View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Crossref

Mohan and Greer *Journal of Software Engineering Research and Development* (2018) 6:3
DOI 10.1186/s40411-018-0046-4

Journal of Software Engineering Research and Development

REVIEW

Open Access

A survey of search-based refactoring for software maintenance

Michael Mohan* and Des Greer

* Correspondence: mmohan03@qub.ac.uk
Queen's University Belfast, Belfast, Northern Ireland, UK

Abstract

This survey reviews published materials related to the specific area of Search-Based Software Engineering that concerns software maintenance and, in particular, refactoring. The survey aims to give a comprehensive review of the use of search-based refactoring to maintain software. Fifty different papers have been selected from online databases to analyze and review the use of search-based refactoring in software engineering. The current state of the research is analyzed and patterns in the studies are investigated in order to assess gaps in the area and suggest opportunities for future research. The papers reviewed are tabulated in order to aid researchers in quickly referencing studies. The literature addresses different methods using search-based refactoring for software maintenance, as well as studies that investigate the optimization process and discuss components of the search. There are studies that analyze different software metrics, experiment with multi-objective techniques and propose refactoring tools for use. Analysis of the literature has indicated some opportunities for future research in the area. More experimentation of the techniques in an industrial environment and feedback from software developers is needed to support the approaches. Also, recent work with multi-objective techniques has shown that there are exciting possibilities for future research using these techniques with refactoring. This survey is beneficial as an introduction for any researchers aiming to work in the area of Search-Based Software Engineering with respect to software maintenance and will allow them to gain an understanding of the current landscape of the research and the insights gathered.

Keywords: Review, Search-based software engineering, Software maintenance, Refactoring, Software metrics, Metaheuristic search, Multi-objective optimization

1 Introduction

SEARCH-Based Software Engineering (SBSE) concerns itself with the resolution of software engineering optimization problems by restructuring them as combinatorial optimization problems. The topic has been addressed and researched in a number of different areas of the software development life cycle, including requirements optimization, software code maintenance and refactoring, test case optimization and debugging. While the area has existed since the early 1990s and the term "search-based software engineering" was originally coined by (Harman & Jones, 2001), most work in this area has been recent with the number of published papers on the topic exploding in the last number of years (De Freitas & De Souza, 2011). Many of the papers in the area of SBSE propose using an automated approach to increase the efficiency of the

Springer Open

© The Author(s). 2018 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

area of the software process looked at. Of the papers published concerning SBSE, a relatively small amount is related to software maintenance. This is despite the fact that it is estimated that the maintenance process takes 70–75% of development effort (Bell, 2000; Pressman & Maxim, 2000) of the development process.

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 "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. This costs the developer time as the overall development time for functionality is offset by this obligatory cleaning up of code.

SBSE has been used to automate this process, thus decreasing time taken to restructure a program. SBSE can be applied to software maintenance by applying refactorings to the code to reduce technical debt. 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 is an uncertain aspect and can be subjective and so can be done in a variety of different ways. The developer needs to devise a heuristic, or most 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 consist of 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 (Martin, 2003). The developer then needs to devise a number of changes that can be made to the software to refactor it in order to enforce the heuristics. A refactoring action modifies the structure of the code without changing the external functionality of the program. When the refactorings are applied to the software they may either improve or impair the quality, but regardless, they act as tools used to modify the solution.

The refactorings are applied stochastically to the original software solution and then the software is measured to see if the quality of the solution has improved or degraded. A "fitness function" combining one or more software metrics is generally used to measure the quality. These metrics are very important as they heavily influence how the software is modified. 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 affect on the outcome. The CK (Chidamber & Kemerer, 1994) and QMOOD (Quality Model for Object-Oriented Design) (Bansiya & Davis, 2002) 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. One common approach is to give weights to the metrics on which heuristics are more important to maintain and combine them into one weighted sum (although this weighting process is often subjective). This weighting may be inappropriate 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 (Harman & Tratt, 2007) 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 used.

Using the metric or metrics to give an overall fitness value, the fitness function of the search-based technique measures the quality of the software solution and generates a numerical value to represent it. 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 applied, this becomes the new solution to compare against. If not, the changes are discarded and the previous solution is kept. This approach is followed over a number of iterations, causing the software solution to gradually improve 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 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 a more optimal result.

This survey aims to review and analyze papers that use search-based refactoring. We apply certain inclusion and exclusion criteria to find relevant papers across a number of research databases. We highlight different aspects of the research to inspect and analyze through a set of research questions. We also identify related work and highlight the differences between those papers and this survey (e.g. many of the related reviews investigate other areas of SBSE or different aspects of refactoring like UML model refactoring or refactoring opportunities. One study that looks at search-based refactoring is investigated in more detail in Section 7.1 to compare similar aspects of the analysis conducted in the paper). Each paper is reviewed and summarised to give an overview of any experiments conducted and the results gained. The overview of the papers is organised into five different groups to cluster together related studies. The papers are then analyzed to address the research questions outlined and derive similarities and differences between the studies. Various different aspects of the papers are analyzed as well as research gaps and possible areas for future work in the area.

The remainder of the survey is structured as follows. Section 2 gives an overview of some of the more common search techniques used for refactoring in SBSE. Section 3 gives an outline of how the survey is conducted, along with an outline of aspects to be measured and analyzed, and introduces the research questions. Section 4 gives a synopsis of the analyzed papers. Section 5 analyzes the papers reviewed and measures patterns that can be derived from the work conducted in the literature. Section 6 discusses and addresses the research questions outlined in Section 3. Section 7 gives an overview of related work along with a discussion of the differences and similarities. Section 8 looks at validity threats in the survey and Section 9 concludes the survey.

2 Search techniques

There are numerous different metaheuristic algorithms available to use in the SBSE field. These methods are used to automate search-based problems through gradual quality increases. Random search is used as a benchmark for most search-based metaheuristic algorithms to compare against. Although most metaheuristics use a non-deterministic approach to making choices, 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. Below, the most common metaheuristic algorithms used in the literature are discussed:

2.1 Hill climbing

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. A change is then made, and the fitness function is used to compare the two 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 the same functionality but a better structure. This is considered a fast algorithm in relation to the other metaheuristic choices but, as with other local search algorithms, it has the risk of being restricted to local optima. The algorithm may "peak" at a less optimal solution (akin to reaching a peak after climbing a hill). There are two main types of HC search algorithm that differ in one aspect. First-ascent HC is the simpler version of the algorithm, whereas steepest-ascent HC has a slightly more sophisticated search method and is a superior choice for quality. Other variations are stochastic HC (neighbors are chosen at random and compared) or random-restart HC (algorithm is restarted at different points to explore the search space and improve the local optima reached). HC is one of the more common search algorithms used in SBSE, and has similarities to other search techniques. The HC technique may not produce solutions as effective as some others do, but it does tend to find a suitable solution faster and more consistently (O'Keeffe & Cinnéide, 2006).

2.2 Simulated annealing

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 introduces 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 optimization 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 neighborhood in which to optimize. This is then generally decreased gradually

until it is negligible. 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". Although the SA process may come up with a better solution compared to the HC process, HC is a lot more reliable as the SA process may struggle to settle on a solution.

2.3 Genetic algorithms

Genetic algorithms (GAs) are a class of evolutionary algorithms (EAs) that, much like SA, mimic a process used elsewhere in science, namely the reproduction and mutation processes in genetics and natural selection. GAs use a fitness function to measure the quality among a number of different solutions (known as "genes") and prioritize them. At each generation (i.e. each iteration of the search), the genes are measured to determine which are the "fittest". Each 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. With this selection, two steps are used to create the new generation. First, a crossover operator is used to create the child solution(s) 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 solution(s) have been created, the second step is mutation. Again, the mutation implementation depends on the GA written. The mutation is used to provide random changes in the solutions to maintain variation in the selection of solutions and prevent convergence. After mutation is applied to a selection of the child solutions, the newly created solutions 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 this ensures that the weakest solutions are weeded out of the gene pool each generation. This process is repeated until a termination condition is reached.

2.4 Multi-objective evolutionary algorithms

When refactoring a software project, as with other areas of software engineering, there are likely numerous conflicting objectives to address and optimize. A multi-objective algorithm can be used to consider the objectives independently instead of having to combine them into one overarching objective to improve. There are numerous EAs available that are used for multi-objective problems, known as multi-objective evolutionary algorithms (MOEAs). The downside to using multi-objective algorithms for software refactoring over the mono-objective metaheuristic algorithms 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. Whereas a single-objective EA can rank the final population of solutions by a single fitness value, there may be numerous possible choices in the MOEA population depending on which objective fitness is more important. On the other hand, this gives the user multiple options depending on their desire or the situation.

Most MOEAs use Pareto dominance (Coello Coello, 1999) in order to restrict the population of solutions generated. If, for a solution, at least one objective of that

solution is better than in another solution and none of the remaining objectives are worse, that solution is said to dominate the other solution. Therefore, a solution is non-dominated unless another solution in the population dominates it. When Pareto dominance is used to generate an optimal set of solutions, they can be inspected on a Pareto front. When the number of objectives is less than or equal to three, the non-dominated solutions can be visualized on the Pareto front. This allows the user to easily visualize the state of the solutions according to each independent objective and choose the most suitable one along the Pareto front. Multi-objective algorithms that are designed to handle more than three objectives are generally referred to as many-objective algorithms (Deb & Jain, 2013). These tend to avoid using only Pareto dominance as it can have difficulty successfully handling more than three objectives (Jain & Deb, 2014). As the amount of objectives to measure increases, it becomes more difficult to rank the solutions into different fitness fronts (as an increasing amount become non-dominated, which results in many of the solutions being given the same fitness rank). When this happens, the multi-objective algorithm becomes less useful at discerning better populations of solutions. Table 1 lists MOEAs that use Pareto dominance to choose solutions. For further information, a survey of MOEAs is given by (Coello Coello, 1999).

3 Survey outline

In this survey, we aim to aggregate information and patterns about the research that have been conducted related to search-based refactoring for software maintenance and the trends that have been uncovered. As such, we aim to answer the following question:

"To what extent has search-based refactoring been studied in software maintenance?"

In order to address this, the following research questions have been introduced:

- **RQ1:** How many papers were published per year?
- **RQ2:** What are the most common methods of publication for the papers?
- **RQ3:** Who are the most prolific authors investigating search-based refactoring in software maintenance?
- **RQ4:** What types of studies were used in the papers?
- **RQ5:** What refactoring approaches were used in the literature?
- **RQ6:** What search techniques were used in the refactoring studies?
- **RQ7:** What types of programs were used to evaluate the refactoring approaches?

Table 1 MOEAs That Use Pareto Dominance

MOEA	Full Name	Developers
DMOEA	Dynamic Multiobjective Evolutionary Algorithm	Yen and Lu (2003)
M-PAES	Memetic-Pareto Archive Evolutionary Stategy	Knowles and Corne (2000)
NGPA	Niched Pareto Genetic Algorithm	Horn et al. (1994)
NSGA-II	Nondominated Sorting Genetic Algorithm II	Deb et al. (2002)
PAES	Pareto Archive Evolutionary Strategy	Knowles and Corne (2000)
PDE	Pareto-frontier Differential Evolution	Abbass et al. (2001)
PESA	Pareto Envelope-based Selection Algorithm	Corne et al. (2000)
SPEA	Strength Pareto Evolutionary Algorithm	Zitzler and Thiele (1999)
SPEA2	Strength Pareto Evolutionary Algorithm 2	Zitzler et al. (2001)

- **RQ8:** What tools were used for refactoring?
- **RQ9:** What types of metrics were used in the studies?
- **RQ10:** What are the gaps in the literature and available research opportunities in the area?

To answer these questions, we analyze and discuss similarities and patterns in the studies investigated. The research questions are addressed in Section 5 by inspecting the various aspects queried such as search techniques used, popular authors, relevant conferences and journals and open source programs used in the studies.

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 two 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 the alternative of looking only within the article title or elsewhere. The most recent search was implemented in September 2016. The time period for the search in which the papers were published was unrestricted, therefore the period is for papers published up to and including 2016. The amount of papers found in each search repository is given in Table 2. Of the papers found with the search, the results were analyzed and reduced to only include the papers that were relevant to software maintenance and involved one of the following:

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

Likewise, the following papers were excluded:

- Papers that involved defect detection but not resolution.
- Literature reviews, theses, abstracts, tutorials, reports or posters.
- Papers that were written in a language other than English.

Although certain related areas captured in the search (such as modelling, defect detection, software architecture, testing etc.) could have been excluded from the search to reduce the number of hits, the papers were analyzed manually using the inclusion and exclusion criteria due to the similar nature of many of the areas in order to ensure that relevant papers weren't lost from the review. Of the papers analyzed, 34 were found to

Table 2 Amount of Results in Each Repository

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

be applicable. Accompanying this, 16 other relevant papers were found on Google Scholar and the IEEE database by analyzing references, researcher profiles and by discussion with other researchers, as well as conducting similar searches. Overall, the number of papers reviewed came to 50. Table 10 in the appendix gives a list of the papers, as well as the authors and year published. Extra information about the papers surveyed is given in the Additional file 1, as well as a list of literature reviews related to SBSE in general.

4 Refactoring in search-based software engineering

The different papers captured in the search are analyzed below. They have been further categorised into five subsections to capture commonly recurring areas, although there may be some overlap between a paper in one section with another section.

4.1 Refactoring to improve software quality

Ó Cinnéide and Nixon (1999a) developed a methodology to refactor software programs 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 the desired pattern (in the paper 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 analyzed the (Gamma et al., 1994) patterns to determine whether a suitable transformation could be built with the applicable mini transformations. They found that while the tool generally worked well for the creational patterns, structural patterns and behavioral patterns caused problems. In a different paper (Cinnéide & Nixon, 1999b), more detail was given on the tool and how it can be used to apply the Factory Method pattern, and in another subsequent paper (Cinnéide, 2000), Ó Cinnéide defined further steps of work to test the applicability of the tool. He defined plans to apply the patterns to production software to test whether behavior is truly preserved and to create a tool to objectively measure how suitably the pattern has been applied to the software.

O'Keeffe and Ó Cinnéide (2003) continued to research in the area of SBSE relating to software maintenance by developing a tool called Dearthóir. They introduce Dearthóir as a prototype tool used to refactor Java code automatically using SA. The tool used two refactorings, "pullUpMethod" and "pushDownMethod" to modify the hierarchical structure of the target program. Again, the refactorings must preserve the behavior 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 analyze the object-oriented structure of the program. The metrics, "availableMethods" and "methodsInherited" 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 minimize code duplication.

Further work (O'Keeffe & Cinnéide, 2004) introduced more refactorings and different metrics to the tool. Along with method movement refactorings the ability to change a class between abstract and concrete was introduced and to extract or collapse a subclass from an abstract class, as well as the ability to change the position of a class in the hierarchy of the class design. A method was introduced to choose appropriate metrics to use in the metrics suite of the tool. The metrics used measured the methods and classes of the solution, counting the number of rejected, duplicate and unused methods as well as the number of featureless or abstract classes. Due to the possibility for the metrics to conflict with each other they were then given dependencies and weighted according to the authors' judgment, as outlined in the method detailed before. Another case study was used to detail the action 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.

O'Keeffe and Ó Cinnéide continued to work on automated refactoring by developing the Dearthóir prototype into the CODe-Imp platform (Combinatorial Optimisation for Design Improvement). They introduced it initially as a prototype automated design-improvement tool (O'Keeffe & Cinnéide, 2006) using Java 1.4 source code as input. Like Dearthóir, CODe-Imp uses abstract syntax trees to apply refactorings to a previously designed solution, but it has been given the ability to implement HC (first-ascent or steepest-ascent) as well as SA. They based the set of metrics used in the tool on the QMOOD model of software quality (Bansiya & Davis, 2002). 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 each search technique is at refactoring software. The reusability function was found to not be suitable to the requirements of search-based refactoring due to the introduction of a large number of featureless classes. The other two 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. SA produced the greatest quality improvements in some cases whereas first-ascent hill-climbing generally produced quality improvements for the least computational expenditure. They further expanded on this work (O'Keeffe & Cinnéide, 2008a) 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 six additional refactorings. Similar results were found with the reusability function found to be unsuitable for search-based refactoring and all of the available search techniques found to be effective.

