings. A-CMA also has 20 different refactorings but also has 24 metrics available. Finally, CODe-Imp has 14 refactorings and 24 metrics (although 20 of these are related to either cohesion or coupling). This means MultiRefactor has more available refactorings than any of the other tools and has only 1 less metric available than the tools containing the top number of metrics, A-CMA and CODe-Imp.

The searches available include metaheuristic search techniques as well as GAs. There are adaptations of the multi-objective NSGA-II algorithm and the many-objective NSGA-III algorithm, meaning that this tool can be used to apply multi-objective and many-objective approaches. This allows the user to select various different properties that they want to improve in the software structure and allow the tool to generate code that satisfies those aims. Although the CODe-Imp tool has been outfitted to include a basic GA, none of the tools have the capability of using multi-objective or many-objective techniques to refactor the software.

A number of studies have used some form of a quality measure to aid with improving the software. O'Keeffe and Ó Cinnéide [115], [116] used the understandability function of the QMOOD suite to measure quality when comparing different search algorithms and input parameters for refactoring. Koc et al. [117] implemented a measure of quality that incorporates a normalised sum of 17 of the 24 metrics available in the A-CMA tool to give an overall normalised metric score. Likewise, Seng et al. [121] used a weighted sum of 7 normalised metrics to create a fitness function to capture coupling, cohesion, complexity and stability in the software. In numerous studies, Ouni et al.[139], [140] and Mkaouer et al. [144], [145] defined a measure of quality as one

objective in a multi-objective approach. In each of the studies, quality was defined by measuring the number of code smells resolved in a software system, as a ratio of the number of smells corrected over the number detected.

The quality function that was developed from the metric experimentation in Chapter 4 and defined in Table 5.1 is made up of a normalised sum of the available metrics in the tool, similar to how Koc et al. and Seng et al. developed their quality measures. The quality measure used by O'Keeffe and Ó Cinnéide was less a measure of general quality and more a measure of a specific property of the software (this was one of 6 properties – reusability, flexibility, understandability, functionality, extendibility and effectiveness – defined by Bansiya and Davis with their QMOOD suite [5]). For Ouni et al. and Mkaouer et al., the quality was defined by the number of code smells corrected in a system, so its generality is restricted by the number of code smells supported, and the ability to find code smells in the solution. On the other hand, Seng et al. selected a set of metrics to combine coupling, cohesion, complexity and stability into an overall quality measure. Likewise, Koc et al. combined most of the metrics they had available in their tool, A-CMA, to create an overall measure of quality, although how these metrics were selected is unclear. The difference between those 2 approaches and the quality objective used in this thesis for experimentation is that the metrics used in the objective were selected based on preliminary experimentation. Eighteen of the 23 metrics available in the tool were used in the quality function based on the volatility of each of the individual metrics, removing the 5 least effective metrics from the measure and using the metrics that worked better with the MultiRefactor tool at refactoring a range of different software systems.

One other study proposed a priority measure to use in a fitness function for software refactoring. Ouni et al. [147] used a priority measure as one of 4 measures in a multi-objective approach using CRO. For this measure, they ranked the 7 code smells they were aiming to resolve, to give some more importance than others. This is completely different to the priority objective proposed in Chapter 5 of this thesis, which instead gives some of the classes in the software more importance, influencing which classes the refactorings are applied in. Also, the priority objective gives the user of the tool the control to indicate which classes are more important depending on the program being

refactored, to further contrast with the priority measure of Ouni et al. where they provide the ranking the code smells based on their own experience. The objective from Chapter 5 also gives the option to indicate classes that should be avoided in the search.

Although the refactoring coverage objective was entirely novel and unlike other objectives used for SBSM studies, a few other studies relating to SBSM have used version history of the target software to aid in refactoring. Pérez *et al.* [143] proposed an approach that involved reusing complex refactorings that had previously been used. Ouni *et al.* [139], [140] used previous versions of code to find a set of refactorings applied in those previous versions. They also [141] analysed co-change, an attribute that identifies how often 2 objects in a project are refactored together at the same time, as well as the number of refactorings applied in the past to the code elements. An extended study [37] investigated the use of past refactorings from other projects to calculate an objective value when the change history for the applicable project is not available.

These studies investigating software history to aid with maintenance are concerned with the refactorings applied in the past. The difference with these studies and the element recentness objective proposed in Chapter 5 is that the element recentness objective investigates the presence of the code elements that have been refactored in the current solution and not the refactorings that have been applied in the past. The MultiRefactor tool also implements the element recentness objective as a fully automated solution, whereas the studies of Ouni *et al.* (and likely of Pérez *et al.* also if their approach was implemented) do not actually apply the proposed refactorings as part of the solution. Instead, they return a list of refactorings to be attempted.

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. In 2014 and 2015, Mkaouer et al. experimented with many-objective techniques using NSGA-III (which itself had been introduced in 2013 [49]). 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 [25], [146]. 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 [139] and semantic

similarity [138]) together into an approach to re-modularise software [42]. The many-objective experimentation conducted in this thesis with the MultiRefactor tool similarly combines the 4 objectives developed in the previous experiments. The objectives used, though, are different to the ones used by Mkaouer et al. 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.

7.5 Novel Contributions

The primary contributions of the thesis that result from the research are outlined below:

1. A systematic analysis of current opportunities with SBSM. The various tools currently available are analysed and inspected. The different search-based optimisation techniques are also inspected and the different searches are compared against each other to analyse the advantages and disadvantages of different approaches. The limitations of the current approaches are analysed and are either outlined or addressed.
2. A new tool is developed and proposed for fully automated maintenance of Java software using mono-objective, multi-objective and many-objective search techniques. The tool is equipped with numerous refactorings and metrics and is fully configurable. It is available online for use and can be used for research purposes or as a maintenance tool to assist with the improvement and maintenance of Java software.
3. An objective is proposed and tested to measure quality in a software program. It can be used to maintain the software and improve its quality in respect to various software metrics available in the refactoring tool.
4. An objective has been proposed and tested to measure the priority of the classes refactored in a refactored solution. Using as input a list of classes to favour and, optionally, a list of classes to disfavour, the objective will guide the refactorings in the search with respect to the relevant classes.

5. An objective is proposed and tested to measure the code coverage of the refactoring solutions generated in the refactoring tool. The objective will investigate and measure the amount of code coverage given by the refactorings in a refactoring solution by inspecting the code elements that are refactored.
6. An objective is proposed and tested to measure the recentness of the code elements refactored in a refactoring solution, in relation to a set of previous versions of the code. The previous versions of the code are read in as inputs and used to indicate the age of the code elements refactored by tracing back how far among the versions the elements are present. The more previous versions of the code that are read in, the more accurate the objective measurement will be.
7. The objectives proposed are combined into an overall framework to use with software in conjunction with the many-objective functionality in order to improve the software across various different properties. They have also been normalised along with the quality objective in order to be more suitable to use together. The objectives have been tested together in different permutations to suggest the best combinations to use for overall success.
8. The tasks constructed for all of the experimentation are implemented into the tool for other users to take advantage of and the data gathered from the experimentation in the thesis is also included in the online repository hosting the tool. This allows developers to make use of the tasks constructed in the tool and researchers to inspect and build upon the research conducted.

7.6 Limitations & Future Work

There are various limitations in the scope of the research in the thesis and in the experimentation conducted. The following subsections discuss these limitations and possibilities for future research as well as potential extensions and additions to be made to the MultiRefactor tool.

7.6.1 Future Adoption Steps

The refactoring tool constructed for the research only works with Java programs. Therefore if the software program being maintained has been written in another programming language, it cannot be refactored using the tool. A possible extension for the tool is therefore to be able to read in and refactor programs written in other common programming languages, such as C++. The capability of the refactoring tool is also restricted by the number of refactorings available. There are currently 26 refactorings available, although there are dozens of others that could be added to the tool. Likewise, there could be more metrics added in order to improve the customisation possibilities in the refactoring process. There are many potential search techniques to adapt into the tool and experiment with, beyond those that have been used. Swarm algorithms have been used scarcely in the previous literature, but with promising results. Likewise, other multi-objective and many-objective EAs could be used to improve the efficiency of the search process, such as those listed in Section 2.6 and Section 2.7.

One way to use the artefacts of a software project could be to incorporate other techniques to detect refactorings applied by developers in the industrial code (or use change logs to store changes made. This information could then be used to influence in certain ways the refactorings applied in the automated approach, such as in the approaches used by Pérez *et al.* [143] and Ouni *et al.* [37], [139]–[141]. Another option is to incorporate the use of unit tests with the automated refactoring approach to improve the solutions returned. The unit tests could be run to check whether the refactoring solutions generated break any tests, and this could be used to validate the refactorings applied. If the unit tests can be run automatically, during the search the results could be used to influence the population returned at the end, with the top solutions passing all of the tests. Another possibility is to integrate the MultiRefactor approach used to refactor software with other aspects of maintenance to make the process as efficient and automated as possible. As an example, the GenProg tool [104] is used to automated software repair and error resolutions. Further work could be done to incorporate an automated software repair tool such as GenProg with the MultiRefactor to cover more aspects of the maintenance process.

7.6.2 Future Research Directions

The experiments conducted in the thesis, although they have been applied to different software projects and repeated numerous times, are still limited in the scale of the tasks conducted. In order to support the conclusions drawn from the experimentation, the experiments would need to be repeated on a larger scale and under different conditions. The experimentation is also limited by the example programs used. 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). To support the testing conducted in the research, further experimentation should be conducted with larger programs and programs of different types. It would be useful to gain access to proprietary software to see what information can be gleaned from testing programs that are being actively developed by a company. Further experimentation with industrial code and other aspects of the software project could potentially yield more generalisable results than testing open source programs, and the code would likely be of a more realistic size.

Furthermore, it would be helpful to gain the insight of experienced developers by asking for their opinion of the capabilities of the tool and of the output code produced by the refactoring process. This could be achieved through surveys to provide a qualitative study of the effectiveness of the tool. The opinions gained may help to highlight any practical issues that may be present in using an automated refactoring tool such as MultiRefactor to maintain the code as part of the development process. Such an insight may be valuable in suggesting improvements and additions to make to the tool to help equip the user with options. Moreover, if an experienced developer was to use the tool and experiment with its capabilities, as well as the proposed objectives, they could convey valuable opinions on how helpful these capabilities can be, and which capabilities would prove most useful for them. Case studies could be set up using the tasks and capabilities currently available within the tool allow the developers to experiment with it. Also, if the tool could be used to experiment using the software that the developers use, then the combined insight of developer opinion and an industrial target program could be used to gained 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.

There is also room to investigate and build upon the approaches proposed in other SBSM papers. For example, Amal et al. [160], introduced an artificial neural network to choose between solutions in the final population of the search. This could be used with the MultiRefactor tool to replace the more basic heuristics used with the multi-objective and many-objective tasks to choose solutions. Another study by White et al. [158] investigated a multi-objective approach which balanced a functional objective with a non-functional objective. Further experimentation with the MultiRefactor tool could be conducted to incorporate other non-functional measures of aspects like security and performance.

All of the possibilities for future work with the research discussed in the preceding sections are summarised and listed below:

1. Extension of the MultiRefactor tool to be able to refactor other programming languages beyond Java.
2. Further experimentation with larger programs and programs of different types.
3. Qualitative analysis of the practicality of the MultiRefactor tool by using surveys and case studies to gain the opinions of software developers.
4. Experimentation with proprietary software to yield more generalisable results.
5. 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.
6. Further insight into the applicability of the MultiRefactor approach using the opinions and insights of experienced developers.
7. Further additions to the capabilities of the MultiRefactor tool through the implementation of other refactorings, metrics and secondary objectives.
8. Further experimentation with other types of search techniques such as SIAs, as well as other multi-objective and many-objective EAs.
9. Incorporation of other techniques introduced in previous SBSM research such as the use of artificial neural networks to choose between solutions,

or the use of non-functional measures such as security and performance to influence the search.

10. Incorporation of other techniques used to automate other aspects of the maintenance process such as software repair.
11. Further replication of the experimentation conducted with more input programs and in more realistic conditions.

7.7 Final Comments

This work has built on a growing body of research into how to automatically refactor software to aid in software maintenance. There is still plenty of scope for research in this field and the experimentation and analysis performed within this thesis allows for further advancement in the directions taken within. Furthermore, the automated tool that has been developed for the research gives an opportunity for further analysis of the experimental data and replications of the experimentation carried out. The capabilities of the tool provide the freedom to continue investigating this research area with other options and techniques, and build upon the research conducted.

Technical debt can cause the structure of a software project to be degraded over time, making it necessary to restructure the program before new functionality can be added. This costs the developer time as the overall development time for functionality is offset by this obligatory cleaning up of code. It is estimated that the maintenance process takes 70-75% of development effort [173], [174]. Automated refactoring of the software can make the software easier to maintain, and therefore has the potential to drastically reduce this cost. Using search-based refactoring may also make the process of changing the code itself less tedious and allow the developer to concentrate on what aspects of the software need to be improved instead of how to go about improving them. Therefore, the research conducted within this thesis has value to software developers and software companies with its potential to help reduce the cost of development and optimise the developer's skills.

