Search-Based Software Maintenance

Mark O'Keeffe, Mel Ó Cinnéide
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 14, Ireland
mark.okeeffe@ucd.ie, mel.ocinneide@ucd.ie

Abstract

The high cost of software maintenance could potentially be greatly reduced by the automatic refactoring of object-oriented programs to increase their understandability, adaptability and extensibility. This paper describes a novel approach in providing automated refactoring support for software maintenance; the formulation of the task as a search problem in the space of alternative designs. Such a search is guided by a quality evaluation function that must accurately reflect refactoring goals. We have constructed a search-based software maintenance tool and report here the results of experimental refactoring of two Java programs, which yielded improvements in terms of the quality functions used. We also discuss the comparative merits of the three quality functions employed and the actual effect on program design that resulted from their use.

1. Introduction

One measure of the quality of an object-oriented design is the level of difficulty encountered in carrying out maintenance programming. This is because the object-oriented approach is geared towards producing designs which are understandable, modular, and model the real world as closely as possible. However, it is not uncommon to encounter designs that have become weakened as a side-effect of the repeated addition of functionality during development (a problem referred to as *design erosion*), or have not been properly maintained in the past. Such designs can require significant refactoring in order to increase their maintainability to an acceptable level, thus increasing the cost of carrying out maintenance tasks.

The ideal solution to this problem would be the automation of some portion of the refactoring step by the application of an automated design improvement tool. Such a tool would take the current set of classes as input and output a set with the same external behaviour, but having a design that is more easily comprehended, adapted and extended. In this context, the application of refactorings can be considered movement in the space of alternative designs.

Our novel approach to automated design improvement is the formulation of the refactoring task as a search problem; given a design quality function we apply automated refactorings to a program in order to move through the space of alternative designs and search for those of highest quality. The effectiveness of the search can be measured in terms of the change in quality function, but the effectiveness of the approach itself can only be judged in terms of the actual changes made to the program and to what extend it is more maintainable than the original. For this reason, choice of design quality function is a key facet of this work.

While there exists a large body of work dealing with the measurement of design quality in terms of a set of metrics (see section 2.3), there are few examples of attempts to capture complex properties such as maintainability as a single value, as required for an evaluation function. This is perhaps not surprising, given that comparison of the design of unrelated programs with different purposes has little meaning. However, for the purpose of search-based software maintenance the evaluation function need only give meaningful quality values for alternative designs of the same program.

One model of software quality that incorporates suitable evaluation functions is Bansiya's 'Hierarchical Model for Object-Oriented Design Quality Assessment' [2], or QMOOD, which defines evaluation functions for such quality attributes as reusability, flexibility and understandability, based on eleven object-oriented design metrics. We have examined this model through experimentation, and determined that it partially meets our requirements. In the process we have demonstrated a secondary function of the search-based software maintenance approach; that by refactoring programs to comply with a given quality model we gain an additional mechanism for validation of that model.

While our ultimate goal is to determine the extent to which the refactoring step of software maintenance can be

automated, the specific goals of the work reported in this paper are:

- To determine whether a small set of automatable refactorings are sufficient to explore the suitability of proposed evaluation functions in general and QMOOD evaluation functions in particular.
- To assess the relative performance of a small set of search techniques in the context of automated refactoring of Java programs guided by QMOOD evaluation functions.
- To subjectively assess the performance of a prototype automated refactoring tool employing QMOOD evaluation functions.

2. Related Work

2.1. Search-Based Software Engineering

Search-Based Software Engineering (SBSE) can be defined as the application of search-based approaches in solving optimisation problems in software engineering [13]. Such problems include *module clustering*, where a software system is reorganised into loosely coupled clusters of highly cohesive modules to aid reengineering [11, 14, 15, 17], test data generation [16], automated testing [20] and project management problems such as requirements scheduling [1] and project cost estimation [5, 9, 10]. An overview of such work and comprehensive recent references can be found in [8] and [13] respectively. Of particular relevance to this work is [13], in which Harman proposes 'Metrics as Fitness Functions' (MAFF). Harman states that a metric can be used as the evaluation function driving a search-based software optimisation; our approach involves using a combination of a *set* of metric values to guide a search for optimal design.