They subsequently (O'Keeffe & Cinnéide, 2007a) used the CODe-Imp platform to conduct an empirical comparison of three methods of metaheuristic search in search-based refactoring; multiple-ascent (as well as steepest-ascent) HC, SA and a GA. To conduct the comparison, four Java programs were taken from SourceForge and Spec-Benchmarks and the mean quality change was measured across the program solutions

for each of the three metaheuristic techniques. These results were then normalized for each metaheuristic technique and then compared against each other. They analyzed the results to conclude 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 (O'Keeffe & Cinnéide, 2008b) with a larger set of input programs, greater number of data points in each experiment and more detailed discussion of results and conclusions.

At a later point, Koc et al. (2012) also compared metaheuristic search techniques using a tool called A-CMA. They compared five different search techniques by using them to refactor five different open source Java projects and one student project. The techniques used were HC (steepest descent, multiple steepest descent and multiple first descent), SA and artificial bee colony (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. The authors suggested that the effectiveness of these techniques may be due to their ability to expand the search horizon to find higher quality solutions.

Mohan, Greer and McMullan (Mohan et al., 2016) adapted the A-CMA tool to investigate different aspects of software quality. They used combinations of metrics to represent three quality factors; abstraction, coupling and inheritance. They then constructed an experimental fitness function to measure technical debt by combining relevant metrics with influence from the QMOOD suite, as well as the SOLID principles of object-oriented design. The technical debt function was compared against each of the other quality factors by testing them on six different open source systems with a random search, HC and SA. The technical debt function was found to be more successful than the others, although the coupling function was also found to be useful. Of the three searches used, SA was the most effective. The individual metrics of the technical debt function were also compared to deduce which were more volatile.

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 (O'Keeffe & Cinnéide, 2007b). The QMOOD metrics suite was used to compare against previous results, and an overall fitness value was derived from the sum of 11 different metrics. A dissimilarity function was evaluated to measure the absolute differences between the metric values of the programs tested, where a lower dissimilarity value meant the programs were more similar. CODe-Imp was then used to refactor the input program to reduce its dissimilarity value to the target program. This was tested with three open source Java programs, with six different tests overall (testing each program against the other two). Two of the programs had been 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 (2011) 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 three categories; method-level, field-level and class-level. The number of metrics had also been expanded to 28, measuring mainly 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.

Moghadam and Ó Cinnéide used CODE-Imp along with JDEvAn (Xing & Stroulia, 2008) to attempt to refactor code towards a desired design using design differencing (Moghadam & Cinnéide, 2012). The JDEvAn tool is used to extract the UML models of two 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. Six open source examples were used to test the efficiency of these tools to create the desired solutions. The number of refactorings detected and applied in each program using the above approach were collected, and in each case a high percentage of refactorings were shown to have been applied.

Seng, Stammel and Burkhart (Seng et al., 2006) 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 was a list of refactorings the software engineer could apply to improve a set of metrics. They used class level refactorings, noting the difficulty of providing refactorings of this type that were behavior preserving. 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 reorganize manually misplaced methods, and it was successfully able to suggest that the methods are moved back to their original position.

Harman and Tratt (Harman & Tratt, 2007) 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 visualize 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.

4.2 Refactoring for testability

Harman (Harman, 2011) 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 the two areas (testing and maintenance) of SBSE. As testability transformation uses refactorings to modify the structure of a program the same technique

can be used for program maintenance, although the two aims may be conflicting. Here a testability refactoring will refer to a process that satisfies both objectives. Harman mentioned that these two possibly conflicting objectives form a multi-objective scenario. He explained that the problem would be well suited to Pareto optimal search-based refactoring and also mentioned a number of ways in which testability transformation may be suited to testability refactoring.

Morales et al. (2016) investigated the use of a multi-objective approach that takes into consideration the testing effort on a system. They used their approach to minimize the occurrence of five well known anti-patterns (i.e. types of design defect), while also attempting to reduce the testing effort. Three different multi-objective algorithms were tested and compared; NSGA-II, SPEA2 and MOCell. This approach was tested on four open source systems. Of the three options, MOCell was found to be the metaheuristic that provided the best performance.

Ó Cinnéide, Boyle and Moghadam (2011) 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. They refactored a small Java program with cohesive defects introduced. Ten volunteers with varying years of industrial experience constructed test cases for the 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. The programmers surveyed also mentioned the use of modern IDE's helped to reduce the issues with the initial code and alleviated any predicted problems with producing test cases for the program in this state.

4.3 Testing metric effectiveness with refactoring

Ghaith and Ó Cinnéide (2012) 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 16 metrics on an example Java application by using them separately at first. After determining that only four of the metrics were affected with the refactoring selection available, they were combined together to form a fitness function to represent security. To avoid the problems related to using a weighted sum approach to combining the metrics, they instead used a Pareto optimal approach. This ensured that no refactoring would be chosen that would cause a decrease in any of the individual metrics in the function. The function was then tested on the Java program using first-ascent HC, steepest-ascent HC and SA. The results for the three searches were mostly identical except that SA caused a higher improvement in one of the metrics. Conversely, the SA solution entailed a far larger number of refactorings than the other two options (2196 compared to 42 and 57). The effectiveness of these metrics was also analyzed 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. (2012) conducted an investigation to measure and compare different cohesion metrics with the help of the CODe-Imp platform. Five popular cohesion metrics were used across eight different real world Java programs to measure the volatility of the metrics. It was found that the five 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, LSCC and TCC (Tight Class Cohesion), were then studied in more detail to determine where the contradictions were in the code that caused the conflicts. Different variations of the metrics were used to compare them in two different ways This study was extended (Cinnéide et al., 2016) to use 10 real world Java programs. Two new techniques, exploring areas referred to as Iterative Refactoring Agreement/Disagreement and Gap Opening/Closing Refactoring, were used to compare the metrics and the number of metric pairs compared was increased to 90 pairs. Among the metrics compared, LSCC was found to be the most representative, while SCOM (Sensitive Class Cohesion) was found to be the least.

Veerappa and Harrison (2013) 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 four standard coupling metrics and to compare the metrics with each other. Eight open source Java projects were used, with all but one of the programs being the same as those used in Ó Cinnéide et al.'s experiment. To measure volatility, they calculated the percentage of refactorings that caused a change in the metrics, and from these a mean value was calculated across the eight projects. The amount of spread between these values was calculated for each metric using standard deviation, as well as the correlation values between each metric. This experiment resulted in less divergence between metrics, with only 7.28% of changes in direct conflicting, but in 55.23% of cases the changes were dissonant, meaning that there was a larger chance that 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 cohesion will continue to increase as the coupling decreases, minimizing the effectiveness of the changes.

Simons, Singer and White (2015) compared metric values with professional opinions to deduce whether metrics alone are enough to helpfully refactor a program. They constructed a number of software examples and used a selection of metrics to measure them. A survey was then conducted and responded to by 50 experienced software engineers. They were asked on their opinion of the quality of the solutions by asking whether they agree or disagree that a solution was reusable, flexible or understandable. The metrics were corresponded to the quality attributes 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 two, leading the authors to suggest that metrics alone are insufficient to optimize software quality as they do not fully capture the judgments of human engineers when refactoring software.

Vivanco and Pizzi (2004) 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 metric. Firstly, they asked an experienced software architect to rank the 366 components of a software system in difficulty, from 1 to 5. The GA was then run for the set of metrics in sequential and parallel, using C++ for the GA and MPI to implement the parallel improvements. 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. (2012a) attempted to outline a set of guidelines to select the best metrics for measuring maintainability in open source software. An EA was used to optimize and rank the set of metrics, which were listed in previous work (Bakar et al., 2012b). An analysis was conducted to validate the quality model using the CK metric suite (Chidamber & Kemerer, 1994) of Object-Oriented Metrics (also known as MOOSE – Metrics for Object-Oriented Software Engineering). The CKJM tool proposed by (Spinellis, 2005) was used to calculate the values of the CK metrics for the open source software under inspection. These values were then used in the EA as ranking criteria in selecting the best metrics to measure maintainability in the software product. This proposed approach had not yet been empirically validated, and had presented the outcome of ongoing research.

Harman, Clark and Ó Cinnéide (2013) 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 (e.g. if a mobile device is used) mean limited time or power, 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 (Harman et al., 2012).

4.4 Refactoring to correct software defects

Kessentini et al. (2011) 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 blob, spaghetti code or functional decomposition design defects. Before the GA was used, a GP approach experimented with different rules than can 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 that reduce the number of design defects in the program. The approach was compared against a different rules-based approach to defects detection with four open source Java programs and was found to be more precise with the design defects found.

Further work with this approach to design smell (defect) correction was then investigated (Kessentini et al., 2011; Ouni et al., 2013; Kessentini et al., 2012). First,

(Kessentini et al., 2011) extended the experimental code base to six different open source Java programs, with the results further supporting the approach. Ouni et al. (2013) replaced the GA used in the code smell correction approach with a multi-objective GA (NSGA-II). They used the previous objective function to minimize design defects as one of two 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, Mahaouachi and Ghedira (2012) 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 used adapted the Needleman-Wunsch alignment algorithm (a dynamic programming algorithm used to efficiently find similar regions between two sequences of DNA, RNA or protein. It can be used to efficiently compare code fragments) to increase the similarity between the two sets of code, allowing the derived refactoring sequences to remove code smells.

Ouni et al. (2012) 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 new approach used two main methods to measure semantic similarity. The first method measures vocabulary based similarity by inspecting the names given to the software elements and comparing them using cosine similarity. The other method measures the dependencies between objects in the program by calculating the shared method calls of two objects and the shared field accesses and combining them into a single function. An overall objective for semantics similarity is derived from these measures by finding the average, and this is then used to help the NSGA-II algorithm find more meaningful solutions. These solutions were analyzed manually to derive the percentage of meaningful refactorings suggested. The results across two different open source programs were then compared against a previous mono-objective and previous multi-objective approach, and, while the number of defects resolved was moderately smaller, the meaningful refactorings were increased.

Ouni, Kessentini and Sahraoui (2013) 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 three 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 on three open source Java projects 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.

They further explored this approach (Ouni et al., 2013) by analyzing co-change that identified how often two objects in a project were refactored together at the same time and also by analyzing 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 on open source Java projects showed a slight improvement in quality values and semantics preservation with these additional considerations. Another study (Ouni et al., 2015) investigated the use of past refactorings borrowed from different software projects when the change history for 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. (2015) combined the previous approach by (Kessentini et al., 2011) 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 minimizing 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 minimize the number of refactorings necessary to improve the software. The approach was tested on four open source Java programs and one industrial Java project. The programs were chosen based on the number of previous versions of the software available, as the success of the approach would depend on this input. The experimental results were compared against previous mono-objective and multi-objective approaches and were found to have better results with less refactorings, but also took longer to run.

Pérez, Murgia and Demeyer (2013) presented a short position paper to propose an approach to resolving design smells in software. They proposed using the version control repository to find and use previously effective refactorings in the code and apply them to the current design as "Refactoring Strategies". Refactoring strategies are defined as heuristic-based, automation-suitable specifications of complex behavior-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.

Morales (2015) defined his aim to create an Eclipse plug-in to help with refactoring in a doctoral paper. He aimed to compare different metaheuristic approaches and use a metaheuristic search to detect anti-patterns in code. The plugin would then use automated refactoring to help remove the anti-patterns and improve the design of the code. Morales et al. (2016) 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 one of three 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). To test the approach, it was applied to three open source Java programs with sufficient information to deduce developer's context. They adapted the approach to look for refactorings that can reduce four types of anti-pattern; lazy class, long parameter list, spaghetti code, and speculative generality. They also aimed to improve five of the quality attributes defined in the

QMOOD model. The results showed that ReCon can successfully correct more than 50% of anti-patterns in a project using less 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.

Mkaouer et al. experimented with combining quality measurement with robustness (Mkaouer et al., 2014) 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 on six different open source Java programs of different sizes and domains to create a population of solutions that used robustness as well as software quality in the fitness measurement. To measure robustness, they used formulas to approximate *smell severity (*by prioritizing four different code smell types with scores between 0 and 1) and *importance of code smells fixed* (by measuring the activity of the code modified via number of comments, relationships and methods) as well as measuring the amount of fixed code smells. They also used a number of multi-objective performance measurements (hypervolume, inverse generational distance and contribution) to compare against other multi-objective algorithms. To analyze 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 one 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 two mono-objective approaches that use a weighted combination of metrics (the same ones used above). 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 (Mkaouer et al., 2016) by testing eight open source systems and one industrial project, and by increasing the number of code smell types analyzed to seven.

They also experimented with the newly proposed evolutionary optimization method NSGA-III (Mkaouer et al., 2014), which uses a GA to balance multiple objectives through non-dominated sorting. They used the algorithm to remove detected code smells in seven open source Java programs through a set of refactorings. 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. Three other MOEAs were compared; IBEA, MOEA/D and NSGA-II. The other MOEAs 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 two 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 (Mkaouer et al., 2015) by

also experimenting on an industrial project and increasing the number of many-objective techniques compared against from two to four. The number of objectives was reduced to eight 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 the NSGA-III algorithm for remodularization (Mkaouer et al., 2015). They used four open source Java systems in the experimentation along with one industrial system provided by Ford Motor Company. They compared the technique against other approaches by looking at up to seven 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 open source systems.

More recently, Ouni et al. (2015) adapted the chemical reaction optimization (CRO) algorithm for search-based refactoring and explored the benefits of this approach. They compared this search technique against more standard optimization techniques used in SBSE; a GA, SA and particle swarm optimization (PSO). They combined four different prioritization measures to make up a fitness function that aimed to reduce seven different types of code smells. The four measures were priority, severity, risk and importance. Each of the code smell types were given a priority score of 1 to 7 to represent their opinion of which smells are more urgent from previous experience in the field. The inFusion tool (a design flaw detection tool) was used to deduce severity scores to represent how critical a code smell is in comparison with others of the same type. The risk score was calculated to represent riskier code as code that deviated from well-designed code. Code smells that related to more frequently changed classes were considered more important as code smells that hadn't undergone changes were considered to have been created intentionally. The approach was applied to five different open source Java programs and was compared against a previous study and a variation of the approach that didn't use prioritization. The approach was superior using the relevant measures to the other two solutions compared against it. It was also shown to give better solutions in larger systems than the other optimization algorithms tested.

Amal et al. (2014) used an Artificial Neural Network (ANN) 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 to 15 years, to manually evaluate the refactoring solutions generated for the first few iterations by marking each refactoring as good or bad. The ANN 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 ANN worked to replace the definition of a fitness function. The approach was tested on six 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.

4.5 Refactoring tools

Fatiregun, Harman and Hierons (2004) 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 the 20 transformations (refactorings) available in the tool, to refactor the program and optimize its length by comparing 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.

DiPenta (2005) proposed another refactoring framework, Evolution Doctor, to handle clones and unused objects, remove circular dependencies and reorganize source code files. Afterwards, a hybridisation of HC and GAs is used to reorganize libraries. The fitness function of the algorithm was created to balance four 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 three open source applications to demonstrate its effectiveness in each of the areas of design flaw detection and removal.

