Getting the Most from Search-Based Refactoring

Mark O'Keeffe and Mel O Cinneide
School of Computer Science and Informatics, University College Dublin
Belfield, Dublin 4, Ireland
mark.okeeffe@ucd.ie, mel.ocinneide@ucd.ie

ABSTRACT

Object-oriented systems that undergo repeated addition of functionality commonly suffer a loss of quality in their underlying design. This problem must often be remedied in a costly refactoring phase before further maintenance programming can take place. Recently search-based approaches to automating the task of software refactoring, based on the concept of treating object-oriented design as a combinatorial optimisation problem, have been proposed. However, because search-based refactoring is a novel approach it has yet to be established which search techniques are most suitable for the task.

In this paper we report the results of an empirical comparison of simulated annealing, genetic algorithm and multiple ascent hill-climbing in search-based refactoring. A prototype automated refactoring tool is employed, capable of making radical changes to the design of an existing program in order that it conform more closely to a contemporary quality model. Results show multiple-ascent hill climbing to outperform both simulated annealing and genetic algorithm over a set of four input programs.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance & Enhancement — *restructuring, reverse engineering and reengineering*

General Terms

Design

Keywords

Search – Based Software Engineering, Refactoring, Object – Oriented Product Metrics, Automated Design Improvement

1. INTRODUCTION

Object-oriented systems that undergo repeated addition of functionality commonly suffer a loss of quality in their underlying design. This problem, known as *design erosion* [21] or *software decay* [9], occurs when changes are made to a program without due consideration to its overall structure and design rationale. Design erosion can be combatted by refactoring, or improving the design of a program without changing its behaviour, but even with the use of contemporary programming tools this requires significant effort on the part of the maintenance programmer.

Recently, search-based approaches to automating the task of software refactoring have been proposed by the authors [21] and Seng et al [22]. These approaches are based on the concept of object-oriented design as a combinatorial optimisation problem, where a fitness function defining design quality is constructed from a weighted sum of object-oriented metrics, and are inspired by the successful application of search-based approaches in other areas of software engineering such as subsystem clustering and test-data generation.

Once formulated suitably as a solution representation, change-effecting operator and fitness function, the problem of automated refactoring can be tackled using a wide variety of search techniques. As is the case with other search-based software engineering applications, however, the effectiveness of the various techniques may vary considerably. Because search-based refactoring is a novel approach it remains to be established which search techniques are most suitable in the general case; while the authors have compared the differing performance of hill-climbing and simulated annealing searches in two case studies [21], no thorough comparison of the effectiveness of local and evolutionary search techniques for this problem has yet been carried out.

In this paper we report the results of an empirical comparison of simulated annealing, genetic algorithm and multiple ascent hill-climbing searches. We have extended the CODe-Imp search-based refactoring tool [21] to employ a genetic algorithm search with a similar representation, crossover operator and mutation operator to that described by Seng et al [22], as well as increasing the power of the tool by adding to the number of different refactorings available for use in searching for a superior design.

The remainder of this paper is structured as follows: in section 2 we outline related work in search-based software engineering, such as module clustering, as well as discussing the state of the art in search-based refactoring. In section 3 we describe the experimental methodology for the study reported here, including the extensions made to the CODe-Imp tool. Section 4 contains the results of the study, comprised of comparisons of various parameter sets for each of

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

the search techniques employed, and a comparison of the relative performance of these techniques. We conclude and suggest some directions for future work in section 5.

2. RELATED WORK

Search-Based Software Engineering (SBSE) can be defined as the application of search-based approaches to solving optimisation problems in software engineering [10]. Such problems include *module clustering*, where a software system is reorganised into loosely coupled clusters of highly cohesive modules to aid reengineering [8, 11, 14, 19], test data generation [16], automated testing [23] and project management problems such as requirements scheduling [1] and project cost estimation [3, 6, 7]. Papers by Clark et al and Harman provide a thorough review of work in the field [5, 10].

