Automated Design Improvement by Example

Mark O'KEEFFE [a,1] and Mel Ó CINNÉIDE [a]

[a] *School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 14, Ireland*

Abstract. The high cost of software maintenance could potentially 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 design quality models based on a set of metrics, by formulating the task as a search problem in the space of alternative designs. However, no consensus exists on a single quality model for object-oriented design, since the definition of 'quality' can depend on the purpose, pedigree and perception of the maintenance programmer. We therefore demonstrate here the flexibility of our approach by automatically refactoring several Java programs to conform with quality models based on the metric values of example programs. Results show that an object-oriented program can be automatically refactored to reduce its dissimilarity in terms of a set of design metrics to another program having some desirable trait, such as ease of maintenance.

Keywords. Automated Design Improvement, Refactoring, Metrics

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, flexible, and modular. 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 *software decay* [16]), 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 or extended. In this context, the application of refactorings can be considered movement in the space of alternative designs.

[1] Corresponding author; email: mark.okeeffe@ucd.ie

Our novel approach to automated design improvement is the formulation of the refactoring task as a search problem; given a design quality model we apply automated refactorings to a program in order to move through the space of alternative designs and search for those of highest quality. However, no consensus exists on a single model of object-oriented design quality, and none is likely to develop since the perceived quality of a design is highly subjective. In other words, that which makes a design understandable may depend on the skills of the programmer, that which makes a design flexible may depend on his/her purpose, and that which makes a design reusable may depend on the nature of the proposed reuse.

Previous search-based refactoring work by the authors has shown that object-oriented programs can be automatically refactored in order to conform more closely to a given quality model [27]. However, because the priorities of the maintenance programmer can change from day to day and best programming practice can vary between domains, it is unlikely that a single quality model can be defined that will maximise the effectiveness of search-based refactoring across the board. The programmer wishing to take advantage of search-based refactoring is therefore left with the problem of developing a suitable quality model. This paper shows how that complex process can be circumvented by basing a quality model on the characteristics of some other program in the problem domain that is known to exhibit the desired property. Therefore, if any one program is found to have a desirable trait such as high maintainability or reusability other programs can be quickly refactored to more closely resemble the example, using our approach.

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 [17]. Such problems include *module clustering*, where a software system is reorganised into loosely coupled clusters of highly cohesive modules to aid reengineering [14,18,20,23], test data generation [21], automated testing [31] and project management problems such as requirements scheduling [1] and project cost estimation [9,13]. An overview of such work and comprehensive recent references can be found in [12] and [17] respectively. Of particular relevance to this work is [17], 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 [10], who proposed algorithms to restructure class hierarchies in order to maximise abstrac-

tion, and Moore [24], 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 [24].

In addition to previous fully-automated approaches to design improvement, semi-automated approaches that require user interaction have been reported. Such semi-automated approaches 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 [15], and by Tahvildari [30], whose system also recommends 'meta-pattern transformations' that can be applied to ameliorate the defect. Another aspect of semi-automated design improvement is simply the automation of application of refactorings, with the particular refactorings performed determined entirely by the user [28]. The drawback of all semi-automated tools is, of course, that they reduce the level of programmer intervention required somewhat less than fully-automated tools have the potential to do.

Seng et. al. [29] describe a similar approach to ours [25,26,27] but use a genetic algorithm rather than local search or simulated annealing to solve the combinatorial optimisation problem. The evaluation function employed is novel rather than previously validated, but is based on well-known metrics such as *Response For a Class* (RFC) and *Weighted Methods per Class* (WMC) from Chidamber & Kemerer's MOOSE suite [11], among others. The authors report success in automatically repositioning displaced methods in the class structure, not limited to movement within inheritance hierarchies. However, only the Move Method refactoring is considered so the extent of change within the class structure is limited.

3. Experimental Methodology

3.1. CODe-Imp

We have constructed a prototype automated design-improvement tool called CODe-Imp[2] in order to facilitate experimentation with search-based software maintenance. CODe-Imp takes Java 1.4 source code as input and extracts design metric information via a Java Program Model (JPM), calculates quality values according to an evaluation 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.

[2]Combinatorial Optimisation Design-Improvement

3.2. Refactorings

Many refactorings are described in the literature, in particular by Fowler [16]. The precise definition of refactorings is an area of research by itself [28]. Fowler, among others, defines refactorings in natural language, so a degree of interpretation is required in automating these refactorings. In addition, language-specific features such as 'package' visibility in Java must be taken into account.

The refactoring configuration of CODe-Imp for the experiments reported here consisted of the fourteen 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 such as simulated annealing to move freely through the solution space, though not for the hill-climbing approach employed here.