Griffith, Wahl and Izurieta (2011) introduced the TrueRefactor tool to find and remove a set of code smells from a program in order to increase comprehensibility. TrueRefactor can detect lazy classes, large classes, long methods, temporary fields or instances of shotgun surgery in Java programs and uses a GA to help remove them. The GA is utilized to search for the best sequence of refactorings that removes the highest number of code smells from the original source code. 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 the code smells. 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 analyze the effectiveness of the tool. The number of code smells of each type 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.

5 Analysis

To address the research questions outlined in Section 3, each subsection analyzes the relevant aspect of the papers.

5.1 Papers published over time

Figure 1 shows the amount of papers published each year, going as far back as 1999. From 1999 to 2008, the largest amount of papers published in a year is four. In 2009 and 2010, there are no papers published, but from 2011 there is an increased amount of search-based refactoring research. From 2011 to 2016, there are at least four papers published a year. The most prolific year for search-based refactoring research was 2012 with eight papers published that year. The years 2001, 2002, 2009 and 2010 are notable for not having any search-based refactoring papers published. Overall, from 1999 to 2016, there is an average of three papers published per year. If we compare the popularity of search-based refactoring research before 2010 with after, we can see that the average has increased from one paper a year to six. The earliest refactoring tool proposed among the papers was DPT (Cinnéide & Nixon, 1999a), in 1999.

Figure 2 gives a look at the amount of papers that use each type of search technique per year. In Fig. 2 GAs, GP and general evolutionary algorithms (GEAs) are encapsulated as EAs and PSO and ABC are grouped together as swarm optimization algorithms (SOAs). These are compared against the other search algorithms used (HC, SA, CRO and VNS). Before 2003, there were no search algorithms involved in any of the published papers. The earliest paper to use search techniques was "A Stochastic Approach To Automated Design Improvement" (O'Keeffe & Cinnéide, 2003), with SA being used. The earliest paper to actually compare search techniques, "Search-Based Software Maintenance" (O'Keeffe & Cinnéide, 2006), wasn't until 2006.

5.2 Types of paper

Figure 3 gives the different types of paper analyzed in the literature. All but one of the papers were published in journals or featured in conferences. The other (Koc et al., 2012) was included as a book section. Of the journal and conference papers, the majority are from conferences. Table 3 gives the list of conferences that at least two of the analyzed papers are from. GECCO has seven papers (Harman & Tratt, 2007; O'Keeffe & Cinnéide, 2007a; Seng et al., 2006; Vivanco & Pizzi, 2004; Ouni et al., 2013; Mkaouer et al., 2014; Mkaouer et al., 2014). This conference, which is concerned primarily with EAs, contains more papers than conferences related to maintenance (CSMR, ICSM and

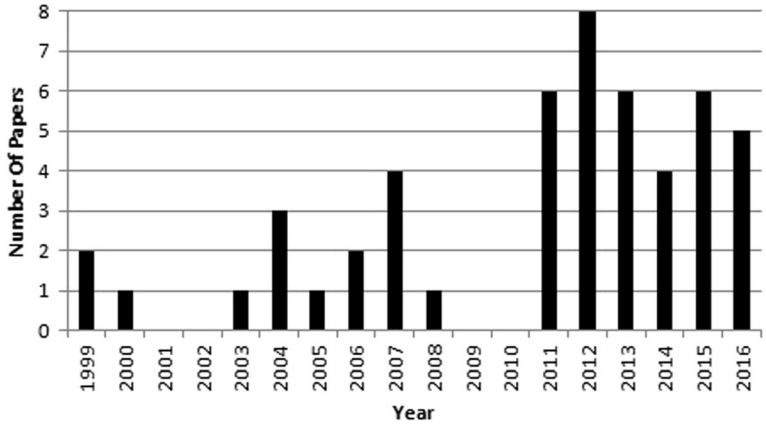

Fig. 1 Number of Papers Published Each Year

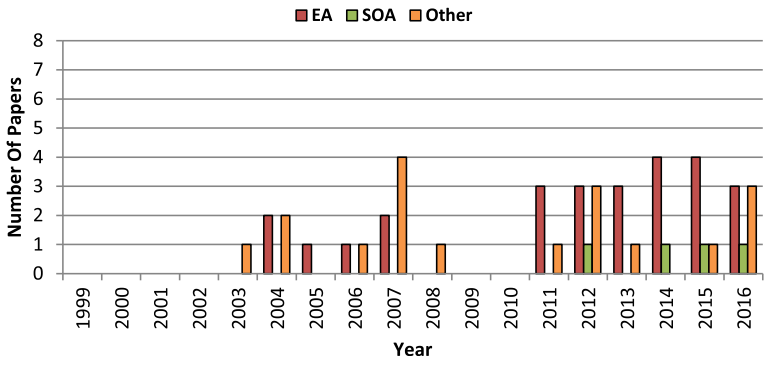

Fig. 2 Number of Papers Using Each Type of Search Technique Per Year

SANER) and even SBSE (SSBSE), demonstrating how ubiquitous evolutionary computation is with search-based refactoring. The journals published in are listed in Table 4, along with the number of papers published in each. The most widely published journals are Empirical Software Engineering (Cinnéide et al., 2012; Cinnéide et al., 2016; Mkaouer et al., 2016; Mkaouer et al., 2015) and the Journal of Systems and Software (O'Keeffe & Cinnéide, 2008a; Mohan et al., 2016; Ouni et al., 2015; Morales et al., 2016) with four papers apiece.

5.3 Authors

Figure 4 gives the number of papers each author has published. The majority of authors have only published one paper. Of the remaining authors, six have published two papers. Only 11 of the 70 authors have published more than two papers. Table 5 lists these authors and gives the number of papers published for each. Mel Ó Cinnéide has published more papers than the other authors at 21. Only the top two authors have

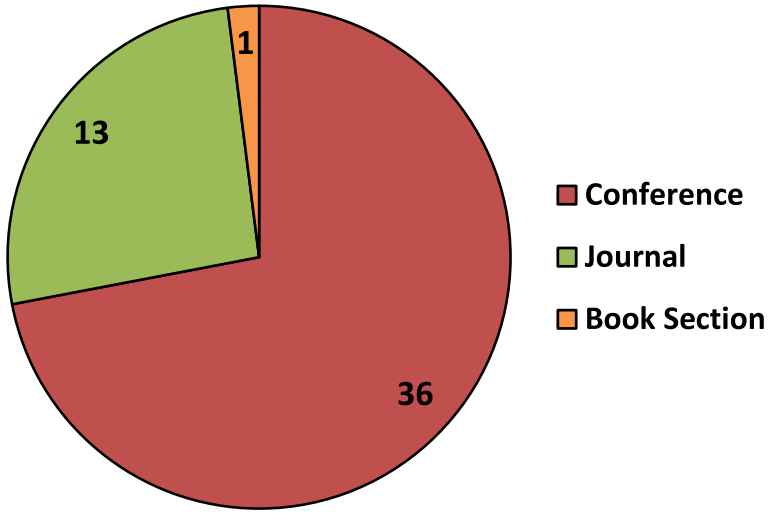

Fig. 3 Types of Paper Analyzed

Table 3 Number of Papers per Conference

Conference	Number of Papers
Genetic and Evolutionary Computation Conference (GECCO)	7
European Conference on Software Maintenance and Reengineering (CSMR)	4
International Conference on Software Maintenance (ISCM)	3
Symposium on Search-Based Software Engineering (SSBSE)	3
International Conference on Software Analysis, Evolution, and Reengineering (SANER)	2
International Conference on Software Testing, Verification and Validation (ICST/ICSTW)	2
International Symposium on Empirical Software Engineering and Measurement (ESEM)	2

published more than 10 papers. Between the two of them, Mel Ó Cinnéide and Marouane Kessentini are authors on 33 of the 54 papers analyzed, with Ó Cinnéide and Kessentini sharing authorship on four of the papers.

5.4 Types of study

Most of the studies in the papers were quantitative. Three were qualitative in comparison to the 37 quantitative studies. A further 10 were discussion based papers with no experimental portions. Of the quantitative papers, most of the studies tested different refactorings approaches, but a number of papers (Harman & Tratt, 2007; Vivanco & Pizzi, 2004; Amal et al., 2014) investigated other factors. Various studies examined the setup of the search approach. Amal et al. (2014) investigated the fitness function used in a GA to choose solutions, by incorporating an ANN into the search. Harman and Tratt (2007) tested the Pareto optimal approach to combine software metrics in a search. Vivanco and Pizzi (2004) used a GA to test metrics and choose the most suitable ones to use (Bakar et al. (2012a) also proposed a method to do this).

5.5 Refactoring approaches

In many cases, the studies conducted proposed and used tools in order to detect issues in the code and of these tools some [Di Penta, 2005, Griffith et al., 2011] were used to find specific issues, like god classes or data classes in the program. The studies are listed in Table 6. One of the studies (Moghadam & Cinnéide, 2012) 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, one to represent the current solution and one to represent the desired solution. This was created with the assistance of the programmer. Using these two 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 three main methods of automated maintenance from the analyzed 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 (Kessentini et al., 2011; Kessentini et al., 2011; Ouni et al., 2013; Kessentini et al., 2012; Ouni et al., 2012; Ouni et al., 2013; Ouni et al., 2013; Ouni et al., 2015; Wang et al., 2015; Morales et al., 2016; Mkaouer et al., 2014; Mkaouer et al., 2016; Mkaouer et al., 2015; Ouni et al., 2015; Griffith et al., 2011) or the problems are addressed automatically (Di Penta, 2005).

Table 4 Number of Papers per Journal

Journals	Number of Papers
Empirical Software Engineering	4
Journal of Systems and Software	4
Software Quality Journal	2
ACM Transactions on Software Engineering and Methodology	1
Automated Software Engineering	1
Journal of Software Maintenance and Evolution: Research and Practice	1

Finally, there is the method of using quality metrics to refactor the program stochastically and work towards a better solution (O'Keeffe & Cinnéide, 2006; O'Keeffe & Cinnéide, 2004; O'Keeffe & Cinnéide, 2008a; O'Keeffe & Cinnéide, 2007a; O'Keeffe & Cinnéide, 2008b; Koc et al., 2012; Mohan et al., 2016; O'Keeffe & Cinnéide, 2007b; Ghaith & Cinnéide, 2012; Cinnéide et al., 2012; Cinnéide et al., 2016; Veerappa & Harrison, 2013; Fatiregun et al., 2004) or again, using this approach to suggest refactorings to apply (Harman & Tratt, 2007; Seng et al., 2006; Mkaouer et al., 2014; Mkaouer et al., 2015).

5.6 Search techniques

Figure 5 identifies the number of papers that use each type of search technique, with GAs, GP and GEAs again encapsulated as EAs and PSO and ABC encapsulated as SOAs. Among the algorithms, the EAs were used the most, at 26 studies. EAs became more prominent in the research after 2010, with three to four papers per year involving them, whereas there had been six studies involving EAs altogether between 1999 and 2010. The next most common technique, HC, was used in 14 studies, with SA being used in 11. There has been a fairly consistent presence of HC and SA over the years. The largest amount of studies looking at HC or SA was in 2007 at four. In comparison, there were four studies involving EAs in both 2014 and 2015. The SOAs, together, were used in only five studies. The SOAs have been more frequently investigated in recent years, with a paper involving one of them in 2012, 2014, 2015 and 2016. CRO and VNS were used in one study

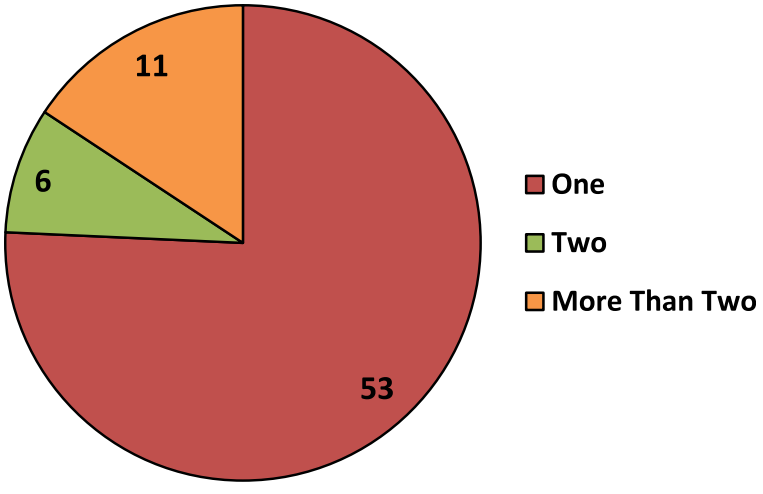

Fig. 4 Number of Papers per Author

Table 5 Number of Papers per Author

Authors	Number of Papers
Mel Ó Cinnéide	21
Marouane Kessentini	16
Ali Ouni	8
Houari Sahraoui	7
Mark Harman	7
Mark O' Keeffe	7
Slim Bechikh	6
Iman Hemati Moghadam	5
Kalyanmoy Deb	5
Wiem Mkaouer	5
Laurence Tratt	3

Table 6 Studies Proposing and Using Tools to Detect Issues in the Code

Authors [Ref]	Year	Title
Fatiregun et al. (2004)	2004	Evolving Transformation Sequences Using Genetic Algorithms
O'Keeffe and Ó Cinnéide (2004)	2004	Towards Automated Design Improvement Through Combinatorial Optimisation
Di Penta (2005)	2005	Evolution Doctor: A Framework To Control Software System Evolution
O'Keeffe and Ó Cinnéide (2006)	2006	Search-Based Software Maintenance
O'Keeffe and Ó Cinnéide (2007a)	2007	Getting The Most From Search-Based Refactoring
O'Keeffe and Ó Cinnéide (2008b)	2007	Search-Based Refactoring: An Empirical Study
O'Keeffe and Ó Cinnéide (2007b)	2007	Automated Design Improvement By Example
O'Keeffe and Ó Cinnéide (2008a)	2008	Search-Based Refactoring For Software Maintenance
Griffith et al. (2011)	2011	TrueRefactor: An Automated Refactoring Tool To Improve Legacy System And Application Comprehensibility
Ghaith and Ó Cinnéide (2012)	2012	Improving Software Security Using Search-Based Refactoring
Koc et al. (2012)	2012	An Empirical Study About Search-Based Refactoring Using Alternative Multiple And Population-Based Search Techniques
Moghadam and Ó Cinnéide (2012)	2012	Automated Refactoring Using Design Differencing
Ó Cinnéide et al. (2012)	2012	Experimental Assessment Of Software Metrics Using Automated Refactoring
Veerappa and Harrison (2013)	2013	An Empirical Validation Of Coupling Metrics Using Automated Refactoring
Mohan et al. (2016)	2016	Technical Debt Reduction Using Search Based Automated Refactoring
Ó Cinnéide et al. (2016)	2016	An Experimental Search-Based Approach To Cohesion Metric Evaluation

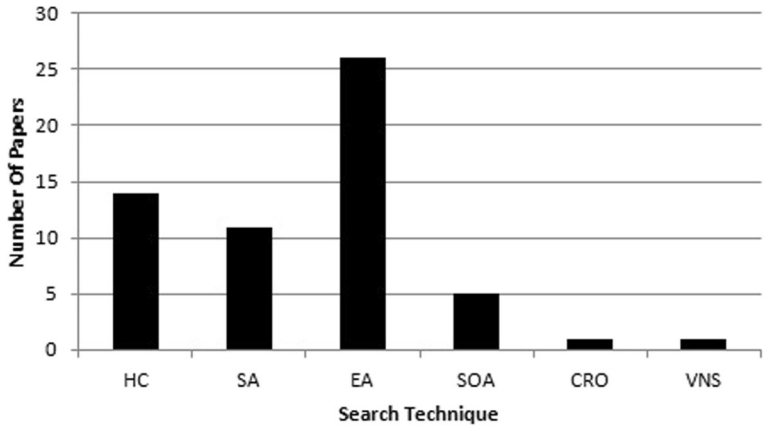

Fig. 5 Types of Search Technique used in the Analyzed Papers

each. Each of these studies (Morales et al., 2016; Ouni et al., 2015) are recent as well (2015 and 2016), suggesting a possibility for the CRO and VNS searches to be explored more in future research.