References

[1] M. Harman and B. F. Jones, "Search-Based Software Engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, Dec. 2001.

[2] Y. Zhang, "CREST SBSE Repository," 2017. [Online]. Available: http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/. [Accessed: 27-Sep-2017].

[3] R. C. Martin, *Agile Software Development, Principles, Patterns, And Practices*. 2003.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] O. Räihä, "An Updated Survey On Search-Based Software Design," 2009.

[9] C. L. Simons and I. C. Parmee, "Single And Multi-Objective Genetic Operators In Object-Oriented Conceptual Software Design," in *8th Annual Conference on Genetic and Evolutionary Computation, GECCO 2006.*, 2006, pp. 1957–1958.

[10] S. Yoo and M. Harman, "Pareto Efficient Multi-Objective Test Case Selection," in *International Symposium On Software Testing And Analysis, ISSTA 2007.*, 2007.

[11] Y. Zhang, M. Harman, and S. A. Mansouri, "The Multi-Objective Next Release Problem," in *9th Annual Conference on Genetic and Evolutionary Computation, GECCO 2007.*, 2007.

[12] C. L. Simons and I. C. Parmee, "A Cross-Disciplinary Technology Transfer For Search-Based Evolutionary Computing: From Engineering Design To Software Engineering Design," *Eng. Optim.*, vol. 39, no. 5, pp. 631–648, 2007.

[13] A. Finkelstein, M. Harman, S. A. Mansouri, J. Ren, and Y. Zhang, "'Fairness Analysis' In Requirements Assignments," in *16th IEEE International Requirements Engineering Conference.*, 2008.

[14] C. L. Simons and I. C. Parmee, "Agent-Based Support For Interactive Search In Conceptual Software Engineering Design," in *Genetic And Evolutionary Computation Conference, GECCO 2008*, 2008, pp. 1785–1786.

[15] Z. Wang, K. Tang, and X. Yao, "A Multi-Objective Approach To Testing Resource Allocation In Modular Software Systems," in *IEEE Congress on Evolutionary Computation, CEC 2008.*, 2008, pp. 1148–1153.

[16] J. J. Durillo, Y. Zhang, E. Alba, and A. J. Nebro, "A Study Of The Multi-Objective Next Release Problem," in *1st International Symposium On Search-Based*

Software Engineering, SSBSE 2009., 2009, pp. 49–58.

[17] C. L. B. Maia, R. A. F. Do Carmo, F. G. De Freitas, G. A. L. De Campos, and J. T. De Souza, "A Multi-Objective Approach For The Regression Test Case Selection Problem," in *XLI Brazilian Symposium of Operational Research, XLI SBPO 2009.*, 2009, pp. 1824–1835.

[18] C. L. B. Maia, F. G. De Freitas, and J. T. De Souza, "Applying Search-Based Techniques For Requirements- Based Test Case Prioritization," in *Proceedings of the Brazilian Workshop on Optimization in Software Engineering, WOES 2010.*, 2010, pp. 24–31.

[19] M. Bowman, L. C. Briand, and Y. Labiche, "Solving The Class Responsibility Assignment Problem In Object-Oriented Analysis With Multi-Objective Genetic Algorithms," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 817–837, 2010.

[20] J. J. Durillo, Y. Zhang, E. Alba, M. Harman, and A. J. Nebro, "A Study Of The Bi-Objective Next Release Problem," *Empir. Softw. Eng.*, vol. 16, no. 1, pp. 29–60, 2011.

[21] M. M. A. Brasil, T. G. N. Da Silva, F. G. De Freitas, J. T. De Souza, and M. I. Cortés, "Multiobjective Software Release Planning With Dependent Requirements And Undefined Number Of Releases," in *13th International Conference on Enterprise Information Systems, ICEIS 2011.*, 2011, pp. 97–107.

[22] T. E. Colanzi, W. K. G. Assunção, S. R. Vergilio, and A. T. R. Pozo, "Generating Integration Test Orders For Aspect-Oriented Software With Multi-Objective Algorithms," in *Latin-American Workshop on Aspect-Oriented Software Development, LA-WASP 2011.*, 2011.

[23] W. K. G. Assunção, T. E. Colanzi, A. T. R. Pozo, and S. R. Vergilio, "Establishing Integration Test Orders Of Classes With Several Coupling Measures," in *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011.*, 2011, pp. 1867–1874.

[24] S. Yoo, M. Harman, and S. Ur, "GPGPU Test Suite Minimisation: Search Based Software Engineering Performance Improvement Using Graphics Cards," *Empir. Softw. Eng.*, vol. 18, no. 3, Mar. 2013.

[25] 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.

[26] C. A. Coello Coello, "A Comprehensive Survey Of Evolutionary-Based Multiobjective Optimization Techniques," *Knowl. Inf. Syst.* ., vol. 1, no. 3, 1999.

[27] G. G. Yen and H. Lu, "Dynamic Multiobjective Evolutionary Algorithm: Adaptive Cell-Based Rank And Density Estimation," *IEEE Trans. Evol. Comput.*, vol. 7, no. 3, pp. 253–274, 2003.

[28] J. D. Knowles and D. W. Corne, "M-PAES: A Memetic Algorithm For Multiobjective Optimization," in *IEEE Congress on Evolutionary Computation, CEC 2000.*, 2000, pp. 325–332.

[29] J. Horn, N. Nafpliotis, and D. E. Goldberg, "A Niched Pareto Genetic Algorithm For Multiobjective Optimization," in *IEEE World Conference On Computational Intelligence.*, 1994.

[30] 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.

[31] J. D. Knowles and D. W. Corne, "Approximating The Nondominated Front Using The Pareto Archived Evolution Strategy," *J. Evol. Comput.*, vol. 8, no. 2, pp. 149–172, 2000.

[32] H. A. Abbass, R. Sarker, and C. Newton, "PDE: A Pareto–Frontier Differential Evolution Approach For Multi-Objective Optimization Problems," in *IEEE Congress on Evolutionary Computation, CEC 2001.*, 2001.

[33] D. W. Corne, J. D. Knowles, and M. J. Oates, "The Pareto Envelope-Based Selection Algorithm For Multiobjective Optimization," in *6th International Conference on Parallel Problem Solving from Nature.*, 2000.

[34] E. Zitzler and L. Thiele, "Multiobjective Evolutionary Algorithms: A Comparative Case Study And The Strength Pareto Approach," *IEEE Trans. Evol. Comput.*, vol. 3, no. 4, pp. 257–271, 1999.

[35] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving The Strength Pareto Evolutionary Algorithm," in *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems, EUROGENS 2001.*, 2001.

[36] N. Srinivas and K. Deb, "Multiobjective Optimization Using Nondominated Sorting In Genetic Algorithms," *J. Evol. Comput.*, vol. 2, no. 3, 1994.

[37] 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.

[38] 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.

[39] M. Garza-Fabre, G. T. Pulido, and C. A. Coello Coello, "Ranking Methods For Many-Objective Optimization," in *Mexican International Conference On Artificial Intelligence, MICAI 2009.*, 2009, pp. 633–645.

[40] A. S. Sayyad, T. Menzies, and H. Ammar, "On The Value Of User Preferences In Search-Based Software Engineering: A Case Study In Software Product Lines," in *35th International Conference on Software Engineering, ICSE 2013.*, 2013, pp. 492–501.

[41] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar, "Scalable Product Line Configuration: A Straw To Break The Camel's Back," in *28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013.*, 2013.

[42] 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.

[43] S. Yang, M. Li, L. Xiaohui, and J. Zheng, "A Grid-Based Evolutionary Algorithm For Many-Objective Optimization," *IEEE Trans. Evol. Comput.*, vol. 17, no. 5, pp. 1–16, 2013.

[44] J. Bader and E. Zitzler, "HypE : An Algorithm For Fast Hypervolume-Based Many-Objective Optimisation," *Evol. Comput.*, vol. 19, no. 1, pp. 1–25, 2011.

[45] E. Zitzler and S. Künzli, "Indicator-Based Selection In Multiobjective Search," *8th*

Int. Conf. Parallel Probl. Solving from Nat. (PPSN VIII), pp. 1–11, 2004.

[46] 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.

[47] E. J. Hughes, "Multiple Single Objective Pareto Sampling," in *Congress on Evolutionary Computation, CEC 2003.*, 2003.

[48] S. Kukkonen and J. Lampinen, "Ranking-Dominance And Many-Objective Optimization," in *IEEE Congress on Evolutionary Computation, CEC 2007.*, 2007, pp. 3983–3990.

[49] 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.

[50] 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.

[51] 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.

[52] R. Wang, R. C. Purshouse, and P. J. Fleming, "Preference-Inspired Coevolutionary Algorithms For Many-Objective Optimization," *IEEE Trans. Evol. Comput.*, vol. 17, no. 4, pp. 474–494, 2013.

[53] 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.

[54] L. Ben Said, S. Bechikh, and K. Ghédira, "The r-Dominance: A New Dominance Relation For Interactive Evolutionary Multicriteria Decision Making," *IEEE Trans. Evol. Comput.*, vol. 14, no. 5, pp. 801–818, 2010.

[55] K. Deb and J. Sundar, "Reference Point Based Multi-Objective Optimization Using Evolutionary Algorithms," in *Genetic and Evolutionary Computation Conference, GECCO 2006.*, 2006, pp. 635–642.

[56] 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.

[57] H. Jain and K. Deb, "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part II: Handling Constraints And Extending To An Adaptive Approach," *IEEE Trans. Evol. Comput.*, vol. 18, no. 4, pp. 602–622, 2014.

[58] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating Software Engineering As A Search Problem," *IEE Proc. - Software.*, vol. 150, no. 3, pp. 1–25, 2003.

[59] M. Harman and J. Clark, "Metrics Are Fitness Functions Too," in *10th International Symposium on Software Metrics, METRICS 2004.*, 2004, pp. 1–12.

[60] M. Harman, "The Current State And Future Of Search Based Software Engineering," in *Future Of Software Engineering, FOSE 2007.*, 2007, pp. 342–

357.

[61] M. Harman, “Search Based Software Engineering For Program Comprehension,” in *15th IEEE International Conference on Program Comprehension, ICPC 2007.*, 2007, pp. 3–13.

[62] M. Harman, “Why The Virtual Nature Of Software Makes It Ideal For Search Based Optimization,” in *13th International Conference on Fundamental Approaches to Software Engineering, FASE 2010.*, 2010, pp. 1–13.

[63] M. Harman, “Software Engineering Meets Evolutionary Computation,” *Computer.*, vol. 44, no. 10, pp. 31–39, 2011.

[64] M. D. O. Barros and A. C. Dias Neto, “Threats To Validity In Search-Based Software Engineering Empirical Studies,” Rio de Janeiro, Brazil, 2011.

[65] F. G. De Freitas and J. T. De Souza, “Ten Years Of Search Based Software Engineering: A Bibliometric Analysis,” in *3rd International Symposium On Search-Based Software Engineering, SSBSE 2011.*, 2011, pp. 18–32.

[66] S. R. Vergilio, T. E. Colanzi, A. T. R. Pozo, and W. K. G. Assunção, “Search Based Software Engineering: A Review From The Brazilian Symposium On Software Engineering,” in *25th Brazilian Symposium on Software Engineering, SBES 2011.*, 2011, pp. 50–55.

[67] 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.

[68] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, “Managing Technical Debt In Software-Reliant Systems,” in *FSE/SDP Workshop on Future of Software Engineering Research, FoSER 2010.*, 2010, pp. 47--52.

[69] E. Allman, “Managing Technical Debt,” *Queue.*, vol. 10, no. 3, pp. 50–55, 01-Mar-2012.

[70] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou, and T. Amanatidis, “Estimating The Breaking Point For Technical Debt,” in *7th International Workshop on Managing Technical Debt, MTD 2015.*, 2015, pp. 53–56.

[71] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, “Searching For Build Debt: Experiences Managing Technical Debt At Google,” in *3rd International Workshop on Managing Technical Debt, MTD 2012.*, 2012.

[72] D. Fatiregun, M. Harman, and R. Hierons, “Search Based Transformations,” in *Genetic and Evolutionary Computation Conference, GECCO 2003.*, 2003, pp. 2511–2512.

[73] 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.

[74] H. Jiang, “Can The Genetic Algorithm Be A Good Tool For Software Engineering Searching Problems?,” in *30th Annual International Computer Software and Applications Conference, COMPSAC 2006.*, 2006, pp. 362–366.

