Search-Based Refactoring for Software Maintenance

Mark O'Keeffe, Mel Ó Cinnéide

School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland

Abstract

The high cost of software maintenance could be reduced by automatically improving the design of object-oriented programs without altering their behaviour. We have constructed a software tool capable of refactoring object-oriented programs to conform more closely to a given design quality model, by formulating the task as a search problem in the space of alternative designs. This novel approach is validated by two case studies, where programs are automatically refactored to increase flexibility, reusability and understandability as defined by a contemporary quality model. Both local and simulated annealing searches were found to be effective in this task.

Key words: search based software engineering, automated design improvement, refactoring

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 goal of the object-oriented approach is to produce understandable, modular designs in order to minimise the cognitive complexity of programming tasks. 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 program entities as input and output a set with the same external behaviour, but having a design that conforms more closely to a given quality model. Maintenance programming or manual refactoring could then begin from a more advantageous point, thus reducing the costs involved.

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 extent 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 be capable of ranking 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 flexibility, reusability and understandability, based on eleven object-oriented design metrics. We have examined these evaluation functions through experimentation described in this article and determined that their level of suitability for this approach varies considerably. 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 a valuable mechanism for validation of that model.

While our primary goal in this work was to demonstrate that object-oriented programs can be automatically refactored to conform more closely to a given quality model using a search-based approach, we also report here on several other significant contributions:

- An assessment of the suitability and effectiveness of several contemporary evaluation functions for the purpose of search-based software maintenance.

- A comparison of the performance of a set of search techniques in the context of automated refactoring of Java programs guided by contemporary evaluation functions.

- A subjective assessment of the performance of a prototype automated refactoring tool from a software engineer's perspective.

This article expands on the CSMR 2006 paper 'Search-Based Software Maintenance' [33] in which we report similar success in automatically refactoring Java packages to increase maintainability. While the same metric suite is employed here, we have substantially expanded the refactoring capability of the prototype design improvement tool, with the addition of six refactorings described in section 4.1. We also report here on larger case studies involving complete open-source programs, rather than individual packages, and have employed a wider variety of search techniques. Our research on this novel concept was first published in 2003 [32].

The remainder of this article is structured as follows: in section 2 we survey related work in the fields of search-based software engineering, automated and semi-automated design improvement, and design quality measurement. In section 3 we discuss the limitations of our approach. In section 4 we describe our experimental methodology, with particular emphasis on the prototype design-improvement tool CODe-Imp. In section 5 we present the results of search-based refactoring of two Java programs, with regard to overall quality function gain and relative performance of search techniques. In section 6 we present our observations on search-based maintenance of the same two Java programs, discussed in terms of individual metrics and direct code examination, and compare the differing effects of three evaluation functions. We describe directions for future work in section 7 and conclude in section 8.

2 Related Work

Work related to this project can be divided broadly into three areas: Search-Based Software Engineering, Automated Design Improvement and Design Quality Measurement. These topics are discussed below.

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 [19]. Such problems include *module clustering*, where a software system is reorganised into loosely coupled clusters of highly cohesive modules to aid reengineering [16,20,24,29], test data generation [26], automated testing [37] and project management problems such as requirements scheduling [1] and project cost estimation [9,14,15]. An overview of such work and comprehensive recent references can be found in [13] and [19] respectively. Of particular relevance to this work is ‘Metrics Are Fitness Functions Too’ [19], in which Harman states that any product or process metric can be used as the evaluation function driving a search-based optimisation.

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 (GAs) and ant colony optimisation. Module clustering, for example, has been addressed using exhaustive search [25], hill-climbing [20,23,25,28], genetic algorithms [16,20,25,28] and simulated annealing (SA) [28]. In those studies that compared search techniques, hill-climbing was, perhaps surprisingly, found to produce better results than meta-heuristic GA searches [20,27]. These results were echoed in search-based auto-parallelisation [39], where local searches similarly out-performed GA. In software clustering the meta-heuristic *simulated annealing* search was found by Mitchell et al. [28] to perform similarly to hill-climbing in terms of solution quality, but better in terms of search efficiency.

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. Examples of such work include that of Casais [11], who proposed algorithms to restructure class hierarchies in order to maximise abstraction, and Moore [30], who proposed a system where existing classes are discarded and replaced with a new set with optimal method factorisation – meaning code duplication is minimised. However, since object-oriented design involves numerous trade-offs, this narrow focus can result in overall quality loss.

Our approach has two main advantages over previous fully automated restructuring work. Firstly, and most significantly, the use of evaluation functions consisting of combinations of multiple 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 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 Moore's work [30].

While the term *refactoring* was popularised by Opdyke [34] as a verb meaning 'to improve the design of a program without altering its behaviour', the word has subsequently come to be used as a noun meaning 'a code change that can be made in order to improve design while preserving behaviour'. An example of a refactoring is Pull Up Method, meaning the repositioning of a method at a higher level in an inheritance hierarchy. Catalogues of refactorings such as Fowler's 'Refactoring: Improving the Design of Existing Code' [18] are available that have provided a useful standard for reference and communication.

The application of many of the refactorings prescribed by Fowler and others can be automated to some extent, given user interaction. Robert's Refactoring Browser [35] was one of the first software tools to provide automated assistance for the application of refactorings; today most IDEs provide some form of automated refactoring support. While such tools reduce the effort involved in refactoring, they do not assist the programmer in the vital task of determining where it is advantageous to apply refactorings. Some semi-automated approaches to design improvement, however, attempt do just that.

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 determine precisely what changes should be made. Such 'bad smell' detection has been proposed by van Emden [17], and by Tahvildari [36], whose system also recommends 'meta-pattern transformations' that can be applied to ameliorate the defect. The proviso 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 cost. We have conducted a survey of 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. The principle criteria for assessment were: firstly, that the model comes as close as possible to providing complete evaluation functions, and secondly, that the constituent metrics are well-defined and well-established.

2.3.1 CK

The Metrics Suite for Object-Oriented Design (known as CK) of Chidamber and Kemerer [12] is a seminal work in object-oriented quality measurement and is still frequently cited today. Metrics are defined for properties such as complexity, inheritance, coupling, cohesion and messaging. The CK metrics and subsequent modifications by Li et al. [38] have been independently validated as indicators of such characteristics as fault-proneness [3], but no attempt has been made to combine them in the form of an evaluation function. Several interpretations exist of some CK metrics, such as Lack of Cohesion of Methods (LCOM).

2.3.2 MOOD2

The MOOD (Metrics for Object-Oriented Design) metrics suite [7] was introduced by Fernando Brito e Abreu et al. in 1994 and was subsequently evaluated by the author [6] and others [21]. Because some deficiencies were identified, namely the lack of measures of reuse, polymorphism and external coupling, the MOOD suite was superseded by the MOOD2 metrics suite in 1998 [4]. The MOOD2 metrics are also defined in an English–language paper [5] through extended OCL and the GOODLY design language [8].