Figure 6 gives the dispersion of the EAs between studies that use GAs, GPs and GEAs. The majority of the EAs used are GAs, at 24 studies. The GP algorithm and GEA was only used in three studies and two studies respectively. Three of the studies involved both GP and a GA. A big reason for the prominence of the GA among the studies is that it is present across the studies of (Kessentini et al., 2011; Kessentini et al., 2011; Kessentini et al., 2012), (Mkaouer et al., 2014; Mkaouer et al., 2016; Mkaouer et al., 2014; Mkaouer et al., 2015; Mkaouer et al., 2015) and (Ouni et al., 2013; Ouni et al., 2012; Ouni et al., 2013; Ouni et al., 2013; Ouni et al., 2015; Ouni et al., 2015). This group of authors regularly used the GA in their papers, amounting to 14 different instances among the 25. Of the 26 studies containing EAs 12 used MOEAs. Figure 7 shows the dispersion of the SOAs between studies using PSOs and ABCs. ABC was used in one study (Koc et al., 2012), but The PSOs were used more frequently, with three studies (Mkaouer et al., 2014; Mkaouer et al., 2016; Ouni et al., 2015).

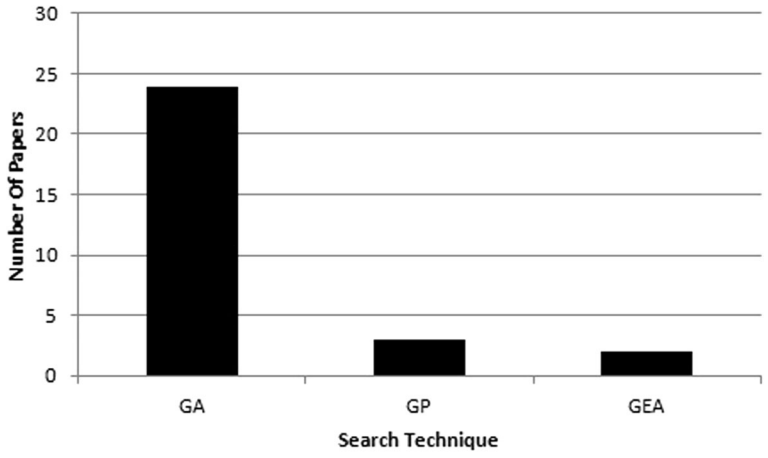

Fig. 6 Dispersion of Evolutionary Algorithms from Fig. 5

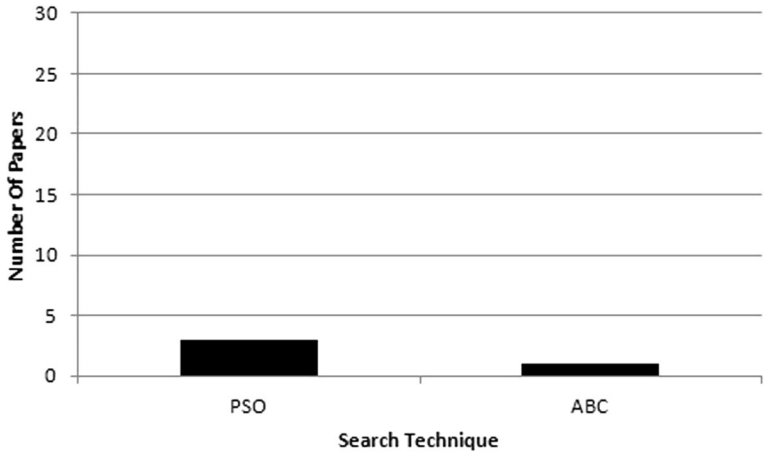

Fig. 7 Dispersion of Swarm Algorithms from Fig. 5

Among the studies using search techniques, some were used to test or present an approach proposed in the paper, and others compared different search techniques against each other. As such, different papers using search techniques contained a different number of search techniques within the paper. Figure 8 shows the number of papers that contained a certain number of search techniques, ranging from no search techniques to the maximum amount of four different techniques in one paper. 11 of the papers didn't contain any search techniques. The majority only used one, at 24 papers, although 15 other papers that did use search techniques used more than one. Of these, 12 papers (O'Keeffe & Cinnéide, 2006; O'Keeffe & Cinnéide, 2008a; O'Keeffe & Cinnéide, 2007a; O'Keeffe & Cinnéide, 2008b; Koc et al., 2012; Mohan et al., 2016; Ghaith & Cinnéide, 2012; Morales et al., 2016; Mkaouer et al., 2014; Mkaouer et al., 2016; Ouni et al., 2015; Fatiregun et al., 2004) directly compared the

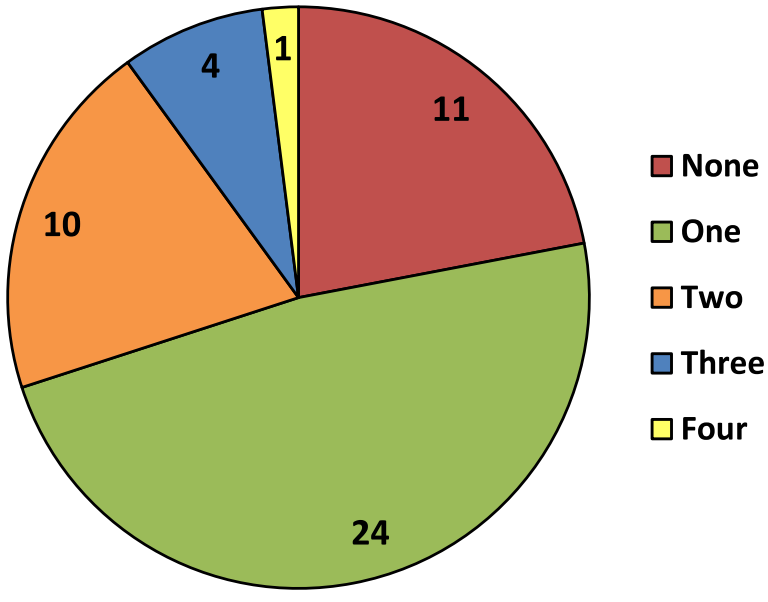

Fig. 8 Number of Search Techniques Used/Analyzed in Each Paper

different search techniques against each other to speculate on the most applicable. Only one study (Ouni et al., 2015) looked at four separate search techniques.

Four of the papers O'Keeffe and Cinnéide, 2006, O'Keeffe and Cinnéide, 2007a, O'Keeffe and Cinnéide, 2008b, Koc et al., 2012, focused mainly on comparing search techniques. These studies compared HC with GAs, HC with SA or all three with each other. One (Koc et al., 2012) 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 depend highly upon the configuration of the search beforehand. Therefore, it seems evident that while these options may be useful, to get good results there will be overhead involved in finding the suitable parameters that yield high quality results for the problem in question. In the study that included the ABC search, it (along with the multiple steepest descent HC) did seem to be more successful. However, as this is the only study to use the ABC technique, further insight into the technique cannot be derived from the literature. Table 10 in the appendix gives the search techniques, if any, used within each paper.

5.7 Input programs used

Figure 9 shows the different types of program used to test the approaches examined among the studies. Most of the programs used are open source. The remaining programs used consist of test programs developed for the study O'Keeffe and Cinnéide, 2003, O'Keeffe and Cinnéide, 2004, Cinnéide et al., 2011, Simons et al., 2015, Fatiregun et al., 2004, Griffith et al., 2011, in-house programs Koc et al., 2012, Vivanco and Pizzi, 2004 and industrial programs. Four studies Wang et al., 2015, Mkaouer et al., 2016, Mkaouer et al., 2015, Mkaouer et al., 2015 used an industrial program by the Ford Motor Company, referred to as JDI-Ford. As the vast majority of the frameworks used dealt with Java code,

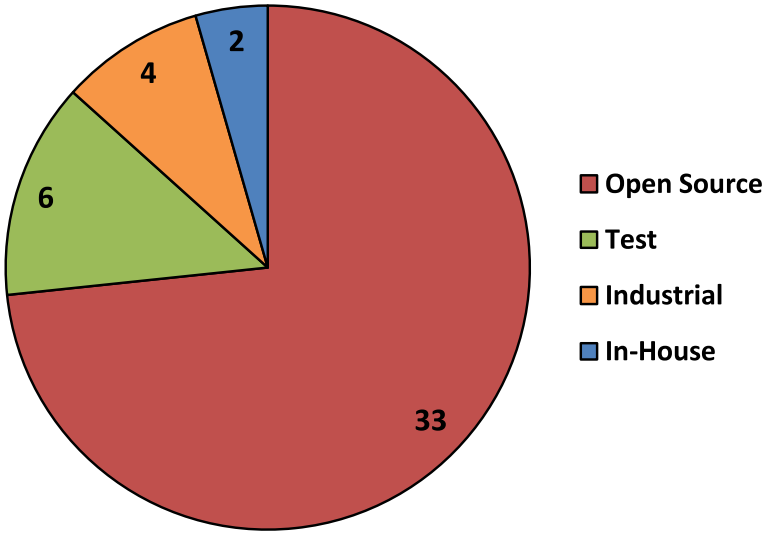

Fig. 9 Types of Benchmark Program Used in Experimental Studies in the Analyzed Papers

the open source programs used are in Java. Table 7 lists the open source programs used among the papers, along with references to the studies that used them.

The project sizes are generally adequate for the experiments as they are large enough to justify representation of a real project. The sizes of these projects generally tend to be tens of thousands of lines of code with hundreds of classes. In the case of the test programs, small programs are used or constructed to demonstrate the applicability of the program or technique with an example. One study (O'Keeffe & Cinnéide, 2007a) deliberately kept the programs smaller than 51 top-level classes. In contrast, another study (Di Penta, 2005) used a large system with over one million lines of code.

5.8 Tools

There were a number of tools Cinnéide and Nixon, 1999a, O'Keeffe and Cinnéide, 2003, Koc et al., 2012, Moghadam and Cinnéide, 2011, Fatiregun et al., 2004, Di Penta, 2005, Griffith et al., 2011 proposed in the literature, using various different approaches to refactoring code. They are listed in Table 8. While most of the seven tools were developed for Java code, FermaT (Fatiregun et al., 2004) is used to provide more low-level changes with wide-spectrum language (WSL) transformations. For one of the tools, TrueRefactor, plans were expressed to adapt the tool to be applicable to multiple different programming languages (Griffith et al., 2011). A number of the proposed tools identified design defects first before attempting to resolve them. DPT (Cinnéide & Nixon, 1999a) was proposed to apply design patterns to the code in an automated manner. It uses mini transformations built from refactorings to apply the patterns. Similarly, Evolution Doctor (Di Penta, 2005) is used to diagnose issues in the software first, before restructuring it to ameliorate those issues. Likewise, TrueRefactor (Griffith et al., 2011) finds instances of five different types of code smells before finding refactorings to resolve them. Other tools O'Keeffe and Cinnéide, 2003, Koc et al., 2012, Moghadam and Cinnéide, 2011, Fatiregun et al., 2004 use refactorings to improve the code according to metric functions. Instead of analyzing the code for issues beforehand, they refactor the code up front in order to resolve issues as they go along. A few of the proposed tools were used in multiple papers. The A-CMA tool was proposed by (Koc et al., 2012) and then adapted and used by Mohan, Greer and McMullan (Mohan et al., 2016). The CODe-Imp tool was used in a myriad of studies O'Keeffe and Cinnéide, 2006, O'Keeffe and Cinnéide, 2008a, O'Keeffe and Cinnéide, 2007a, O'Keeffe and Cinnéide, 2008b, O'Keeffe and Cinnéide, 2007b, Moghadam and Cinnéide, 2011, Moghadam and Cinnéide, 2012, Cinnéide et al., 2011, Ghaith and Cinnéide, 2012, Cinnéide et al., 2012, Cinnéide et al., 2016, Veerappa and Harrison, 2013. A precursor to CODe-Imp, DPT, was also present in three different papers Cinnéide and Nixon, 1999a, Cinnéide and Nixon, 1999b, Cinnéide, 2000.

5.9 Metrics

Of the papers, there have been a number that have investigated O'Keeffe and Cinnéide, 2006, Mohan et al., 2016, O'Keeffe and Cinnéide, 2007b, Ghaith and Cinnéide, 2012, Cinnéide et al., 2012, Cinnéide et al., 2016, Veerappa and Harrison, 2013, Vivanco and Pizzi, 2004, Bakar et al., 2012a or discussed Simons et al., 2015, Harman et al., 2013, Harman et al., 2012 the metrics used in search-based approaches. Many of the programs analyzed in the experiments and case studies conducted have been using Java (or one using C++

Table 7 Open Source Test Programs Used in the Literature (and the Papers They are Used in)

GanttProject (Moghadam & Cinnéide, 2012; Morales et al., 2016; Cinnéide et al., 2012; Cinnéide et al., 2016; Veerappa & Harrison, 2013; Kessentini et al., 2011; Kessentini et al., 2011; Ouni et al., 2013; Kessentini et al., 2012; Ouni et al., 2012; Ouni et al., 2013; Ouni et al., 2015; Mkaouer et al., 2014; Mkaouer et al., 2016; Mkaouer et al., 2014; Mkaouer et al., 2015; Mkaouer et al., 2015; Ouni et al., 2015; Amal et al., 2014)	JHotDraw (Harman & Tratt, 2007; Mohan et al., 2016; Moghadam & Cinnéide, 2012; Seng et al., 2006; Morales et al., 2016; Cinnéide et al., 2012; Cinnéide et al., 2016; Veerappa & Harrison, 2013; Kessentini et al., 2012; Ouni et al., 2013; Ouni et al., 2015; Mkaouer et al., 2014; Mkaouer et al., 2016; Mkaouer et al., 2015; Ouni et al., 2015; Amal et al., 2014)	Xerces-J (Kessentini et al., 2011; Kessentini et al., 2011; Ouni et al., 2013; Kessentini et al., 2012; Ouni et al., 2012; Ouni et al., 2013; Ouni et al., 2013; Ouni et al., 2015; Mkaouer et al., 2014; Mkaouer et al., 2016; Mkaouer et al., 2014; Mkaouer et al., 2015; Mkaouer et al., 2015; Ouni et al., 2015; Amal et al., 2014)
JFreeChart (Veerappa & Harrison, 2013; Ouni et al., 2013; Ouni et al., 2015; Wang et al., 2015; Mkaouer et al., 2014; Mkaouer et al., 2016; Mkaouer et al., 2015; Ouni et al., 2015; Amal et al., 2014)	ArgoUML (Morales et al., 2016; Kessentini et al., 2011; Kessentini et al., 2011; Ouni et al., 2013; Mkaouer et al., 2014; Mkaouer et al., 2015)	Beaver (O'Keeffe & Cinnéide, 2008a; O'Keeffe & Cinnéide, 2007a; O'Keeffe & Cinnéide, 2008b; Koc et al., 2012; Mohan et al., 2016; O'Keeffe & Cinnéide, 2007b)
Apache Ant (Mkaouer et al., 2014; Mkaouer et al., 2016; Mkaouer et al., 2014; Mkaouer et al., 2015; Amal et al., 2014)	Art of Illusion (Cinnéide et al., 2012; Cinnéide et al., 2016; Veerappa & Harrison, 2013; Ouni et al., 2015; Ouni et al., 2015)	Mango (O'Keeffe & Cinnéide, 2007a; O'Keeffe & Cinnéide, 2008b; Koc et al., 2012; Mohan et al., 2016; O'Keeffe & Cinnéide, 2007b)
Spec-Check (O'Keeffe & Cinnéide, 2006; O'Keeffe & Cinnéide, 2008a; O'Keeffe & Cinnéide, 2007a; O'Keeffe & Cinnéide, 2008b; O'Keeffe & Cinnéide, 2007b)	XOM (Harman & Tratt, 2007; Moghadam & Cinnéide, 2012; Cinnéide et al., 2012; Cinnéide et al., 2016; Veerappa & Harrison, 2013)	Azureus (Kessentini et al., 2011; Ouni et al., 2013; Mkaouer et al., 2014; Mkaouer et al., 2015)
JGraphX (Moghadam & Cinnéide, 2012; Cinnéide et al., 2012; Cinnéide et al., 2016; Veerappa & Harrison, 2013)	QuickUML (Kessentini et al., 2011; Kessentini et al., 2011; Ouni et al., 2013; Kessentini et al., 2012)	JabRef (Cinnéide et al., 2012; Cinnéide et al., 2016; Veerappa & Harrison, 2013)
JRDF (Cinnéide et al., 2012; Cinnéide et al., 2016; Veerappa & Harrison, 2013)	JTar (Moghadam & Cinnéide, 2012; Cinnéide et al., 2012; Cinnéide et al., 2016)	Log4j (Kessentini et al., 2011; Ouni et al., 2013; Mkaouer et al., 2016)
Rhino (Mkaouer et al., n.d.; Mkaouer et al., 2016; Amal et al., 2014)	Apache XML-RPC (Koc et al., 2012; Mohan et al., 2016)	EAOP (O'Keeffe & Cinnéide, 2007a; O'Keeffe & Cinnéide, 2008b)
JFlex (Koc et al., 2012; Mohan et al., 2016)	HTMLUnit (Moghadam & Cinnéide, 2012; Cinnéide et al., 2016)	JSON (Koc et al., 2012; Mohan et al., 2016)
Mylyn (Morales et al., 2016; Morales et al., 2016)	FindBugs (Wang et al., 2015)	Grammatica (O'Keeffe & Cinnéide, 2008b)
GRASS (Di Penta, 2005)	Hibernate (Wang et al., 2015)	jSMPP (Cinnéide et al., 2016)
KDE (Di Penta, 2005)	Maven (Harman & Tratt, 2007)	MySQL (Di Penta, 2005)

