Solving the Class Responsibility Assignment Problem in Object-oriented Analysis with Multi-Objective Genetic Algorithms

Michael Bowman[1] Lionel C. Briand[2] Yvan Labiche[1]

[1] Carleton University, Software Quality Engineering Lab, 1125 Colonel By Drive Ottawa, ON K1S 5B6, Canada {mgbowman, labiche}@sce.carleton.ca

[2] Simula Research Laboratory & University of Oslo, P.O. Box 134, Lysaker, Norway briand@simula.no

Abstract

In the context of object-oriented analysis and design (OOAD), class responsibility assignment is not an easy skill to acquire. Though there are many methodologies for assigning responsibilities to classes, they all rely on human judgment and decision making. Our objective is to provide decision-making support to re-assign methods and attributes to classes in a class diagram. Our solution is based on a multi-objective genetic algorithm (MOGA) and uses class coupling and cohesion measurement for defining fitness functions. Our MOGA takes as input a class diagram to be optimized and suggests possible improvements to it. The choice of a MOGA stems from the fact that there are typically many evaluation criteria that cannot be easily combined into one objective, and several alternative solutions are acceptable for a given OO domain model. Using a carefully selected case study, this article investigates the application of our proposed MOGA to the class responsibility assignment problem, in the context of object-oriented analysis and domain class models. Our results suggest that the MOGA can help correct suboptimal class responsibility assignment decisions and perform far better than simpler alternative heuristics such as hill climbing and a single objective GA.

1 INTRODUCTION

Class responsibility assignment (CRA) is about deciding where responsibilities, under the form of class operations (as well as the attributes they manipulate), belong and how objects should interact (by using those operations). CRA is often identified as the most important learning goal in object-oriented analysis and design (OOAD) since it "tends to be a challenging skill to master (with many "degrees of freedom" or alternatives), and yet is vitally important." [23] There is indeed evidence that this is hard to teach and apply (e.g., [35]). Not only is this vital during initial analysis/design phases, but also during maintenance when new responsibilities have to be assigned to (new) classes, or existing

responsibilities have to be changed (e.g., moved to other classes). Though there are many (incremental and iterative) methodologies to help assign responsibilities to classes (e.g., [8]), they all rely on human judgment and decision making, primarily based on heuristics. In this paper, our objective is to present and evaluate a multi-objective genetic algorithm to provide decision-making support for class responsibility assignment in UML class diagrams. Our work takes place in the context of Model Driven Architecture/Development (MDD) [21], whereby class responsibility assignment is first performed when creating (or modifying) the Platform Independent Model (PIM) before the PIM is automatically transformed into a Platform Specific Model (PSM), which will eventually be the basis for code generation. Note that in the MDD context, software evolution consists of changing models, not code, which is then re-generated. In this paper, we first focus on diagrams exclusively containing domain classes [8] (the PIM), which are often referred to as analysis or domain models and which are usually part of early analysis steps [23]. Future work will explore similar solutions for lower-level design class diagrams.

Our work bears some similarity to refactoring. Most of the work in this area has considered source code refactoring, though there is a trend to also consider refactorings at higher levels of abstraction, such as refactorings of UML models [26]. There are however important differences between our approach and UML refactorings, as further discussed in Section 2.

Our approach is based on a multi-objective genetic algorithm (MOGA) [36], uses class coupling and cohesion measurement [4, 5], and aims at providing interactive feedback to designers. The MOGA takes as input a class diagram to be optimized, and information about method and attribute dependencies which can be extracted from other UML diagrams, e.g., Sequence diagrams. It also accepts user defined constraints on what can and cannot change in the class diagram. It then evaluates the class diagram based on multiple, complementary measures of coupling and cohesion, and suggests possible improvements using these measures as evaluation criteria. The MOGA provides alternative solutions to the user for her perusal and may ask for feedback to get further guidance, though the latter is not addressed in this paper. The goal is therefore to discover optimal assignments of attributes and methods to classes in regards to various aspects of coupling and cohesion, thus leading to a more maintainable model [4], while accounting for user defined constraints on the class diagram.

Our main motivation for using the more complex MOGAs is practical since it is very difficult to combine the many criteria used to assess an analysis class diagram into one unique fitness function. Furthermore, by allowing the user to specify some constraints on the model, along with interacting with the GA itself, the search will be guided towards an optimal class diagram that will be based on both coupling and cohesion and additional designer inputs. The motivation is again practical since no matter how complete our list of objectives and fitness functions, there will always be additional practical considerations that the designers will need to account for when selecting a specific solution.

The rest of the paper is structured as follows. Section 2 describes related work. Sections 3 to 6 describe our approach. A case study is described in Section 7 and conclusions are drawn in Section 8.