The MOOD2 suite is a comprehensive, modern metrics suite including several measures each of coupling, reuse, polymorphism, data-hiding and inheritance. MOOD2 metrics are formally defined, and hence can be directly implemented without resolution of ambiguity. However, nowhere in the literature are evaluation functions defined that combine MOOD2 metric values to give an overall quality index. As a result MOOD2 does not provide a complete quality model suitable for use in search-based refactoring.

2.3.3 QMOOD

The QMOOD (Quality Model for Object-Oriented Design) model of Bansiya [2] was introduced in 2002 and consists of a hierarchy of four levels. The levels in descending order are: Design Quality Attributes such as ‘understandability’, Object-Oriented Design Properties such as ‘encapsulation’, Object-Oriented Design Metrics, and Object-Oriented Design Components such as ‘class’.

For the purpose of search-based refactoring, the QMOOD model has the advantage that it defines functions from metric values to Quality Attribute Indices (QAIs) for such design attributes as flexibility, reusability and understandability. This provides an excellent foundation for experimentation in automatically refactoring a design to conform to this quality model. However, while QMOOD provides a detailed model of object-oriented design quality, it is lacking in the area of effective metric definition. Metrics in QMOOD literature [2] are defined in natural language, and are in some cases ambiguous. In order to implement the QMOOD metrics for replicable studies it is necessary to define them more precisely.

The QMOOD quality model was selected for this work as it includes predefined evaluation functions. Were another metric suite selected it would have been necessary to define evaluation functions on those metrics, thus reducing the independence of our work from the field of software product measurement. In order to ameliorate the problem of ambiguous metric definitions in QMOOD, we have precisely defined the QMOOD metrics as implemented in CODe-Imp later in this article. It should be noted that other metrics for individual design properties such as cohesion could be substituted for the corresponding QMOOD metrics without alteration to the approach as a whole.

3 Limitations of this approach

In common with all fully automated refactoring tools CODe-Imp has the drawback that changes in the design must be communicated to the programmer. Several issues such as the relocation of program comments and need to introduce new identifiers such as class names can complicate this task [10]. These issues could be addressed by a programmer review of the refactored code, but our assumption that this would be less expensive than completely manual refactoring is unproven.

As automatic refactoring with CODe-Imp improves program design with respect to a well-defined quality model rather than in an absolute sense, the effectiveness of the approach hinges on how accurately that quality model reflects the refactoring goals of the user. Quality models of sufficient detail to be employed in our case studies are extremely rare in the literature, largely because the definition of quality varies not only between software domains but also between refactoring tasks of differing motivation. Early in the software life-cycle, for example, the main motivation for refactoring may be to preserve flexibility so that further functionality can be easily added, while later in the cycle reusability may be paramount. For this reason, we ultimately see the automated refactoring approach described here being employed by software companies that have developed their own domain-specific quality models.

4 Experimental Methodology

In order to test the thesis that object-oriented programs can be automatically refactored so that their design conforms more closely to a given quality model we have constructed a prototype search-based design improvement tool called CODe-Imp [1] . CODe-Imp can be configured to operate using various subsets of its available automated refactorings, various search techniques, and various evaluation functions based on combinations of established metrics. In the remainder of this section we describe the configuration of CODe-Imp that yielded the results reported in this article.

4.1 Refactorings

The refactoring configuration of CODe-Imp was constant throughout the case studies reported here, and consisted of the fourteen refactorings described below. Complementary pairs of refactorings were selected 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 have access to the field.

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

Increase Field Security increases the security of a field from public to protected or from protected to private. This refactoring increases data encapsulation.

Decrease Field Security decreases the security of a field from private to protected or from protected to public. This refactoring reduces data encapsulation.

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. This refactoring is used to rectify a situation where a subclass does not use enough of a superclass's features to justify the specialisation relationship [18].

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. This refactoring can be used in a situation where a delegating class is using enough features of a delegate class that a specialisation relationship would be more appropriate [18].

Increase Method Security increases the security of a method from protected to private or from public to protected. This refactoring can reduce the size of the public interface of a class.

Decrease Method Security decreases the security of a method from protected to public or from private to protected. This refactoring can increase the size of the public interface of a class.

Make Superclass Abstract declares a constructorless class explicitly abstract. This increases some measures of abstraction, and facilitates other refactorings.

Make Superclass Concrete removes the explicit 'abstract' declaration of an abstract class without abstract methods. This decreases some measures of abstraction.

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.

During the search process alternative designs are repeatedly generated by the application of a refactoring to the existing design, evaluated for quality, and either accepted as the new current design or rejected. As the current design changes, the number of points at which each refactoring can be applied will also change. One the functions of CODe-Imp's Java Program Model (JPM), an abstraction of the AST automatically enriched with program facts at run-time, 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 similar to that developed by Roberts for the Smalltalk 'Refactoring Browser' [35] and subsequently extended by Ó Cinnéide [31], but using static rather than dynamic analysis. Further details are omitted here due to space constraints.

4.2 Search Techniques

In order to provide an insight into which search techniques are most effective in a search-based software maintenance context we have replicated the case studies reported here across four; three local and one meta-heuristic. The search techniques selected were the following:

First-Ascent Hill Climbing (HC1) A local search algorithm where the search examines neighbouring solutions until a higher quality solution is discovered. This neighbour then becomes the current solution. Local search algorithms were selected as they have been shown to produce good results in other SBSE applications, as discussed in section 2.1.

Steepest-Ascent Hill Climbing (HC2) A second local search algorithm, where the search examines all neighbouring solutions and moves to the solution of highest quality.

Multiple-Restart Hill Climbing (HCM) A variation of first-ascent hill climbing where a number of 'restarts' are made when the search reaches an apparent optimum. In the experiments described here three restarts of a depth of five random refactorings were made in each case.

Low-Temperature Simulated Annealing (SA) A meta-heuristic search technique described below. Simulated annealing was selected as it has previously been found to perform well in the context of software clustering [28].

A Simulated Annealing [22] 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).

$$p = e^{-\frac{\delta q}{T}} \tag{1}$$

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 employs the standard 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, a simulated annealing search yields optimum results when M tends towards infinity and f towards one; in practice the cooling schedule should be as slow as possible within the time available.

A number of different cooling schedules were tested in order to establish a useful candidate for the experiments described later in this article. Mean quality increase over three runs for Markov chain lengths 1 and 2 and various cooling factors in the range 0.990 – 0.999 was recorded for input B under the QMOOD Understandability evaluation function, which we describe in section 4.3. A cooling factor of f=0.9975 was found to be most effective while Markov chain lengths of 1 and 2 were equally effective for that value. A cooling schedule of f=0.9975 and M=1 was therefore used in the experiments described in the remainder of this article.