[75] H. Jiang, C. K. Chang, D. Zhu, and C. Shuxing, “A Foundational Study On The Applicability Of Genetic Algorithm To Software Engineering Problems,” in *IEEE*

Congress on Evolutionary Computation, CEC 2007., 2007, pp. 2210–2219.

[76] 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.

[77] D. Doval, S. Mancoridis, and B. S. Mitchell, "Automatic Clustering Of Software Systems Using A Genetic Algorithm," in *9th International Workshop on Software Technology and Engineering Practice, STEP 1999.*, 1999, pp. 1–9.

[78] H. A. Sahraoui, P. Valtchev, I. Konkobo, and S. Shen, "Object Identification In Legacy Code As A Grouping Problem," in *Computer Software and Applications Conference, COMPSAC 2002.*, 2002, pp. 1–14.

[79] K. Mahdavi, M. Harman, and R. M. Hierons, "A Multiple Hill Climbing Approach To Software Module Clustering," in *International Conference on Software Maintenance, ICSM 2003.*, 2003, pp. 1–10.

[80] G. Antoniol, M. Di Penta, and M. Neteler, "Moving To Smaller Libraries Via Clustering And Genetic Algorithms," in *7th European Conference on Software Maintenance and Reengineering, CSMR 2003.*, 2003, pp. 307–316.

[81] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo, "A Language-Independent Software Renovation Framework," *J. Syst. Software.*, vol. 77, no. 3, pp. 225–240, Sep. 2004.

[82] O. Seng, M. Bauer, M. Biehl, and G. Pache, "Search-Based Improvement Of Subsystem Decompositions," in *Conference on Genetic and Evolutionary Computation, GECCO 2005.*, 2005, pp. 1045–1051.

[83] N. Gold, M. Harman, and Z. Li, "Allowing Overlapping Boundaries In Source Code Using A Search Based Approach To Concept Binding," in *22nd IEEE International Conference on Software Maintenance, ICSM 2006.*, 2006, pp. 310–319.

[84] S. Huynh and Y. Cai, "An Evolutionary Approach To Software Modularity Analysis," in *1st International Workshop on Assessment of Contemporary Modularization Techniques, ACoM 2007.*, 2007.

[85] Amarjeet and J. K. Chhabra, "Improving Package Structure Of Object-Oriented Software Using Multi-Objective Optimization And Weighted Class Connections," *J. King Saud Univ. - Comput. Inf. Sci.*, 2015.

[86] B. S. Mitchell, "A Heuristic Search Approach To Solving The Software Clustering Problem," 2002.

[87] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using Automatic Clustering To Produce High-Level System Organizations Of Source Code," in *6th International Workshop on Program Comprehension, IWPC 1998.*, 1998, pp. 45–52.

[88] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A Clustering Tool For The Recovery And Maintenance Of Software System Structures," in *IEEE International Conference on Software Engineering, ICSM 1999.*, 1999, pp. 50–59.

[89] B. S. Mitchell and S. Mancoridis, "Using Heuristic Search Techniques To Extract Design Abstractions From Source Code," in *Genetic and Evolutionary Computation Conference, GECCO 2002.*, 2002, pp. 1375–1382.

[90] B. S. Mitchell, S. Mancoridis, and M. Traverso, “Search Based Reverse Engineering,” in *14th International Conference On Software Engineering And Knowledge Engineering, SEKE 2002.*, 2002, pp. 431–438.

[91] B. S. Mitchell and S. Mancoridis, “Modeling The Search Landscape Of Metaheuristic Software Clustering Algorithms,” in *Genetic and Evolutionary Computation Conference, GECCO 2003.*, 2003, pp. 2499–2510.

[92] B. S. Mitchell, S. Mancoridis, and M. Traverso, “Using Interconnection Style Rules To Infer Software Architecture Relations,” in *Genetic and Evolutionary Computation Conference, GECCO 2004.*, 2004.

[93] B. S. Mitchell and S. Mancoridis, “On the Automatic Modularization Of Software Systems Using The Bunch Tool,” *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 193–208, 2006.

[94] B. S. Mitchell and S. Mancoridis, “On The Evaluation Of The Bunch Search-Based Software Modularization Algorithm,” *Soft Comput. - A Fusion Found. Methodol. Appl.*, vol. 12, no. 1, pp. 77–93, Jun. 2007.

[95] R. Marinescu, “Detecting Design Flaws Via Metrics In Object-Oriented Systems,” in *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, TOOLS 2001.*, 2001, pp. 173–182.

[96] D. Kirk, M. Roper, and M. Wood, “A Heuristic-Based Approach To Code-Smell Detection,” in *1st Workshop On Refactoring Tools, WRT 2007.*, 2007, pp. 54–55.

[97] M. Kessentini, S. Vaucher, and H. Sahraoui, “Deviance From Perfection Is A Better Criterion Than Closeness To Evil When Identifying Risky Code,” in *IEEE/ACM International Conference on Automated Software Engineering, ASM 2010.*, 2010, pp. 113–122.

[98] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, “Search-Based Design Defects Detection By Example,” in *14th International Conference on Fundamental Approaches to Software Engineering, FASE 2011.*, 2011, pp. 401–415.

[99] U. Mansoor, M. Kessentini, S. Bechikh, and K. Deb, “Code-Smells Detection Using Good And Bad Software Design Examples,” 2013.

[100] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, “A Cooperative Parallel Search-Based Software Engineering Approach For Code-Smells Detection,” *IEEE Trans. Softw. Eng.*, vol. 40, no. 9, pp. 841–861, 2014.

[101] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, “Multi-Objective Code-Smells Detection Using Good And Bad Design Examples,” *Softw. Qual. Journal.*, pp. 1–24, 2016.

[102] T. Dudziak and J. Wloka, “Tool-Supported Discovery And Refactoring Of Structural Weaknesses In Code,” 2002.

[103] T. Nguyen, W. Weimer, C. Le Goues, and S. Forrest, “Using Execution Paths To Evolve Software Patches,” in *International Conference on Software Testing, Verification, and Validation.*, 2009, pp. 152–153.

[104] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “GenProg: A Generic Method For Automatic Software Repair,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[105] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, “A Systematic Study Of

Automated Program Repair: Fixing 55 Out Of 105 Bugs For $8 Each," in *34th International Conference on Software Engineering, ICSE 2012.*, 2012, pp. 3–13.

[106] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably Optimal And Human-Competitive Results In SBSE For Spectrum Based Fault Localisation," in *5th International Symposium On Search-Based Software Engineering, SSBSE 2013.*, 2013.

[107] M. Ó Cinnéide and P. Nixon, "Automated Application Of Design Patterns To Legacy Code," in *Workshop on Object-Oriented Technology.*, 1999, pp. 1–5.

[108] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements Of Reusable Object-Oriented Software*. 1994.

[109] M. Ó Cinnéide and P. Nixon, "A Methodology For The Automated Introduction Of Design Patterns," in *IEEE International Conference on Software Maintenance, ICSM 1999.*, 1999, pp. 1–10.

[110] M. Ó Cinnéide, "Automated Refactoring To Introduce Design Patterns," in *International Conference on Software Engineering, ICSE 2000.*, 2000, pp. 722–724.

[111] M. O'Keeffe and M. Ó Cinnéide, "A Stochastic Approach To Automated Design Improvement," in *2nd International Conference on Principles and Practice of Programming in Java, PPPJ 2003.*, 2003, pp. 59–62.

[112] M. O'Keeffe and M. Ó Cinnéide, "Towards Automated Design Improvement Through Combinatorial Optimisation," in *Workshop on Directions in Software Engineering Environments, WoDiSEE 2004.*, 2004.

[113] M. O'Keeffe and M. Ó Cinnéide, "Search-Based Software Maintenance," in *10th European Conference on Software Maintenance and Reengineering, CSMR 2006.*, 2006, pp. 251–260.

[114] M. O'Keeffe and M. Ó Cinnéide, "Search-Based Refactoring For Software Maintenance," *J. Syst. Software.*, vol. 81, no. 4, pp. 502–516, Apr. 2008.

[115] 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.

[116] 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.

[117] 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.

[118] M. O'Keeffe and M. Ó Cinnéide, "Automated Design Improvement By Example," in *6th Conference on New Trends in Software Methodologies, Tools and Techniques, SoMeT 2007.*, 2007, pp. 315–329.

[119] Z. Xing and E. Stroulia, "The JDEvAn Tool Suite In Support Of Object-Oriented Evolutionary Development," in *13th International Conference on Software Engineering, ICSE 2008.*, 2008, pp. 951–952.

[120] I. H. Moghadam and M. Ó. Cinnéide, "Automated Refactoring Using Design Differencing," in *16th Conference on Software Maintenance and Reengineering,*

CSMR 2012., 2012, pp. 43–52.

[121] 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.

[122] M. Harman, "Refactoring As Testability Transformation," in *4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTVV 2011.*, 2011, pp. 1–8.

[123] R. Morales, A. Sabané, P. Musavi, F. Khomh, F. Chicano, and G. Antoniol, "Finding The Best Compromise Between Design Quality And Testing Effort During Refactoring," in *23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016.*, 2016, pp. 24–35.

[124] M. Ó Cinnéide, D. Boyle, and I. H. Moghadam, "Automated Refactoring For Testability," in *4th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTVV 2011.*, 2011, pp. 437–443.

[125] S. Ghaith and M. Ó Cinnéide, "Improving Software Security Using Search-Based Refactoring," in *4th International Symposium On Search-Based Software Engineering, SSBSE 2012.*, 2012, pp. 121–135.

[126] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam, "Experimental Assessment Of Software Metrics Using Automated Refactoring," in *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2012.*, 2012, pp. 49–58.

[127] M. Ó Cinnéide, I. H. Moghadam, M. Harman, S. Counsell, and L. Tratt, "An Experimental Search-Based Approach To Cohesion Metric Evaluation," *Empir. Softw. Eng.*, 2016.

[128] V. Veerappa and R. Harrison, "An Empirical Validation Of Coupling Metrics Using Automated Refactoring," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2013.*, 2013, pp. 271–274.

[129] C. Simons, J. Singer, and D. R. White, "Search-Based Refactoring: Metrics Are Not Enough," in *7th International Symposium On Search-Based Software Engineering, SSBSE 2015.*, 2015, pp. 1–14.

[130] A. D. Bakar, A. B. Sultan, H. Zulzalil, and J. Din, "Applying Evolution Programming Search Based Software Engineering (SBSE) In Selecting The Best Open Source Software Maintainability Metrics," in *International Symposium on Computer Applications and Industrial Electronics, ISCAIE 2012.*, 2012, no. Iscaie, pp. 70–73.

[131] A. D. Bakar, A. B. M. Sultan, H. Zulzalil, and J. Din, "Review On 'Maintainability' Metrics In Open Source Software," *Int. Rev. Comput. Software.*, vol. 7, no. 3, pp. 903–908, 2012.

[132] M. Harman, J. Clark, and M. Ó Cinnéide, "Dynamic Adaptive Search Based Software Engineering Needs Fast Approximate Metrics," in *4th International Workshop on Emerging Trends in Software Metrics, WETSoM 2013.*, 2013, pp. 1–6.

[133] M. Harman, E. Burke, J. A. Clark, and X. Yao, "Dynamic Adaptive Search Based Software Engineering," in *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2012.*, 2012, pp. 1–8.

[134] 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.

[135] 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.

[136] 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.

[137] 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.

[138] 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.

[139] 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.

[140] 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.

[141] 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.

[142] H. Wang, M. Kessentini, W. Grosky, and H. Meddeb, "On The Use Of Time Series And Search Based Software Engineering For Refactoring Recommendation," in *7th International Conference on Management of computational and collective intElligence in Digital EcoSystems, MEDES 2015.*, 2015, no. October, pp. 35–42.

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

[144] 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.

[145] 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.

[146] 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.

[147] 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.

[148] 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.

[149] 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.

[150] 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.

[151] 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.

[152] 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.

[153] 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.

[154] R. Morales, "Towards A Framework For Automatic Correction Of Anti-Patterns," in *22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015.*, 2015, pp. 603–604.

[155] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On The Use Of Developers' Context For Automatic Refactoring Of Software Anti-Patterns," *J. Syst. Software.*, pp. 2–58, 2016.

[156] 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.

[157] M. Harman, R. Hierons, and M. Proctor, "A New Representation And Crossover Operator For Search-Based Optimization Of Software Modularization," in *Genetic and Evolutionary Computation Conference, GECCO 2002.*, 2002, pp. 1–8.

[158] 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.

[159] F. Qayum and R. Heckel, "Local Search-Based Refactoring As Graph Transformation," in *1st International Symposium On Search-Based Software Engineering, SSBSE 2009.*, 2009, pp. 43–46.

[160] 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 2014.*, 2014, pp. 31–45.

[161] 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.

[162] 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.

[163] M. Harman, S. A. Mansouri, and Y. Zhang, "Search Based Software Engineering:

A Comprehensive Analysis And Review Of Trends Techniques And Applications," 2009.