2 RELATED WORK

A wide range of possible applications of meta-heuristic search techniques, such as GAs, to the field of software engineering are discussed in [10], e.g., the maintenance and re-engineering of software using program transformations. This idea is expanded upon in [27] where the authors use a simulated annealing algorithm to automatically improve the structure of an existing inheritance hierarchy. The design measures are expressed as a sum of weighted objectives in order to measure the designs and suggest improvements. This is further expanded in [32], where the authors use a GA to automatically determine potential refactorings of a (reverse-engineered) class structure, not just an inheritance hierarchy. The authors consider a subset of Fowler's refactorings [16]: moving a method from a class to another class, moving methods/attributes up/down in an inheritance hierarchy. Applying these refactorings is constrained as the authors consider specific, predefined code refactorings, thereby limiting the search space for the assignment of class responsibilities. For instance, if a method moves, call sites have to be updated and therefore the caller needs to have some visibility to where the method has been moved. Therefore, a non-static method can only be moved from class X to class Y if there is already a relationship (association through attribute, dependency through parameter) between X and Y. We will see that we do not have such constraints as we work on analysis class diagrams instead of the source code. As a fitness function, the paper uses a sum of weighted objectives that measures the coupling, cohesion, complexity and stability of the system's source code. The algorithm then searches the source code for the possible refactorings mentioned above that will improve these objectives

according to the fitness function, and finally presents these refactorings to the designer as potential improvements to the system. The focus is to help prevent code decay.

The above approaches both use a sum of weighted objectives to balance the influence of various quality measures on the fitness function. While this is clearly helpful, it can only take into account one possible, predetermined tradeoff among objectives, whereas the Pareto based multi-objective GA [39] we use is able to present a number of possible tradeoffs to the designer. We think this is paramount in our context as it is a priori difficult for any designer to weigh different design properties based on any objective criteria. Multi-objective simulated annealing (SA) algorithms are an alternative to consider, but this field of work is not as extensive and mature as for multi-objective GAs, thus our choice of the latter. Additionally, as opposed to focusing on the prevention of code decay during iterative development, we aim at providing decision aid to improve early OOAD models.

Recently, a similar Pareto optimal search based approach to refactorings has been investigated in [19]. The authors use multiple runs of a random mutation hill climber in order to generate a Pareto optimal front of possible refactorings. Each execution uses a single objective fitness function, but as the random mutation hill climber is non-deterministic, it is able to generate a wider range of Pareto optimal solutions. The authors only consider a restricted set of refactorings, moving a single method from one class to another, and do so at the code level. Our approach focuses on refactorings at the model level rather than the code level. This allows us to consider more potential refactorings. We also make use of a multi-objective algorithm [39], which is able to present a Pareto optimal front in a single execution, and does not require several runs to determine a possible front. We compare our MOGA approach to the class assignment problem with a similar random mutation hill climber to the one presented in [19]. Our findings are discussed in Section 7.2.3.

Refactoring [16] and reengineering [13] are activities usually performed during maintenance, and driven by the need to fix the code (more recently, the need to refactor models has also been recognized [26]) when so-called “bad-smells” (e.g., a god class) have been identified (e.g., using metrics [22]). Although some refactorings [16, 26] and reengineering patterns [13] change class responsibility assignment, this is not their main objective, as they are problem-driven (e.g., by specific “bad-smells”). Instead, our approach specifically addresses the class responsibility assignment problem, without being

driven by the search for specific anti-patterns, and does so at the model level during early software life-cycle phases.

Multi-objective genetic algorithms have been applied to other areas of search based software engineering. In particular, the work in [38] demonstrates the application of a MOGA to the problem of test case selection. Although the Strength Pareto approach (SPEA2) has been recently introduced [39], several applications of the technique have already been reported (e.g., [25, 30]).

3 THE CLASS RESPONSIBILITY ASSIGNMENT PROBLEM

The class responsibility assignment (CRA) problem is encountered in the early stages of software design [23]. CRA is about deciding where responsibilities, under the form of class operations (as well as the attributes they manipulate), belong and how objects should interact (by using those operations).