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. This is because standard simulated annealing is usually employed where the starting solution is of extremely low quality, such as a timetable with a large number of clashes in the context of a timetabling problem. In contrast, object-oriented designs which are of low quality from a programmer's perspective are nonetheless of relatively high quality, when we consider the potentially infinite size of the search space of all functionally equivalent designs. In the experiments reported in this article that used simulated annealing 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.

4.3 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 in table 1. QMOOD evaluation functions determine the relative quality attributes of two designs, presumed to be similar in purpose. For this reason, each metric value for the refactored design D' is divided by the corresponding value for the original design D to give the metric change quotient. Metric weights for each evaluation function are shown in table 2; a positive weight corresponds to a metric that should be increased in order to enhance the design property in question, while a negative weight corresponds to a metric that should be decreased.

Of course, the concept of design quality is quite ephemeral and even sub-concepts such as understandability cannot easily be defined. We consider the QMOOD evaluation functions examples of how some desirable design property can be precisely expressed, rather than definitive metrics for the design properties they are named for. We therefore attempt to optimise these values not in order to guarantee an improvement in terms of the subjective concepts of flexibility, reusability and understandability, but rather to demonstrate that in the general case a well-defined design property can be optimised using our approach. However, in section 6 we do subjectively assess whether the QMOOD evaluation functions lead to improvements in the corresponding design properties, in the context of our approach.

4.4 Input & Hardware

Input consisted of one program from the Spec Benchmarks[2] standard performance evaluation framework and one program taken from SourceForge[3] via java-source.net. These programs were selected because a large number of refactorings could be applied to them. Input A (*Spec-Check*) consisted of 41

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

Metric	Acronym	Description	Design Property
Design Size in Classes	DSC	A count of the total number of classes in the design. Interpreted as excluding imported library classes.	Design Size
Number Of Hierarchies	NOH	A count of the number of class hierarchies in the design. Interpreted as excluding hierarchies that consist of a specialised class within the design and a generalised class outside.	Hierarchies
Average Number of Ancestors	ANA	The average number of classes from which each class inherits information.	Abstraction
Number of Polymorphic Methods	NOP	A count of the number of the methods that can exhibit polymorphic 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.	Polymorphism
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.	Messaging
Number Of Methods	NOM	A count of all the methods defined in a class. Interpreted as the average across all classes in a design.	Complexity
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.	Encapsulation
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.	Coupling
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.	Cohesion
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.	Composition
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.	Inheritance

Table 1
QMOOD metrics [2].

function	DSC	NOH	ANA	DAM	DCC	CAM	MOA	MFA	NOP	CIS	NOM
Flexibility	0	0	0	0.25	-0.25	0	0.5	0	0.5	0	0
Reusability	0.5	0	0	0	-0.25	0.25	0	0	0	0.5	0
Understandability	-0.33	0	-0.33	0.33	-0.33	0.33	0	0	-0.33	0	-0.33

Table 2
Metric weights of QMOOD evaluation functions [2].

classes to which 351 distinct refactorings could initially be applied, while input B (*Beaver*)consisted of 30 classes to which 190 distinct refactorings could initially be applied.

Experiments were carried out on a 2.2GHz AMD Athlon powered PC with 1GB RAM. Mean processing time per solution examined was approximately one second, including model building, metric extraction, quality assessment, discovery of legal refactorings, and actual (Abstract Syntax Tree) refactoring. Total run times varied between approximately one hour and almost two hours depending on input size and search algorithm for local searches, with simulated annealing notably requiring a disproportionately long run time of up to ten hours due to processing overheads required by the algorithm. However, CODe-Imp was designed with robustness rather than speed as a priority and makes no use of concurrent processes, so there is potential to greatly decrease these run-times.

In the following two sections we present case studies of two facets of the search-based refactoring of Java programs. In section 5 we present the results of refactoring of two programs with regard to quality gain and relative performance of search techniques, while in section 6 we present our observations on search-based maintenance of the same two Java programs, discussed in terms of individual metrics and direct code examination, and compare the differing effects of three evaluation functions.

5 Case Studies I; comparison of search techniques

In this section we present the quality changes observed using each of the four search techniques described in section 4.2. Two input programs and the QMOOD Flexibility, Reusability and Understandability evaluation functions described in section 4.3 were examined, giving a total of six distinct cases for comparison. These results indicate the level of success achieved in refactoring the input programs to improve design as measured by the evaluation functions, and allow us to compare the performance of the three search techniques employed.

The results described in this section are mean values of at least three replications of each run, the only variation being in random decisions required by the search algorithms. Figures show standard deviation ‘error’ bars; where these

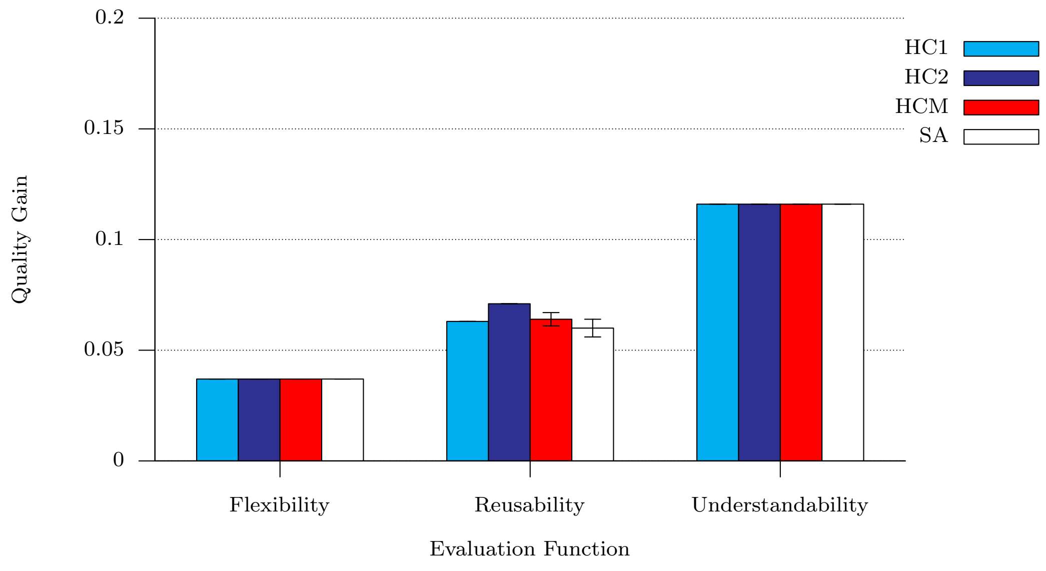

Fig. 1. Mean quality change – Input A

are absent no deviation from the mean was observed. Statistical significance was established in all cases by performing student's t-test for unpaired data assuming unequal variance, in order to establish that perceived differences were not due to chance alone. A confidence interval of 95% was used.