The CODe-Imp refactorings are, in general, based on a Fowler refactoring of the same name. Some reverse refactorings were added, as were some obvious alternative refactorings. Refactorings were selected according to the criteria below:

1. GRANULARITY – only refactorings operating at the level of methods/ attributes and higher were accepted. Lower-level refactorings are generally concerned with rectifying implementation mistakes, and do not have a large impact on the design of a program. An example refactoring that does not meet this criterion is Introduce Assertion ([16], p.267), in which an 'assert' statement is added at the start of a method to ensure some necessary conditions hold.

2. REVERSIBILITY – only refactorings that could be reversed by some other acceptable refactoring were accepted. The preconditions for all accepted refactorings were formulated to ensure this property. An example refactoring that cannot be applied in a reversible fashion, and so fails to meet this criterion, is Remove Parameter ([16], p.277).

3. AUTOMATION – the motivation for some refactorings necessitates programmer input; for example, the refactoring Extract Method ([16], p.110) is useful because the name given to a new method replacing a complex statement tells the programmer what the statement does. An extracted method with a meaningless name would actually make the code harder to understand. For this reason, some refactorings were not selected simply because it does not make sense to automate them.

The refactorings implemented in CODe-Imp are:

1. **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 ([16], p.329).
2. **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 ([16], p.320).
3. **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 ([16], p.328).

4. **Pull Up Method** – moves a method from some class(es) to their immediate superclass. This refactoring is intended to help eliminate duplicate methods among sibling classes, and hence reduce code duplication in general ([16], p.322).
5. **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.
6. **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.
7. **Increase Field Security** – increases the security of a field from public to protected, from protected to package or from package to private. This refactoring increases data encapsulation.
8. **Decrease Field Security** – decreases the security of a field from private to package, from package to protected or from protected to public. This refactoring reduces data encapsulation.
9. **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 ([16], p.352).
10. **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 ([16], p.355).
11. **Increase Method Security** – increases the security of a method from public to protected, from protected to package or from package to private. This refactoring can reduce the size of the public interface of a class ([16], p.303).
12. **Decrease Method Security** – decreases the security of a method from private to package, from package to protected or from protected to public. This refactoring can increase the size of the public interface of a class.
13. **Make Superclass Abstract** – declares a constructorless class explicitly abstract. This increases some measures of abstraction, and can facilitate other refactorings.
14. **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.

One of the functions of CODe-Imp's Java Program Model (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 static precondition checking, the details of which are beyond the scope of this paper.

3.3. Dissimilarity Function

Several metric suites were considered, as described below, but QMOOD was selected in order that the results of this work could be readily compared with the results of previous search-based refactoring approaches [27].

3.3.1. CK

The Metrics Suite for Object-Oriented Design (known as CK) of Chidamber and Kemerer [11] 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. [32] 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).

3.3.2. 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 [19]. 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.

3.3.3. 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 metrics are as follows:

1. **Design Size in Classes (DSC)**– "A count of the total number of classes in the design."[2] Interpreted as excluding imported library classes. Corresponds to the object-oriented design property of 'design size' in QMOOD.

2. **Number Of Hierarchies (NOH)**– "A count of the number of class hierarchies in the design."[2] Interpreted as excluding hierarchies that consist of a specialised class within the design and a generalised class outside. Corresponds to the object-oriented design property of 'Hierarchies' in QMOOD.

3. **Average Number of Ancestors (ANA)**– "The average number of classes from which each class inherits information."[2] Corresponds to the object-oriented design property of 'Abstraction' in QMOOD.

4. **Number of Polymorphic Methods (NOP)**– "A count of the number of the methods that can exhibit polymorphic behaviour."[2] Interpreted as the sum over all classes, where a method can exhibit polymorphic behaviour if it is overridden by one or more descendent classes. Corresponds to the object-oriented design property of 'Polymorphism' in QMOOD.

5. **Class Interface Size (CIS)**– "A count of the number of public methods in a class."[2] Interpreted as the average over all classes in a design. Corresponds to the object-oriented design property of 'Messaging' in QMOOD.

6. **Number Of Methods (NOM)**– "A count of all the methods defined in a class."[2] Interpreted as the sum over all classes in a design. Corresponds to the object-oriented design property of 'Complexity' in QMOOD.

7. **Data Access Metric (DAM)**– "The ratio of the number of private (protected) attributes to the total number of attributes declared in the class."[2] Interpreted as the average over all design classes *with at least one attribute*, of the ratio of non-public to total attributes in a class. Corresponds to the object-oriented design property of 'Encapsulation' in QMOOD.