Let us illustrate the CRA problem with a simple problem: the Monopoly game model [3, 23]. Suppose two alternative (excerpt) CRAs for Monopoly: operation `roll()` is assigned to `Player` (Figure 1(a)) or `Die` (Figure 1(b)); note that `roll()` uses attributes `MAX` and `faceValue`. Let us now analyze these alternative CRAs in the light of the general responsibility assignment software patterns (GRASP) [23], a collection of patterns to support object-oriented design. According to the *Information Expert* pattern, one should assign a responsibility to a class that has the information (attribute) needed to fulfill it: operation `roll()` should therefore be in class `Die` (where `MAX` and `faceValue` are located). According to the *Low Coupling* pattern, one should assign responsibilities so that coupling remains low: Figure (b) shows higher class coupling than Figure (a) since `roll()` in `Player` has to ask `Die` for attribute values (`MAX` and `faceValue`). According to the *High Cohesion* pattern, one should assign responsibilities so that cohesion remains high: Figure (b) shows lower cohesion than Figure (a) since `roll()` is functionally related to `MAX` and `faveValue` and therefore the three should be in the same class.

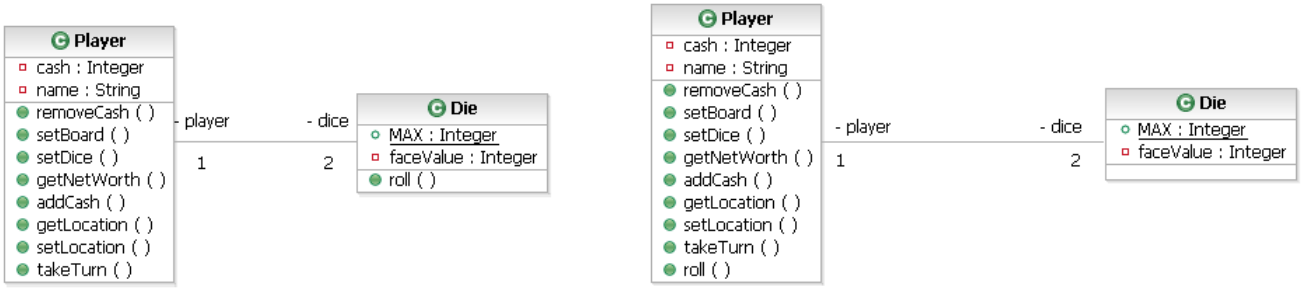


Figure 1 Two alternative CRAs for the Monopoly game

The CRA problem has a great impact on the overall design of the application [23]. However, the assignment of responsibilities to classes is a very difficult skill both to teach and to master [35]. Approaches to CRA are subjective, relying upon human decision-making, experience and judgment (e.g. [8]). As a result, the design is highly dependent on the skill of the designer. While the assignment of responsibilities to classes is identified as a key aspect of object-oriented analysis and design, there is little to aid a designer at this early stage of development.

However, we believe the CRA problem is well suited for applying a search-based technique [10] to aid designers find good solutions. In [10], the authors outline four characteristics of problems that may benefit from a search-based approach, and we explain below why the CRA problem exhibits such characteristics:

(1) It must have a very large search space. In our context, the size of the search space is the total number of possible class assignments that can be formed from the class members (methods, attributes and association ends, see Section 5). The search space consists of every valid design involving these classes, methods and attributes;

(2) There is no known efficient and complete solution. As discussed earlier, the optimal assignment of responsibility to classes is subjective and based on the designer's judgment. While guidelines exist to aid in the design process, there is no one solution that guarantees the best possible design. Similarly, in the context of the quality measures used (Section 4), there is no known efficient algorithm that can generate an optimal solution or solution set in terms of these measures;

(3) One can define suitable fitness functions. While the overall quality of a design is ultimately subjective, there are known metrics that can be used to measure various aspects design quality [37] in fitness functions;
(4) Generating candidate solutions should be inexpensive. The CRA problem can be represented as a sequence of the classes to which the class members (methods, attributes and association ends) are assigned. This sequence can then easily be altered to generate new candidate solutions.

In this paper, we focus our approach on optimizing the domain model [8] at the analysis level (using the terminology in [23]). At this point in the design process, many of the classes, attributes and methods have been defined, and dependency information is available from sequence diagrams and method contracts, e.g., defined in the Object Constraint Language (OCL) [8]. Our objective is then to provide alternative CRA solutions to be analyzed by a designer.

One problem of applying the search-based heuristic is measuring the fitness of the candidate solutions. While many possible quality measures exist that could be used [37], most are not comparable with each other, and it is difficult to balance these quality metrics into a single fitness function. Additionally, the final say in the optimum trade-off between these quality metrics should be left with the designer, not with the heuristic. In order to overcome these difficulties, we formulate the CRA problem as a multi-objective problem. The fitness function will consider a subset of quality metrics, and present the designer with a number of possible candidate solutions, each representing a different trade-off. In order to determine the quality of the candidate solutions, in this paper we will use complementary coupling and cohesion quality metrics to define multiple objectives for the search.