Once a software engineering task is framed as a search problem there are numerous approaches that can be applied to solving that problem, from local searches such as exhaustive search and hill-climbing to meta-heuristic searches such as genetic algorithms (GA) and ant colony optimisation. Module clustering, for example, has been addressed using exhaustive search [15], hill-climbing [11, 13, 15, 18], genetic algorithms [8, 11, 15, 18] and simulated annealing (SA) [18]. In those studies that compared search techniques, hill-climbing was, perhaps surprisingly, found to produce better results than meta-heuristic GA searches [11, 17]. These results were echoed in search-based auto-parallelisation [24], where local searches also out-performed GA. In software clustering the meta-heuristic *simulated annealing* search was found by Mitchell et al. [18] to perform similarly to hill-climbing in terms of solution quality, but better in terms of search efficiency.

The concept of treating object-oriented design as a combinatorial optimisation problem that can be solved using a search-based approach was introduced by the authors [20], and later small case studies based on the QMOOD quality model [2] were conducted which suggested that both simulated annealing and hill-climbing are effective in solving this problem [21]. Seng et. al. [22] describe a similar approach but use a genetic algorithm to solve the combinatorial optimisation problem. The evaluation function employed by Seng is novel rather than previously validated, but is based on well-known metrics such as *Response For a Class* (RFC) and *Weighted Methods per Class* (WMC) from Chidamber & Kemerer's object-oriented metrics suite [4], among others. The authors report success in automatically repositioning displaced methods in the class structure using a GA search. However, as only the Move Method refactoring is considered the extent of change within the class structure is limited.

3. EXPERIMENTAL METHODOLOGY

In this section we describe CODe-Imp, a prototype search-based refactoring tool designed to facilitate experimentation in automatically improving the design of existing programs. In common with other search-based software engineering applications, search-based refactoring requires a solution representation, a change operator that allows for movement in the space of alternative solutions and a fitness or evaluation function that allows solutions to be ranked in terms of desirability. With these three elements in place, various search techniques can be applied in solving the problem. In sections 3.1, 3.2 and 3.3 respectively we describe the solution representation, change operator and fitness function employed in this study. In section 3.4 we briefly discuss the four search techniques employed, while in section 3.5 we describe the input programs used.

3.1 Solution Representation

In search-based refactoring, the solution representation can be a program itself, its Abstract Syntax Tree (AST) or a more abstract model. The key requirements are that it must be possible to determine what transformations can be made to the representation in order to move through the space of alternative solutions, and it must be possible to apply corresponding refactorings to the program in question in order to implement the solution.

This study employs the tool CODe-Imp (Combinatorial Optimisation Design-Improvement), developed by the authors in order to test the thesis that the maintainability of object-oriented programs can be improved by automatically refactoring them to adhere more closely to a pre-defined quality model. CODe-Imp takes Java 1.4 source code as input and extracts design metric information via a Java Program Model (JPM), calculates quality values according to an evaluation or fitness function and effects change in the current solution by applying refactorings to the AST as required by a given search technique. Output consists of the refactored input code as well as a design improvement report including quality change and metric information [21].

CODe-Imp uses a two-level representation; the actual program to be refactored is represented as its AST, but a more abstract model called the JPM is also maintained from which metric values are determined and refactoring preconditions are checked. Where possible, the JPM alone is refactored when a new solution is to be examined, but for the most complex refactorings the more robust approach of refactoring the AST and regenerating the JPM is adopted.

3.2 Change Operator

In the context of search-based refactoring, the change operator is a transformation of the solution representation that corresponds to a refactoring that can be carried out on the source code. The refactoring configuration of CODe-Imp, which was extended by the authors for the experiments reported here, consists of the fourteen refactorings described below. In CODe-Imp, complementary pairs of refactorings are employed in order that all changes made to the input design during the course of the search be reversible. This is a requirement of some search techniques that must move freely through the solution space, such as simulated annealing. All refactorings employed operate at the method/field level of granularity and higher, in order to focus on improvement of design rather than implementation issues such as correct factorisation of methods. The two principles of implementing complementary pairs of refactorings and operating at the method/field level of granularity have been upheld in extending the refactoring set; those refactorings that have been added to CODe-Imp for this study are indicated by asterisks below.

Push Down Field moves a field from some class to those subclasses that require it.

Pull Up Field moves a field from some class(es) to the immediate superclass.

Push Down Method moves a method from some class to those subclasses that require it.

Pull Up Method moves a method from some class(es) to the immediate superclass.