8. **Direct Class Coupling (DCC)**– "A count of the number of different 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."[2] 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 imported library classes from the computation. Corresponds to the object-oriented design property of 'Coupling' in QMOOD.

9. **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."[2] Interpreted as an average over all classes having at least one method. We have excluded constructors and implicit 'this' parameters from the computation. Cor-

responds to the object-oriented design property of 'Cohesion' in QMOOD.

10. **Measure Of Aggregation (MOA)**– "A count of the number of data declarations whose types are user-defined classes."[2] Interpreted as the sum of values for all design classes. We define 'user defined classes' as classes defined in the source code of the input program. Corresponds to the object-oriented design property of 'Composition' in QMOOD.

11. **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."[2] Interpreted as the average over all classes in a design (with at least one method available) 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. Corresponds to the object-oriented design property of 'Inheritance' in QMOOD.

The dissimilarity function itself gives the sum of absolute differences between the quotients for subject program metric over example program metric and the identity value of one. More precisely: the dissimilarity value $d(P_s P_e)$ for a subject program P_s and an example program P_e, where the metric value for metric m on program P is given by $m(P)$, is defined as

$$d(P_s P_e) = \sum_{n=1}^{11} \left| \frac{m_n(P_s)}{m_n(P_e)} - 1 \right|$$

and is minimised by combinatorial optimisation in a refactored program ${P_s}'$.

3.4. Search Technique

Previous work has shown that a steepest-ascent hill-climbing search is sufficient to produce consistently good results in automated refactoring, though simulated annealing is more effective in some cases [27]. Hill climbing has also been found to be surprisingly effective in solving similar combinatorial optimisation problems such as module clustering [18,22]. For clarity, we have employed steepest ascent hill-climbing alone in this study.

Steepest ascent hill-climbing (HCS) is a local search algorithm where the search examines all neighbouring solutions and moves to the neighbour of highest quality. This is repeated until no neighbour of higher quality can be found, at which point the search terminates. A neighbour of a solution S is defined as a solution that can be generated by one application of one refactoring to S.

3.5. Input

Input consisted of two open-source Java 1.4 programs of a maximum size of one hundred classes randomly selected from SourceForge[3] via java-source.net, and a self-contained

[3] http://sourceforge.net/

subset of the *Spec-Benchmarks*[4] standard performance evaluation framework, to which it was known a large number of refactorings could be applied. The programs selected were:

1. **Input A:** *Beaver*, a parser generator
   - 93 classes
   - 4999 SLOC
   - 9 inheritance hierarchies
   - 177 refactorings could initially be applied
2. **Input B:** *SpecCheck*, a benchmarking program
   - 41 classes
   - 4836 SLOC
   - 5 inheritance hierarchies
   - 351 refactorings could initially be applied
3. **Input C:** *Mango*, a collections library
   - 51 classes
   - 1131 SLOC
   - 0 inheritance hierarchies
   - 28 refactorings could initially be applied

4. Results

4.1. Overview

The aim of the study described here was to demonstrate that Java programs can be automatically refactored to conform more closely to a quality model extracted from an example program and defined in terms of a set of object-oriented metrics. No assumptions were made as to the quality (or otherwise) of the input programs, nor was the range of dissimilarity values controlled by selection of input programs, which was random. Each of the three input programs was automatically refactored with CODe-Imp, using metric values of the other two programs as alternative quality models.

Experiments were carried out on a 2.2GHz AMD Athlon powered PC with 1GB CL2 RAM. Mean processing time per solution examined was approximately one second, including model building, metric extraction, quality assessment, discovery of legal refactorings, and actual (AST) refactoring. Total run-time varied between less than one minute and 66 minutes, depending on the number of refactorings possible for the input program and the number of refactorings applied. 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.

Tables 1– 3 show the total difference function values as defined in section 3.3 for input program and refactored input program, as well as individual metric components given by

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

	DSC	NOH	ANA	DAM	DCC	CAM	MOA	MFA	NOP	CIS	NOM	$d\,(P_s\,P_e)$
						input *Beaver*, example *Spec-Check*						
Beaver	0.268	0.5	1.343	0.296	2.256	0.67	1.952	1	0.401	0.355	0.465	**8.903**
Beaver'	0.268	0	0.952	0.156	2.417	0.059	2.061	0	0.066	0.334	0.422	**6.736**
change	0	-0.5	-0.390	-0.140	0.161	-0.009	0.109	-1	-0.335	-0.021	-0.043	**-2.167**
						input *Beaver*, example *Mango*						
Beaver	0.25	1	1	0.332	1.455	0.248	2.273	1	1	0.266	0.022	**8.846**
Beaver'	0.25	1	1	0.193	1.455	0.216	2.273	0	1	0.258	0.022	**7.667**
change	0	0	0	-0.138	0	-0.0323	0	-1	0	-0.008	0	**-1.178**

Dissimilarity values for input program *Beaver* and refactored program *Beaver'*. Note that dissimilarity quotients rather than actual metric values are shown, so the identity value 1 indicates a perfect match between subject program and example program for that metric. Negative values in the 'change' rows indicate metrics that have been brought closer to the desired value.