4 QUALITY MEASUREMENT

Our MOGA fitness function is based on five measures capturing different and complementary aspects of coupling and cohesion. These measures are presented below. We summarize the notation in Table 1.

Table 1 Notation summary

Notation	Definition	Defining section
C	the set of classes in the assessed class diagram	4.1
M(c)	the sets of methods of a class, or a set of classes	
A(c)	the sets of attributes of a class, or a set of classes	
AR(m)	the set of attributes directly accessed by method m	
SM(m)	the set of methods statically invoked by method m	
PM(m)	the set of methods dynamically invoked by method m	
LSM(m)	set of methods called by m in the same class as m	
DMA(m,a)	direct method–attribute dependency between method m and attribute a	
DMM(m_1,m_2)	direct method–method dependency between m_1 and m_2	
AE(c)	the set of association ends of a class	
AER(m)	the set of association ends directly accessed by method m	
LR(m,a)	the set of local (in-) direct access dependencies of attribute (or association end) a by method m	
MAI(c_1,c_2)	the set of method-attribute interactions between classes c_1 and c_2	4.2.1
MMI(c_1,c_2)	the set of method-method interactions between classes c_1 and c_2	
MAC(c)	method-attribute coupling between class c and other classes	
MMC(c)	method-method coupling between class c and other classes	
MGC(c)	method-generalization coupling	
CI(c)	cohesive interaction: the set of indirect method-attribute or method-association end dependencies between the methods of c and the attributes of c	4.2.2
CI_{max}(c)	the set of all possible cohesive interactions in class c	
RCI(c)	ratio of cohesive interactions of class c	
cu(m_1,m_2)	common usage: m_1, and m_2 of the same class (in)directly use an attribute (or association end) of that class, or m_1 (in)directly invokes m_2	
TCC(c)	tight class cohesion	

4.1 Basic Definitions

The information we are using to optimize the model are dependencies among methods and attributes. These dependencies need to be defined precisely as they will constitute the basis of our coupling and cohesion measurement in Section 4.2.

Let us first define our basic notation by defining a number of sets. C is the set of classes in the assessed class diagram. M() and A() refer to the sets of methods and attributes of a class, or a set of classes (e.g., M(C) and A(C) refer to all the methods and attributes in the assessed class diagram, respectively). (Note that A(c) contains attributes inherited by class c.) For a class c, M(c) is the set of newly defined and overridden methods in c. AR(m) refers to the set of attributes directly accessed (read or updated) by a method m. For the set of methods invoked by a method m, we differentiate methods that are

statically invoked from those that are *polymorphically* invoked by *m*: denoted SM(*m*) and PM(*m*), respectively, with SM(*m*) $\subseteq$ PM(*m*). A method *m'* in class *c'* is statically invoked by *m* when *m* invokes *m'* on an instance of type *c'*. In addition *m'* can be invoked on an instance of any subclass of *c'* that overrides *m'* and these invocations are referred to as polymorphic. Calls within the same class are denoted as LSM, or local SM[1]. A "*" appended to the above set names denotes *indirect* accesses, invocations, or dependencies (i.e., transitive closure).

Definition 1. Method–Attribute Dependency (DMA). A direct method–attribute dependency exists between $m \in M(C)$ and $a \in A(C)$ if $a \in AR(m)$. This is denoted *DMA(m,a)*.

Definition 2. Method–Method Dependency (DMM). A direct method–method dependency exists between $m_1 \in M(C)$ and $m_2 \in M(C)$ if $m_2 \in PM(m_1)$, and is denoted $DMM(m_1,m_2)$.

In addition to methods, attributes, and their dependencies, we need to consider three types of relationships that we expect to be part of a typical class model: association, usage dependency, and generalization [23]. Note that we do not differentiate between associations, aggregation, and composition relationship, the two latter ones being a specialization of the first[2]. Association ends will be handled like attributes, which is not surprising as both are usually implemented as references to instances. In other words, a bidirectional, binary association will translate into two attributes, one in each class at its ends. Each association end can therefore move from one class to another during the search. For reasons explained below, association ends are only accounted for in cohesion measures. We therefore need to distinguish them from attributes: AE() refers to the set of association ends of a class, or a set of classes; AER(*m*) refers to the association ends directly accessed by method *m*.