2.2. Automated Design Improvement

Previous approaches to the fully automated restructuring of software have focussed on improving one particular aspect of design, such as method reuse or code factorisation. However, since object-oriented design involves numerous trade-offs, this narrow focus could result in overall quality loss. Examples of such work include that of Casais [6], who proposed algorithms to restructure class hierarchies in order to maximise abstraction, and Moore [18], who proposed a system where existing classes are discarded and replaced with a new set where methods are optimally factored – meaning code duplication is minimised.

Our approach has two main advantages over previous fully automated refactoring work. Firstly, and most significantly, the use of evaluation functions consisting of combinations of various metric values allows us to employ much richer quality models than the single-goal approaches mentioned above, which do not take into account the numerous trade-offs involved in object-oriented design. Secondly, by careful choice and precise definition of the refactorings employed we can make design-quality affecting changes to an object-oriented program without loss of domain-specific information such as class and member names; a particular disadvantage of [18].

Semi-automated approaches to design improvement mainly involve the use of metric-based rules to identify areas in need of improvement, the onus then being on the programmer to make the necessary changes. Such 'bad smell' detection has been proposed by Van Emden [12], and by Tahvildari [19], whose system also recommends 'meta-pattern transformations' that can be applied to ameliorate the defect. The drawback of such tools is, of course, that they reduce the need for programmer intervention rather than eliminate it.

2.3. Design Quality Measurement

In order to treat object-oriented design as a search problem, it is necessary to define a quality evaluation function that will serve to rank alternative designs. Furthermore, in order for an effective search to be carried out this quality function must be automatically computable from the design model at a minimal computational cost. We have conducted a survey of current metric-based object-oriented quality models and selected three of the most prominent, which are described below and assessed as to their suitability for the task in hand.

- MOOD and MOOD2 suites of Fernando Brito é Abreu et al. [3, 4] – well-established contemporary metric suite but does not define evaluation functions.
- QMOOD model of Bansiya [2] – hierarchical quality model including evaluation function definitions, but does not formally define metrics.
- MOOSE suite of Chidamber and Kemerer [7] and subsequent modifications by Li et al. [21] – well known metric suite that has been independently validated, but consists of only six metrics and does not define evaluation functions.

The QMOOD quality model was selected for this experiment because it most closely matches our requirements; implementation of QMOOD metrics implies their precise definition in the context of this work.

3. Experimental Methodology

3.1. CODe-Imp

We have constructed a prototype automated design-improvement tool called CODe-Imp[1] in order to facilitate experimentation with search-based software maintenance. CODe-Imp takes Java 1.4 source code as input and extracts metric information via a Java Program Model (JPM), calculates quality values according to an evaluation function and applies refactorings to the Abstract Syntax Tree (AST), as required by the search technique employed. Output consists of the refactored input code as well as a design improvement report including quality change and metric information.

3.2. Refactorings

The refactoring configuration of CODe-Imp for the experiments reported here consisted of the six refactorings described below. We have selected complementary pairs of refactorings so that changes made to the input design during the course of the search could be reversed. This is necessary for some search techniques (e.g. Simulated Annealing) to move freely through the space of alternative designs.

Push Down Field moves a field from some class to those subclasses that require it. This refactoring is intended to simplify the design by reducing the number of classes that possess the field.

Pull Up Field moves a field from some class(es) to the immediate superclass. This refactoring is intended to help eliminate duplicate field declarations in sibling classes.

Push Down Method moves a method from some class to those subclasses that require it. This refactoring is intended to simplify the design by reducing the size of class interfaces.

Pull Up Method moves a method from some class(es) to the immediate superclass. This refactoring is intended to help eliminate duplicate methods among sibling classes, and hence reduce code duplication in general.

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. This method is intended to help improve class cohesion and modularity by increasing abstraction in the class hierarchy.

[1] Combinatorial Optimisation Design-Improvement

Collapse Hierarchy removes a non-leaf class from an inheritance hierarchy. This refactoring is intended to reduce design complexity by removing superfluous classes from the design.

We have deliberately chosen refactorings that operate at the method/field level of granularity and higher because our focus is on the automatic improvement of the design encapsulated in a program rather than implementation issues such as correct factorisation of methods.

One of the functions of CODe-Imp's JPM is to determine where refactorings can legally be applied – in other words, where the corresponding code alterations can be made without altering program behaviour. In order to achieve this we have employed a system of conservative precondition checking, the details of which are beyond the scope of this paper.