5.1 Input A – Spec-Check

Figure 1 shows the mean overall quality changes observed for each search technique and evaluation function for input A. An increase in evaluation function value was observed for all four search techniques for each of the three evaluation functions Flexibility, Reusability and Understandability. The magnitude of quality function change was greatest in the case of the Understandability function and smallest in the case of the Flexibility function, but as QMOOD evaluation functions are based on a weighted sum of metric quotients this does not necessarily mean that more extensive changes were made to the design in any particular case.

The four search techniques yielded identical results in terms of mean solution quality increase across the Flexibility and Understandability evaluation functions. For the Reusability function HC2 produced significantly greater quality increases than HC1, HCM or SA. No statistically significant difference in quality increase was observed between the other three search techniques.

Figure 2 shows the mean number of solutions examined in each of the cases graphed in figure 1. For all three evaluation functions, HC1 and SA examined

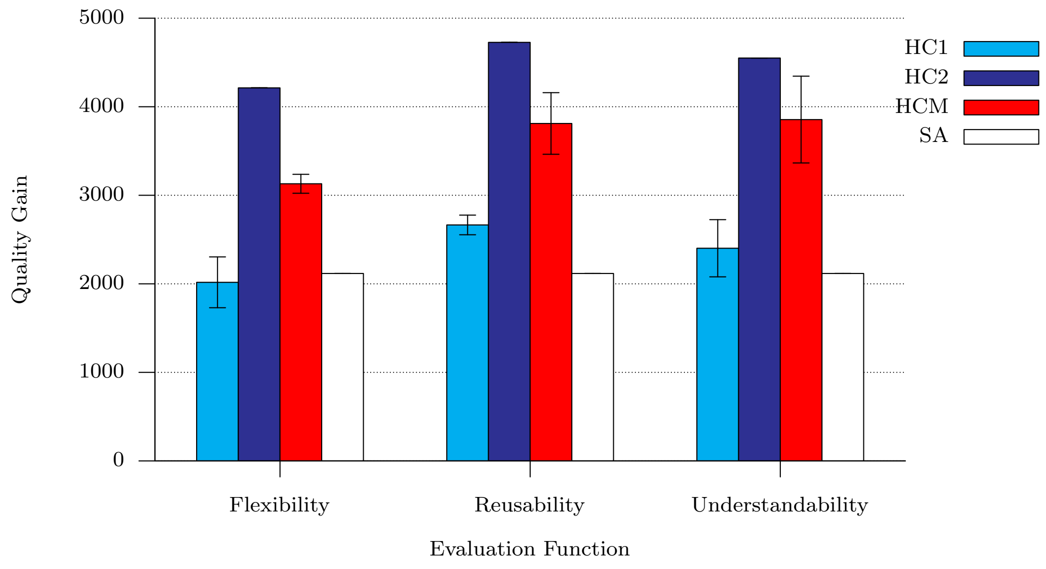

Fig. 2. Mean solutions examined – Input A

the fewest solutions, with a statistically significant difference occurring only for the Reusability function, where SA examined fewest solutions. For all three functions HC2 examined the greatest number of solutions, with HCM examining significantly fewer than HC2 but significantly more than HC1 or SA. So, while the four search techniques performed similarly in terms of mean quality gain for this input, the number of solutions examined in the search process varied considerably, with HC2 examining approximately twice as many as SA in each case.

For this input all four search techniques produced quality improvements, but HC1 and SA were the most efficient in terms of number of solutions examined. Since SA incurred high processing overheads as mentioned in section 4.4, HC1 must be considered the most efficient search technique for this input. As neither HCM or SA discovered solutions of higher quality it is likely that the search space was smooth for these three combinations of input and evaluation function.

5.2 Input B – Beaver

Figure 3 shows the mean overall quality changes observed for each search technique and evaluation function for input B. An increase in evaluation function value was observed for all three search techniques for each of the three evaluation functions Flexibility, Reusability and Understandability.

The three search techniques yielded varying results in terms of mean solu-

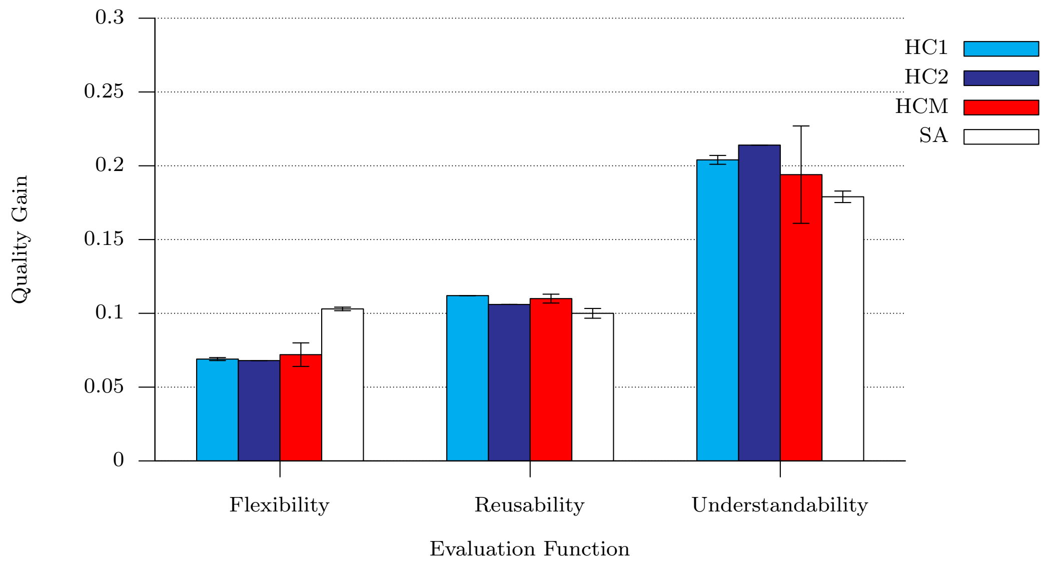

Fig. 3. Mean quality change – Input B

tion quality increase across the three evaluation functions. For the Flexibility function, SA produced the greatest quality increase by a clear margin, while no statistically significant difference was observed between the other three search techniques. For the Reusability function, HC1 and HCM produced the greatest quality increases, with no statistically significant difference between them, while HC2 and SA produced lesser quality increases, with no statistically significant difference between them. For the Understandability function, HC2 produced the greatest mean quality increase. Although HCM produced the highest quality individual solution observed, the large variation in solution quality in this case caused the mean quality increase to be significantly lower than HC2. HC1 and SA also performed significantly worse than HC2 for this evaluation function.