Definition 3. Local (in-) direct access (LR). An (in-) direct local access dependency exists between $m \in M(C)$ and $a \in A(C) \cup AE(C)$ if *m* and *a* are in the same class and *m* (in-) directly accesses *a* within its class. This is denoted *LR(m,a)*. More formally[3]: $(a \in \mathrm{AR} \circ \mathrm{LSM}^*(m) \vee a \in \mathrm{AER} \circ \mathrm{LSM}^*(m)) \wedge \exists c \in \mathrm{C}, m \in \mathrm{M}(c) \wedge (a \in \mathrm{A}(c) \cup \mathrm{AE}(c))$

[1] $m' \in \mathrm{LSM}(m) \Leftrightarrow m' \in \mathrm{SM}(m) \wedge \exists c \in C, m \in \mathrm{M}(c) \wedge m' \in \mathrm{M}(c)$

[2] It could be argued that a composition usually entails more coupling than associations and aggregations, and that we fail to account for that. However, we consider that the higher coupling entailed by a composition is already accounted for since it will translate into more method-method and method-attribute dependencies.

[3] Here, $\circ$ is the standard notation to denote the composition of two binary relations.

Generalization relationships will not change during the GA search but are nevertheless accounted for when we modify the class model (Section 5.1). Usage dependencies among classes are already accounted for when we consider Method-Attribute and Method-Method dependencies.

The information on which we rely can be retrieved from UML models [28], and in particular interaction diagrams and method contracts. These are typical components of an analysis or design model expressed with the UML [8]. For example, sequence diagrams tell us what methods can invoke other methods, OCL contracts suggest what attributes (and association ends) can be accessed by which methods, and class diagrams tell us about the M() and A() sets. To summarize, although the goal of our approach is to improve CRA, as modeled by a class diagram, we still need to rely on information provided by other components of a UML model.

4.2 Coupling and Cohesion Measurement

4.2.1 Coupling Measures

Many measures for cohesion and coupling have been proposed in the literature, and frameworks to support their selection exist [4, 5]. Using these frameworks we selected three coupling measures, defined at the class level though the entire class diagram coupling is computed to assess improvements. We first re-express dependencies as a set of interactions between pairs of classes. Note that we only account for methods defined in a class, i.e., M(), the ones inherited being accounted for in the context of the parent class.

Definition 4. Set of Method–Attribute Interactions (MAI). For two classes $c_1,c_2 \in C$, MAI(c_1,c_2) the set of MAIs between c_1 and c_2 is defined as $MAI(c_1,c_2) = \bigcup_{m \in M(c_1)} \{(m,a) \mid a \in A(c_2) \wedge DMA(m,a)\}$.

Definition 5. Set of Method–Method Interactions (MMI). For two classes $c_1,c_2 \in C$, the set of MMIs between c_1 and c_2 is defined as $MMI(c1,c2) = \bigcup_{m \in M(c1)} \bigcup_{m' \in M(c2)} \{(m,m') \mid DMM(m,m')\}$.

The two coupling measures below are based on the summation of the interactions between classes, which are not in the same generalization hierarchy, i.e., the two classes do not have any common ancestor (denoted *Others(c)* for any class *c*).

Definition 6. Method–Attribute Coupling (MAC). For a given class $c_1 \in C$, $MAC(c_1)$ counts all MAI from c_1 to classes that do not have a common ancestor with c_1: $MAC(c_1) = \sum_{c_2 \in Others(c_1)} |MAI(c_1, c_2)|$.

Definition 7. Method–Method Coupling (MMC). For a given class $c_1 \in C$, $MMC(c_1)$ counts all MMI from c_1 to classes that do not have a common ancestor with c_1: $MMC(c_1) = \sum_{c_2 \in Others(c_1)} |MMI(c_1, c_2)|$.

When method-method and method-attribute interactions occur within a generalization hierarchy, we define a specific coupling measure to account for coupling between siblings and coupling between ancestors and descendants (but not between descendants and ancestors since ancestors' fields are inherited, and such interactions therefore pertain to cohesion rather than coupling). These classes are denoted *OthersGen(c)* for any class *c*.

Definition 8. Method–Generalization Coupling (MGC). For a given class $c_1 \in C$, $MGC(c_1)$ counts all MMI and MAI from c_1 to classes that are in the same generalization as c_1 but are not ancestors of c_1: $MGC(c_1) = \sum_{c_2 \in OthersGen(c_1)} |MMI(c_1, c_2)| + |MAI(c_1, c_2)|$.

Model level coupling measures are obtained by summing up the class level measurements for all the classes in the model.

In [5] the authors present five mathematical properties that all valid coupling measures must adhere to: non-negativity, null value, monotonicity (adding relationships to a class increases its coupling), merging of classes (combining two existing classes cannot increase the coupling of the classes or the system), and merging of unconnected classes (the union of two unconnected classes does not change the total coupling of the system) [5]. MAC, MMC and MGC are based on a suite of measures that adhere to these properties [6] and therefore adhere to them too.