3.3. Search Techniques

For this experiment two local and one meta-heuristic search algorithms were selected. The local search algorithms were selected as they are easily implemented and have low resource requirements. Significant quality increases achieved using such techniques would be evidence in favour of scalability of the approach. A meta-heuristic algorithm was also selected since it was expected that local search techniques would not be sufficient to provide significant quality increases. The search techniques selected were the following:

First-Ascent Hill Climbing (HC1) A local search algorithm where the search moves to the first neighbouring solution of a higher quality discovered.

Steepest-Ascent Hill Climbing (HC2) A local search algorithm where the search moves to the neighbouring solution of highest quality.

Low-Temperature Simulated Annealing (SA) A meta-heuristic search technique described below.

A Simulated Annealing search essentially involves making series of tentative changes to some solution of a combinatorial optimisation problem. Changes which increase the quality of the solution are accepted, and the changed solution becomes the starting point for the next series of tentative changes. In addition, some changes which reduce the quality of the solution are accepted in order to allow the search to escape from local minima. Such (negative) changes are accepted with a probability that decreases steadily during the annealing process (equation 1; where p is the probability of accepting a given solution, δq is the magnitude of quality reduction relative to the current solution, and T is the temperature value).

$$p = e^{-\frac{\delta q}{T}} \quad (1)$$

In common with other search techniques simulated annealing requires an evaluation function and a problem representation with a means of altering solutions. In addition, a *cooling schedule* is required that determines how quickly the annealing runs, and hence how likely the solution is to be of high quality. CODe-Imp currently employs a geometric cooling schedule, meaning the temperature is reduced by a constant factor after each step in the annealing process.

The parameters of a geometric cooling schedule are: T_{start}, the starting value for the temperature variable; Markov chain length (M), the number of tentative changes that will be made at each temperature; and f, the geometric cooling factor. Theoretically, M should tend towards infinity and f towards one in order to produce the best possible solution. In practice the cooling schedule should be as slow as possible within the time available; values of M=1 and f=0.995 were used in this experiment. It should also be noted that a *low temperature* simulated annealing was employed, meaning that the value of $T_{start}$was adjusted to give large quality drops a lower than normal chance of being accepted. Initial acceptance probabilities of approximately 0.2 were observed for large quality drops, whereas a standard annealing schedule would result in initial probabilities of approximately 0.8.

3.4. Evaluation Functions

The evaluation functions employed in the CODe-Imp prototype described here are the Flexibility, Reusability and Understandability functions defined as part of the QMOOD hierarchical design quality model [2]. Each evaluation function in the model is based on a weighted sum of quotients on the eleven metrics described below. QMOOD evaluation functions determine the relative quality attributes of two designs, presumed to be similar in purpose. For this reason, each metric value for design A is divided by the corresponding value for design B to give the metric change quotient. Metric weights for each evaluation function are show in tables 1, 2, and 3.

Design Size in Classes (DSC) A count of the total number of classes in the design. Interpreted as excluding imported library classes.

Number Of Hierarchies (NOH) A count of the number of class hierarchies in the design. Interpreted as excluding hierarchies that consist of a generalised class within the design and a specialised class outside.

Average Number of Ancestors (ANA) The average number of classes from which each class inherits information.

Number of Polymorphic Methods (NOP) A count of the number of the methods that can exhibit polymorhic behaviour. Interpreted as the average across all classes, where a method can exhibit polymorphic behaviour if it is overridden by one or more descendent classes.

Class Interface Size (CIS) A count of the number of public methods in a class. Interpreted as the average across all classes in a design.

Number Of Methods (NOM) A count of all the methods defined in a class. Interpreted as the average across all classes in a design.

Data Access Metric (DAM) The ratio of the number of private (protected) attributes to the total number of attributes declared in the class. Interpreted as the average across all design classes *with at least one attribute*, of the ratio of non-public to total attributes in a class.

Direct Class Coupling (DCC) 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. Interpreted as an average over all classes when applied to a design as a whole; a count of the number of distinct user-defined classes a class is coupled to by method parameter or attribute type. We exclude standard Java library classes from the computation.

Cohesion Among Methods of Class (CAM) 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. We have excluded constructors and implicit 'this' parameters from the computation.

Measure Of Aggregation (MOA) A count of the number of data declarations whose types are user-defined classes. Interpreted as the average value across all design classes. We define 'user defined classes' as non-primitive types that are not included in the Java standard libraries.