Table 7 Open Source Test Programs Used in the Literature (and the Papers They are Used in) *(Continued)*

Nutch (Mkaouer et al., 2016)	PDE (Morales et al., 2016)	Pixelitor (Wang et al., 2015)
Platform (Morales et al., 2016)	Samba (Di Penta, 2005)	Spec-Raytrace (O'Keeffe & Cinnéide, 2006)
Wife (Ghaith & Cinnéide, 2012)		

(Vivanco & Pizzi, 2004)) and likewise, many of the metrics investigated have been related to object-oriented behaviors. The most commonly used metrics were ones that measured cohesion or coupling. Numerous different metrics are available to measure these qualities and one study (Cinnéide et al., 2012) compared different cohesion metrics to determine how similar they are, finding conflicting behaviors. Another study (Veerappa & Harrison, 2013) 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 (Vivanco & Pizzi, 2004) to compare 64 different metrics in an attempt to determine the most effective ones for search-based optimization. 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 (O'Keeffe & Cinnéide, 2006) used the QMOOD model to represent the properties flexibility, reusability and understandability with various weighted combinations of metrics to analyze which ones were most useful.

5.10 Research gaps and opportunities

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. 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

Table 8 List Of Automated Refactoring Tools With Brief Descriptions

Refactoring Tool [Ref]	Year	Purpose
A-CMA (Koc et al., 2012)	2012	Refactors Java bytecode using a selection of refactorings and metrics.
CODe-Imp (Moghadam & Cinnéide, 2011)	2011	Automated refactoring tool containing numerous metrics and refactorings.
Dearthóir (O'Keeffe & Cinnéide, 2003)	2003	Improves the design of an object-oriented program.
DPT (Cinnéide & Nixon, 1999a)	1999	Applies design pattern transformations to Java programs.
Evolution Doctor (Di Penta, 2005)	2005	Diagnoses reorganization opportunities and performs reengineering actions.
FermaT (Fatiregun et al., 2004)	2004	Transformation tool for migration of legacy systems from assembly code to higher level languages.
TrueRefactor (Griffith et al., 2011)	2011	Identifies and removes five different design smells in Java.

usefulness of the metaheuristic optimization technique, depending on how accurately they portray quality in the eyes of the user. We need explicit metrics to guide the optimization of a solution, but one developer's view of quality may be different to another's. Similarly, a programmer's opinion of quality may change from project to project or over time. It would be useful to have some form of explicit guidance on how to choose metrics for a search-based optimization technique. What is the general view of software quality? How is this affected from one programming language to another? Most previous research has been applied to object-oriented programs and as such most fitness functions aim to improve object-oriented behaviors like cohesion or flexibility. Even defining these aspects has proven to be difficult. The field would gain valuable insight with research into developer opinions on software quality and on how technical debt is currently handled in the business environment.

Experimentation has been done 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 and into how metrics that aim to measure similar aspects differ from each other. There is an opportunity for research into using different combinations to improve the software in different ways, similarly to how a human assisted tool can guide the improvement of the software design to a suitable solution for the user.

Another important aspect to research is the applicability of these techniques in a company environment. There are various things to consider here. Is the refactored code actually useful for a maintenance programmer? There is a risk of the code being modified so much that it becomes incomprehensible. It has been suggested that the automated maintenance of code could possibly be viewed in a similar way to a compiler that makes changes to the code behind the scenes that a programmer needs not worry about. This further abstraction of the code may be the future of software design, as metamodels become more involved in the coding aspect of the project development cycle. In the analyzed literature, experimentation with industrial code has been lacking. An increase in the use of industrial code and the opinions and expertise of experienced software developers may help to simulate the company environment and uncover possible issues or problems to address. In addition to this, the majority of studies have been concerned with the Java programming language. As this doesn't accurately reflect the range of programming languages used in the software environment, increased support for other programming languages is desirable.

Of the different search techniques used to address software maintenance, a large proportion of the analyzed literature used EAs. Among these studies, a lot of recent work has looked at multi-objective approaches. Multi-objective algorithms have been applied sparsely to SBSE problems (Simons & Parmee, 2006; Yoo & Harman, 2007; Zhang et al., 2007; Simons & Parmee, 2007; Finkelstein et al., 2008; Simons & Parmee, 2008; Wang et al., 2008; Durillo et al., 2009; Maia et al., 2009; Maia et al., 2010; Bowman et al., 2010; Durillo et al., 2011; Brasil et al., 2011; Colanzi et al., 2011; Assunção et al., 2011; Yoo et al., 2013) and only recently have been used to address issues relating to maintenance. This indicates a promising evolution of automated maintenance in SBSE 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.

6 Discussion

The survey aim was outlined in Section 3 when it was asked "To what extent has search-based refactoring been studied in software maintenance?" In order to address this, 10 research questions were proposed. **RQ1** asked "How many papers were published per year?" From 1999 to 2016, there is an average of three papers published per year, with a notable increase in the amount being published after a drought in 2009 and 2010. Papers involving the use of EAs also increased notably after 2010, along with studies of more obscure search techniques (ABC, CRO, VNS). The largest amount of papers published in one year is eight papers, in 2012.

RQ2 asked "What are the most common methods of publication for the papers?" The majority of papers were published in conferences with the evolutionary computation conference GECCO publishing the most papers at seven. Only one paper was published as a book section and 13 were published in journals in comparison to the 36 papers (72% of those analyzed) that were published in conferences. The Empirical Software Engineering journal and Journal of Systems and Software published the largest amount of journal papers at four each.

RQ3 asked "Who are the most prolific authors investigating search-based refactoring in software maintenance?" 76% of the authors of the papers analyzed had only been involved in one of the published papers, but 17 were authors in at least two. Eleven authors published more than two papers each. Of these authors, Mel Ó Cinnéide and Marouane Kessentini authored 33 of the papers between them, and worked together on four of them.

RQ4 was interested in the types of papers analyzed, asking "What types of studies were used in the papers?" The majority of the papers (37) had quantitative studies, with three papers been qualitative studies and 10 being discussion papers with no experimental portions. Most of the studies investigated refactoring approaches, but three of the quantitative studies examined other aspects, such as the setup of the search process itself or the metrics used for maintenance.

RQ5 asked "What refactoring approaches were used in the literature?" We isolated three main methods for using search-based refactoring to improve the maintenance of software. Design defects can be found in the code first and then removed, or the software can be refactored up front to improve certain software metrics or sets of metrics. The less common approach used in only one study (Moghadam & Cinnéide, 2012) refactored software towards a previously generated ideal software design using UML models. Other papers also investigated the optimization process itself or tested the metrics available to measure maintenance.

RQ6 was interested in "What search techniques were used in the refactoring studies?" The majority of the studies involved EAs of some sort (mostly GAs). HC and SA were also popular, being used in 14 studies and 11 studies respectively. Other search techniques were used less frequently. PSO was used in three studies while ABC was experimented with in one study. Studies also experimented with CRO and VNS in 2015 and 2016.

RQ7 asked "What types of programs were used to evaluate the refactoring approaches?" in the experimentation. Most of the studies (33) used open source Java programs, but a selection of studies used test programs, in-house code or an industrial program called JDI-Ford. Five of the studies used open source programs in conjunction with an in-house program or the JDI-Ford industrial program. The program sizes were generally around tens of thousands of lines of code.

RQ8 asked "What tools were used for refactoring?" There were seven different refactoring tools proposed and used among the studies. Most of them worked with Java code. Half of the tools (Cinnéide & Nixon, 1999a; Di Penta, 2005; Griffith et al., 2011) used the design defect approach to refactoring the code and the other half (O'Keeffe & Cinnéide, 2003; Koc et al., 2012; Moghadam & Cinnéide, 2011; Fatiregun et al., 2004) used the approach that modified code based on a software quality measure. Of the tools, A-CMA was used in two papers, and CODe-Imp was used in 12.

RQ9 asked "What types of metrics were used in the studies?" Most studies have used metrics that examine object-oriented behaviours, cohesion and coupling in particular. Some studies also used metrics to represent the class structure of a program or for inheritance based observations. A number of the studies used metrics from the QMOOD metric suite (Mohan et al., 2016; O'Keeffe & Cinnéide, 2007b) or the quality attributes constructed by (Bansiya & Davis, 2002) using their QMOOD suite (O'Keeffe & Cinnéide, 2006; O'Keeffe & Cinnéide, 2008a; O'Keeffe & Cinnéide, 2007a; O'Keeffe & Cinnéide, 2008b; Simons et al., 2015; Ouni et al., 2015; Wang et al., 2015; Morales et al., 2016).

Finally, **RQ10** asked "What are the gaps in the literature and available research opportunities in the area?" Analysis of the literature and measurement of features of the research has derived various observations of the developments in the area and isolated a few aspects wherein there is an opportunity for future work and experimentation. Search-based refactoring to automate software maintenance has been shown to work for experimental examples and various tools have been developed to tackle the maintenance issue using automated means, but more work needs to be done to measure empirical examples. It needs to be evaluated whether the search-based refactoring techniques that have been developed can carry over to the business environment or whether real-world application scenarios will bring to light further issues. Also, the metrics used to test the experimental approaches and aid with refactoring could also be further examined. There have been a few studies investigating the metrics used, but as they can be subjective, further inspection is necessary. Likewise, recent work has experimented with the use of MOEAs for refactoring, and this is an exciting area to investigate for further research.

7 Related work

There has already been a number of literature reviews related to the field of SBSE as well as to various aspects of refactoring. Table 9 lists the other literature reviews related to SBSE or refactoring.

Harman, Mansouri and Zhang wrote a general review of SBSE in 2009 (Harman et al., 2009) before updating the review in 2012 (Harman et al., 2012). Another review of the area of SBSE was done by (Harman et al., 2012). These reviews give an overview of the different areas of SBSE and discuss research done in those areas up to that point in time. The literature review conducted in this survey sticks out from them as it focuses of SBSE in relation to maintenance and, in particular, refactoring. Räihä wrote a report (Räihä, 2009) in 2009 that was later released in a journal (Räihä, 2010) in 2010 that focused on the areas of architecture design, software clustering, software refactoring and software quality. Although this review focuses on similar areas to maintenance in SBSE and looks at refactoring, it was still a bit too general in comparison with the current survey.

A few recent reviews were more focused and looked at various aspects of refactoring. Misbhauddin and Alshayeb looked at UML model refactoring in 2015 (Misbhauddin &

Table 9 Literature Reviews

Authors [Ref]	Year	Title
Al Dallal (2015)	2015	Identifying Refactoring Opportunities In Object-Oriented Code: A Systematic Literature Review
Harman et al. (2009), (2012)	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. (2012)	2012	Search Based Software Engineering: Techniques, Taxonomy, Tutorial
Mariani and Vergilio (2017)	2017	A Systematic Review On Search-Based Refactoring
Misbhauddin and Alshayeb (2015)	2015	UML Model Refactoring: A Systematic Literature Review
Räihä (2009), (2010)	2009/ 2010	An Updated Survey On Search Based Software Engineering/A Survey On Search Based Software Engineering

Alshayeb, 2015), concluding that UML model refactoring is a highly active area of research. They noted that, while a number of approaches have been proposed in the area, there are still some important issues and limitations to be addressed. Al Dallal also reviewed studies that identified refactoring opportunities in 2015 (Al Dallal, 2015). They too concluded that the analyzed area is a highly active area of research. They found that many of the studies used open source systems and that they were limited in their size, recommending that further studies involve industrial systems and systems of greater size to improve the generality of the results. They also encouraged researchers to expand the coverage of their work to include more refactoring activities. Mariani and Vergilio gave a review of search-based refactoring in 2017 (Mariani & Vergilio, 2017). They found that most of the search-based refactoring approaches were more recent and that many of the studies used EAs. They also found that the most commonly used refactorings were part of Fowler's (Fowler, 1999) refactoring catalogue and that the most common type of solution produced in the studies is a sequence of refactorings to apply to the system.

Although the reviews of Misbhauddin and Alshayeb and of Al Dallal looked at aspects of refactoring, unlike this survey they did not focus on the use of search-based refactoring applied to software maintenance. They looked at aspects of refactoring in software engineering that was more abstract. Misbhauddin and Alshayeb investigated the use of refactoring to modify UML models, as an aspect of model-driven engineering. This survey looks more directly at refactoring to improve aspects of the software itself. Al Dallal investigates studies that identify opportunities for refactoring in object-oriented code. Again, this is less distinct than the investigation in this paper that analyzes papers that provide actual refactoring solutions in software.

On the other hand, Mariani and Vergilio did look at search-based refactoring. However, their study focuses only on the analysis of the studies, whereas this paper gives a detailed review of the studies being analyzed. Thus, this survey can work as an introduction to the area of SBSE relating to maintenance for researchers aiming to work in the area by giving an overview of the research actually conducted, as well as analyzing this research and drawing conclusions to derive gaps and possibilities for future work. On the other hand, the literature review by Mariani and Vergilio is not a survey and as such is focused only on the analysis. Not only is the review of the literature itself more in depth but the analysis also investigates other aspects of the literature. There is a more in depth investigation of the tools used for

refactoring in the studies and also an examination into how metrics have been tested and discussed in the literature. This survey also investigates how the search techniques used in the literature have changed over time. Similar aspects of the literature that have been investigated in both papers are compared below in order to examine whether related trends have been isolated across the two sets of analyzed papers.