[164] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting Empirical Methods For Software Engineering Research," in *Guide to Advanced Empirical Software Engineering.*, 2008, pp. 285–311.

[165] M. Mohan, D. Greer, and P. McMullan, "Technical Debt Reduction Using Search Based Automated Refactoring," *J. Syst. Software.*, vol. 120, pp. 183–194, 2016.

[166] A. Murgia, R. Tonelli, G. Concas, M. Marchesi, and S. Counsell, "Parameter-Based Refactoring And The Relationship With Fan-In/Fan-Out Coupling," *J. Object Technol.*, vol. 11, no. 2, pp. 1–24, 2012.

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

[168] M. Fowler, "Refactoring Catalog," 2015. [Online]. Available: http://refactoring.com/catalog/. [Accessed: 22-Apr-2015].

[169] D. Heuzeroth and U. Aßmann, "The COMPOST, COMPASS, InjectJ And RECODER Tool Suite For Invasive Software Composition - Invasive Composition With COMPASS Aspect-Oriented Connectors," in *International Summer School on Generative and Transformational Techniques in Software Engineering, GTTSE 2005.*, 2005.

[170] N. Mohd Razali and J. Geraghty, "Genetic Algorithm Performance With Different Selection Strategies In Solving TSP," in *World Congress on Engineering, WCE 2011.*, 2011.

[171] A. Ouni, "A Mono- And Multi-Objective Approach For Recommending Software Refactoring," 2014.

[172] 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.

[173] D. Bell, *Software Engineering: A Programming Approach*. Prentice Hall, 2000.

[174] R. S. Pressman and B. R. Maxim, *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2000.

[175] M. Bozkurt and M. Harman, "Optimised Realistic Test Input Generation," *4th Int. Symp. Search-Based Softw. Eng. SSBSE 2012.*, vol. 7515, pp. 1–6, 2012.

[176] K. Lakhotia, M. Harman, and H. Gross, "AUSTIN: A Tool For Search Based Software Testing For The C Language And Its Evaluation On Deployed Automotive Systems," in *2nd International Symposium On Search-Based Software Engineering, SSBSE 2010.*, 2010, pp. 101–110.

[177] B. Korel, "Automated Software Test Data Generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[178] K. Sen and G. Agha, "CUTE And jCUTE : Concolic Unit Testing And Explicit Path Model-Checking Tools (Tools Paper)," in *18th International Conference on Computer Aided Verification, CAV 2006.*, 2006, pp. 419–423.

[179] K. Lakhotia, M. Harman, and P. McMinn, "Handling Dynamic Data Structures In Search Based Testing," in *Genetic and Evolutionary Computation Conference, GECCO 2008.*, 2008, pp. 1759–1766.

[180] D. Greer and G. Ruhe, "Software Release Planning: An Evolutionary And

Iterative Approach," *Inf. Softw. Technol.*, vol. 46, no. 4, pp. 243–253, Mar. 2004.

[181] A. Ngo-The and G. Ruhe, "A Systematic Approach For Solving The Wicked Problem Of Software Release Planning," *Soft Comput.*, vol. 12, no. 1, pp. 95–108, Jun. 2007.

[182] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation For Object-Oriented Software," in *13th European Software Engineering Conference and 19th ACM SIGSOFT Symposium on Foundations of Software Engineering, ESEC/FSE 2011.*, 2011, pp. 416–419.

[183] K. Lakhotia, N. Tillmann, M. Harman, and J. De Halleux, "FloPSy - Search-Based Floating Point Constraint Solving For Symbolic Execution," in *22nd IFIP WG 6.1 International Conference on Testing Software and Systems, ICTSS 2010.*, 2010, pp. 1–16.

[184] G. Ganea, I. Verebi, and R. Marinescu, "Continuous Quality Assessment With inCode," in *14th European Conference on Software Maintenance and Reengineering, CSMR 2010.*, 2013, pp. 1–10.

[185] J. Yue and M. Harman, "MiLu : A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool For The Full C Language," in *Testing: Academic & Industrial Conference - Practice and Research Techniques, TIAC PART 2008.*, 2008, pp. 94–98.

[186] N. Alshahwan and M. Harman, "Automated Web Application Testing Using Search Based Software Engineering," in *26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011.*, 2011, pp. 3–12.

[187] R. Feldt, "An Interactive Software Development Workbench Based On Biomimetic Algorithms," 2002.

[188] M. Di Penta and S. Poulding, "Introduction To The Special Issue On Search Based Software Engineering," *Empir. Softw. Eng.*, vol. 16, no. 1, pp. 1–4, Jan. 2011.

[189] M. Di Penta, G. Antoniol, and M. Harman, "Special Issue On Search-Based Software Maintenance," *J. Softw. Maint. Evol. Res. Pract.*, vol. 20, no. 5, pp. 317–319, Sep. 2008.

[190] W. J. Gutjahr and M. Harman, "Search-Based Software Engineering," *Comput. Oper. Res.*, vol. 35, no. 10, pp. 3049–3051, Oct. 2008.

[191] M. Harman and B. F. Jones, "The SEMINAL Workshop: Reformulating Software Engineering As A Metaheuristic Search Problem," *Softw. Eng. Notes.*, vol. 26, no. 6, pp. 62–66, 2001.

[192] M. Harman and B. F. Jones, "Software Engineering Using Metaheuristic Innovative Algorithms: Workshop Report," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 905–907, Dec. 2001.

[193] M. Harman and A. Mansouri, "Search Based Software Engineering: Introduction To The Special Issue Of The IEEE Transactions On Software Engineering," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 737–741, 2010.

[194] M. Harman and J. Wegener, "Getting Results From Search-Based Approaches To Software Engineering," in *26th International Conference On Software Engineering, ICSE 2004.*, 2004, pp. 728–729.

[195] M. Harman, B. Korel, and P. K. Linos, "Guest Editorial: Special Issue On

Software Maintenance And Evolution," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 801–803, 2005.

[196] M. Harman, "Search-Based Software Engineering For Maintenance And Reengineering," in *Conference On Software Maintenance And Reengineering, CSMR 2006.*, 2006, p. 311.

[197] M. Harman, "The Importance Of Metrics In Search Based Software Engineering," in *International Conference on Software Process And Product Measurement, MENSURA 2006.*, 2006, pp. 15–20.

[198] M. Harman, "Overview Of TASE 2012 Talk On Search Based Software Engineering," in *6th International Symposium on Theoretical Aspects of Software Engineering, TASE 2012.*, 2012, pp. 3–4.

[199] M. Harman, "Software Engineering: An Ideal Set Of Challenges For Evolutionary Computation," in *15th International Conference on Genetic and Evolutionary Computation, GECCO 2013.*, 2013, pp. 1759–1760.

[200] M. Ó Cinnéide and M. B. Cohen, "Introduction To The Special Issue On Search Based Software Engineering," *Empir. Softw. Eng.*, vol. 18, no. 3, pp. 547–549, May 2013.

[201] 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.

[202] 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.

[203] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Softw. Testing, Verif. Reliab.*, vol. 14, no. 2, pp. 1–58, 2004.

[204] 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.

[205] O. Räihä, "A Survey On Search-Based Software Design," *Comput. Sci. Rev.*, vol. 4, no. 4, pp. 203–249, 2010.

[206] 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.

Acknowledgements

The author would like to thank his supervisors, Dr. Des Greer and Dr. Paul McMullan for giving valuable advice and reviewing his work to date. In particular, Dr. Greer has been consistently helpful throughout the course of the PhD and has remained open and inquisitive during the many meetings held. The author would also like to thank Queen's University Belfast for the facilities they allowed him to have at his disposal and especially for providing the funding to attend a workshop in University College London that allowed him to discuss the research area with other researchers at an early stage. Acknowledgements must also go out to the other attendees at said workshop as well as the organisers. Gratitude goes out to Ekin Koc, who gave the author permission to use the A-CMA refactoring tool for experimentation, and even updated the developer licence in order for the author to modify the tool for use. The author thanks the benefactors of the Emily Sarah Montgomery travel scholarship for the award which allowed him to attend a conference in Germany and gain greater exposure to the research conducted as part of the PhD. The work undertaken for this review was funded by an EPSRC PhD studentship (the duration of which was 2014-2018).

Appendix A – Literature Review Quantitative Analysis

Various aspects of the analysed SBSM literature were measured and are outlined in more detail below. Figure A.1 shows the number of papers published per year among the main SBSM papers and Figure A.3 shows the number of search techniques analysed per year among them. Figure A.2 also shows the number of papers published per year among all 99 of the papers analysed. Figure A.4 shows the different ways the papers have been published, with the majority being published in journals or featured in conferences. Table A.1 lists the more popular conferences that have featured 2 or more of the papers and Table A.2 list the different journals that the papers have been published in. Figure A.5 outlines the number of authors that have a certain number of papers published, with the majority of authors only having 1. Table A.3 lists the authors that have 4 or more papers published. Table A.4 list the qualitative or discussion papers among the main SBSE papers analysed. Figures A.6-A.8 display visualisations of the number of SBSM papers each type of search technique was present in. Figure A.9 shows the number of papers that involve a certain number of search techniques among the main SBSM papers (there are anywhere from none to 4 different search techniques in any 1 paper). Figure A.10 shows the different types of program used to test the approaches in the main SBSM papers, with the majority being open source Java programs. Table A.5 lists the different open source programs used across the analysed studies, as well as the studies that each program has been used in.

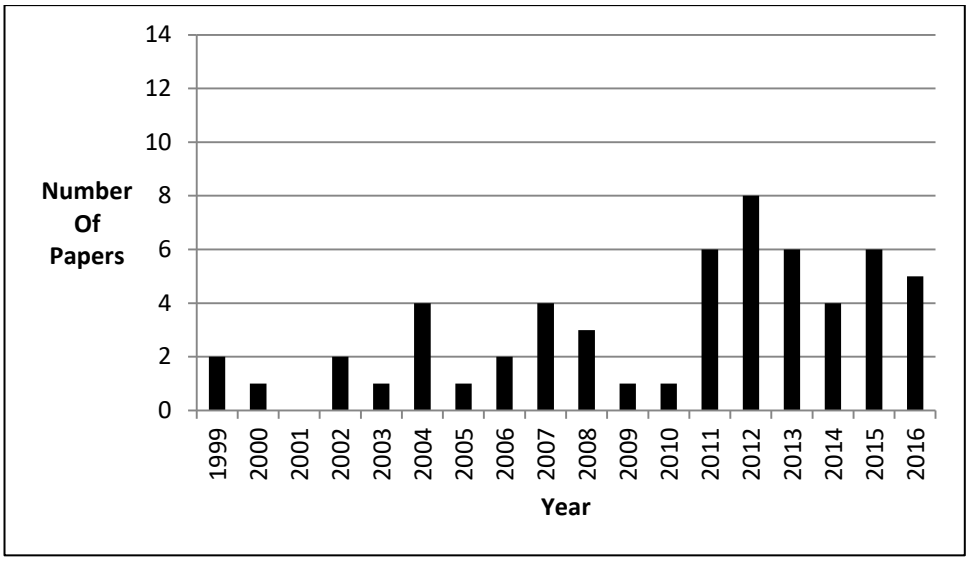

Figure A.1 – Number of the Main Search-Based Software Maintenance Papers Published Each Year