Measure of Functional Abstraction (MFA) The ratio of the number of methods inherited by a class to the number of methods accessible by member methods of the class. Interpreted as the average across all classes in a design of the ratio of the number of methods inherited by a class to the total number of methods available to that class, i.e. inherited and defined methods.

3.5. Input

Input consisted in both cases of one package from the `spec.benchmarks`[2] standard performance evaluation framework. The packages selected were those to which the greatest number of refactorings could be applied. Input A (*spec.benchmarks._200_check*) consisted of 16 classes to which 14 distinct refactorings could initially be applied, while input B (*spec.benchmarks._205_raytrace*) consisted of 25 classes to which 53 distinct refactorings could initially be applied.

4. Case Studies

4.1. Overview

The results described in this section are mean values of ten replications of each run, the only variation being in random decisions required by the search algorithms. Figures show standard deviation 'error' bars; where these are absent no deviation from the mean value was observed. Statistical significance was established in all cases by performing student's t-test for unpaired data assuming unequal variance, with a confidence interval of 95%.

Experiments were carried out on a 2.2GHz AMD Athlon powered PC with 1GB RAM. Mean processing time per solution examined was approximately five seconds, including model building, metric extraction, quality assessment, discovery of legal refactorings, and actual (AST) refactoring.

In the remainder of this section we discuss two aspects of the results of the experiment; firstly, in section 4.2 we present the overall quality changes observed as measured by the three evaluation functions for each of the three search techniques. These results indicate the level of success achieved in refactoring the input programs to improve design as measured by the evaluation functions. Secondly, in section 4.3 we present the observed changes in metric value for each of the three evaluation functions, in the case of the most consistent search technique. These results demonstrate the differing effects of the various evaluation functions, and along with an inspection of the output code, allow us to discuss the effectiveness of the evaluation functions in actually increasing design quality.

4.2. Overall Quality Changes

4.2.1 Input A

Figure 1 shows the mean overall quality changes observed for each search technique and evaluation function for input A. A significant increase in evaluation function value was observed with all three search techniques for the evaluation

[2]http://www.spec.org/

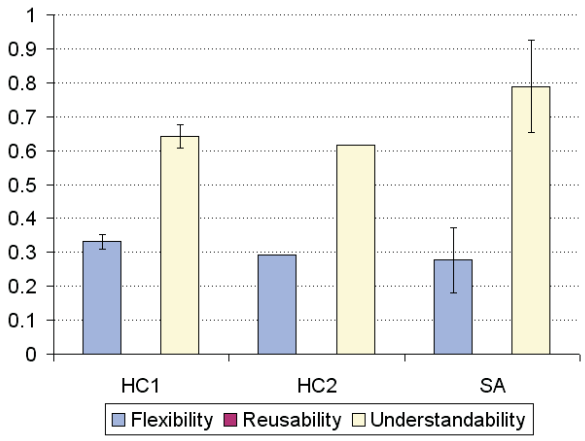

Figure 1. Mean quality change – Input A

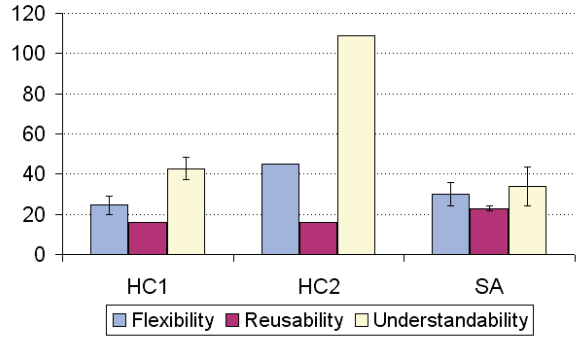

Figure 2. Mean solutions examined – Input A

functions Flexibility and Understandability. No increase in Reusability evaluation function value was observed with any search technique for this input - either no improvement was possible using the refactorings employed or the search failed to escape from a local maximum in each case.

The three search techniques yielded similar results in terms of mean solution quality increase, with HC1 performing a small but statistically significant amount better than HC2 on both Flexibility and Understandability, while SA performed better than either on Understandability. SA search showed greatest standard deviation in solution quality for both Flexibility and Understandability, while HC2 showed no deviation at all.