Extract Hierarchy adds a new subclass to a non-leaf class C in an inheritance hierarchy. A subset of the subclasses of C will inherit from the new class.

Collapse Hierarchy removes a non-leaf class from an inheritance hierarchy.

Increase Field Security* increases the security of a field from public to protected or from protected to private.

Decrease Field Security* decreases the security of a field from private to protected or from protected to public.

Increase Method Security* increases the security of a method from protected to private or from public to protected.

Decrease Method Security* decreases the security of a method from protected to public or from private to protected.

Replace Inheritance with Delegation* replaces an inheritance relationship between two classes with a delegation relationship; the former subclass will have a field of the type of the former superclass.

Replace Delegation with Inheritance* replaces a delegation relationship between two classes with an inheritance relationship; the delegating class becomes a subclass of the former delegate class.

Make Superclass Abstract* declares a constructor-less class explicitly abstract.

Make Superclass Concrete* removes the explicit declaration of an abstract class without abstract methods.

3.3 Fitness Function

The fitness function employed here is an implementation of the Understandability function from Bansiya's QMOOD[1] hierarchical design quality model [2], consisting of a weighted sum of metric quotients between two designs. Use of this design quality evaluation function was previously found by the authors to result in tangible improvements to object-oriented program design in the context of search-based refactoring [21]. The QMOOD model includes the eleven metrics described below. The weight for each metric in the Understandability function is listed beside each metric acronym; metrics are weighted positively where high values are considered to contribute to understandability and negatively where low values contribute.

- **Data Access Metric (DAM, 0.33)**

  The ratio of the number of non-public attributes to the total number of attributes declared in the class. This metric corresponds to the property *encapsulation*.

- **Cohesion Among Methods of Class (CAM, 0.33)**

  The relatedness among methods of a class, computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class. This metric corresponds to the property *cohesion*.

- **Number Of Methods (NOM, -0.33)**

  A count of all the methods defined in a class. This metric corresponds to the property *complexity*.

- **Number of Polymorphic Methods (NOP, -0.33)**

  A count of the number of methods that can exhibit polymorphic behaviour. This metric corresponds to the property *polymorphism*.

- **Direct Class Coupling (DCC, -0.33)**

  A count of the different number of classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods. This metric corresponds to the property *coupling*.

- **Design Size in Classes (DSC, -0.33)**

  A count of the total number of classes in the design. This metric corresponds to the property *design size*.

- **Average Number of Ancestors (ANA, -0.33)**

  The average number of classes from which each class inherits information. This metric corresponds to the property *abstraction*.

In addition, metrics for the design properties *messaging*, *inheritance*, *aggregation* and *hierarchies* are included in the QMOOD model but unweighted in the Understandability function.

3.4 Search Techniques

3.4.1 Genetic Algorithm

For this study we have extended the set of search techniques available to CODe-Imp by including a Genetic Algorithm (GA) implementation similar to that of Seng et al [22]. For the purpose of genetic algorithm implementation, the standard solution representation (the AST) can be considered the *phenotype*, while the sequence of refactorings carried out in order to reach that solution can be considered the *genotype*. The mutation operator employed here is identical to that described by Seng, and simply consists of adding one random refactoring to the genotype. Our crossover operator is similar to Seng's, and consists of 'cut and splice' crossover of two genotypes, so the length of the offspring genotypes can differ from that of the parents. It is of course likely that in the process of splicing genotypes we will encounter a situation where the necessary preconditions for some refactoring are not met. In such a case we discard the refactoring in question, rather than discard the entire genotype.

[1]Quality Model for Object-Oriented Design

3.4.2 Simulated Annealing

Simulated Annealing (SA) is a meta-heuristic search technique inspired by the metallurgic process of annealing, where a molten metal is cooled slowly in order to produce or preserve certain characteristics in the solid form [12]. SA has been applied to a wide range of search-based software engineering problems, and has been found to be effective in the context of software clustering [18]. SA has the advantage that it is very robust against local optima in the search space, but the disadvantage that its many parameters can make it hard to configure for any given problem.

In this study we have employed the standard geometric cooling schedule, but a very low starting temperature. This is due to the assumption that even a design of low quality from a human perspective is in fact in the top few percent when we consider the space of all possible designs.