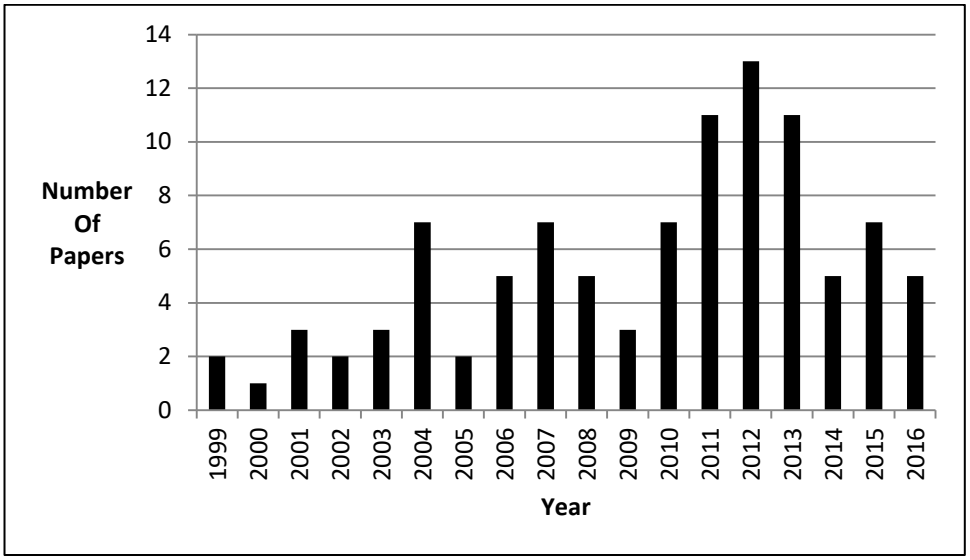

Figure A.2 – Number of Papers Published Each Year

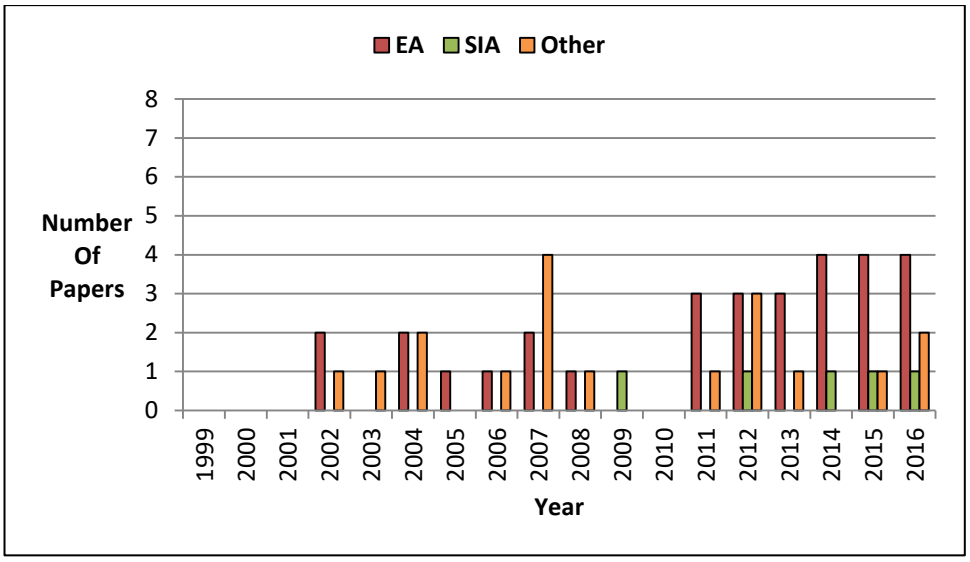

Figure A.3 – Number of the Main Search-Based Software Maintenance Papers Using Each Type of Search Technique per Year

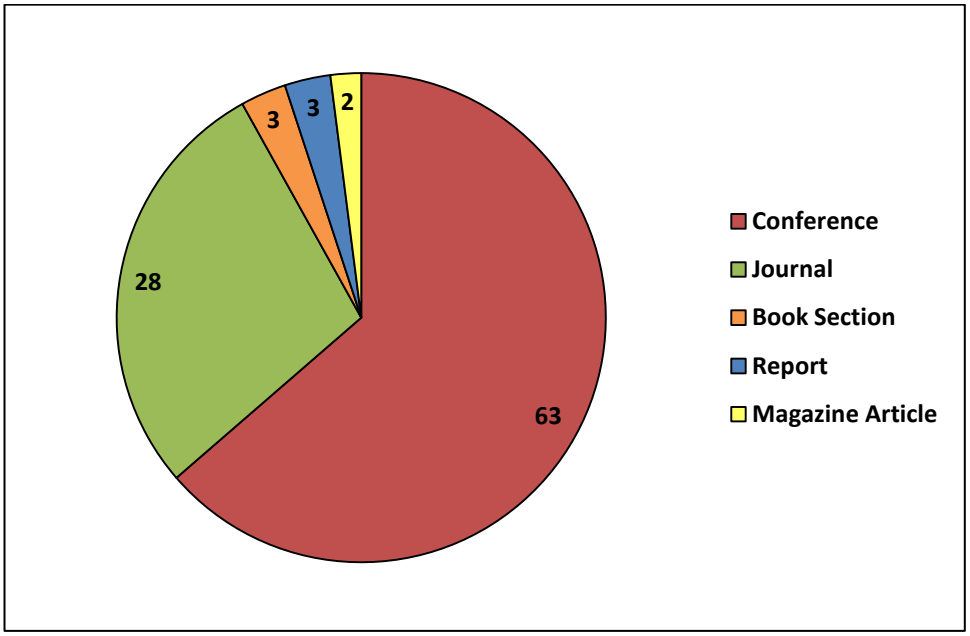

Figure A.4 – Types of Paper Analysed

Table A.1 – Number of Papers per Conference

Conference	Number Of Papers
Genetic and Evolutionary Computation Conference (GECCO)	12
European Conference on Software Maintenance and Reengineering (CSMR)	7
Symposium on Search-Based Software Engineering (SSBSE)	7
International Conference on Software Maintenance (ISCM)	3
International Symposium on Empirical Software Engineering and Measurement (ESEM)	2
International Conference on Program Comprehension (ICPC)	2
International Conference on Software Engineering (ICSE)	2
International Conference on Software Testing, Verification and Validation (ICST/ICSTVV)	2
International Workshop on Managing Technical Debt (MTD)	2
International Conference on Software Analysis, Evolution, and Reengineering (SANER)	2
International Working Conference on Source Code Analysis and Manipulation (SCAM)	2

Table A.2 – Number of Papers per Journal

Journals	Number Of Papers
Empirical Software Engineering	6
Journal of Systems and Software	4
IEEE Transactions on Software Engineering	3
ACM Transactions on Software Engineering and Methodology	2
Information and Software Technology	2
Journal of Software Maintenance and Evolution: Research and Practice	2
Software Quality Journal	2
ACM Computing Surveys	1
Automated Software Engineering	1
Computer Science Review	1
Computers & Operations Research	1
IEE Proceedings – Software	1
Software Engineering Notes	1
Software Testing, Verification and Reliability	1

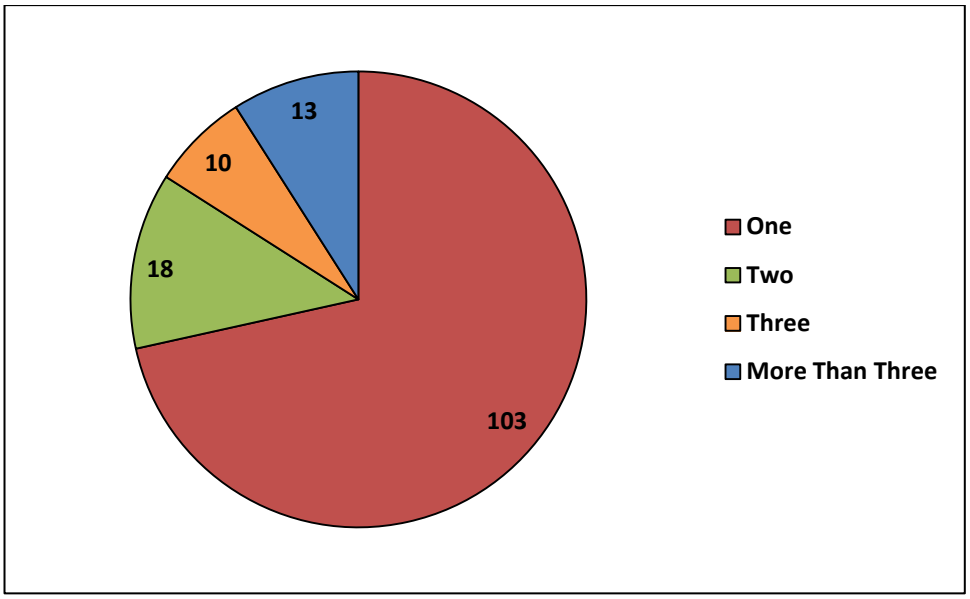

Figure A.5 – Number of Papers per Author

Table A.3 – Number of Papers per Author

Authors	Number Of Papers
Mark Harman	32
Mel Ó Cinnéide	22
Marouane Kessentini	17
Ali Ouni	9
Houari Sahraoui	8
Mark O' Keeffe	7
Kalyanmoy Deb	6
Slim Bechikh	6
Iman Hemati Moghadam	5
John A. Clark	5
Wiem Mkaouer	5
Bryan Jones	4
Robert M. Hierons	4

Table A.4 – Analysed Papers from the Main Search-Based Software Maintenance Papers That Are Not Quantitative

Authors [Ref]	Year	Type	Title
Bakar *et al.* [130]	2012	Discussion	Applying Evolution Programming Search Based Software Engineering (SBSE) In Selecting The Best Open Source Software Maintainability Metrics
Harman [122]	2011	Discussion	Refactoring As Testability Transformation
Harman *et al.* [133]	2012	Discussion	Dynamic Adaptive Search Based Software Engineering
Harman *et al.* [132]	2013	Discussion	Dynamic Adaptive Search Based Software Engineering Needs Fast Approximate Metrics
Moghadam and Ó Cinnéide [152]	2011	Discussion	Code-Imp - A Tool For Automated Search-Based Refactoring
Morales [154]	2015	Discussion	Towards A Framework For Automatic Correction Of Anti-Patterns
Ó Cinnéide [110]	2000	Discussion	Automated Refactoring To Introduce Design Patterns
Ó Cinnéide and Nixon [109]	1999	Discussion	A Methodology For The Automated Introduction Of Design Patterns
Ó Cinnéide and Nixon [107]	1999	Discussion	Automated Application Of Design Patterns To Legacy Code
Pérez *et al.* [143]	2013	Discussion	A Proposal For Fixing Design Smells Using Software Refactoring History
Tsantalis *et al.* [150]	2008	Discussion	JDeodorant: Identification And Removal Of Type-Checking Bad Smells
Ó Cinnéide *et al.* [124]	2011	Qualitative	Automated Refactoring For Testability
O'Keeffe and Ó Cinnéide [111]	2003	Qualitative	A Stochastic Approach To Automated Design Improvement
Qayum and Heckel [159]	2009	Qualitative	Local Search-Based Refactoring As Graph Transformation
Simons *et al.* [129]	2015	Qualitative	Search-Based Refactoring - Metrics Are Not Enough

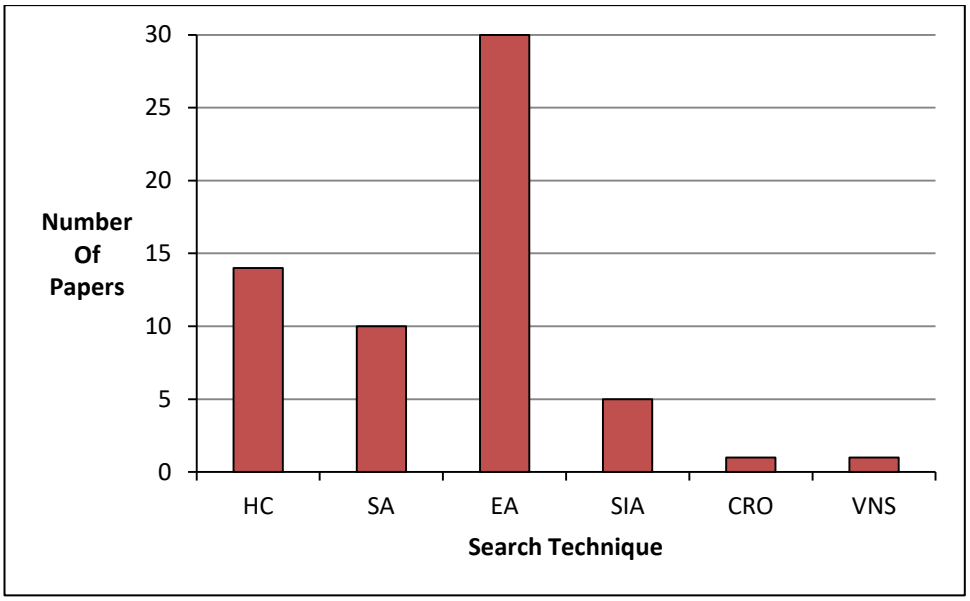

Figure A.6 – Types of Search Technique Used in the Main Search-Based Software Maintenance Papers