Figure 2 shows the mean number of solutions examined in each case graphed in figure 1. The mean number of solutions examined varied across evaluation functions and

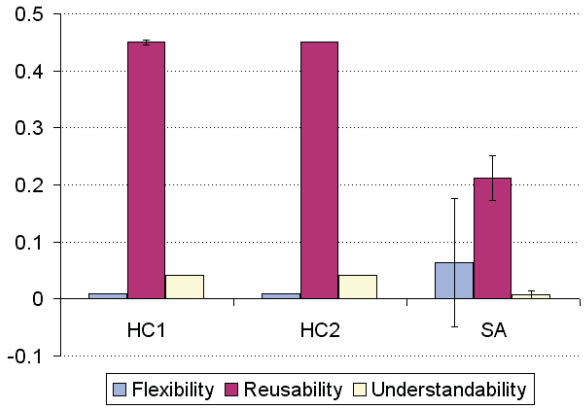

Figure 3. Mean quality change – Input B

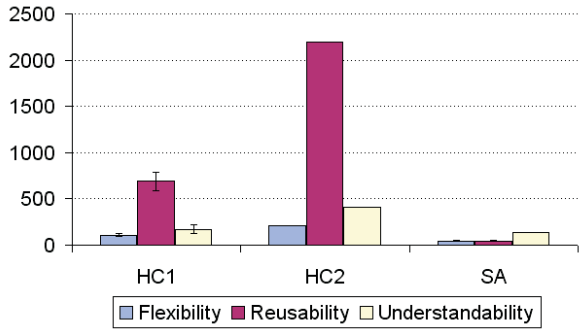

Figure 4. Mean solutions examined – Input B

search technique, the most notable feature for this input being that HC2 examined significantly more solutions than the other two search techniques for the two evaluation functions where quality increases were observed.

In summation, there is little to choose between search techniques in terms of efficiency or effectiveness for this input. The only clear conclusion that can be drawn is that HC2 examined more solutions to arrive at comparable results to the other two algorithms. However, the most significant observation is that quality increases were obtained for two of the three evaluation functions.

4.2.2 Input B

Figure 3 shows the mean overall quality changes observed for each search technique and evaluation function for input B. A significant increase in evaluation function value was observed for all three evaluation functions using both hill-climbing search algorithms, and for Reusability and Understandability using SA. Observed increases in the case of the reusability function were particularly large. The two hill-climbing search techniques performed similarly in terms of mean quality increase, and produced larger quality increases for Reusability and Understandability

Figure 4 shows the mean number of solutions examined in each of the cases graphed in figure 3. The mean number of solutions examined varied across evaluation functions and search technique, the most notable feature for this input being that HC2 examined considerably more solutions than either of the other search techniques. Simulated Annealing examined the fewest solutions in each case.

In summation, HC1 produced joint greatest evaluation function increases with HC2, but examined far fewer solutions. We therefore consider HC1 to have performed best for this input. SA produced solutions of a reasonable quality given the small number of solutions examined. We conclude that the parameters of the SA search cooling schedule were not suitable for this particular input. Again, the most significant observation is that quality increases were obtained for all three evaluation functions.

4.3. Comparison of Metric Changes by Evaluation Function

4.3.1 Flexibility

The Flexibility quality attribute in QMOOD is defined as "The ability of a design to be adapted to provide functionally related capabilities"[2]. Metric quotient changes resulting from use of the Flexibility function in CODe-Imp are shown in figure 5. It should be noted that these are metric *quotient* changes; differences from the identity value of 1 are graphed. A graphed value of 1 equates to a doubling of the metric value from input to output. The actual metric weights comprising the Flexibility function are shown in table 1.

In the case of input A, use of the QMOOD Flexibility function resulted in a large increase in the DAM metric, with no change to any other metric. The metric's corresponding weight in the function is positive (0.25), so it is clear how the evaluation function increase has been obtained. In real terms, the DAM metric value increased by approximately 0.09, so the output solution consisted of classes with an average ratio of non-public to public attributes increased by 9%. Inspection of the solution code revealed that this change was effected by the removal of unnecessary public fields from three classes by the Push Down Field refactoring. The refactored design was therefore less complex, and hence slightly more flexible.

In the case of input B, use of the Flexibility function resulted in a small decrease in the positively-weighted (0.25) DAM metric, but a greater decrease in the nega-

tively weighted (-0.25) DCC metric. An increase in the unweighted CAM metric was also observed. In real terms, the DCC metric value decreased by approximately 0.05, so the output solution consisted of classes coupled to, on average, 5% fewer other classes by attribute declarations and message passing. Inspection of the solution code revealed that these changes were brought about primarily by the removal of unnecessary public fields and methods from solution classes by the Push Down Field and Push Down Method refactorings. The small decrease in coupling combined with the increase in cohesion indicated a more modular design that appears to be slightly more flexible.