7.1 Comparison

The review by Mariani and Vergilio (Mariani & Vergilio, 2017) analyzed some similar aspects of the literature, finding related trends. They investigated the search techniques used in the papers they examined. They observed that EAs and, in particular, the GA were the most commonly used algorithms in the studies. They observed HC in 16 studies and SA in 10, and observed ABC and CRO in a small subset of studies (two and one respectively). These trends within their review mirror this paper with EAs being observed more commonly in both papers and HC and SA being observed in a similar amount of studies, while other search techniques (e.g. ABC and CRO) are seen less commonly.

Another aspect investigated in their review is the systems used in the experiments to validate the approached proposed in the papers. Again, the observations they made were similar. Many of the systems noted were open source Java programs. Four of the papers they analyzed used the JDI-Ford program, and many of the most commonly used Java programs analyzed in this paper were also analyzed in their review (for example, Xerces-J, GanttProject, JHotDraw and JFreeChart were the most commonly used open source programs among the studies in both papers). They listed a smaller amount of open source Java programs tested, at 14, whereas in this paper there are 40 different programs used to test the approaches in the literature. They also investigated the tools used for refactoring and found that the CODe-Imp tool was used in nine of their studies, similarly to how 12 of the papers analyzed in this paper used the tool.

They examined the distribution of the studies analyzed per year, between 2005 and 2016. Multi-objective and many-objective approaches were observed from 2014, and more papers were observed in general from 2010 onwards. Before 2010, there were at most four papers per year whereas there were at least six papers analyzed per year afterwards. Again, these trends are mirrored across their paper and this one. The majority of the papers analyzed in their review were from conferences, with the remaining being published in journals (nine papers) or being from workshops (two papers). Lastly, they investigated the most prominent researchers in the studies they analyzed, with the seven authors they listed publishing more than five papers. Six of the seven authors are also listed among the seven most prominent authors of the studies analyzed in this paper, with the top two being noted in their paper and this one as authors that stand out due to publishing at least 15 of the analyzed studies each. The one author among the top seven in their survey who did not appear in this paper was Camelia Chisalita-Cretu, who had published six of the papers they had analyzed.

8 Threats to validity

There are number of elements of how the literature search has been conducted that may contribute threats to its validity. The methods used to address the aim of the survey "To what extent has search-based refactoring been studied in software maintenance?" may

provide a validity threat to the conclusions made. In this survey, we split the aim into a set of 10 research questions, which each investigate some element of the papers analyzed. Each research question is explored individually within the analysis and answered separately in the discussion section. The papers captured in the search can also be affected by a number of attributes related to how the search has been conducted. First, the search repository or repositories used to find the papers may provide different results and could prevent the identification of certain papers. To minimize this issue, we have used five popular search repositories to look for papers related to the areas of focus. We also used a snowballing approach to finding further related papers by investigating references in the papers and conducting similar searches.

Another element that could affect the papers returned is the search string used in the search. The search string used in this survey was constructed in order to reduce the returned results from hundreds of thousands of hits to something more feasible to sift through. As such, some papers may have been filtered out of the search that could be relevant. This is also somewhat appeased by the snowballing approach used so that other potentially related papers will likely be found regardless. The time period used to filter the results can prove a threat to the validity of the search, but the search used for this survey was conducted up to September 2016, so many of the more recent results will not be filtered out. Finally, the process used to search through and pick out relevant papers from the search results could affect the analysis, depending on with papers are chosen to investigate. To maximise the repeatability of the process and minimize the validity threat, a set of inclusion and exclusion criteria have been used to aid in picking out the relevant papers for the survey.

9 Conclusion

This survey investigates the question "To what extent has search-based refactoring been studied in software maintenance?" and introduces a set of research questions to help address it. Five different search repositories are used with the aid of a set of inclusion and exclusion criteria to find 50 different papers that use search-based refactoring for purposes of software maintenance. Before the papers are examined, the most common search techniques used in the studies are quickly discussed and described. Each of the papers are examined and their research output is discussed. Then, different aspects of the set of papers are analyzed according to the set of research questions outlined. The research questions are addressed using each relevant aspect of the analysis. Related reviews are also discussed and compared with the survey after it is conducted, describing the similarities and differences between them.

This survey is a valuable resource for researchers planning with investigate the use of search-based refactoring techniques for software maintenance and gives an overview of the field as well as the relevant work in the area. The analysis made within the paper allows readers to be aware of how the research has progressed and addresses the aim of finding the extent that search-based refactoring has been studied in software maintenance. The identified gaps and recommended areas for future work allow researchers to investigate other aspects of the research area. Work in these areas could contribute towards progression in the use of search-based refactoring for software maintenance and could aid in making the approach more feasible for use in the development of software products, therefore saving time and developer resources.

10 Appendix

Table 10 Papers On Search-Based Refactoring

Authors [Ref]	Year	Title	Search Technique
Amal et al. (2014)	2014	On The Use Of Machine Learning And Search-Based Software Engineering For Ill-Defined Fitness Function: A Case Study On Software Refactoring	GA
Bakar et al. (2012a)	2012	Applying Evolution Programming Search Based Software Engineering (SBSE) In Selecting The Best Open Source Software Maintainability Metrics	EA
Di Penta (2005)	2005	Evolution Doctor: A Framework To Control Software System Evolution	GA
Fatiregun et al. (2004)	2004	Evolving Transformation Sequences Using Genetic Algorithms	HC/GA
Ghaith and Ó Cinnéide (2012)	2012	Improving Software Security Using Search-Based Refactoring	HC/SA
Griffith et al. (2011)	2011	TrueRefactor: An Automated Refactoring Tool To Improve Legacy System And Application Comprehensibility	GA
Harman (2011)	2011	Refactoring As Testability Transformation	
Harman and Tratt (2007)	2007	Pareto Optimal Search Based Refactoring At The Design Level	HC
Harman et al. (2012)	2012	Dynamic Adaptive Search Based Software Engineering	
Harman et al. (2013)	2013	Dynamic Adaptive Search Based Software Engineering Needs Fast Approximate Metrics	
Kessentini et al. (2011)	2011	Design Defects Detection And Correction By Example	GA/GP
Kessentini et al. (2011)	2011	Example-Based Design Defects Detection And Correction	GA/GP
Kessentini et al. (2012)	2012	What You Like In Design Use To Correct Bad-Smells	GA
Koc et al. (2012)	2012	An Empirical Study About Search-Based Refactoring Using Alternative Multiple And Population-Based Search Techniques	HC/SA/ ABC
Mkaouer et al., 2015	2014	Many-Objective Software Remodularization Using NSGA-III	GA
Mkaouer et al., 2014	2014	High Dimensional Search-Based Software Engineering: Finding Tradeoffs Among 15 Objectives For Automating Software Refactoring Using NSGAIII	GA
Mkaouer et al., (2014)	2014	Software Refactoring Under Uncertainty: A Robust Multi-Objective Approach	GA/PSO
Mkaouer et al., (2015)	2015	On The Use Of Many Quality Attributes For Software Refactoring: A Many Objective Search-Based Software Engineering Approach	GA
Mkaouer et al. (2016)	2016	A Robust Multi-Objective Approach To Balance Severity And Importance Of Refactoring Opportunities	GA/PSO
Moghadam and Ó Cinnéide (2011)	2011	Code-Imp: A Tool For Automated Search-Based Refactoring	
Moghadam and Ó Cinnéide (2012)	2012	Automated Refactoring Using Design Differencing	
Mohan et al. (2016)	2016	Technical Debt Reduction Using Search Based Automated Refactoring	HC/SA
Morales (2015)	2015	Towards A Framework For Automatic Correction Of Anti-Patterns	
Morales et al. (2016)	2016	Finding The Best Compromise Between Design Quality And Testing Effort During Refactoring	GA
Morales et al. (2016)	2016	On The Use Of Developers' Context For Automatic Refactoring Of Software Anti-Patterns	SA/GA/ VNS
Ó Cinnéide and Nixon (1999a)	1999	Automated Application Of Design Patterns To Legacy Code	

Table 10 Papers On Search-Based Refactoring *(Continued)*

Authors [Ref]	Year	Title	Search Technique
Ó Cinnéide and Nixon (1999b)	1999	A Methodology For The Automated Introduction Of Design Patterns	
Ó Cinnéide et al. (2011)	2011	Automated Refactoring For Testability	HC
Ó Cinnéide et al. (2012)	2012	Experimental Assessment Of Software Metrics Using Automated Refactoring	HC
Ó Cinnéide et al. (2016)	2016	An Experimental Search-Based Approach To Cohesion Metric Evaluation	HC
Cinnéide 2000	2000	Automated Refactoring To Introduce Design Patterns	
O'Keeffe and Ó Cinnéide (2003)	2003	A Stochastic Approach To Automated Design Improvement	SA
O'Keeffe and Ó Cinnéide (2004)	2004	Towards Automated Design Improvement Through Combinatorial Optimisation	SA
O'Keeffe and Ó Cinnéide (2006)	2006	Search-Based Software Maintenance	HC/SA
O'Keeffe and Ó Cinnéide (2008b)	2007	Search-Based Refactoring: An Empirical Study	HC/SA/GA
O'Keeffe and Ó Cinnéide (2007a)	2007	Getting The Most From Search-Based Refactoring	HC/SA/GA
O'Keeffe and Ó Cinnéide (2007b)	2007	Automated Design Improvement By Example	HC
O'Keeffe and Ó Cinnéide (2008a)	2008	Search-Based Refactoring For Software Maintenance	HC/SA
Ouni et al. (2012)	2012	Search-Based Refactoring: Towards Semantics Preservation	GA
Ouni et al. (2013)	2013	Maintainability Defects Detection And Correction: A Multi-Objective Approach	GA/GP
Ouni et al. (2013)	2013	The Use Of Development History In Software Refactoring Using A Multi-Objective Evolutionary Algorithm	GA
Ouni et al. (2013)	2013	Search-Based Refactoring Using Recorded Code Changes	GA
Ouni et al. (2015)	2015	Improving Multi-Objective Code-Smells Correction Using Development History	GA
Ouni et al. (2015)	2015	Prioritizing Code-Smells Correction Tasks Using Chemical Reaction Optimization	CRO/GA/ SA/ PSO
Pérez et al. (2013)	2013	A Proposal For Fixing Design Smells Using Software Refactoring History	
Seng et al. (2006)	2006	Search-Based Determination Of Refactorings For Improving The Class Structure Of Object-Oriented Systems	EA
Simons et al. (2015)	2015	Search-Based Refactoring: Metrics Are Not Enough	
Veerappa and Harrison (2013)	2013	An Empirical Validation Of Coupling Metrics Using Automated Refactoring	HC
Vivanco and Pizzi (2004)	2004	Finding Effective Software Metrics To Classify Maintainability Using A Parallel Genetic Algorithm	GA
Wang et al. (2015)	2015	On The Use Of Time Series And Search Based Software Engineering For Refactoring Recommendation	GA

Table 11 Papers On SBSM

Authors [Ref]	Title	Paper Type	Journal/ Conference	Benchmark Type	Benchmarks
Ó Cinnéide and Nixon (2007b)	A Methodology For The Automated Introduction Of Design Patterns	Conference	ICSM	None	
Pérez et al. (2015)	A Proposal For Fixing Design Smells Using Software Refactoring History	Conference	RefTest	None	
Mkaouer et al. (2004)	A Robust Multi-Objective Approach To Balance Severity And Importance Of Refactoring Opportunities	Journal	Empirical Software Engineering.	Open-Source/ Industrial	Xerces-J JFreeChart GanttProject ApacheAnt JHotDraw Rhino Log4J Nutch JDI-Ford
O'Keeffe and Ó Cinnéide (2008)	A Stochastic Approach To Automated Design Improvement	Conference	PPPJ	Test	
Koc et al. (2011)	An Empirical Study About Search-Based Refactoring Using Alternative Multiple And Population-Based Search Techniques	Book Section	pp. 59–66	Open-Source/ In-House	Beaver Mango JFlex Apache Xml-Rpc JSON Mosaic
Veerappa and Harrison (2011)	An Empirical Validation Of Coupling Metrics Using Automated Refactoring	Conference	ESEM	Open-Source	GanttProject JabRef JHotDraw JFreeChart XOM JRDF Art of Illusion JGraphX
Ó Cinnéide et al. (2012)	An Experimental Search-Based Approach To Cohesion Metric Evaluation	Journal	Empirical Software Engineering.	Open-Source	JHotDraw XOM Art of Illusion GanttProject JabRef JRDF JTar JGraphX HTMLUnit jSMPP
Bakar et al. (2012)	Applying Evolution Programming Search Based Software Engineering (SBSE) In Selecting The Best Open Source Software	Conference	ISCAIE	None	

Table 11 Papers On SBSM *(Continued)*

Authors [Ref]	Title	Paper Type	Journal/ Conference	Benchmark Type	Benchmarks
	Maintainability Metrics				
Ó Cinnéide and Nixon (2012)	Automated Application Of Design Patterns To Legacy Code	Conference	Workshop on Object-Oriented Technology.	None	
O'Keeffe and Ó Cinnéide (2012)	Automated Design Improvement By Example	Conference	SoMeT	Open-Source	Beaver Spec-Check Mango
Ó Cinnéide et al. (2012b)	Automated Refactoring For Testability	Conference	ICST	Test	
Ó Cinnéide (2011)	Automated Refactoring To Introduce Design Patterns	Conference	ICSE	None	
Moghadam and Ó Cinnéide (2013)	Automated Refactoring Using Design Differencing	Conference	CSMR	Open-Source	JHotDraw JGraphX JTar HTMLUnit GanttProject XOM
Moghadam and Ó Cinnéide (2008)	Code-Imp: A Tool For Automated Search-Based Refactoring	Conference	WRT	None	
Kessentini et al. (2015)	Design Defects Detection And Correction By Example	Conference	ICSM	Open-Source	GanttProject Xerces-J ArgoUML QuickUML
Harman et al. (2015)	Dynamic Adaptive Search Based Software Engineering	Conference	ESEM	None	
Harman et al. (2013)	Dynamic Adaptive Search Based Software Engineering Needs Fast Approximate Metrics	Conference	WETSoM	None	
Di Penta (2008)	Evolution Doctor: A Framework To Control Software System Evolution	Conference	CSMR	Open-Source	GRASS KDE MySQL Samba
Fatiregun et al. (2007)	Evolving Transformation Sequences Using Genetic Algorithms	Conference	SCAM	Test	
Kessentini et al. (2013)	Example-Based Design Defects Detection And Correction	Conference	ICPC	Open-Source	GanttProject Xerces-J ArgoUML QuickUML Log4J Azureus

Table 11 Papers On SBSM *(Continued)*

Authors [Ref]	Title	Paper Type	Journal/ Conference	Benchmark Type	Benchmarks
Ó Cinnéide et al. (2013)	Experimental Assessment Of Software Metrics Using Automated Refactoring	Journal	Empirical Software Engineering.	Open-Source	JHotDraw XOM Art of Illusion GanttProject JabRef JRDF JTar JGraphX
Vivanco and Pizzi (2013)	Finding Effective Software Metrics To Classify Maintainability Using A Parallel Genetic Algorithm	Conference	GECCO	In-House	Unspecified
Morales et al. (2012a)	Finding The Best Compromise Between Design Quality And Testing Effort During Refactoring	Conference	SANER	Open-Source	ArgoUML GanttProject JHotDraw Mylyn
O'Keeffe and Ó Cinnéide (2011)	Getting The Most From Search-Based Refactoring	Conference	GECCO	Open-Source	Beaver Mango EAOP Spec-Check
Mkaouer et al. (2005)	High Dimensional Search-Based Software Engineering: Finding Tradeoffs Among 15 Objectives For Automating Software Refactoring Using NSGA-III	Conference	GECCO	Open-Source	Apache Ant ArgoUML GanttProject Azureus Xerces-J
Ouni et al. (2015)	Improving Multi-Objective Code-Smells Correction Using Development History	Journal	Journal Of Systems And Software.	Open-Source	Xerces-J JFreeChart GanttProject Art of Illusion JHotDraw
Ghaith and Ó Cinnéide (2005)	Improving Software Security Using Search-Based Refactoring	Conference	SSBSE	Open-Source	Wife
Ouni et al. (2015)	Maintainability Defects Detection And Correction: A Multi-Objective Approach	Journal	Automated Software Engineering.	Open-Source	GanttProject QuickUML Azureus Log4J ArgoUML Xerces-J
Mkaouer et al. (2006)	Many-Objective Software Remodularization Using NSGA-III	Journal	ACM Transactions On Software Engineering And Methodology.	Open-Source/ Industrial	Xerces-J JHotDraw JFreeChart GanttProject JDI-Ford
Morales et al. (2009)	On The Use Of Developers' Context For Automatic Refactoring Of	Journal	Journal Of Systems And Software.	Open-Source	Mylyn PDE Platform