Using these coupling measures on the alternative CRAs of Figure 1 (dependencies are available in [2]), we obtain (subscripts indicate CRA alternatives of Figure 1(a) and Figure 1(b): $MAC(Die_a)=MAC(Die_b)=0$; $MAC(Player_a)=1$; $MAC(Player_b)=3$; $MMC(Die_a)=MMC(Die_b)=0$; $MMC(Player_a)=1$, and $MMC(Player_b)=0$ (MGC does not apply). These confirm our discussion in Section 3 regarding the fact that Figure 1(b) shows higher coupling than Figure 1(a).

4.2.2 Cohesion Measures

Similarly to our coupling measures, we measure cohesion at the class level, considering the method-based dependencies defined above (DMA, DMM, and LR). Though it is not realistic to expect every class member to be directly related to another, by considering indirect dependencies between class members, our assumption is that, within a class, each class member must depend on all of the other class members directly or indirectly through other members to achieve perfect cohesion.

There are two aspects related to inheritance that should be taken into consideration in the analysis of cohesion. Within an inheritance hierarchy, each child class is representing a specialized aspect of a given domain concept. The classes in the hierarchy represent a single abstraction, at various levels of specialization. Then, to assess how cohesive a class is, both the methods and attributes that are locally defined or inherited within that class must be considered. Furthermore, since it is not possible to polymorphically invoke a method or attribute of the same class, there is no need to consider polymorphic dependencies to measure the cohesion of a class. Second, we should determine how to handle accessor methods and constructors. Accessor methods can cause problems for measures which count references to attributes [4]. The reason is that accessor methods can artificially lower the cohesion value by hiding a method's access to an attribute. However, because we consider indirect dependencies, the use of accessor methods does not hide, in our specific context, the attribute reference. Constructors are not considered since the analysis and early design models typically do not list them.

The first measure we consider is based on the concept of cohesive interactions. A cohesive interaction is defined as a (in)direct method-attribute or method-association end dependency in a class (LR) as it is considered to contribute to the cohesion of that class. This measure is normalized, as all cohesion measures should be [4], and is computed as the percentage of cohesive interactions in a class relative to all possible cohesive interactions in the same class.

Definition 9. Cohesive Interaction (CI). For a given class $c \in C$, $CI(c)$ is the set of all indirect method-attribute or method-association end dependencies between the methods $m \in M(c)$ and the attributes $a \in A(c)$ or association-ends $a \in AE(c)$. $CI(c) = \bigcup_{m \in M(c)} \{(m,a) \mid a \in A(c) \cup AE(c) \wedge LR(m,a)\}$.

Assuming $CI_{max}(c)$ is the set of all possible cohesive interactions in the class, accounting for (in)direct method–attribute and method-association end dependencies, we define the ratio of cohesive interactions.

Definition 10. Ratio of Cohesive Interactions (RCI). $RCI(c)$ for class $c \in C$ is the number of cohesive interactions in c, over the number of possible such interactions: $RCI(c) = |CI(c)| / |CI_{max}(c)|$.

Note that when no method (or no attribute) is present in a class, we set its RCI measure to 0. This is to penalize data container classes (i.e., classes with only attributes) and service classes (i.e., classes with only methods). In order to compute the cohesion value across the entire class diagram the RCI values for all the classes in C are averaged.

We also use a complementary measure, tight class cohesion (TCC) [4], which is based on the concept of common attribute (or association-end) usage. The idea is that methods which use common attributes (association-ends) should be together in the same class, and represent a single abstraction. We extended this notion to also include methods that invoke one another, in order to account for classes without attributes, and refer to this as common usage. Common usage occurs when two methods of a class (in)directly use a common attribute (or association end) of that class, or when one method (in)directly invokes the other.

Definition 11. Common usage (cu). The predicate $cu(m_1,m_2)$ is true if $m_1,m_2 \in M(c)$ (in)directly use an attribute or association end of class c, or if m_1 (in)directly invokes m_2:

$$cu(m1, m2) \Leftrightarrow \begin{cases} \bigcup_{m \in m_1 \cup LSIM^*(m_1)} LR(m) \cap \bigcup_{m \in m_2 \cup LSIM^*(m_2)} LR(m) \cap (A(c) \cup AE(c)) \neq \varnothing \\ or \quad m_2 \in LSM^*(m_1) \end{cases}$$

The TCC metric is then defined as the percentage of pairs of methods of a class with common usage.

Definition 12. Tight Class Cohesion (TCC). TCC(c) is the pairs of methods, m_1 *and* m_2, of a class $c \in C$ with common usage: (It is normalized.) $TCC(c) = 2 \frac{\left|\left\{(m_1, m_2) \mid m_1, m_2 \in M(c) \wedge m_1 \neq m_2 \wedge cu(m_1, m_2)\right\}\right|}{|M(c)|\,(|M(c)|-1)}$.