4.3.2 Reusability

Metric quotient changes resulting from use of the Reusability function in CODe-Imp are shown in figure 6. Note that these are metric quotient changes; differences from the identity value of 1 are graphed. The actual metric weights comprising the Reusability function are shown in table 2.

In the case of input A, use of the QMOOD Reusability function resulted in increases in the positively weighted metrics DSC (0.5) and CAM (0.25), and also in the unweighted metric ANA. Decreases were observed in the negatively weighted metric DCC (-0.25), the unweighted metrics DAM, MOA, NOP and NOM, and the positively weighted (0.5) metric CIS. The most prominent changes are the increase in DSC, which reached the imposed solution size limit of 200% original size, and the 605% increase in ANA (truncated on graph) which corresponded to a jump in the average number of ancestors per class from less than 0.45 to approximately 2.8.

Predictably, inspection of the output code revealed that the large increases in design size in classes and average number of ancestors were due to a large number of additional non-leaf classes in the four inheritance hierarchies, some entirely devoid of fields or methods. This also explains the decrease in values for unweighted metrics, as any metric that is an average over the number of classes can decrease as the Extract Hierarchy refactoring adds classes. The dominance of the DSC metric in this evaluation function indicates a flaw in the QMOOD Reusabilty function, since the reusability of a design cannot be said to increase as featureless classes are added.

No change in metric values was observed for input B with the Reusability evaluation function.

4.3.3 Understandability

Metric quotient changes resulting from use of the Understandability function in CODe-Imp are shown in figure 7. Again, these are metric quotient changes; differences from the identity value of 1 are graphed. The actual metric weights comprising the Understandability function are shown in table 3.

In the case of input A the Understandability function produced increases in the positively weighted metrics DAM (0.33) and CAM (0.33) as well as the unweighted metric MFA, and decreases in the negatively weighted metrics NOP (-0.33) and NOM (-0.33) as well as the unweighted CIS. The most prominent metric changes were the 0.09 rise in DAM, corresponding to an increase in the average ratio of non-public to public fields in each class of 9%, the 0.04 rise in MFA, corresponding to a 4% increase in the average ratio of the number of methods inherited by a class to the total number of methods accessible by the class, and the 0.36 drop in both CIS and NOM, corresponding to 36% decreases in the average number of public methods and average number of methods per class, respectively. Inspection of the output code revealed that the decreases in CIS and NOM were due mainly to the removal of methods from classes where they were not required by the Push Down Method refactoring, while the increase in DAM was mainly due to the removal of unrequired public fields by the Push Down Field refactoring. These results show that understandability has increased, since smaller classes and their interaction via correspondingly smaller public interfaces are easier to comprehend.

In the case of input B the Understandability function produced an increase in the positively weighted metric CAM (0.33), and decreases in the negatively weighted metrics DCC (-0.33) and NOM (-0.33). A small decrease was also observed in the positively weighted metric DAM (0.33). The increase in cohesion and decrease in coupling indicate a slightly more modular and hence more understandable design; this slight improvement was confirmed by inspection of the output code.

5. Conclusion

The results reported above show that some Java 1.4 programs can be automatically refactored to improve quality as measured by QMOOD evaluation functions. As such, they partially validate the search-based software maintenance approach. We have shown that evaluation function increases can be obtained in most cases examined using simple search techniques, and that variation in weights on evaluation function components has a significant effect on the overall refactoring process.

To address the points of inquiry mentioned in section 1;

- The small set of refactorings employed by the CODe-Imp prototype gave rise to observed changes in ten out of eleven QMOOD metrics, the exception being NOH (Number Of Hierarchies). While this indicates a solid foundation for the exploration of proposed quality functions, it is clear that the addition of further