Figure 4 shows the mean number of solutions examined in each of the cases graphed in figure 3. For all three evaluation functions HC1 examined the fewest solutions, although SA did not examine significantly more in the case of the Flexibility function. For the Flexibility and Understandability functions HC2 examined the greatest number of solutions, while for the Reusability function SA examined more by a small but statistically significant amount. In all three cases HCM examined the second-highest number of solutions, although in the case of the Reusability function it did not examine significantly more than HC2.

For this input all four search techniques produced quality improvements, but HC1 was most efficient in terms of number of solutions examined. In the case of the Flexibility function SA produced the greatest quality increase while examining the second fewest mean number of solutions, but still required the

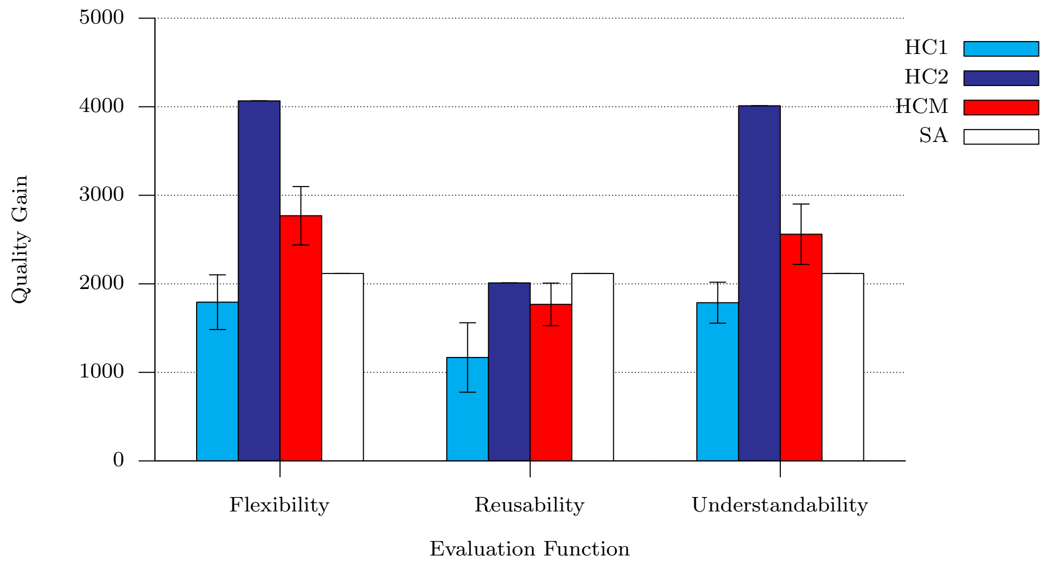

Fig. 4. Mean solutions examined – Input B

greatest run-time due to the processing overheads mentioned in section 4.4. In the case of the Understandability function HC2 produced the greatest quality increase, but at the cost of examining approximately twice as many solutions as each of the other search techniques.

In summary, the search techniques employed all demonstrated strengths in this experiment: first-ascent hill climbing consistently produced quality improvements at a relatively low cost, steepest-ascent hill climbing produced the greatest mean quality improvements in two of the six cases, multiple-ascent hill climbing produced individual solutions of highest quality in two cases, and simulated annealing produced the greatest mean quality improvement in one case.

6 Case Studies II; comparison of evaluation functions

In this section we present the observed changes in the metric values that comprise each of the three QMOOD evaluation functions Flexibility, Reusability and Understandability, for two Java programs after search-based refactoring. These results demonstrate the differing effects of the various evaluation functions on the output design and, along with an examination of the output code, allow us to discuss the effectiveness of the evaluation functions in actually increasing design quality. For clarity, we present the results of the best single run for each input in each of the following sections. As can be seen from figures 1 and 3, little variation was observed in mean quality increase in most cases.

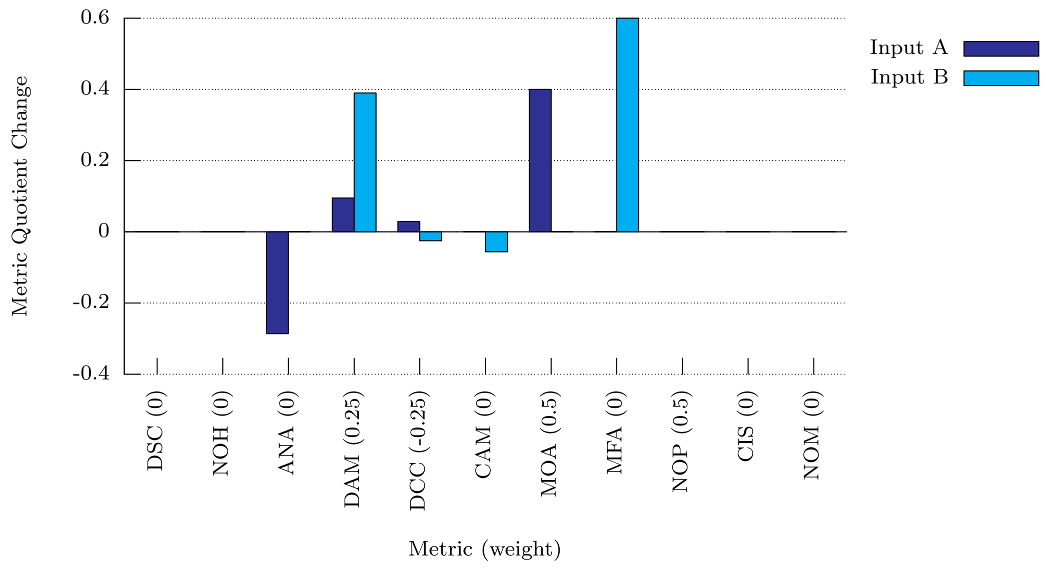

Fig. 5. Metric quotient changes, Flexibility function

For each evaluation function we will present the observed changes in metric values and describe how they contributed to the increase in quality function value. We also report the refactorings that were actually applied to the design in order to achieve these changes, and discuss the impact on the design from a programmer's perspective, having examined the refactored source code. The relevant subsections will be titled 'metric quotient changes', 'concrete design changes' and 'analysis of refactored design', respectively.

6.1 Flexibility

The Flexibility quality attribute in QMOOD is defined as the readiness of a design for adaptation to provide functionally related capabilities [2]. Metric changes resulting from use of the Flexibility function in CODe-Imp are shown in figure 5. Input A values are from one solution obtained using HC1; input B values are from one solution obtained using SA. It should be noted that these are metric *quotient* changes; differences from the identity value of 1 are graphed. A graphed value of 1 would equate to a doubling of the metric value from input to output. The actual metric weights comprising the Flexibility function are shown in table 2 and are repeated in figure 5. Details of the individual metrics can be found in table 1.