3.4.3 Multiple Ascent Hill-Climbing

A variation on the standard first ascent hill-climbing algorithm, multiple ascent hill-climbing (HCM) is capable of achieving improved results due to its ability to escape from local optima. HCM initially acts identically to a first ascent hill-climbing search, but when a local optimum is reached a pre-defined number of random refactorings are carried out in order to move away from that point in the solution space. The search is then restarted from the randomly chosen solution. This procedure is repeated a set number of times, depending on the *number of restarts* parameter. The number of random refactorings made each time is the *restart depth* parameter. This search technique is considered a primary candidate for search-based refactoring because local searches have been shown to be effective in the similar domain of module clustering [11, 17], as well as in previous exploratory work [21].

3.4.4 Steepest Ascent Hill-Climbing

Although not considered a primary candidate for use in search-based refactoring due to its long run-times and inability to escape local optima, the performance of steepest ascent hill-climbing was used as a reference point in this study in order to carry out normalisation of results across different input programs. Because the extent to which a design can be improved varies greatly depending on such factors as how many refactorings can legally be applied, this normalisation was vital in order to establish relative performance of the search techniques mentioned above in the general case. Steepest ascent hill-climbing provided an ideal reference point due to its deterministic nature, as the same quality gain is obtained from each run on a given input program.

3.5 Input

Input consisted of four Java 1.4 programs of a maximum size of fifty top-level classes; three randomly selected from SourceForge[2] via java-source.net, and a self-contained subset of the *Spec-Benchmarks*[3] standard performance evaluation framework, to which it was known a large number of refactorings could be applied. Clearly, some refactorings reduce the number of legal refactorings that can be applied to a design, while some increase the number and others have no effect. We indicate below the number of distinct refactorings that could be applied to each design on input. The programs selected were:

1. *Beaver*, a parser generator
   - 30 top-level classes
   - 4999 SLOC
   - 177 refactorings could initially be applied
2. *Mango*, a collections library
   - 45 top-level classes
   - 1131 SLOC
   - 28 refactorings could initially be applied
3. *EAOP*, an aspect-oriented programming library
   - 40 top-level classes
   - 1164 SLOC
   - 200 refactorings could initially be applied
4. *Spec-Check*, a benchmarking program
   - 41 top-level classes
   - 4836 SLOC
   - 351 refactorings could initially be applied

4. RESULTS

Experiments were carried out on a 2.2GHz AMD Athlon powered PC with 2GB CL2 RAM. Mean processing time per solution examined was less than one second, including model building, metric extraction, quality assessment, discovery of legal refactorings, and actual (AST) refactoring. Total runtime varied between six minutes and twelve hours as discussed below, depending on the search technique employed, number of refactorings possible for the input program and the number of refactorings applied. However, CODe-Imp makes no use of concurrent processes, so there is potential to greatly decrease these run-times.

Statistical analyses were carried out using Graphpad Prism. A confidence interval of 95% was used in all statistical tests. Error bars on all figures indicate standard deviation from the mean; three replications of each experiment were performed.

4.1 Simulated Annealing

In order to determine a suitable simulated annealing cooling schedule for use in the general case of search-based refactoring, the set of input programs were automatically refactored to conform more closely to the QMOOD Understandability model under all nine permutations of cooling factor (CF) 0.990, 0.993 or 0.996 and Markov chain length (MCL) 1, 2 or 3. Figure 1 shows the mean quality increase observed across the set of input programs, with each data point normalised against the highest observed value under SA for the particular input program. Statistical analysis of these data using 2-way ANOVA revealed no significant correlation between quality gain and either cooling factor or Markov chain length, in the range investigated. Furthermore, Bonferroni post-tests revealed no significant difference between the results achieved for any pair of cooling schedules.