When a class contains fewer than two methods, TCC is undefined. As for RCI, to measure TCC across a class diagram, the TCC class values are averaged over all of the classes in the diagram.

In [4], the authors present four properties that all valid cohesion measures must adhere to: Non-negativity and normalization, null value and maximum value, monotonicity (adding relationships to a system must not decrease cohesion) and merging of unrelated classes (combining two unrelated classes must not increase cohesion). TCC and RCI comply with these properties [4].

Using these cohesion measures on the alternative CRAs of Figure 1 (dependencies are available in [2]), we obtain (subscripts indicate CRA alternatives of Figure 1(a) and Figure 1(b): $RCI(Die_a)=1$; $RCI(Die_b)=0$; $RCI(Player_a)=0.18$; $RCI(Player_b)=0.16$ (TCC is only defined for Player). These confirm our discussion in Section 3 regarding the fact that Figure 1(b) shows lower cohesion than Figure 1(a).

5 CHANGE MODEL

The goal of our GA is to optimize an analysis or domain model by finding an optimal assignment of methods and attributes to classes. To perform a search for such an assignment, it is necessary to define the search space by determining possible changes to the model. (See [2] for examples.)

5.1 Changes to methods, attributes, association ends, classes and relationships

Since the search is based on method-method, method-attribute, and method-association end dependencies, this information cannot, at this stage of our research, be subject to change. The change model also does not include the addition and removal of methods, attributes, or association-ends. The main mechanism for our search of better domain models is therefore to move methods, attributes, and association ends from one class to another, thus affecting the measures of coupling and cohesion. Note that, once method-method, method-attribute, and method-association end dependencies have been identified (e.g., from UML documents), only the ownership of methods and attributes matters and we do not need to know about attribute types and method signatures.

Since our goal is to determine the optimal assignment of methods, attributes, and association ends to classes, we have to acknowledge that this may result in some new classes being added or existing classes being removed. More specifically, classes should be removed from the model when they are empty, as they serve no purpose any longer. The addition of classes is a necessity since optimal class assignments may require additional classes that were not identified in the first place. Our strategy is,

when a method, an attribute, or an association end is moved, to allow a move to a new class. Note that finding a meaningful name for every created class will be the responsibility of the designer who, in the end, is presented with the GA solution(s).

Though association ends are handled like attributes and usage dependencies are already accounted for through method-method and method-attribute dependencies, generalization relationships are treated differently in the change model. Let us first consider moving any method that belongs to a generalization hierarchy, including abstract methods. This would have an important impact on the dependencies we have to maintain. Client methods invoking a moved abstract method would then have to invoke concrete implementations of the abstract method in child classes. Additionally, the class receiving the moved method should then either provide an implementation of the method (which would then become concrete) or provide concrete implementations of the abstract method in its own (existing or to be created) child classes. Alternatively, we could create a new generalization hierarchy that would receive the abstract method and all its concrete implementations in child classes.

At this stage of the research, we consider these changes to the class diagram too complex and we are not sure of their impact on the search. A simplifying assumption, which will be addressed in future work, is that we limit modifications to generalization hierarchies to attributes, association ends, and concrete methods that are not overridden. Other class elements in hierarchies cannot be moved during the search.

5.2 Constraints

Though allowed changes to a domain model have been described above, such changes must comply with a number of constraints. First, methods that simply use the "reference" corresponding to an association end (e.g., adding, or removing an element to the collection represented by the association end) are conceptually related. They are grouped together with the association end and can only be moved together. Second, classes cannot be empty (an empty class is simply removed). Third, classes should not be stand alone classes: they must be involved in at least one dependency, either as a client or server, or be a child class.

We also allow the user to specify that certain methods and attributes are conceptually related (though not necessarily dependent on each other) and can only be moved together. This allows the user to

identify parts of the model that are satisfactory and should not undergo change, thus limiting the search space for new solutions.

6 MULTI-OBJECTIVE GA (MOGA)

This section presents the basic principle of multi-objective genetic algorithms, the specific technique we use (SPEA2), and how it was parameterized. A discussion of its implementation can be found in [2].

6.1 Basic Principles

We assume here that the reader is familiar with the basic notions of genetic algorithms (population, operators, fitness function). The objective of our search is to optimize the coupling and cohesion of a given class diagram based on five distinct measures (Section 4.2). However, to address those objectives at the same time, it may be necessary to consider tradeoffs between them to find the best model. This type of problem is referred to as a multi-objective problem (MOP) [36].