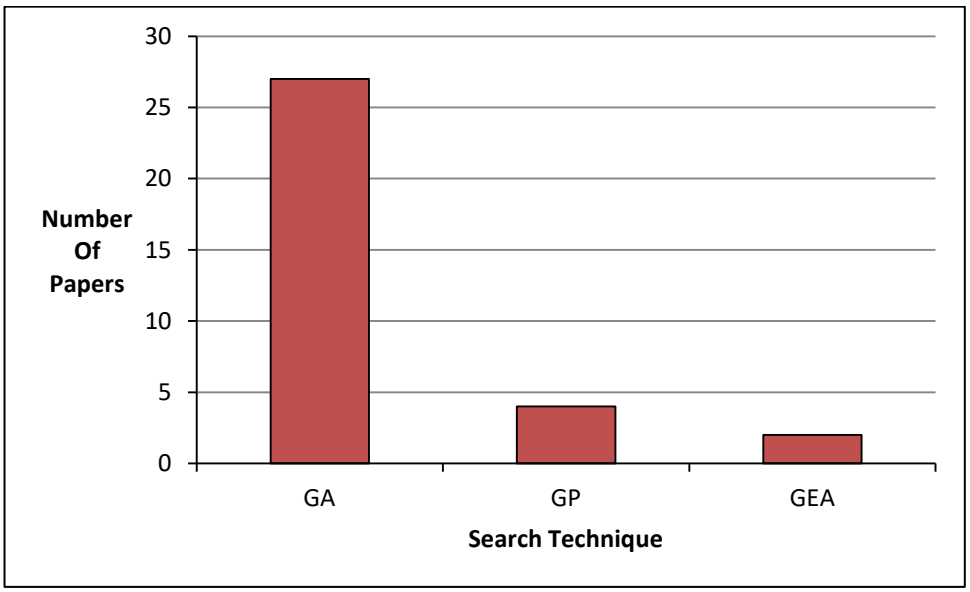

Figure A.7 – Dispersion of Evolutionary Algorithms from Figure A.6 (Some Papers Contain More Than One Search Technique)

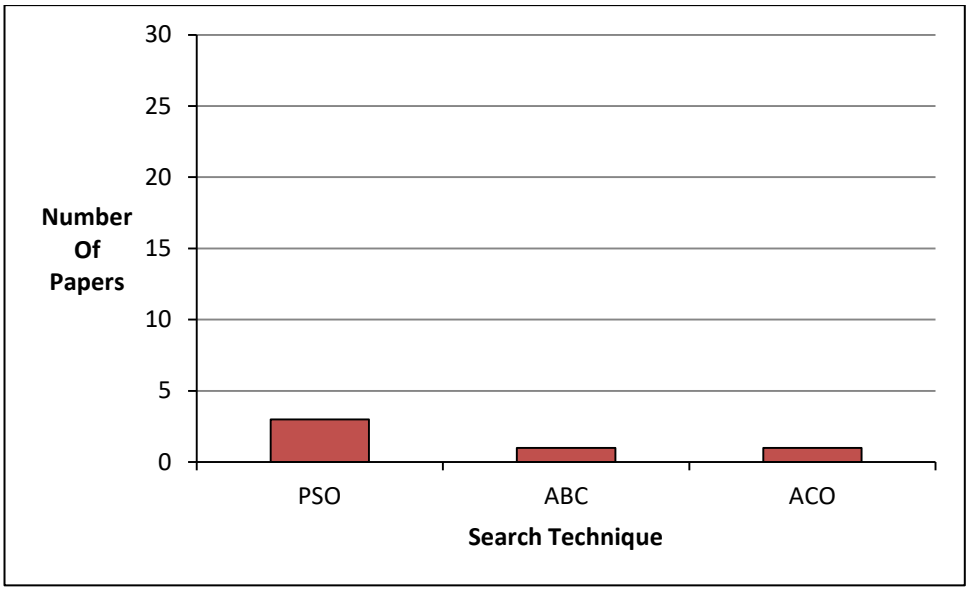

Figure A.8 – Dispersion of Swarm Intelligence Algorithms from Figure A.6

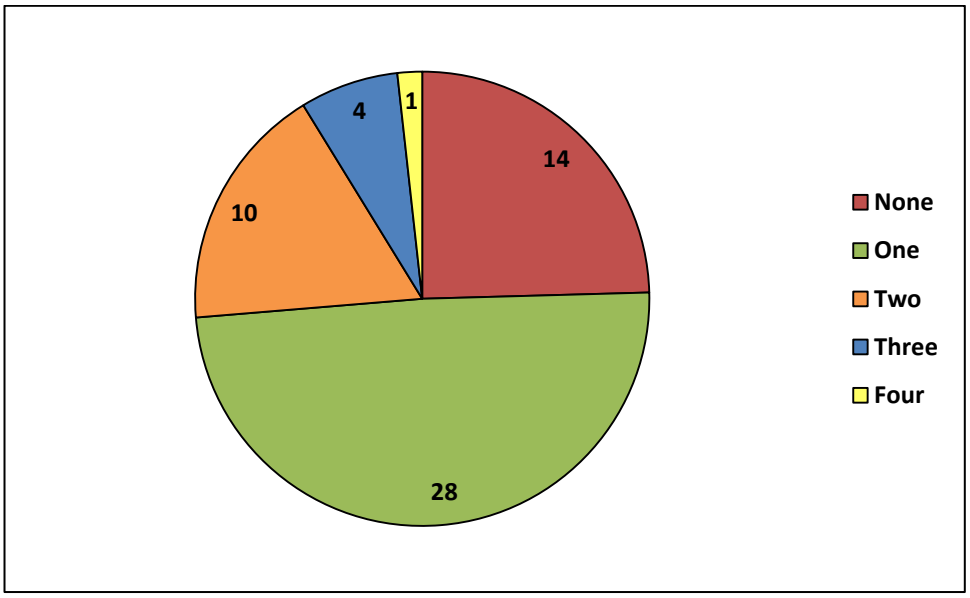

Figure A.9 – Number of Search Techniques Used/Analysed in Each Search-Based Software Maintenance Paper

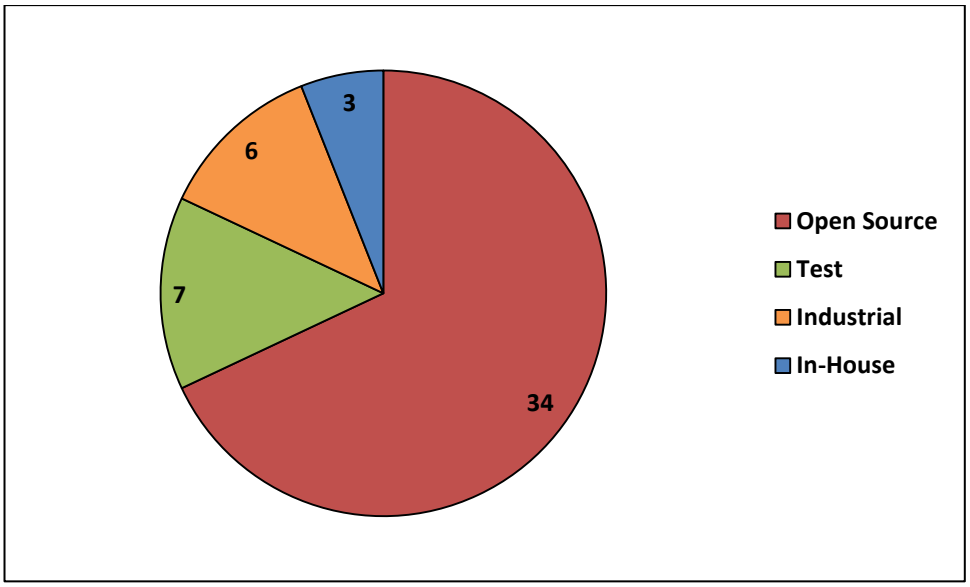

Figure A.10 – Types of Benchmark Program Used in Experimental Studies in the Main Search-Based Software Maintenance Papers

Table A.5 – Open Source Test Programs Used in the Literature

Apache Ant [25], [144]–[146], [160]	Apache XML-RPC [117]	ArgoUML [25], [123], [134]–[136], [146]
Art of Illusion [37], [126]–[128], [147]	Azureus [25], [135], [136], [146]	Beaver [114]–[118]
EAOP [115], [116]	FindBugs [142]	GanttProject [25], [37], [42], [120], [123], [126]–[128], [134]–[139], [144]–[147], [160]
Grammatica [116]	GRASS [149]	Hibernate [142]
HTMLUnit [120], [127]	JabRef [126]–[128]	JFlex [117]
JFreeChart [37], [42], [128], [141], [142], [144], [145], [147], [160]	JGraphX [120], [126]–[128]	JHotDraw [6], [37], [42], [120], [121], [123], [126]–[128], [137], [139], [144], [145], [147], [160]
JRDF [126]–[128]	jSMPP [127]	JSON [117]
JTar [120], [126], [127]	KDE [149]	Log4j [135], [136], [145]
Mango [115]–[118]	Maven [6]	Mylyn [123], [155]
MySQL [149]	Nutch [145]	PDE [155]
Pixelitor [142]	Platform [155]	QuickUML [134]–[137]
Rhino [144], [145], [160]	Samba [149]	Spec-Check [113]–[116], [118]
Spec-Raytrace [113]	Wife [125]	Wrangler [151]
Xerces-J [25], [37], [42], [134]–[139], [141], [144]–[147], [160]	XOM [6], [120], [126]–[128]	

Appendix B – SBSE Software Packages From Literature

Various software packages have been created and proposed in the literature to assist with the SBSE research done, as listed in Table B.1. Descriptions are given for the identified tools below.

Table B.1 – List of Search-Based Software Engineering Tools with Brief Description and Search-Based Software Engineering Area

Software Package	Area Of Software Engineering	Purpose
WISE	Design	Interactive software development workbench.
GenProg	Error Resolution	Generic tool for automated software repair.
A-CMA	Maintenance	Refactors Java bytecode using a selection of refactorings and metrics.
Advanced Refactoring Wizard	Maintenance	Integration platform for problem detection and refactoring using jGoose Echidna, Costrat and Inject/J.
Bunch	Maintenance	Optimises software programs with module clustering.
CODe-Imp	Maintenance	Automated refactoring tool containing numerous metrics and refactorings.
Dearthóir	Maintenance	Improves the design of an object-oriented program.
DPT	Maintenance	Applies design pattern transformations to Java programs.
Evolution Doctor	Maintenance	Used to diagnose reorganisation opportunities and perform reengineering actions.
FermaT	Maintenance	Transformation tool for migration of legacy systems from assembly code to higher level languages.
J/Art	Maintenance	Detects structural weaknesses in code.
JDeodorant	Maintenance	Identifies and removes 4 different types of design smell.
TrueRefactor	Maintenance	Identifies and removes 5 different design smells in Java.
Wrangler	Maintenance	Provides general purpose refactorings in Erlang.
EVOLVE	Requirements	Offers decision support for software release planning.
ATAM	Testing	Provides realistic test data to test other services.
AUSTIN	Testing	Helps to achieve branch coverage with testing.
CUTE/jCUTE	Testing	Concolic unit testing engine.
eTOC	Testing	Evolutionary test data generation tool.
EvoSUITE	Testing	Automatic test suite generation for object-oriented software.
FLoPSy	Testing	Search-based floating point constraint solver.

Software Package	Area Of Software Engineering	Purpose
inCode	Testing	Continuously assesses the quality of Java systems and identifies design flaws as they appear.
MiLu	Testing	Customisable higher order mutation testing tool.
SWAT	Testing	Automated web application testing tool.

The A-CMA tool was developed by Koc *et al.* [117] to analyse and improve Java bytecode. It contains 24 different software metrics and uses 20 refactoring actions in order to carry out its function. It can choose between 5 different search techniques (random search, multiple variations of a HC search, SA, ABC and beam search) in order to find improved solutions from the input. The tool uses the ASM framework to get access to the bytecode and form a virtual representation of the code. From here the search technique can be chosen to analyse and modify the virtual design in order to find an improved version of the code. It has a GUI interface to use for structuring optimisation tasks or analysing the metric results.

The Advanced Refactoring Wizard is actually a combination of tools used by Trifu *et al.* [148] in order to use to detect and correct design flaws in object-oriented systems. The tool chain contains 3 tools for each separate phase of the process. Problem detection and analysis is done with jGoose Echidna, solution analysis is handled with Costrat and the reorganisation itself is executed with Inject/J. The Advanced Refactoring Wizard serves as an integration platform for the process. The problem detection phase is able to look for 19 different design flaws and the solution analysis phase has developed correction strategies for 4 design problems. The tool chain supports Java code although there is potential for other languages to be supported.

ATAM is a tool developed by Bozkurt and Harman [175] to help collect realistic test input data. The tool uses existing semantic web services online like Google, Bing and Yahoo to extract realistic data like ZIP codes or IP addresses. ATAM requires the ability to select and use services with higher reliability and with low price. This allows the services to be used with lower cost and increased efficiency. ATAM is used to minimise the manual input required by discovering services automatically and invoking them dynamically. It is also able to discover relations between service inputs and outputs using ontological descriptions.

The AUSTIN tool, developed by Lakhotia *et al.* [176] uses search-based software testing on C programs to help achieve branch coverage. AUSTIN (AUgmented Search-based TestINg) is a publicly available tool that uses a variation of Korel's Alternating Variable Method [177]. The tool is used to generate a set of input data for a given function to achieve some level of branch coverage for that function. It uses a HC algorithm (combined with a set of constraint solving rules for pointer type inputs) to work as a unit testing tool for C programs. The search is guided by an objective function that uses 2 metrics to evaluate an input against a target branch; approach level and branch distance. The tool is not able to generate meaningful inputs for strings, void and function pointers or union constructs but despite this, it has been applied to open source programs as well as real industrial code successfully.