[2]http://sourceforge.net/
[3]http://www.spec.org/

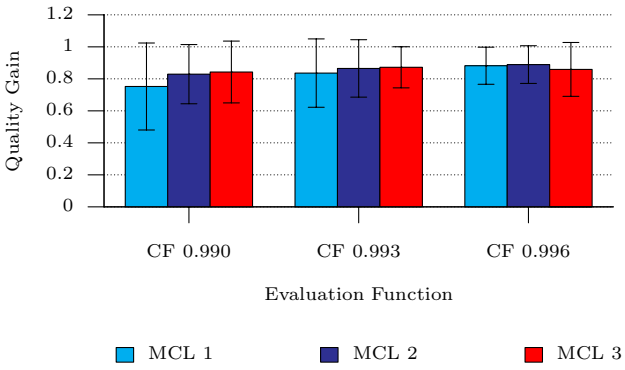

Figure 1: Mean quality change – SA

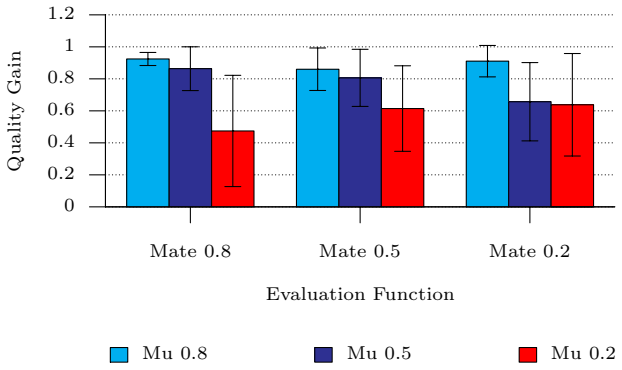

Figure 2: Mean quality change – GA

These results support the recommendation of low values for both cooling factor and Markov chain length, since more computationally expensive parameters do not yield greater quality function gains. However, the variation in mean quality gain for most of the cooling schedules investigated was large, indicating that the relative performance of SA varies greatly depending on the particular input program in question. As a result, no strong recommendation of schedule for use in the general case can be made, and use of SA in search-based refactoring would likely require a large volume of searches with different parameters in order to be confident that high-quality solutions were obtained.

4.2 Genetic Algorithm

In order to determine the ideal parameters for a genetic algorithm in the general case of search-based refactoring, the set of input programs were automatically refactored to conform more closely to the QMOOD Understandability model under all nine permutations of mutation probability (Mu) 0.8, 0.5 or 0.2 and mating probability (Mate) 0.8, 0.5 or 0.2. Figure 2 shows the mean quality increase observed across the set of input programs, with each data point normalised against the highest observed value under GA for the particular input program. Statistical analysis of these data using 2-way ANOVA revealed an extremely significant correlation between quality gain and the combination of mutation probability and mating probability (P less than 0.0001). Bonferroni post-tests indicated that the combination of Mu 0.8 and Mate 0.8 was significantly more effective over the set of input programs than any other set of parameters.

These results support the recommendation of high mutation and mating probabilities, with values of 0.8 & 0.8 yielding high mean quality increases with little deviation. However, high mutation and mating parameters result in a high computational expenditure, as discussed below. It should be noted that in many cases the application of a single refactoring did not affect the QMOOD Understandability function value, so the mutation operator employed did not always produce a change in solution quality. This is likely a factor in the observed ineffectiveness of low mutation probabilities in this study.

4.3 Multiple Ascent Hill-Climbing

In order to determine suitable parameters for multiple ascent hill-climbing in the general case of search-based refactoring, the set of input programs were automatically refactored to conform more closely to the QMOOD Understandability model under nine permutations of number of restarts (R) 1, 2 or 3 and restart depth 5, 10 or 20. Figure 3 shows the mean quality increase observed across the set of input programs, with each data point normalised against the highest observed value under HCM for the particular input program. Statistical analysis of these data using 2-way ANOVA revealed no significant correlation between quality gain and either number of restarts or restart depth. Bonferroni post-tests indicated that no parameter set produced significantly greater mean quality gains than any other.

These results support the recommendation of low values for both number of restarts and restart depth, since more computationally expensive parameters do not yield greater quality function gains.

4.4 Comparison of Searches

Figure 4 shows the mean quality increase for each search technique for the entire set of input programs. For each program/search pair the set of parameters having the highest mean quality gain were taken as the performance of that search technique on that program. These values were then normalised against the performance of steepest ascent hill-climbing (HC) for each input program, before mean quality gain over the set of input programs was calculated. Statistical analysis using Student's T-test for unpaired data assuming equal variance revealed no significant difference in mean quality increase over the set of input programs between the four search techniques. However, two other criteria are pertinent in comparing the relative performance.