Table 11 Papers On SBSM *(Continued)*

Authors [Ref]	Title	Paper Type	Journal/ Conference	Benchmark Type	Benchmarks
	Software Anti-Patterns				
Amal et al. (2011)	On The Use Of Machine Learning And Search-Based Software Engineering For Ill-Defined Fitness Function: A Case Study On Software Refactoring	Conference	SSBSE	Open-Source	Xerces-J JFreeChart GanttProject Apache Ant JHotDraw Rhino
Mkaouer et al. (2011)	On The Use Of Many Quality Attributes For Software Refactoring: A Many Objective Search-Based Software Engineering Approach	Journal	Empirical Software Engineering.	Open-Source/ Industrial	ArgoUML Xerces-J Apache Ant GanttProject Azureus JDI-Ford
Wang et al. (2015)	On The Use Of Time Series And Search Based Software Engineering For Refactoring Recommendation	Conference	MEDES	Open-Source/ Industrial	JFreeChart FindBugs Hibernate Pixelitor JDI-Ford
Harman and Tratt (2007)	Pareto Optimal Search Based Refactoring At The Design Level	Conference	GECCO	Open-Source	JHotDraw Maven XOM
Ouni et al. (2007)	Prioritizing Code-Smells Correction Tasks Using Chemical Reaction Optimization	Journal	Software Quality Journal.	Open-Source	Xerces-J JFreeChart GanttProject Art of Illusion JHotDraw
Harman (2004)	Refactoring As Testability Transformation	Conference	ICSTW	None	
Seng et al. (2015)	Search-Based Determination Of Refactorings For Improving The Class Structure Of Object-Oriented Systems	Conference	GECCO	Open-Source	JHotDraw
O'Keeffe and Ó Cinnéide (2006)	Search-Based Refactoring For Software Maintenance	Journal	Journal Of Systems And Software.	Open-Source	Spec-Check Beaver
Ouni et al. (2016)	Search-Based Refactoring Using Recorded Code Changes	Conference	CSMR	Open-Source	GanttProject Xerces-J JHotDraw
O'Keeffe and Ó		Journal	Journal Of Software	Open-Source	Spec-Check Beaver

Table 11 Papers On SBSM *(Continued)*

Authors [Ref]	Title	Paper Type	Journal/ Conference	Benchmark Type	Benchmarks
Cinnéide (2016)	Search-Based Refactoring: An Empirical Study		Maintenance And Evolution: Research and Practice.		EAOP Mango Grammatica
Simons et al. (2011)	Search-Based Refactoring: Metrics Are Not Enough	Conference	SSBSE	Test	
Ouni et al., 2012	Search-Based Refactoring: Towards Semantics Preservation	Conference	ICSM	Open-Source	GanttProject Xerces-J
O'Keeffe and Ó Cinnéide (2006)	Search-Based Software Maintenance	Conference	CSMR	Open-Source	Spec-Check Spec Raytrace
Mkaouer et al. (2014)	Software Refactoring Under Uncertainty: A Robust Multi-Objective Approach	Conference	GECCO	Open-Source	Xerces-J JFreeChart GanttProject Apache Ant JHotDraw Rhino
Mohan et al. (2012)	Technical Debt Reduction Using Search Based Automated Refactoring	Journal	Journal Of Systems And Software.	Open-Source	JSON JFlex Apache Xml-Rpc Mango Beaver JHotDraw
Ouni et al. (2014)	The Use Of Development History In Software Refactoring Using A Multi-Objective Evolutionary Algorithm	Conference	GECCO	Open-Source	JFreeChart Xerces-J
Morales (2009)	Towards A Framework For Automatic Correction Of Anti-Patterns	Conference	SANER	None	
O'Keeffe and Ó Cinnéide (2012)	Towards Automated Design Improvement Through Combinatorial Optimisation	Conference	WoDiSEE	Test	
Griffith et al. (2008)	TrueRefactor: An Automated Refactoring Tool To Improve Legacy System And Application Comprehensibility	Conference	ISCA	Test	
Kessentini et al. (2016)	What You Like In Design Use To Correct Bad-Smells	Journal	Software Quality Journal.	Open-Source	GanttProject Xerces-J QuickUML JHotDraw

Table 12 Literature Reviews

Authors [Ref]	Title	Paper Type	Journal/Conference
Al Dallal (2015)	Identifying Refactoring Opportunities In Object-Oriented Code: A Systematic Literature Review	Journal	Information And Software Technology.
Ferrucci et al. (2010)	Search-Based Software Project Management	Book Section	pp. 373–399
Harman and McMinn (2009)	A Theoretical And Empirical Study Of Search-Based Testing: Local, Global And Hybrid Search	Journal	IEEE Transactions On Software Engineering.
Harman et al. (1999a)	Search Based Software Engineering: A Comprehensive Analysis And Review Of Trends Techniques And Applications	Report	
Harman et al. (1999)	Search Based Software Engineering: Techniques, Taxonomy, Tutorial	Book Section	pp. 1–59
Harman et al. (2012)	Search Based Software Engineering: Trends, Techniques And Applications	Journal	ACM Computing Surveys.
Mariani and Vergilio (2012)	A Systematic Review On Search-Based Refactoring	Journal	Information And Software Technology.
McMinn (2017)	Search-Based Software Test Data Generation: A Survey	Journal	Software Testing, Verification And Reliability.
Misbhauddin and Alshayeb (2015)	UML Model Refactoring: A Systematic Literature Review	Journal	Empirical Software Engineering.
Pitangueira et al. (1999b)	A Systematic Review Of Software Requirements Selection And Prioritization Using SBSE Approaches	Conference	SSBSE
Räihä (2009)	A Survey On Search Based Software Engineering	Journal	Computer Science Review.
Räihä (1994)	An Updated Survey On Search Based Software Engineering	Report	
Sayyad and Ammar (2000)	Pareto-Optimal Search-Based Software Engineering (POSBSE): A Literature Survey	Conference	RAISE

Table 13 Amount Of Papers Per Author

Authors	Amount Of Papers
Mel Ó Cinnéide	21
Marouane Kessentini	16
Ali Ouni	8
Houari Sahraoui	7
Mark Harman	7
Mark O′ Keeffe	7
Slim Bechikh	6
Iman Hemati Moghadam	5
Kalyanmoy Deb	5
Wiem Mkaouer	5
Laurence Tratt	3
John A. Clark	2
Mohamed Salah Hamdi	2
Mounir Boukadoum	2
Paddy Nixon	2
Steve Counsell	2
Wael Kessentini	2
A. B. Sultan	1
A. D. Bakar	1
Abdelkarim Erradi	1
Alessandro Murgia	1
Ali Andac	1
Boukhdhir Amal	1
Clemente Izurieta	1
Chris Simons	1
David Burkhart	1
David R. White	1
Deji Fatiregun	1
Dermot Boyle	1
Des Greer	1
Edmund Burke	1
Ekin Koc	1
Foutse Khomh	1
Francisco Chicano	1
Giuliano Antoniol	1
H. Zulzalil	1
Hanzhang Wang	1
Haythem Meddeb	1
Hurevren Kilic	1
Ibrahim Cereci	1
Isaac Griffith	1
J. Din	1
Javier Pérez	1
Jeremy Singer	1

Table 13 Amount Of Papers Per Author *(Continued)*

Authors	Amount Of Papers
Johannes Stammel	1
Josselin Dea	1
Katsuro Inoue	1
Khaled Ghedira	1
Lamjed Ben Said	1
Massimiliano Di Penta	1
Michael Mohan	1
Nicolino Pizzi	1
Nur Ersoy	1
Olaf Seng	1
Patrice Kontchou	1
Paul McMullan	1
Rachel Harrison	1
Rim Mahaouachi	1
Robert M. Hierons	1
Rodrigo Morales	1
Rodrigo Vivanco	1
Scott Wahl	1
Serge Demeyer	1
Shadi Ghaith	1
Shinpei Hayashi	1
Varsha Veerappa	1
William Grosky	1
Xin Yao	1
Zelal Seda Camlidere	1
Zéphyrin Soh	1

Table 14 Amount Of Papers Per Conference

Conference	Amount Of Papers
GECCO	7
CSMR	4
ICSM	3
SSBSE	3
ESEM	2
ICST/ICSTW	2
SANER	2
ICPC	1
ICSE	1
ISCA	1
ISCAIE	1
MEDES	1
PPPJ	1
RefTest	1
SCAM	1
SoMeT	1
WETSoM	1
WoDiSEE	1
Workshop on Object-Oriented Technology	1
WRT	1

Table 15 Amount Of Papers Per Open Source Benchmark

Benchmarks	Studies Used
GanttProject	19
JHotDraw	16
Xerces-J	15
JFreeChart	9
ArgoUML	6
Beaver	6
Apache Ant	5
Art of Illusion	5
Mango	5
Spec-Check	5
XOM	5
Azureus	4
JGraphX	4
QuickUML	4
JabRef	3
JRDF	3
JTar	3
Log4j	3
Rhino	3
Apache XML-RPC	2
EAOP	2
HTMLUnit	2
JFlex	2
JSON	2
Mylyn	2
FindBugs	1
Grammatica	1
GRASS	1
Hibernate	1
jSMPP	1
KDE	1
Maven	1
MySQL	1
Nutch	1
PDE	1
Pixelitor	1
Platform	1
Samba	1
Spec-Raytrace	1
Wife	1

11 Additional file

Additional file 1: Extra analysis of SBSM papers and a listing of literature reviews related to SBSE. (XLSX 22 kb)

Abbreviations
ABC: Artificial Bee Colony; ANN: Artificial Neural Network; CBO: Coupling Between Objects; 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; LSCC: Low-level Similarity-based Class Cohesion; MOEA: Multi-Objective Evolutionary Algorithm; PSO: Particle Swarm Optimization; QMOOD: Quality Model for Object-Oriented Design; ReCon: Refactoring approach based on task Context; RFC: Response For Class; SA: Simulated Annealing; SBSE: Search-Based Software Engineering; SCOM: Sensitive Class Cohesion; SDMPC: Standard Deviation of Methods Per Class; SOA: Swarm Optimisation Algorithm; TCC: Tight Class Cohesion; VNS: Variable Neighborhood Search; WSL: Wide-Spectrum Language

Acknowledgments
Not applicable.

Funding
The research for this paper contributes to a PhD project funded by the EPSRC grant EP/M506400/1. The funding body did not contribute to the design of the study, the writing of the manuscript or the collection, analysis or interpretation of data.

Availability of data and materials
The datasets supporting the conclusions of this article are included within the article and its additional files.

Authors' contributions
MM carried out the planning, execution and data analysis of the research as part of his ongoing doctoral thesis and worked on the manuscript. DG supervised the research and also worked on the manuscript. All authors read and approved the final manuscript.

Author's information
MM received his MEng degree in Computer Games Design And Development at Queen's University Belfast in 2014. During the degree, he undertook a 12 month placement in industry as a software engineer. He is currently a Ph.D. student at Queen's University. He has previously published one other peer-reviewed paper related to technical debt in software maintenance. His research interests include search-based software engineering, software maintenance, automated refactoring and multi-objective search techniques.
DG is a senior lecturer at Queens University, Belfast. He earned a Master of Science and Doctorate at University of Ulster. His research can be collectively described as how to better manage and adapt to change, both in the software product and the software development process. He has published over 50 research papers, many of which have arisen from research in an empirical context, in collaboration with industry.

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 25 May 2017 Accepted: 19 January 2018
Published online: 07 February 2018

References
Abbass HA, Sarker R, Newton C (2001) PDE: a Pareto–frontier differential evolution approach for multi-objective optimization problems. In: IEEE Congr. Evol. Comput. CEC 2001.

Al Dallal J (2015) Identifying refactoring opportunities in object-oriented code: a systematic literature review. Inf Softw Technol 58:231–249. https://doi.org/10.1016/j.infsof.2014.08.002.

Amal B, Kessentini M, Bechikh S, Dea J, Ben Said L (2014) On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring. In: 6th Int. Symp. Search-based Softw. Eng. SSBSE 2014, pp 31–45.

Assunção WKG, Colanzi TE, Pozo ATR, Vergilio SR (2011) Establishing integration test orders of classes with several coupling measures. In: 13th Annu. Genet. Evol. Comput. Conf. GECCO. McGraw-Hill Higher Education; 2011, pp 1867–1874. https://doi.org/10.1145/2001576.2001827.

Bakar AD, Sultan AB, Zulzalil H, Din J (2012a) Applying evolution programming search based software engineering (SBSE) in selecting the best open source software maintainability metrics. International Symposium on Computer Applications and Industrial Electronics, ISCAIE, IEEE; 2012. pp 70–73. https://doi.org/10.1109/ISCAIE.2012.6482071.

Bakar AD, Sultan ABM, Zulzalil H, Din J (2012b) Review on "maintainability" metrics in open source software. Int Rev Comput Softw 7:903–908.