Table 1. Results for input program Beaver

	DSC	NOH	ANA	DAM	DCC	CAM	MOA	MFA	NOP	CIS	NOM	$d\,(P_s\,P_e)$
						input *Spec-Check*, example *Beaver*						
Spec-Check	0.367	0.333	0.573	0.421	0.693	0.063	0.661	1	0.286	0.550	0.869	**5.816**
Spec-Check'	0.467	0.333	0.091	0.000	0.714	0.063	0.684	0.237	0.380	0.400	0.869	**4.238**
change	0.1	0	-0.482	-0.421	0.021	0	0.023	-0.763	0.094	-0.150	0	**-1.578**
						input *Spec-Check*, example *Mango*						
Spec-Check	0.025	1	1	0.050	0.246	0.296	0.109	0	1	0.137	0.827	**4.690**
Spec-Check'	0.125	1	1	0.000	0.313	0.295	0.010	0	1	0.005	0.827	**4.575**
change	0.1	0	0	-0.050	0.067	-0.001	-0.099	0	0	-0.133	0	**-0.115**

Dissimilarity values for input program *Spec-Check* and refactored program *Spec-Check'*. Note that dissimilarity quotients rather than actual metric values are shown, so the identity value 1 indicates a perfect match between subject program and example program for that metric. Negative values in the 'change' rows indicate metrics that have been brought closer to the desired value.

Table 2. Results for input program Spec-Check

$$d_{m_n}\left(P_s P_e\right) = \left| \frac{m_n\left(P_s\right)}{m_n\left(P_e\right)} - 1 \right|$$

where $d_{m_n}\left(P_s P_e\right)$ is the individual metric difference value of metric m_n for a subject program P_s and an example program P_e. Negative values in the 'change' rows indicate a metric that has been brought closer to the desired value, positive values indicate a metric that has been taken further from the desired value, and zero indicates no metric change.

	DSC	NOH	ANA	DAM	DCC	CAM	MOA	MFA	NOP	CIS	NOM	$d(P_s P_e)$
						input *Mango*, example *Spec-Check*						
Mango	0.0244	1	1	0.052	0.326	0.419	0.098	0	1	0.120	0.452	**4.495**
Mango'	0.0244	1	1	0.052	0.326	0.419	0.098	0	1	0.120	0.452	**4.495**
change	0	0	0	0	0	0	0	0	0	0	0	**0.000**
						input *Mango*, example *Beaver*						
Mango	0.333	1	1	0.495	0.592	0.330	0.694	1	1	0.362	0.022	**6.833**
Mango'	0.333	1	1	0.495	0.592	0.330	0.694	1	1	0.362	0.022	**6.833**
change	0	0	0	0	0	0	0	0	0	0	0	**0.000**

Dissimilarity values for input program *Mango* and refactored program *Mango'*. Note that dissimilarity quotients rather than actual metric values are shown, so the identity value 1 indicates a perfect match between subject program and example program for that metric. Negative values in the 'change' rows indicate metrics that have been brought closer to the desired value.

Table 3. Results for input program Mango

4.2. Beaver

Results for the input program *Beaver*, a parser generator of 93 classes, are shown in table 1. The dissimilarity value for this program, taking *Spec-Check* as example program, decreased by 24.3%, from 8.903 to 6.736. Eight metrics were brought closer to the example program values, while two moved farther away. These metric changes were effected by one Pull Up Field, two Push Down Method, seven Increase Field Security, one Decrease Method Security and two Replace Inheritance with Delegation refactorings, so very significant changes were made to the class structure.

Taking *Mango* as example program, the dissimilarity value for *Beaver* decreased by 13.3%, from 8.903 to 7.667. Four metrics were brought closer to the example program values, with none moved farther away. These metric changes were effected by two Pull Up Field, one Push Down Method, six Increase Field Security and one Decrease Method Security refactorings, so significant changes were made to the class structure.