Firstly, a search technique upon which a search-based refactoring tool is based must be capable of producing good results for any input, as the user will not wish to run the tool several times with different search techniques in order to obtain high-quality results. It is therefore important that quality gain is consistently high across the set of possible input programs. As can be seen from figure 4, the standard deviation of quality gain is quite large for both simulated annealing and genetic algorithm searches, small for multiple ascent hill-climbing, and zero for steepest-ascent hill climbing.

Secondly, the computational cost of the various searches is of course a factor in choosing between them. Longest relevant run times for the various search techniques were as follows:

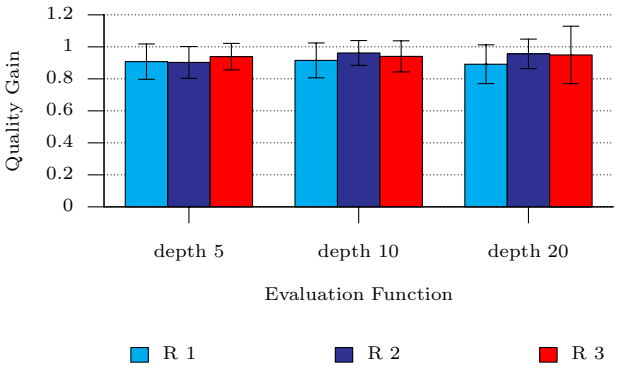

Figure 3: Mean quality change – HCM

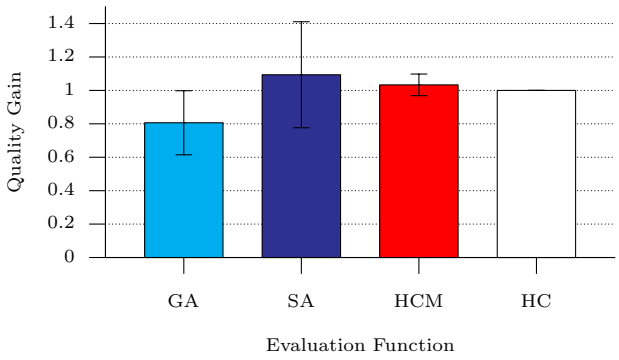

Figure 4: Overall mean quality change

SA Up to two hours for *spec-check* with most effective parameters for that input (CF 0.996, M 3).

GA Up to twelve hours before peak values were observed for most effective parameters (mut 0.8, mate 0.8) for *spec-check*.

HCM Approximately one hour for *spec-check* with most effective parameters (3 restarts, depth 5).

HC Approximately one hour in the case of *spec-check* input.

While precise runtimes are clearly implementation dependant, these results are representative of the expected relative efficiency of the algorithms. HC, for example, must examine every possible refactoring before accepting one so it is unsurprising that it requires more time than HCM with a low restart depth parameter. GA is extremely inefficient in the number of solutions it examines relative to its total runtime because the entire population of solutions must be stored in memory, resulting in a huge memory-access overhead. This is a result of the complexity of the solution representation in this domain, which necessitates more intense computation than the binary string representation used in other GA applications.

In summary, simulated annealing has several disadvantages; it is hard to recommend a cooling schedule that will generally be effective, results varied considerably across input programs and the search is quite slow. No significant advantage in terms of quality gain was observed that would make up for these shortcomings.

The genetic algorithm has the advantage that it is easy to establish a set of parameters that work well in the general case, but the disadvantages that it is costly to run and varies greatly in effectiveness for different input programs. Again, no significant advantage in terms of quality gain was observed that would make up for these shortcomings.

Multiple-ascent hill climbing stood out as the most efficient search technique in this study; it produced high-quality results across all the input programs, is relatively easy to recommend parameters for, and runs more quickly than any of the other techniques examined.

Steepest ascent hill-climbing produced surprisingly high quality solutions, suggesting that the search space is less complex than might be expected, but is slow when we consider its known inability to escape local optima.

5. CONCLUSIONS & FUTURE WORK

In this study we have investigated how best to parameterise three search techniques for use in search-based refactoring, and compared the overall performance of four. High values for mating and mutation probability are shown to produce the best results for a genetic algorithm, while low values for restart depth and numbers of restarts are shown to be most efficient for multiple ascent hill-climbing. The effectiveness of simulated annealing cooling schedules was shown to vary depending on input program.