6.1.1 Metric quotient changes

In the case of input A, use of the QMOOD Flexibility function resulted in increases in the positively weighted DAM (0.25) and MOA (0.5) metrics, as well as the negatively weighted DCC (-0.25) metric. A decrease was also observed in the unweighted ANA metric.

In the case of input B, use of the Flexibility function resulted in increases in the positively weighted DAM (0.25) metric and the unweighted MFA metric, and decreases in the negatively weighted DCC (-0.25) metric and unweighted CAM metric.

6.1.2 Concrete design changes

In more concrete terms, for input A the DAM metric value increased from 0.82 to 0.90, so the output solution consisted of classes with a higher average ratio of non-public to public attributes. MOA for the program increased from 0.61 to 0.63, while DCC increased from 0.83 to 0.85. ANA decreased from 0.17 to 0.12. Examination of the refactored code revealed that nine applications of the Increase Field Security refactoring had increased encapsulation in eight different classes, while one Pull Up Field and one Replace Inheritance With Delegation refactoring had increased aggregation for one class at the cost of one additional coupling link.

In the case of input B the DAM metric value increased from 0.58 to 0.81, so again the output solution consisted of classes with a considerably higher average ratio of non-public to public attributes. In addition, DCC fell from 2.70 to 2.63 and CAM fell from 0.67 to 0.62, meaning that average coupling and average cohesion decreased slightly, while MFA increased from 0.03 to 0.10. Examination of the source code revealed that eleven applications (net) of the Pull Up Method and seven applications (net) of the Pull Up Field refactorings had reduced coupling in the case of six different classes but increased coupling in the case of three classes, for a net decrease of three classes coupled to one other. One class displayed slightly reduced cohesion. In addition, application of the Increase Field Security refactoring had improved encapsulation in seven classes.

6.1.3 Analysis of refactored designs

The refactored design in the case of input A exhibited improved data-hiding, with seven classes having a greater proportion of non-public methods compared to the input design. In addition, use of the inheritance mechanism had decreased by one instance, which had been replaced by aggregation. Coupling had also increased slightly. Examination of the refactored code revealed that where the inheritance relationship had been replaced by delegation the former subclass made no use of any of the former superclass's features, so the change was justified. The refactored design in this case was superior in terms of general object-oriented design principles such as the maximisation of encapsulation and the use of inheritance only where it is suitable, so there was some evidence that general maintainability had increased. There was no conclusive evidence that the refactored design would be more flexible in particular.

In the case of input B, the refactored design exhibited improved data-hiding, with seven classes having a greater proportion of non-public methods compared to the input design. In addition, coupling had been reduced for three classes net, at the cost of slightly reduced cohesion for one class. While high cohesion is valued in object-oriented design, low coupling is perhaps a greater priority when flexibility of design is paramount. Therefore, the refactored design in this case was not only better in terms of general object-oriented principles, but also could be regarded as more flexible than the input design.

Although the NOP metric is positively weighted in the QMOOD Flexibility function, no increase in NOP was observed for either input, nor was any increase observed for the positively weighted MOA metric in the case of input A. There are two possible explanations for this; firstly, there may not have been any legal refactorings that would increase these values or, secondly, any refactoring that increased these values may have also caused an undesired change of greater magnitude in other weighted metrics and hence been rejected.

6.2 Reusability

Metric changes resulting from use of the Reusability function in CODe-Imp are shown in figure 6. Input A values are from one solution obtained using HC2; input B values are from one solution obtained using HC1. Again, 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 and repeated in figure 6. It should be noted that it was necessary to impose a limit on the number of classes in refactored designs considered in the search space; this is discussed further below.

6.2.1 Metric quotient changes

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 the unweighted metrics ANA, DAM and MFA. Decreases were observed for the positively weighted metric CIS (0.5), the negatively weighted metric DCC (-0.25) and the unweighted metrics MOA and NOP.

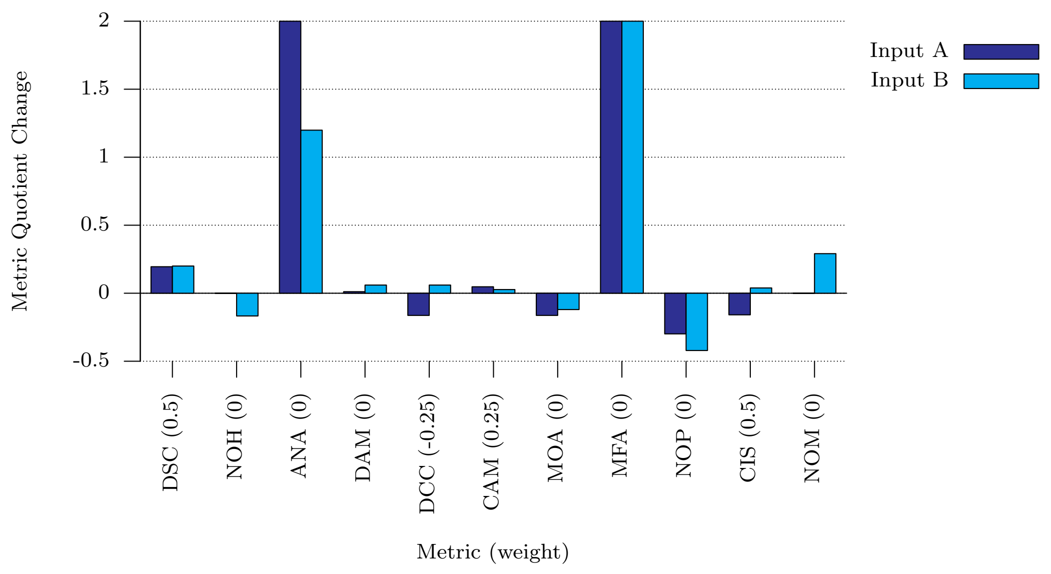

Fig. 6. Metric quotient changes, Reusability function

In the case of input B, use of the QMOOD Reusability function resulted in increases in the positively weighted metrics DSC (0.5), CAM (0.25) and CIS (0.5), the negatively weighted metric DCC (-0.25), and the unweighted metrics ANA, DAM, MFA and NOM. Decreases were observed in the unweighted metrics NOH, MOA and NOP.

In the cases of both input A and input B the most prominent changes are the increases in ANA and MFA (truncated on graph) which corresponded to greater than two-fold increases in these metric values in the case of each input. However, as these metrics are unweighted in the QMOOD Reusability function, their values had no impact on the search process.