4.3. Spec-Check

Results for the input program *Spec-Check*, a benchmarking program of 41 classes, are shown in table 2. The dissimilarity value for this program, taking *Beaver* as example program, decreased by 27.1%, from 5.816 to 4.238. Surprisingly, given the large decrease in dissimilarity value, the same number of metrics were brought closer and moved away from the example program values (4). However, the magnitude of absolute metric quotient change was much greater in the negative/closer changes, with metrics such as ANA and DAM changing from approximately half the desired value to within 1%. These metric changes were effected by one Pull Up Field, three Extract Hierarchy, eleven Decrease Field Security and six Increase Method Security refactorings, so very significant changes were made to the class structure.

Taking *Mango* as example program, the dissimilarity value decreased by only 2.45%, from 4.690 to 4.575, although the input/example dissimilarity value was close to

the lowest observed, in this case. Nevertheless, four metrics were brought closer to the example program values while two were taken farther away. These metric changes were effected by one Pull Up Method, four Extract Hierarchy, four Increase Field Security, one Decrease Field Security and six Increase Method Security refactorings, so despite the small change in dissimilarity value very significant changes were made to the class structure.

4.4. Mango

Results for the input program *Mango*, a collections library of 51 classes, are shown in table 3. No decrease in dissimilarity value between the input and refactored program was observed for this input, with either example program. However, only 28 refactorings were possible for this program, compared to 190 for *Beaver* and 351 for *Spec-Check*. This lack of instances where the preconditions for potential refactoring were met in the *Mango* program is the most likely explanation for the failure of CODe-Imp to reduce the dissimilarity value. Inspection of the *Mango* program revealed that its designers had made no use of the inheritance mechanism. Since most of CODe-Imp's refactorings operate on inheritance hierarchies, this explains the small number of possible refactorings for this program.

4.5. Summary

Of three input programs automatically refactored by CODe-Imp, one was made significantly more similar in terms of the metrics suite employed to two example programs, one was made significantly more similar to one example program and slightly more similar to the other, and one was not made any more similar to either example program. Unsurprisingly, the greatest reductions in dissimilarity value were observed where a large number of refactorings were possible for the input program, and there was a relatively high dissimilarity value between input program and example program.

5. Conclusions & Future Work

In this study we have demonstrated that Java programs can be automatically refactored to reduce their dissimilarity to other programs in terms of a set of object-oriented metrics, provided the subject program in question makes use of object-oriented features such as inheritance that make refactoring possible. This work has the potential to contribute to the development of domain-specific object-oriented quality models, which will be of use in both traditional and search-based software maintenance.

Future work will include repeating these experiments with larger input programs, alternative search techniques and other metric suites. The limitations of this work centre around the fact that similarity between programs as measured by a metrics suite may not match with the human perception of similarity, particularly in the case of ontological artifacts such as design patterns. We plan to further explore this relationship in the future by supplementing the metrics suite with pattern-detection routines, in order to extract richer quality models from example programs.

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. *Journal of Systems and Software*, 26(1), 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] 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.

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

[12] John A. Clark, José J. 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.

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

[14] 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.

[15] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In Arie van Deursen and Elizabeth Burd, editors, *WCRE*, page 97. IEEE Computer Society, 2002.

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

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

[18] Mark Harman, Robert M. Hierons, and Mark Proctor. A new representation and crossover operator for search-based optimization of software modularization. In William B. Langdon, Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant Honavar, Günter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund K. Burke, and Natasa Jonoska, editors, *GECCO*, pages 1351–1358. Morgan Kaufmann, 2002.

[19] 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.

[20] 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.

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

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

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

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

[25] 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.

[26] M. O'Keeffe and M. Ó Cinnéide. Towards automated design improvement through combinatorial optimisation. In *Proceedings of the 4th International Workshop on Directions in Software Engineering Environments (WoDiSEE 2004)*, May 2004.

[27] 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.

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

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

[30] 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.

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

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

Algorithm 1 Steepest Ascent Hill-Climbing

```
1:  currentNode = startNode;
2:  madeAscent = TRUE;
3:  while madeAscent do
4:     madeAscent = FALSE;
5:     L = NEIGHBOURS(currentNode);
6:     bestNeighbour = null;
7:     bestNeighbourEval = -INFINITY;
8:     for all x in L do
9:        if (EVAL(x) > bestNeighbourEval then
10:          bestNeighbour = x;
11:          bestNeighbourEval = EVAL(bestNeighbour);
12:       end if
13:    end for
14:    if bestNeighbourEval > EVAL(currentNode) then
15:       currentNode = bestNeighbour;
16:       madeAscent = TRUE;
17:    end if
18: end while
19: return currentNode;
```