Bansiya J, Davis CG (2002) A hierarchical model for object-oriented design quality assessment. IEEE Trans Softw Eng 28: 4–17. https://doi.org/10.1109/32.979986.
Bell D (2000) Software engineering: a programming approach. Prentice Hall, Addison.
Bowman M, Briand LC, Labiche Y (2010) Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. IEEE Trans Softw Eng 36:817–837. https://doi.org/10.1109/TSE.2010.70.
Brasil MMA, Da Silva TGN, De Freitas FG, De Souza JT, Cortés MI (2011) Multiobjective software release planning with dependent requirements and undefined number of releases. In: 13th Int. conf. Enterp. Inf. Syst. ICEIS 2011, pp 97–107.
Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20:476–493.
Cinnéide MÓ (2000) Automated refactoring to introduce design patterns. In: Int. conf. Softw. Eng. ICSE 2000, pp 722–724.
Cinnéide MÓ, Boyle D, Moghadam IH (2011) Automated refactoring for testability. In: 4th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2011. IEEE, pp 437–443. https://doi.org/10.1109/ICSTW.2011.23.
Cinnéide MÓ, Moghadam IH, Harman M, Counsell S, Tratt L (2016) An experimental search-based approach to cohesion metric evaluation. Empir Softw Eng. https://doi.org/10.1007/s10664-016-9427-7.
Cinnéide MÓ, Nixon P (1999a) Automated application of design patterns to legacy code. In: Work. Object-oriented Technol, pp 1–5.
Cinnéide MÓ, Nixon P (1999b) A methodology for the automated introduction of design patterns. In: IEEE Int. conf. Softw. Maintenance, ICSM 1999, pp 1–10.
Cinnéide MÓ, Tratt L, Harman M, Counsell S, Moghadam IH (2012) Experimental assessment of software metrics using automated refactoring. In: ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas. ESEM 2012, pp 49–58.
Coello Coello CA (1999) A comprehensive survey of evolutionary-based multiobjective optimization techniques. Knowl Inf Syst. https://doi.org/10.1017/CBO9781107415324.004.
Colanzi TE, Assunção WKG, Vergilio SR, Pozo ATR (2011) Generating integration test orders for aspect-oriented software with multi-objective algorithms. In: Latin-American work. Asp. Softw. Dev. LA-WASP 2011.
Corne DW, Knowles JD, Oates MJ (2000) The Pareto envelope-based selection algorithm for multiobjective optimization. In: 6th Int. conf. Parallel Probl. Solving from nature. https://doi.org/10.1007/s13398-014-0173-7.2.
De Freitas FG, De Souza JT (2011) Ten years of search based software engineering: a Bibliometric analysis. In: 3rd International Symposium On Search-Based Software Engineering, SSBSE 2011, pp 18–32. https://doi.org/10.1007/978-3-642-23716-4_5.
Deb K, Jain H (2013) An evolutionary many-objective optimization algorithm using reference-point based non-dominated sorting approach, part I: solving problems with box constraints. IEEE Trans Evol Comput 18:1–23. https://doi.org/10.1109/TEVC.2013.2281534.
Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans Evol Comput 6:182–197. https://doi.org/10.1109/4235.996017.
Di Penta M (2005) Evolution doctor: a framework to control software system evolution. In: 9th European Conference on Software Maintenance and Reengineering, CSMR 2005. IEEE, pp 280–283. https://doi.org/10.1109/CSMR.2005.29.
Durillo JJ, Zhang Y, Alba E, Harman M, Nebro AJ (2011) A study of the bi-objective next release problem. Empir Softw Eng 16:29–60. https://doi.org/10.1007/s10664-010-9147-3.
Durillo JJ, Zhang Y, Alba E, Nebro AJ (2009) A study of the multi-objective next release problem. In: 1st Int. Symp. Search-based Softw. Eng. SSBSE 2009, pp 49–58. https://doi.org/10.1109/SSBSE.2009.21.
Fatiregun D, Harman M, Hierons RM (2004) Evolving transformation sequences using genetic algorithms. In: 4th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2004. IEEE Comput. Soc, pp 65–74. https://doi.org/10.1109/SCAM.2004.11.
Finkelstein A, Harman M, Mansouri SA, Ren J, Zhang Y (2008) "Fairness analysis" in requirements assignments. In: 16th IEEE Int. Requir. Eng. Conf.
Fowler M, Refactoring: improving the design of existing code, 1999.
Gamma E, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, 1994.
Ghaith S, Cinnéide MÓ (2012) Improving software security using search-based refactoring. In: 4th Int. Symp. Search-based Softw. Eng. SSBSE 2012, pp 121–135. https://doi.org/10.1007/978-3-642-33119-0_10.
Griffith I, Wahl S, Izurieta C (2011) TrueRefactor: an automated refactoring tool to improve legacy system and application comprehensibility. In: 24th Int. conf. Comput. Appl. Ind. Eng. ISCA 2011.
Harman M (2011) Refactoring as testability transformation. In: 4th IEEE Int. conf. Softw. Testing, Verif. Valid. Work. ICSTW 2011, pp 1–8. https://doi.org/10.1109/ICSTW.2011.38.
Harman M, Burke E, Clark JA, Yao X (2012) Dynamic adaptive search based software engineering. In: ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas. ESEM 2012, pp 1–8.
Harman M, Clark J, Cinnéide MÓ (2013) Dynamic adaptive search based software engineering needs fast approximate metrics. In: 4th Int. work. Emerg. Trends Softw. Metrics, WETSoM 2013, pp 1–6.
Harman M, Jones BF (2001) Search-based software engineering. Inf Softw Technol 43:833–839. https://doi.org/10.1016/S0950-5849(01)00189-6.
Harman M, S.A. Mansouri, Y. Zhang, Search based software engineering: a comprehensive analysis and review of trends techniques and applications, 2009.
Harman M, Mansouri SA, Zhang Y (2012) Search based software engineering: trends, techniques and applications. ACM Comput Surv 45:1–64. https://doi.org/10.1145/0000000.0000000.
Harman M, McMinn P, De Souza JT, Yoo S (2012) Search based software engineering: techniques, taxonomy, tutorial. In: Empir. Softw. Eng. Verif, pp 1–59.
Harman M, Tratt L (2007) Pareto optimal search based refactoring at the design level. In: 9th Annu. Conf. Genet. Evol. Comput. GECCO 2007, pp 1106–1113.
Horn J, Nafpliotis N, Goldberg DE (1994) A Niched Pareto genetic algorithm for multiobjective optimization. In: IEEE world conf. Comput. Intell. https://doi.org/10.1109/ICEC.1994.350037.
Jain H, Deb K (2014) 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 18:602–622. https://doi.org/10.1109/TEVC.2013.2281534.

Kessentini M, Kessentini W, Erradi A (2011) Example-based design defects detection and correction. In: 19th Int. conf. Progr. Comprehension, ICPC 2011, pp 1–32.
Kessentini M, Kessentini W, Sahraoui H, Boukadoum M, Ouni A (2011) Design defects detection and correction by example. In: IEEE Int. conf. Softw. Eng. ICSM 2011, pp 81–90. https://doi.org/10.1109/ICPC.2011.22.
Kessentini M, Mahaouachi R, Ghedira K (2012) What you like in design use to correct bad-smells. Softw Qual J 21:551–571. https://doi.org/10.1007/s11219-012-9187-6.
Knowles JD, Corne DW (2000) Approximating the nondominated front using the Pareto archived evolution strategy. J Evol Comput 8:149–172.
Knowles JD, Corne DW, Paes M (2000) A Memetic algorithm for multiobjective optimization. In: IEEE Congr. Evol. Comput. CEC 2000, pp 325–332 doi:citeulike-article-id:8823683.
Koc E, Ersoy N, Andac A, Camlidere ZS, Cereci I, Kilic H (2012) An empirical study about search-based refactoring using alternative multiple and population-based search techniques. In: Gelenbe E, Lent R, Sakellari G (eds) Comput. Inf. Sci. II. Springer London, London, pp 59–66. https://doi.org/10.1007/978-1-4471-2155-8.
Maia CLB, De Freitas FG, De Souza JT (2010) Applying search-based techniques for requirements- based test case prioritization. In: Proc. Brazilian work. Optim. Softw. Eng. WOES 2010, pp 24–31.
Maia CLB, Do Carmo RAF, De Freitas FG, De Campos GAL, De Souza JT. A multi-objective approach for the regression test case selection problem. XLI Brazilian Symp. Oper. Res. XLI SBPO 2009;2009:1824–1835.
Mariani T, Vergilio SR (2017) A systematic review on search-based refactoring. Inf Softw Technol 83:14–34. https://doi.org/10.1016/j.infsof.2016.11.009.
Martin RC, Agile software development, principles, patterns, and practices, 2003.
Misbhauddin M, Alshayeb M (2015) UML model refactoring: a systematic literature review. Empir Softw Eng 20:206–251.
Mkaouer MW, Kessentini M, Bechikh S, Cinnéide MÓ, Deb K (2015) On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. Empir Softw Eng. https://doi.org/10.1007/s10664-015-9414-4.
Mkaouer MW, Kessentini M, Cinnéide MÓ, Hayashi S, Deb K (2016) A robust multi-objective approach to balance severity and importance of refactoring opportunities. Empir Softw Eng. https://doi.org/10.1007/s10664-016-9426-8.
Mkaouer W, Kessentini M, Bechikh S, Cinnéide MÓ, Deb K (2014) Software refactoring under uncertainty: a robust multi-objective approach. In: Genet. Evol. Comput. Conf. GECCO 2014.
Mkaouer W, Kessentini M, Bechikh S, Deb K, Cinnéide MÓ (2014) High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III. In: Genet. Evol. Comput. Conf. GECCO 2014.
Mkaouer W, Kessentini M, Kontchou P, Deb K, Bechikh S, Ouni A (2015) Many-objective software Remodularization using NSGA-III. ACM Trans Softw Eng Methodol 24.
Moghadam IH, Cinnéide MÓ (2011) Code-imp: a tool for automated search-based refactoring. In: 4th work. Refactoring tools, WRT 2011, pp 41–44.
Moghadam IH, Cinnéide MÓ (2012) Automated refactoring using design differencing. In: 16th conf. Softw. Maint. Reengineering, CSMR 2012, pp 43–52.
Mohan M, Greer D, McMullan P (2016) Technical debt reduction using search based automated refactoring. J. Syst. Softw. 120:183–194. https://doi.org/10.1016/j.jss.2016.05.019.
Morales R (2015) Towards a framework for automatic correction of anti-patterns. In: 22nd Int. conf. Softw. Anal. Evol. Reengineering, SANER 2015, pp 603–604. https://doi.org/10.1109/SANER.2015.7081891.
Morales R, Sabané A, Musavi P, Khomh F, Chicano F, Antoniol G (2016) Finding the best compromise between design quality and testing effort during refactoring. In: 23rd Int. conf. Softw. Anal. Evol. Reengineering, SANER 2016, pp 24–35.
Morales R, Soh Z, Khomh F, Antoniol G, Chicano F (2016) On the use of developers' context for automatic refactoring of software anti-patterns. J Syst Softw:2–58. https://doi.org/10.1016/j.neuroimage.2011.12.085.
O'Keeffe M, Cinnéide MÓ (2003) A stochastic approach to automated design improvement. In: 2nd Int. conf. Princ. Pract. Program. Java, PPPJ 2003, pp 59–62.
O'Keeffe M, Cinnéide MÓ (2004) Towards automated design improvement through combinatorial optimisation. In: Work. Dir. Softw. Eng. Environ. WoDiSEE 2004.
O'Keeffe M, Cinnéide MÓ (2006) Search-based software maintenance. In: 10th Eur. Conf. Softw. Maint. Reengineering, CSMR 2006, pp 251–260.
O'Keeffe M, Cinnéide MÓ (2007a) Getting the most from search-based refactoring. In: 9th Annu. Conf. Genet. Evol. Comput. GECCO 2007, pp 1114–1120.
O'Keeffe M, Cinnéide MÓ (2007b) Automated design improvement by example. In: 6th conf. New trends Softw. Methodol. Tools tech. SoMeT 2007, pp 315–329.
O'Keeffe M, Cinnéide MÓ (2008a) Search-based refactoring for software maintenance. J Syst Softw 81:502–516. https://doi.org/10.1016/j.jss.2007.06.003.
O'Keeffe M, Cinnéide MÓ (2008b) Search-based refactoring: an empirical study. J Softw Maint Evol Res Pract 20:1–23.
Ouni A, Kessentini M, Bechikh S, Sahraoui H (2015) Prioritizing code-smells correction tasks using chemical reaction optimization. Softw Qual J 23. https://doi.org/10.1007/s11219-014-9233-7.
Ouni A, Kessentini M, Sahraoui H (2013) Search-based refactoring using recorded code changes. In: Eur. Conf. Softw. Maint. Reengineering, CSMR 2013, pp 221–230. https://doi.org/10.1109/CSMR.2013.31.
Ouni A, Kessentini M, Sahraoui H, Boukadoum M (2013) Maintainability defects detection and correction: a multi-objective approach. Autom Softw Eng 20:47–79. https://doi.org/10.1007/s10515-011-0098-8.
Ouni A, Kessentini M, Sahraoui H, Hamdi MS (2012) Search-based refactoring: towards semantics preservation. In: 28th IEEE Int. conf. Softw. Maintenance, ICSM 2012, pp 347–356.
Ouni A, Kessentini M, Sahraoui H, Hamdi MS (2013) The use of development history in software refactoring using a multi-objective evolutionary algorithm. In: Genet. Evol. Comput. Conf. GECCO 2013, pp 1461–1468. https://doi.org/10.1145/2463372.2463554.
Ouni A, Kessentini M, Sahraoui H, Inoue K, Hamdi MS (2015) Improving multi-objective code-smells correction using development history. J. Syst. Software. 105:18–39. https://doi.org/10.1016/j.jss.2015.03.040.

Pérez J, Murgia A, Demeyer S (2013) A proposal for fixing design smells using software refactoring history. In: Int. work. Refactoring testing, RefTest 2013, pp 1–4.
Pressman RS, Maxim BR (2000) Software engineering: a Practitioner's approach. McGraw-Hill Higher Education
Räihä O, An updated survey on search-based software design, 2009.
Räihä O (2010) A survey on search-based software design. Comput Sci Rev 4:203–249.
Seng O, Stammel J, Burkhart D (2006) Search-based determination of Refactorings for improving the class structure of object-oriented systems. In: Genet. Evol. Comput. Conf. GECCO 2006, pp 1909–1916.
Simons C, Singer J, White DR (2015) Search-based refactoring: metrics are not enough. In: 7th Int. Symp. Search-based Softw. Eng. SSBSE 2015, pp 1–14. https://doi.org/10.1007/978-3-319-22183-0_4.
Simons CL, Parmee IC (2006) Single and multi-objective genetic operators in object-oriented conceptual software design. In: 8th Annu. Conf. Genet. Evol. Comput. GECCO 2006, pp 1957–1958. https://doi.org/10.1145/1143997.1144324.
Simons CL, Parmee IC (2007) A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design. Eng Optim 39:631–648. https://doi.org/10.1080/03052150701382974.
Simons CL, Parmee IC (2008) Agent-based support for interactive search in conceptual software engineering design. In: Genet. Evol. Comput. Conf. GECCO 2008, pp 1785–1786. https://doi.org/10.1145/1389095.1389440.
Spinellis D (2005) Tool writing: a forgotten art? IEEE Softw 22:9–11. https://doi.org/10.1109/MS.2005.111.
Veerappa V, Harrison R (2013) An empirical validation of coupling metrics using automated refactoring. In: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2013. IEEE, pp 271–274. https://doi.org/10.1109/ESEM.2013.37.
Vivanco R, Pizzi N (2004) Finding effective software metrics to classify maintainability using a parallel genetic algorithm. In: 6th Annu. Conf. Genet. Evol. Comput. GECCO 2004, pp 1388–1399.
Wang H, Kessentini M, Grosky W, Meddeb H (2015) On the use of time series and search based software engineering for refactoring recommendation. In: 7th Int. conf. Manag. Comput. Collect. Intell. Digit. Ecosyst. MEDES 2015, pp 35–42.
Wang Z, Tang K, Yao X (2008) A multi-objective approach to testing resource allocation in modular software systems. In: IEEE Congr. Evol. Comput. CEC 2008, pp 1148–1153.
Xing Z, Stroulia E (2008) The JDEvAn tool suite in support of object-oriented evolutionary development. In: 13th Int. conf. Softw. Eng. ICSE 2008. ACM Press, New York, pp 951–952. https://doi.org/10.1145/1370175.1370203.
Yen GG, Lu H (2003) Dynamic multiobjective evolutionary algorithm: adaptive cell-based rank and density estimation. IEEE Trans Evol Comput 7:253–274. https://doi.org/10.1109/TEVC.2003.810068.
Yoo S, Harman M (2007) Pareto efficient multi-objective test case selection. In: Int. Symp. Softw. Test. Anal. ISSTA 2007. https://doi.org/10.1145/1273463.1273483.
Yoo S, Harman M, Ur S (2013) GPGPU test suite minimisation: search based software engineering performance improvement using graphics cards. Empir Softw Eng 18. https://doi.org/10.1007/s10664-013-9247-y.
Zhang Y, Harman M, Mansouri SA (2007) The multi-objective next release problem. In: 9th Annu. Conf. Genet. Evol. Comput. GECCO 2007. https://doi.org/10.1145/1276958.1277179.
Zitzler E, Laumanns M, Thiele L (2001) SPEA2: improving the strength Pareto evolutionary algorithm. In: Evol. Methods des. Optim. Control with Appl. To Ind. Probl. EUROGENS 2001. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.4617.
Zitzler E, Thiele L (1999) Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. IEEE Trans Evol Comput 3:257–271. https://doi.org/10.1109/4235.797969.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com