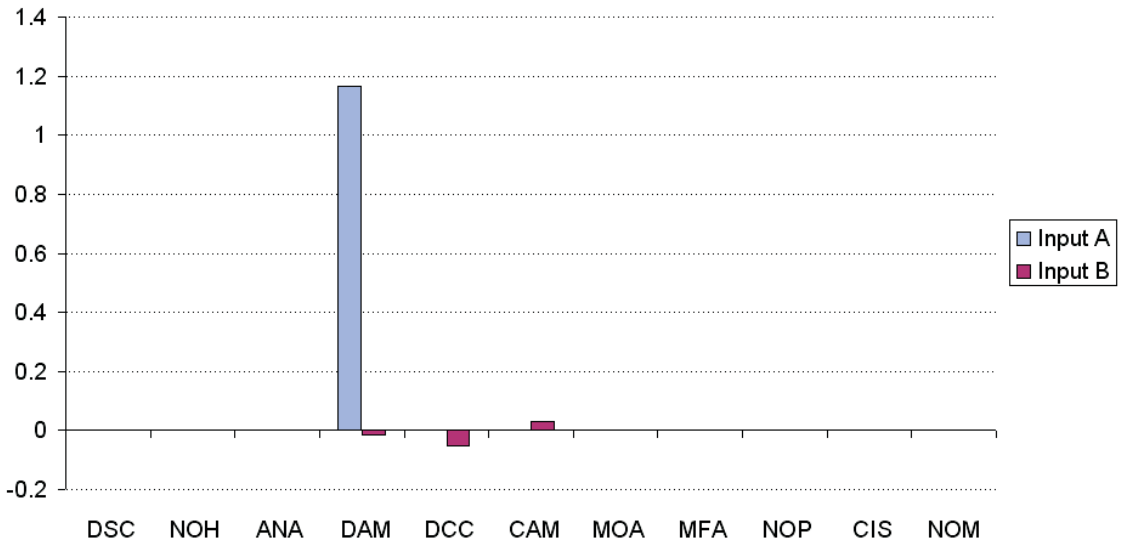

Figure 5. Metric quotient changes, Flexibility function

DSC	NOH	ANA	DAM	DCC	CAM	MOA	MFA	NOP	CIS	NOM
0	0	0	0.25	-0.25	0	0.5	0	0.5	0	0

Table 1. Metric weights of QMOOD Flexibility function

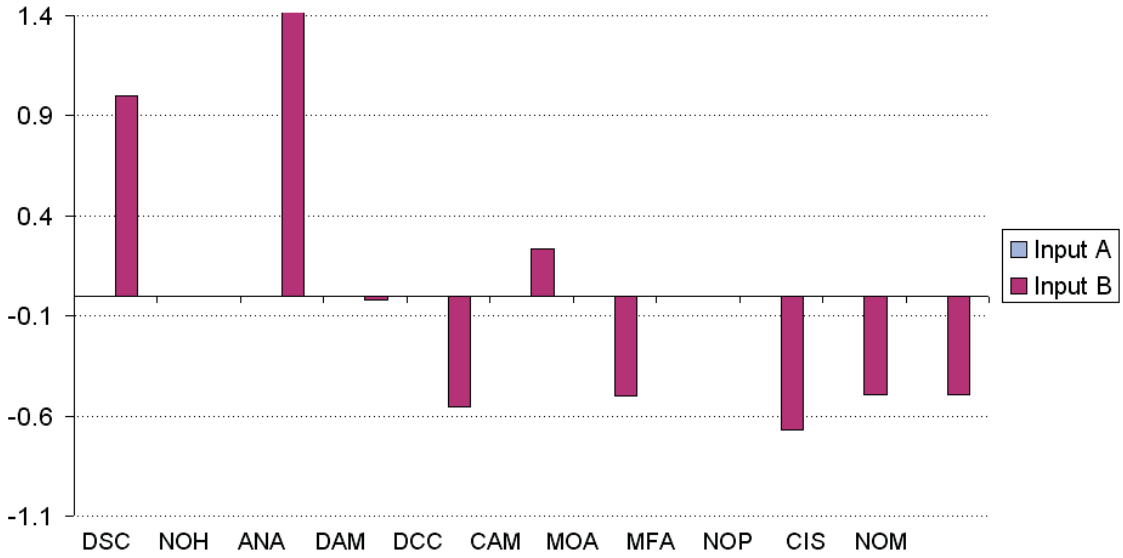

Figure 6. Metric quotient changes, Reusability function

DSC	NOH	ANA	DAM	DCC	CAM	MOA	MFA	NOP	CIS	NOM
0.5	0	0	0	-0.25	0.25	0	0	0	0.5	0

Table 2. Metric weights of QMOOD Reusability function

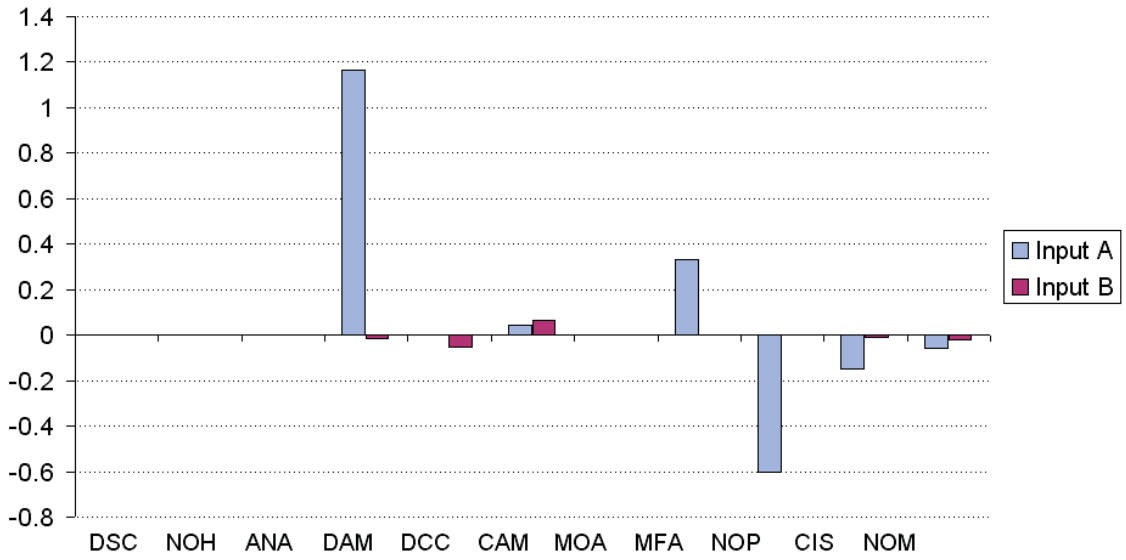

Figure 7. Metric quotient changes, Understandability function

DSC	NOH	ANA	DAM	DCC	CAM	MOA	MFA	NOP	CIS	NOM
-0.33	0	-0.33	0.33	-0.33	0.33	0	0	-0.33	0	-0.33

Table 3. Metric weights of QMOOD Understandability function

refactorings would add to the depth of such exploration.

- The search techniques employed all demonstrated strengths in this experiment; first-ascent hill climbing generally produced quality improvements for the least computational expenditure, steepest-ascent hill climbing produced the most consistent improvements, and Simulated Annealing produced the greatest quality improvements in certain cases. The most significant observation here was that quality improvements were obtained using simple search techniques with manageable run-times, which bodes well for the scalability of the approach.

- Inspection of output code and analysis of solution metrics provided some evidence in favour of use of the QMOOD Flexibility function, and strong evidence in favour of use of the Understandability function. The QMOOD Reusability function was not found to be suitable to the requirements of search-based software maintenance because it resulted in solutions including a large number of featureless classes.

Future work will include adding to the refactoring capabilities of CODe-Imp , experimentation with a wider variety of input programs and further exploration of potential quality evaluation functions in the search-based software maintenance approach.

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] F. Brito e Abreu and W. L. Melo. Evaluating the impact of object-oriented design on software quality. In *IEEE METRICS*, pages 90–99, 1996.

[4] F. Brito e Abreu, L. Ochoa, and M. Goulão. The GOODLY design language for MOOD2 metrics collection. In *ECOOP Workshops*, pages 328–329, 1999.

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

[6] E. Casais. An incremental class reorganization approach. In O. L. Madsen, editor, *Proceedings of the European Conference on Object-Oriented Programming*, pages 114–131, Utrecht, June 1992. LNCS.

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

[8] 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.

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

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

[11] 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)*.

[12] E. V. Emden and L. Moonen. Java quality assurance by detecting code smells. In *WCRE*, pages 97–, 2002.

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

[14] 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.

[15] 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.

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

[18] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA*, pages 235–250, 1996.

[19] L. Tahvildari and K. Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *Journal of Software Maintenance*, 16(4-5):331–361, 2004.

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

[21] Wei Li. Another metric suite for object-oriented programming. *J. Syst. Softw.*, 44(2):155–162, 1998.