Comparison of search techniques showed multiple–ascent hill climbing (HCM) to be most suitable for search-based refactoring. Although no statistically significant difference in mean quality gain was observed between the four search techniques, HCM performed best in terms of speed, consistency of quality gain over various input programs and consistency of quality gain for a particular parameter set. Since parameters of *restart depth* 10 or less and *number of restarts* 2 or less produced large quality gains over all input programs, HCM can be used in search-based refactoring without the large volume of searches with different parameters required by a simulated annealing implementation.

These results are consistent with the search space for the set of refactorings employed under the QMOOD Understandability function being relatively smooth, which suggests that meta-heuristic search techniques are over-powered for this particular application. Further studies using other contemporary evaluation functions will be required in order to determine whether these results can be generalised across the domain of search-based refactoring as a whole. However, the superiority of multiple-ascent hill climbing in this study leads us to conclude that search-based refactoring tools such as CODe-Imp , that utilise the QMOOD Understandability function, should make use of this search technique.

Further work is also required in the related field of automated object-oriented design quality measurement, since search-based refactoring can only be as effective at actually improving design as the quality model employed is at measuring it. Of particular interest would be an analysis of the impact of search-based refactoring on such artifacts as design patterns and 'bad smells'. We believe a synergy exists between search-based refactoring and design quality measurement, since the programmer can be shown the results of rigorously implementing a given quality model. We therefore expect that future collaboration with members of

the object-oriented measurement community will yield the next major advances in search-based refactoring.

6. REFERENCES

[1] A. J. Bagnall, V. J. Rayward-Smith, and I. M. Whittley. The next release problem. *Information & Software Technology*, 43(14):883–890, 2001.

[2] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Software Eng.*, 28(1):4–17, 2002.

[3] C. J. Burgess and M. Lefley. Can genetic programming improve software effort estimation? A comparative evaluation. *Information & Software Technology*, 43(14):863–873, 2001.

[4] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20:476–493, June 1994.

[5] J. A. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. S. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. J. Shepperd. Formulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.

[6] J. J. Dolado. A validation of the component-based method for software size estimation. *IEEE Trans. Software Eng.*, 26(10):1006–1021, 2000.

[7] J. J. Dolado. On the problem of the software cost function. *Information & Software Technology*, 43(1):61–72, 2001.

[8] D. Doval, S. Mancoridis, and B. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *International Conference on Software Tools and Engineering Practice (STEP'99)*, 1999.

[9] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[10] M. Harman and J. A. Clark. Metrics are fitness functions too. In *IEEE METRICS*, pages 58–69, 2004.

[11] M. Harman, R. M. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO*, pages 1351–1358, 2002.

[12] S. Kirkpatrick, D. G. Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[13] K. Mahdavi, M. Harman, and R. M. Hierons. A multiple hill climbing approach to software module clustering. In *ICSM*, pages 315–324, 2003.

[14] S. Mancoridis, B. S. Mitchell, Y.-F. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *ICSM*, pages 50–, 1999.

[15] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC*, pages 45–, 1998.

[16] C. C. Michael, G. McGraw, and M. Schatz. Generating software test data by evolution. *IEEE Trans. Software Eng.*, 27(12):1085–1110, 2001.

[17] B. S. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem*. PhD thesis, Drexel University Philadelphia, USA, 2002.

[18] B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *GECCO*, pages 1375–1382, 2002.

[19] B. S. Mitchell, M. Raverso, and S. Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *WICSA*, pages 181–190, 2001.

[20] M. O'Keeffe and M. Ó Cinnéide. A stochastic approach to automated design improvement. In J. F. Power and J. T. Waldron, editors, *Proceedings of the 2nd International Conference on the Principles and Practice of Programming in Java*, pages 59–62. ACM SIGAPP, Computer Science Press, Trinity College Dublin, Ireland., June 2003.

[21] M. O'Keeffe and M. Ó Cinnéide. Search-based software maintenance. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 249– 260, 2006.

[22] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916, New York, NY, USA, 2006. ACM Press.

[23] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 43(14):841–854, 2001.

[24] K. P. Williams. *Evolutionary algorithms for automatic parallelization*. PhD thesis, University of Reading, UK, Department of Computer Science, September 1998.