Bunch was created by Mitchell and Gansner [88] to cluster the source level modules and dependencies of a software system into sub systems. It creates a system composition automatically by treating clustering as an optimisation problem. It has been used with C, C++ and Turing programs. To cluster the system into cohesive sub systems, the tool maps the modules and dependencies of the system to a Module Dependency Graph. Algorithms based on HC and GAs are then used to find the optimal partitions of the Module Dependency Graph that minimise coupling and increase cohesion. The algorithms determine the quality of the partitions found by measuring the interconnectivity between modules and intra connectivity of the dependencies and combining them into a Modularization Quality value. This is then used to find the optimal solutions where the partitions contain a trade-off between the 2 aspects. As it is not feasible to search through every available partition, more efficient search algorithms are used to find acceptable sub-optimal results in a more acceptable time frame.

CODe-Imp, developed by Moghadam and Ó Cinnéide [152], is an automated refactoring platform developed for Java. This platform uses abstract syntax trees to apply refactorings to Java programs. This, along with the ability to reverse the refactorings (SA demands that any refactoring can be reversed), allows the developers to use a variety of different search-based optimisation techniques with the platform (HC – both first-ascent and steepest-ascent, SA and GAs). There are a number of design level refactorings built as part of the

platform, used mainly to modify the object-oriented properties of the program. Examples of refactorings would be moving methods or attributes between classes, changing privacy settings in a class, or changing the heirarchy of the classes themselves. These refactorings can then be applied randomly and the program can be measured to determine whether its quality has been increased or decreased. Then the search-based algorithm used will find the optimal solution, based on the software metric(s) used. There are various metrics implemented in CODe-Imp to measure cohesion, coupling and other object-oriented properties.

CUTE and jCUTE are developed by Sen and Agha [178] and are concolic unit testing engines that use explicit path model-checking. There are 2 variations; CUTE is used for sequential C programs and jCUTE is used to test concurrent Java programs. The tools can be used to automatically generate test cases to improve test coverage while also avoiding redundant test cases as well as false warnings. The algorithm will be complete only if given an oracle that can solve the constraints in a program, and the length and number of paths is finite.

Dearthóir (Irish for Designer) was created by Ó Cinnéide along with O'Keefe [112] and serves a similar purpose to CODe-Imp. It is a prototype software engineering tool capable of improving a design with respect to a conflicting set of goals. Dearthóir uses SA to apply refactorings to Java code in a similar way to CODe-Imp. A lot of similar refactorings are used in the program (move methods between classes, make classes abstract or concrete etc.). Dearthóir can use a number of metrics to measure fitness by applying weights to the metrics related to their desired influence and combine them into a weighted sum. Refactorings can then be applied that preserve the behaviour of the program but modify the structure. Each time a refactoring is applied the quality of the program is measured and will determine the behaviour of the algorithm.

DPT, developed by Ó Cinnéide and Nixon [107], is used to apply design patterns to a Java program in an automated manner without changing the behaviours of the program. The tool uses a 4-tier design architecture to apply the desired pattern(s) to the program. The design patterns are composed of a number of minitransformations (minipatterns) which themselves are made up of various refactorings. The design patterns can then be applied using these minitransformations along with possibly some extra refactorings and/or helper

functions. The actual refactorings are applied to the program using an abstract syntax tree. Minitransformations will be reused in the program if they have already been composed to save time. DPT mostly implements creational patterns along with some structural and behavioural patterns. These have been taken from the Gang Of Four patterns proposed by Gamma *et al.*

Paolo Tonella [179] developed eToc (Evolutionary Testing Of Classes) for test data generation with a GA. The tool is developed for Java programs and the fitness function is used to prioritise test cases for selection. Fitness is determined in the algorithm by calculating how many new targets are covered by the test case. Superfluous test cases will be de-prioritised for the more useful ones. The GA can then evolve the method sequences along with their parameters as a means to achieve unit testing. A drawback of the tool is that new parameter values are only introduced into the population via the mutation operator, which randomly changes a parameter value within given bounds.

The Evolution Doctor framework was developed by Di Penta [149] to "cure" various maintenance problems in software. The framework improves a software system by detecting potential reorganisation opportunities and then performing reengineering actions to implement them. To monitor and analyse the system the framework looks for the presence of clones, unused objects and circular dependencies, as well as measuring various metrics along with library size, cohesion and coupling (static and dynamic). In order to improve the system, the framework will aim to remove or handle the detected clones, objects and dependencies. It will also reorganise the source files using Formal Concept Analysis.

EVOLVE is a tool proposed by Greer and Ruhe [180] to help prioritise software requirements for continuous planning with incremental software development. The tool uses a GA style approach to order the proposed requirements using the cost and priority for each requirement. These attributes along with certain precedents are used to assist the algorithm. The tool can order the requirements into numerous different releases with a certain cost limit. It can then be told if certain requirements must be grouped in the same release or if certain requirements need to be developed before certain other requirements. The cost and priority can be given in multiple different values to represent different stakeholders. These can then be given a weighting to correspond with their

influence, and an overall cost and priority can be calculated and normalised from these values. These 2 values for each requirement, along with the precedent attributes and coupling constraints will be used to determine the optimal ordering of requirements composed in the algorithm. Then the GA will use crossover and mutation to come up with a set of orderings for the requirements of the given number of releases that provide optimal balances between benefits. The tool has since been enhanced by Ruhe and Ngo-The into the EVOLVE+ tool [181].

EvoSUITE, developed by Fraser and Arcuri [182], is a tool that automatically generates test cases with assertions for classes written in Java code. EvoSUITE can be used as a command line tool or as an Eclipse plugin, producing test suites that achieve high code coverage and are as small as possible. It uses an EA and works to address the issue of the oracle (i.e. whether the results shown by the test cases capture the desired behaviour of the system). After the search a JUnit test suite will be produced for a given class (EvoSUITE will consider 1 class at a time for a given package).

FermaT was developed by Ward and was used by Fatiregun *et al.* [73] to automate the problem of finding good transformation sequences by using search techniques to improve a source code level metric. It can use random search, HC or GAs. The tool has the ability to use over 20 different source code transformations. The majority of these transformations are WSL-to-WSL transformations and are more explicit than traditional refactorings (e.g. else-if-to-elseif).

FLoPSy (search-based Floating Point constraint solving for Symbolic execution), developed by Lakhotia *et al.* [183], is an open source Pex extension that allows constraint solving with floating point operations. The extension has been implemented with a combination of evolutionary strategies and the Alternating Variable Method. Pex is a test input generator for .NET code, where test inputs are generated for parameterised unit tests, or for arbitrary methods of the code under test. If a constraint refers to a floating point operation, Pex performs a 2-phase solving approach. First, all floating point values are approximated by rational numbers, and it is checked whether the resulting constraint system is satisfiable. Second, 1 or more custom arithmetic solvers are invoked in order to

correct a previously computed model at all positions which depend on floating point constraints.

GenProg, developed by Goues *et al.* [104], is used to provide automated program repair for large software programs with reproducible software defects and with implemented version control. The tool uses GP (hence the name) to repair off-the-shelf C programs and can be used in parallel with cloud computing resources. The source code used must contain sufficient C source code and must have a reasonably sized test suite of viable test cases in order for GenProg to be able to use it. The program can then compare the modified program with a set of the positive test cases and all of the negative test cases (where "negative" relates to test cases that are failing due to the software defect) to measure if the solution is optimal and has kept the program functionality intact elsewhere. It can use previous versions of the software and code from elsewhere in the program to make these modifications. If the program passes all the negative test cases, then it will terminate. Otherwise, further restrictions of time or number of iterations can cause the tool to cease execution.

The inCode tool was developed by Ganea *et al.* [184] as an Eclipse plugin aimed to transform quality assessment and code inspections from a standalone activity, into a continuous, agile process, fully integrated in the development life-cycle. inCode identifies and locates specific design flaws as they appear.

Dudziak and Wloka [102] created the J/Art tool to detect structural weaknesses in Java code and for certain problems it can suggest the most beneficial restructuring required. Static analysis is used, along with the representation of abstract syntax trees, to detect numerous code smells with the tool, developed as an add-in for NetBeans[7]. It can perform limited restructuring capabilities for the design defects that are found using refactorings, although this is limited in comparison.

JDeodorant is an Eclipse plugin developed by Tsantalis *et al.* [150] to automatically identify and resolve type checking bad smells in Java source code. The plugin initially identified 2 different types of type checking bad smell and employs a different refactoring for each. The refactorings are "Replace

[7] https://netbeans.org/

Conditional with Polymorphism" and "Replace Type Code with State/Strategy". The tool has since implemented the ability to further identify instances of feature envy, duplicated code, long methods and god classes. The tool also has corresponding refactorings in order to resolve these smells ("Move Method", "Extract Clone", "Extract Method" and "Extract Class" respectively).

MiLu, developed by Jia and Harman [185], is a mutation testing tool designed for both first order and higher order testing in the C language. MiLu is a Chinese term named after a deer composed of 4 other animals. This name represents the rare but valuable nature of the program. It also relates to the mutation operators of nature that the program applies 4 times, as an example of a Higher Order Mutant. Mutation testing is used to measure the quality of a test set and design new software tests. It works by generating a set of faulty programs by making small changes to the original program via the mutation operator. Each faulty program or *mutant* will be run against a test set and depending on the result may *survive* or be *killed*. The adequacy level of the test set can then be measured by a mutation score that is computed in terms of the number of mutants killed by the test set.

SWAT (Search based Web Application Tester), created by Alshahwan and Harman [186], is a web application testing tool written in the PHP scripting language. The algorithm is based on HC using the Alternating Variable Method, but also uses constant seeding and Dynamically Mined Values. When a target branch is selected, the Alternating Variable Method is used to mutate each input in turn while other inputs remain fixed. When the selected mutation is found to improve fitness, the change in the same direction is accelerated. The tool is used to achieve branch coverage when testing web applications.

TrueRefactor was created by Griffith *et al.* [153] with the goal of improving the understandability, maintainability and reusability aspects of legacy software. It uses a GA on Java programs to detect lazy classes, large classes, long methods, temporary fields or instances of shotgun surgery. It contains a set of 12 refactorings (at class level, method level or field level) that are used to remove any code smells found. A set of pre conditions and post conditions are generated for each code smell to ensure that they can be resolved beforehand.

WISE was created by Feldt [187] as an interactive tool to help with software development and design. WISE uses biomemetic algorithms to support the development process. A prototype of the tool, WiseR, has been implemented in Ruby to focus on searching for tests. It is used to evolve test templates and generate tests that add interesting information to the system. It uses this to increase flexibility in the program.

Wrangler was developed by Li and Thompson [151] to support interactive refactorings in Erlang programs. Erlang is a functional language supporting modular programming and the Wrangler refactorings allow the improvement of modularity smells without dramatically changing the existing modular structure. Wrangler supports a variety of elementary structural refactorings, process refactorings and code smell inspection operations, as well as the ability to detect and eliminate duplicated code.

Appendix C – Other Relevant Software Tools

Along with the tools proposed from the literature, there are numerous open source tools available that can assist with automating refactoring, metrics calculations or providing search-based optimisation algorithms. Tables C.1-C.4 list the available tools for each of the 3 main components of SBSE. Commercial refactoring tools are listed as well as open source tools.

Table C.1 – List of Open Source Refactoring Tools

Software Package	**Programming Language**	**Purpose**
AutoRefactor	Java	Implements a set of common refactorings to Java code.
Coccinelle	C	Program matching and transformation engine.
Design Pattern Transformer	Java	Tool for implementing automated program transformations in Java.
Eclipse	Java	IDE containing a selection of manual refactorings.
EMF-Refactor	Java	Tool environment for metrics reporting, smell detection and refactoring.
JavaRefactor	Java	Plugin for jEdit to automatically refactor Java code.
JRefactory	Java	Applies manual refactorings to Java code.
PHP Refactoring Browser	PHP	Command line refactoring tool for PHP.
RefactorIT	Java	Provides automated refactorings, metrics, audits and corrective actions.
Tane Eclipse Refactorings	Java	Eclipse plugin designed to complement the refactorings supplied by Eclipse.
Transmogrify	Java	Java code analysis and manipulation architecture.
Xrefactory	Java	Plugin for jEdit providing code completion, source understanding tools and a refactoring browser.

Table C.2 – List of Commercial Refactoring Tools

Software Package	Programming Language	Purpose
CodeRush	.NET, C++, JavaScript	Refactoring and productivity plugin for Visual Studio.
Visual Assist	C/C++, C#	Productivity tool consisting of a number of refactorings.
Klocwork	C/C++	Source code analysis tool containing refactoring support.
JustCode	.NET	Visual Studio extension containing refactoring capability.
ReSharper	.NET, C++	Productivity tool for Visual Studio with refactoring support.
IntelliJ IDEA	Java	Commercial version of IDE with refactoring support.