6.2.2 Concrete design changes

Examination of the output code for input A revealed that major changes had been made to the design, including the addition of eight new classes within one inheritance hierarchy by means of the Extract Hierarchy refactoring. The addition of subclasses affected not only the DSC and ANA metrics, but also all metrics that are taken as an average over the number of design classes. The observed changes in the DCC, NOP and CIS metrics were due solely to this effect. The observed increase in the CAM metric was due to three applications of the Pull Up Method refactoring, with one input class exhibiting greater cohesion after methods had been repositioned. Three of the eight new classes also exhibited high CAM values, but in each case this was due to the class defining only one method.

Examination of the output code for input B revealed that major changes had also been made to this design by three applications of the Replace Inheritance With Delegation refactoring and the addition of six new classes by means of the Extract Hierarchy refactoring. In contrast to the results for input A, none of the observed changes occurred solely as a side-effect to the increase in the size of the design. Significant reductions were observed for this input in the case of the DCC metric, a result which was not observed for input A: due to repositioning of methods in inheritance hierarchies, four input classes were found to be dependent on fewer other classes after refactoring, though one input class gained a dependency. Improvements were also observed for the CAM metric in five of the input classes, while decreases were observed in two. High CAM values were observed for six of the ten new classes, but in all instances this was a result of the class defining only one method.

6.2.3 Analysis of refactored designs

The striking changes made to the two input designs under the QMOOD Reusability function provide a good illustration of the capacity of CODe-Imp to discover designs that conform more closely to a given quality model. However, in subjectively assessing the level of design improvement in this case we find fault with the evaluation function. Firstly, as mentioned above, it was necessary to impose a limit on the number of classes in the solution design. The reason for this is the large positive weight on the Design Size in Classes metric, which makes it likely that any addition of a class to the design will be interpreted as an improvement in the evaluation function. A runaway search process that adds an infinite number of empty classes to the design therefore becomes a possibility. In the cases of both input A and B, the ‘best’ solutions observed reached the imposed design size limit of 1.2 times the number of input classes, even though this meant including featureless classes in the output design. Secondly, putting aside the featureless classes problem, it is hard to see how a real increase in the reusability of a design is effected by reorganising an inheritance hierarchy to favour classes with only one method. While such classes can be said to be highly cohesive and loosely coupled it is unlikely that they would represent meaningful domain abstractions. We conclude that although some genuine improvements to the input design were observed, the QMOOD Reusability function in its published form is not well–suited to the task of search-based software refactoring with the set of refactorings we have employed.

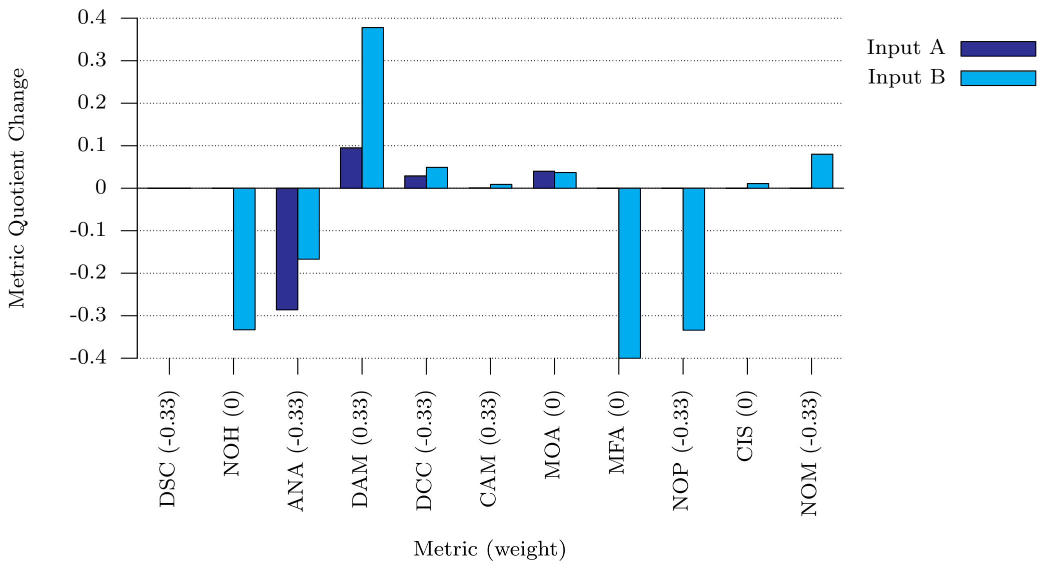

Fig. 7. Metric quotient changes, Understandability function

6.3 Understandability

Metric quotient changes resulting from use of the Understandability function in CODe-Imp are shown in figure 7. Input A values are from one solution obtained using HC1; input B values are from one solution obtained using HCM. 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 2 and repeated in figure 7.

6.3.1 Metric quotient changes

In the case of input A the Understandability function produced increases in the positively weighted metric DAM (weight 0.33) and the negatively weighted metric DCC (weight -0.33), and a decrease in the negatively weighted metric ANA (weight -0.33). A very small increase in the positively weighted metric CAM (0.33) was also observed which is not visible from the graph. In the case of input B the Understandability function produced increases in the positively weighted metrics DAM (0.33) and CAM (0.33), the negatively weighted metrics DCC (-0.33) and NOM (-0.33), and the unweighted metrics MOA and CIS. Decreases were observed for the negatively weighted metrics ANA (-0.33) and NOP (-0.33), and the unweighted metrics NOH and MFA.

6.3.2 Concrete design changes

The average DAM value for classes in the input A design rose from 0.82 to 0.90, so the proportion of non-public to public methods was considerably higher in the output design, indicating a greater level of encapsulation achieved by nine applications of the Increase Field Security refactoring. The average ANA value dropped from 0.17 to 0.12 as a result of one application of the Replace Inheritance with Delegation refactoring, and a slight increase in the average CAM value resulted from an application of the Pull Up Method refactoring.

For input B the average DAM value rose from 0.58 to 0.78, so a considerable improvement in terms of encapsulation was achieved as a result of eighteen applications of the Increase Field Security refactoring. Examination of the output code revealed that the decreases in ANA (0.40 to 0.33) and NOP (0.007 to 0.004), and the increases in DCC (2.70 to 2.83) and NOM (175 to 189) resulted from two applications of the Replace Inheritance with Delegation refactoring. Two applications of Pull Up Field and one of Push Down Method resulted in the increase in CAM (0.66 to 0.67), and helped prevent a greater increase in DCC.