Table C.3 – List of Open Source Search-Based Optimisation Tools

Name	Language
AntClique	C
AntMiner+	MATLAB
AntSolver	C
Beaver	C++
CPLEX	C/C++, C#, VB, Java, MATLAB, Python
Distributed Genetic Programming Framework	Java
ECJ	Java
Epsilon-MOEA	C++
EVA2	Java
Evolving Objects	C++
Example Genetic Algorithm	C++
Example MOPSO	C++
General Simulated Annealing Algorithm	MATLAB
Genetic Algorithm Library	C++
GUI Ant-Miner	Java
GUI-MOO	C++
HeuristicLab	Any
JAnnealer	Java
Java Ant Colony Systems Framework	Java
JCLEC	Java
Jenes	Java
Jenetics	Java
JGAP	Java
JMetal	Java
JNSGA-II	Java
JSwarm-PSO	Java
MAX-MIN Ant System	C++
Micro-GA For MOO	C++
MOEA Framework	Java
MOEA Library	C++
MOEA-D	C++
MOGA With Elitism	C++
mPOEMS	Java

Name	Language
Myra	Java
NSGA	C++
NSGA-II	C++
NSGA-III	C++
OpenTS	Java

Table C.4 – List of Open Source Metrics Tools

Name	Language
CKJM	Java
Eclipse Metrics	Java
inFusion	C/C++/Java
iPlasma	C++/Java
JCosmo	Java
Metrics	Java
Metrics (Extension)	Java
Sonar Qube	C/C++/Objective-C, C#/VB.NET, Java/JavaScript, Python, PHP, Flex/ActionScript, Erlang, Android, SQL, COBOL, XML, CSS

Appendix D – Papers

Tables D.1-D.4 list the papers reviewed in Chapter 2, with the title of each paper given along with the authors and year it was published (along with the citation it relates to for quick referencing). Table D.1 lists the papers related to SBSM that were the main subject of the literature review. Table D.2 lists the other relevant papers in the area of SBSE. Table D.3 lists the editorials for journals containing papers relating to SBSE and reports introducing tutorials and talks given on the research area. Table D.4 gives the other literature reviews that have been published in relation to SBSE or 1 of its sub areas.

Table D.1 – Papers on Search-Based Software Maintenance

Authors [Ref]	Year	Title
Amal *et al.* [160]	2014	On The Use Of Machine Learning And Search-Based Software Engineering For Ill-Defined Fitness Function: A Case Study On Software Refactoring
Bakar *et al.* [130]	2012	Applying Evolution Programming Search Based Software Engineering (SBSE) In Selecting The Best Open Source Software Maintainability Metrics
Di Penta [149]	2005	Evolution Doctor: A Framework To Control Software System Evolution
Fatiregun *et al.* [73]	2004	Evolving Transformation Sequences Using Genetic Algorithms
Ghaith and Ó Cinnéide [125]	2012	Improving Software Security Using Search-Based Refactoring
Griffith *et al.* [153]	2011	TrueRefactor: An Automated Refactoring Tool To Improve Legacy System And Application Comprehensibility
Harman [122]	2011	Refactoring As Testability Transformation
Harman and Tratt [6]	2007	Pareto Optimal Search Based Refactoring At The Design Level
Harman *et al.* [157]	2002	A New Representation And Crossover Operator For Search-Based Optimization Of Software Modularization
Harman *et al.* [133]	2012	Dynamic Adaptive Search Based Software Engineering
Harman *et al.* [132]	2013	Dynamic Adaptive Search Based Software Engineering Needs Fast Approximate Metrics
Kessentini *et al.* [134]	2011	Design Defects Detection And Correction By Example
Kessentini *et al.* [135]	2011	Example-Based Design Defects Detection And Correction
Kessentini *et al.* [137]	2012	What You Like In Design Use To Correct Bad-Smells

Authors [Ref]	Year	Title
Koc *et al.* [117]	2012	An Empirical Study About Search-Based Refactoring Using Alternative Multiple And Population-Based Search Techniques
Li and Thompson [151]	2010	Refactoring Support For Modularity Maintenance In Erlang
Mkaouer *et al.* [25]	2014	High Dimensional Search-Based Software Engineering: Finding Tradeoffs Among 15 Objectives For Automating Software Refactoring Using NSGA-III
Mkaouer *et al.* [42]	2014	Many-Objective Software Remodularization Using NSGA-III
Mkaouer *et al.* [144]	2014	Software Refactoring Under Uncertainty: A Robust Multi-Objective Approach
Mkaouer *et al.* [146]	2015	On The Use Of Many Quality Attributes For Software Refactoring: A Many Objective Search-Based Software Engineering Approach
Mkaouer *et al.* [145]	2016	A Robust Multi-Objective Approach To Balance Severity And Importance Of Refactoring Opportunities
Moghadam and Ó Cinnéide [152]	2011	Code-Imp: A Tool For Automated Search-Based Refactoring
Moghadam and Ó Cinnéide [120]	2012	Automated Refactoring Using Design Differencing
Morales [154]	2015	Towards A Framework For Automatic Correction Of Anti-Patterns
Morales *et al.* [123]	2016	Finding The Best Compromise Between Design Quality And Testing Effort During Refactoring
Morales *et al.* [155]	2016	On The Use Of Developers' Context For Automatic Refactoring Of Software Anti-Patterns
Ó Cinnéide and Nixon [109]	1999	A Methodology For The Automated Introduction Of Design Patterns
Ó Cinnéide and Nixon [107]	1999	Automated Application Of Design Patterns To Legacy Code
Ó Cinnéide *et al.* [124]	2011	Automated Refactoring For Testability
Ó Cinnéide *et al.* [126]	2012	Experimental Assessment Of Software Metrics Using Automated Refactoring
Ó Cinnéide *et al.* [127]	2016	An Experimental Search-Based Approach To Cohesion Metric Evaluation
Ó Cinnéide [110]	2000	Automated Refactoring To Introduce Design Patterns
O'Keeffe and Ó Cinnéide [111]	2003	A Stochastic Approach To Automated Design Improvement
O'Keeffe and Ó Cinnéide [112]	2004	Towards Automated Design Improvement Through Combinatorial Optimisation
O'Keeffe and Ó Cinnéide [113]	2006	Search-Based Software Maintenance
O'Keeffe and Ó Cinnéide [118]	2007	Automated Design Improvement By Example
O'Keeffe and Ó Cinnéide [115]	2007	Getting The Most From Search-Based Refactoring
O'Keeffe and Ó Cinnéide [116]	2007	Search-Based Refactoring: An Empirical Study
O'Keeffe and Ó Cinnéide [114]	2008	Search-Based Refactoring For Software Maintenance
Ouni *et al.* [138]	2012	Search-Based Refactoring: Towards Semantics Preservation
Ouni *et al.* [136]	2013	Maintainability Defects Detection And Correction: A Multi-Objective Approach

Authors [Ref]	Year	Title
Ouni *et al.* [139]	2013	Search-Based Refactoring Using Recorded Code Changes
Ouni *et al.* [141]	2013	The Use Of Development History In Software Refactoring Using A Multi-Objective Evolutionary Algorithm
Ouni *et al.* [37]	2015	Improving Multi-Objective Code-Smells Correction Using Development History
Ouni *et al.* [147]	2015	Prioritizing Code-Smells Correction Tasks Using Chemical Reaction Optimization
Ouni *et al.* [140]	2016	Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study
Pérez *et al.* [143]	2013	A Proposal For Fixing Design Smells Using Software Refactoring History
Qayum and Heckel [159]	2009	Local Search-Based Refactoring As Graph Transformation
Seng *et al.* [121]	2006	Search-Based Determination Of Refactorings For Improving The Class Structure Of Object-Oriented Systems
Simons *et al.* [129]	2015	Search-Based Refactoring: Metrics Are Not Enough
Trifu *et al.* [148]	2004	Automated Design Flaw Correction In Object-Oriented Systems
Tsantalis *et al.* [150]	2008	JDeodorant: Identification And Removal Of Type-Checking Bad Smells
Van Belle and Ackley [156]	2002	Code Factoring And The Evolution Of Evolvability
Veerappa and Harrison [128]	2013	An Empirical Validation Of Coupling Metrics Using Automated Refactoring
Vivanco and Pizzi [7]	2004	Finding Effective Software Metrics To Classify Maintainability Using A Parallel Genetic Algorithm
Wang *et al.* [142]	2015	On The Use Of Time Series And Search Based Software Engineering For Refactoring Recommendation
White *et al.* [158]	2008	Searching for Resource-Efficient Programs: Low-Power Pseudorandom Number Generators

Table D.2 – Papers on General Aspects of Search-Based Software Engineering

Authors [Ref]	Year	Title
Allman [69]	2012	Managing Technical Debt
Barros and Dias Neto [64]	2011	Threats To Validity In Search-Based Software Engineering Empirical Studies
Brown *et al.*[68]	2010	Managing Technical Debt In Software-Reliant Systems
Chatzigeorgiou *et al.* [70]	2015	Estimating The Breaking Point For Technical Debt
Clarke *et al.* [58]	2003	Reformulating Software Engineering As A Search Problem
De Freitas and De Souza [65]	2011	Ten Years Of Search Based Software Engineering: A Bibliometric Analysis
De Souza *et al.* [76]	2010	The Human Competitiveness Of Search Based Software Engineering
Fatiregun *et al.* [72]	2003	Search Based Transformations
Harman and Clark [59]	2004	Metrics Are Fitness Functions Too
Harman and Jones [1]	2001	Search-Based Software Engineering
Harman [61]	2007	Search Based Software Engineering For Program Comprehension
Harman [60]	2007	The Current State And Future Of Search Based Software Engineering
Harman [62]	2010	Why The Virtual Nature Of Software Makes It Ideal For Search Based Optimization
Harman [63]	2011	Software Engineering Meets Evolutionary Computation
Jiang *et al.* [75]	2007	A Foundational Study On The Applicability Of Genetic Algorithms To Software Engineering Programs
Jiang [74]	2006	Can The Genetic Algorithm Be A Good Tool For Software Engineering Searching Problems
Morgenthaler *et al.* [71]	2012	Searching For Build Debt: Experiences Managing Technical Debt At Google
Vergilio *et al.* [66], Colanzi *et al.* [67]	2011/ 2013	Search Based Software Engineering: A Review From The Brazilian Symposium On Software Engineering/Search Based Software Engineering: Review And Analysis Of The Field In Brazil

Table D.3 – Editorials and Reports

Authors [Ref]	Year	Title
Di Penta and Poulding [188]	2011	Introduction To The Special Issue On Search Based Software Engineering
Di Penta *et al.* [189]	2008	Special Issue On Search-Based Software Maintenance
Gutjahr and Harman [190]	2008	Search-Based Software Engineering
Harman and Jones [97], [98]	2001	The SEMINAL Workshop: Reformulating Software Engineering As A Metaheuristic Search Problem/Software Engineering Using Metaheuristic Innovative Algorithms: Workshop Report
Harman and Mansouri [193]	2010	Search Based Software Engineering: Introduction To The Special Issue Of The IEEE Transactions On Software Engineering
Harman and Wegener [194]	2004	Getting Results From Search-Based Approaches To Software Engineering
Harman *et al.* [195]	2005	Guest Editorial: Special Issue On Software Maintenance And Evolution
Harman [196]	2006	Search-Based Software Engineering For Maintenance And Reengineering
Harman [197]	2006	The Importance Of Metrics In Search Based Software Engineering
Harman [198]	2012	Overview Of TASE 2012 Talk On Search Based Software Engineering
Harman [199]	2013	Software Engineering: An Ideal Set Of Challenges For Evolutionary Computation
Ó Cinnéide and Cohen [200]	2013	Introduction To The Special Issue On Search Based Software Engineering

Table D.4 – Literature Reviews

Authors [Ref]	Year	Title
Ferrucci *et al.* [162]	2014	Search-Based Software Project Management
Harman and McMinn [201]	2010	A Theoretical And Empirical Study Of Search-Based Testing: Local, Global And Hybrid Search
Harman *et al.* [163], [202]	2009/ 2012	Search Based Software Engineering: A Comprehensive Analysis And Review Of Trends Techniques And Applications/Search Based Software Engineering: Trends, Techniques And Applications
Harman *et al.* [161]	2012	Search Based Software Engineering: Techniques, Taxonomy, Tutorial
McMinn [203]	2004	Search-Based Software Test Data Generation: A Survey
Pitangueira *et al.* [204]	2013	A Systematic Review Of Software Requirements Selection And Prioritization Using SBSE Approaches
Räihä [8], [205]	2009/ 2010	An Updated Survey On Search Based Software Engineering/A Survey On Search Based Software Engineering
Sayyad and Ammar [206]	2013	Pareto-Optimal Search-Based Software Engineering (POSBSE): A Literature Survey