In each case there was no decrease in DSC, despite the negative weight on this metric. This is because we do not allow the complete destruction of input classes by CODe-Imp, even where it can be done legally. The reason for this is simple; destroying a class that a programmer has identified as an enduring domain abstraction is undesirable because it represents a loss of domain-specific information, and could result in refactored designs containing a large proportion of automatically named classes if classes were repeatedly removed and replaced.

6.3.3 Analysis of refactored designs

From a programmer's perspective, the changes to the two designs observed after running CODe-Imp under the Understandability evaluation function were positive. The increases in encapsulation and method cohesion seen in both cases, as well as the decrease in polymorphism seen in one case, mean an improvement in the understandability of the designs from our subjective viewpoint. In addition, for both inputs unjustified inheritance relationships were replaced with delegation; for input A a subclass using none of its superclass's features and for input B a subclass using only one of its superclass's ten methods were refactored. This shows that a design can be improved in ways that are not directly measured; in this case the redefinition of the relationships between several classes was a product of reducing inheritance and polymorphism while maintaining cohesion and encapsulation. Furthermore, these improvements were made without incurring any loss of domain-specific information encapsulated in the design such as class or member names. For these reasons, the results reported here suggest that the QMOOD Understandability function is a quality model suitable for search-based software refactoring.

7 Future Work

In common with all fully automated refactoring tools CODe-Imp has the drawback that changes in the design must be communicated to the programmer. Several issues such as the relocation of program comments and need to introduce new identifiers such as class names can complicate this task [10]. These issues could be addressed by a programmer review of the refactored code, but our assumption that this would be less expensive than completely manual refactoring has not yet been proved.

To date, the refactoring capacity of CODe-Imp has mainly focussed on transforming the structure of inheritance hierarchies. In order for this technique to become part of software engineering practice it will be necessary to increase the power of the tool. However, only certain refactorings can be fully automated. Further research is required to determine whether the refactoring capacity of CODe-Imp can become diverse enough to satisfy the maintenance programmer.

While we have demonstrated that object-oriented programs can be automatically refactored to improve their design with respect to a given quality model, much work remains to be done in order to provide a quality model that is of use in the general case. Although the QMOOD Flexibility and Understandability evaluation functions appear sufficient to produce genuine improvements in the case studies described here, larger studies with independent assessment of the refactored designs are required in order to fully establish this approach. Further studies are also required to establish the level of resources needed to successfully apply this approach in an industrial setting.

In the future we envisage the search-based software maintenance approach forming a synergistic relationship with operational research in domain-specific quality models. Where a specific model is proposed it would be possible to validate and refine it using the search-based approach; a quality model of sufficient accuracy could then be used to drive the search-based design improvement process. It would also be possible to give the programmer more control over the automated refactoring process, for example by protecting certain methods, fields or classes from alteration. In this way, portions of the design known to be of high quality could be preserved.

8 Conclusion

The results reported here support the thesis that object-oriented programs can be automatically refactored to improve quality as measured by well-defined quality models, and partially validate the search-based software maintenance approach. We have shown that evaluation function increases can be obtained in all 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 elaborate on the contributions stated in section 1; 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 in its present form 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.

The search techniques employed all demonstrated strengths in this experiment: first-ascent hill climbing consistently produced quality improvements at a relatively low cost, steepest-ascent hill climbing produced the greatest mean quality improvements in certain cases, multiple-restart hill climbing produced individual solutions of highest quality in certain cases, and simulated annealing produced the greatest mean quality increase by a clear margin for one input and evaluation function pair. We conclude that both local search and simulated annealing are effective in the context of search-based software refactoring, as did Mitchell et al [28] for the related problem of module clustering. Perhaps the most significant observation here was that quality improvements were obtained using simple search techniques with manageable run-times such as first-ascent hill climbing, which bodes well for the scalability of the approach.

In the case of the Understandability function, genuine improvements were made to the design of both of the programs studied here. In addition, the fact that these improvements were made using local search techniques indicates that the search landscape is not as difficult as might be imagined. We conclude that a search-based software maintenance tool based on the QMOOD Understandability evaluation function has the potential to be of real use to the software engineer faced with a difficult reengineering task.

References

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

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

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[4] Fernando Brito e Abreu. The MOOD2 metrics set (in portuguese); relatorio r7/98, abril, 1998a. Technical report, Grupo de Engenharia de Software, INESC, 1998.

[5] Fernando Brito e Abreu. Using OCL to formalize object oriented metrics definitions. Technical report, Grupo de Engenharia de Software, INESC, 2001.

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

[7] Fernando Brito e Abreu, Luis Ochoa, and Miguel Goulão. Candidate metrics for object oriented software within a taxonomy framework. In *Journal of Systems and Software*, volume 26. North-Holland, Elsevier Science, July 1994.

[8] Fernando Brito e Abreu, Luis Ochoa, and Miguel Goulão. The GOODLY design language for MOOD2 metrics collection. In *ECOOP Workshops*, pages 328–329, 1999.

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

[10] Frank W. Calliss. Problems with automatic restructurers. *SIGPLAN Notices*, 23(3):13–21, 1988.

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

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

[13] John A. Clark, José Javier Dolado, Mark Harman, Robert M. Hierons, B. Jones, M. Lumkin, Brian S. Mitchell, Spiros Mancoridis, K. Rees, Marc Roper, and Martin J. Shepperd. Formulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.

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

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

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

[17] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *WCRE*, page 97, 2002.

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

[19] Mark Harman and John A. Clark. Metrics are fitness functions too. In *IEEE METRICS*, pages 58–69, 2004.

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

[21] R. Harrison, S. Counsell, and R. Nithi. An evaluation of the MOOD set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, 1998.

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

[23] Kiarash Mahdavi, Mark Harman, and Robert M. Hierons. A multiple hill climbing approach to software module clustering. In *ICSM*, pages 315–324, 2003.

[24] Spiros Mancoridis, Brian S. Mitchell, Yih-Farn Chen, and Emden R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *ICSM*, pages 50–, 1999.

[25] Spiros Mancoridis, Brian S. Mitchell, C. Rorres, Yih-Farn Chen, and Emden R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC*, pages 45–, 1998.

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

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

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

[29] Brian S. Mitchell, Martin Raverso, and Spiros Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *WICSA*, pages 181–190, 2001.

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

[31] Mel Ó Cinnéide. *Automated Application of Design Patterns: a Refactoring Approach.* PhD dissertation, University of Dublin, Trinity College, Department of Computer Science, 2000. Available from: http://www.cs.ucd.ie/staff/meloc/home/papers/thesis.

[32] M. O'Keeffe and M. Ó Cinnéide. A stochastic approach to automated design improvement. In James F. Power and John 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.

[33] 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.

[34] William Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[35] Donald Bradley Roberts. *Practical analysis for refactoring.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999. Adviser-Ralph Johnson.

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

[37] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 43(14):841–854, 2001.

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

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