MOPs are mathematically defined as follows [36]: A solution is characterized by a vector $\vec{u} = (u_1,...,u_k)$ of fitness function values and is said to dominate another vector $\vec{v} = (v_1,...,v_k)$ (denoted by $\vec{u} \prec \vec{v}$) if and only if u is partially less than v: $\forall i \in \{1,...,k\}, u_i \leq v_i \wedge \exists i \in \{1,...,k\} : u_i < v_i$. In our context, a solution is a class assignment and fitness functions are coupling and cohesion measures.

The objectives being optimized will often conflict, which places a partial ordering on the search space. This makes the problem of finding a single globally optimum solution in a MOP an NP-Complete problem [36]. Genetic algorithms are well suited to the task of solving MOPs, as they rely not on a single solution but rather a population of solutions. Thus, different individuals in the population can represent solutions that are close to an optimum and represent different tradeoffs among the various objectives.

One key concept related to MOPs is *Pareto optimality*.

Definition 13. Pareto Dominance. A vector $\vec{u} = (u_1,...,u_k)$ is said to dominate $\vec{v} = (v_1,...,v_k)$ (denoted by $\vec{u} \prec \vec{v}$) if and only if u is partially less than v, i.e., $\forall i \in \{1,...,k\}, u_i \leq v_i \wedge \exists i \in \{1,...,k\} : u_i < v_i$.

Definition 14. Pareto Optimal Set. For a given MOP, i.e., a set of fitness functions $F(\vec{x})$, the Pareto optimal set (*PS*) is $PS = \{\vec{x} \in \Omega \mid \neg\exists \vec{x}' \in \Omega : F(\vec{x}') \prec F(\vec{x})\}$, where Ω is the search space.

Definition 15. Pareto (Optimal) Front. For a given MOP $F(\vec{x})$ and Pareto optimal set P^*, the Pareto optimal front (*PF*) is defined as: $PF = \{\vec{u} = F(\vec{x}) \mid \vec{x} \in PS\}$.

In other words, the Pareto (optimal) front refers to optimal solutions whose corresponding vectors are non-dominated by any other solution vector. Because a range of individual solutions are considered in our GA search, rather than a single solution, it is possible to find many points in the Pareto optimal set, and thus present the many possible tradeoffs between the various objectives. The final decision is left with a decision maker, rather than the optimization algorithm, with respect to which solutions to select from the Pareto front.

Another key concept related to MOPs is *range independence*.

Definition 16. Effective range: The effective range of $F(\vec{x})$ is the range determined by min($F(\vec{x})$) and max($F(\vec{x})$) for all values of $\vec{x}$ that are actually generated by the algorithm, and for no other values of $\vec{x}$.

Definition 17. Range independence. Given a MOP $F(\vec{x})$ and a set of solution vectors $\{\vec{u}_1, \vec{u}_2, ..., \vec{u}_n\}$, a multi objective ranking method is range-independent if the fitness ranking of $\{\vec{u}_1, \vec{u}_2, ..., \vec{u}_n\}$ defined by the method does not change when the effective range of $F(\vec{x})$ changes.

One way to use several fitness functions, as we have seen in the Related Work section, is to sum their values. This kind of approach must rely on weights to adjust the objectives, which should be normalized, to make the fitness value meaningful. These weights are typically subjective and problem dependent, making the summation based approach not always easy to apply. On the other hand, approaches that evaluate the fitness as a vector of individual objective values using Pareto dominance do not need normalization (they are range independent) nor weights since the objectives are compared on an individual basis. However, the comparison process of the individuals tends to be more complex and the search tends not to converge on single solutions, but rather a range of Pareto optimal solutions.

In this paper, rather than normalizing or setting weights for our cohesion and coupling measures, we choose to adopt a range independent, vector-based approach.

6.2 The Strength Pareto Approach

There are many vector-based approaches in literature. VEGA [31] was the first one proposed but its performance has been shown to be suboptimal compared to more recent alternatives such as NSGA [34]. NSGA itself has been shown to be outperformed by SPEA [40]. More recently, improvements to both NSGA and SPEA have been proposed: NSGA-II [12] and SPEA2 [39], which have been shown to produce similar results on some MOPs [39]. The time complexity of NSGA-II is better than SPEA2 — SPEA2 is $O(N^3)$ [39], whereas NSGA-II is $O(N^2)$ [12] —but SPEA2 has advantages over NSGA-II in higher dimensional objective spaces (with four objectives and more) [39]. This is why we selected this technique in the case study presented below. SPEA2 has, however, the highest worst-case time complexity of all techniques [39] and its scalability will therefore need to be investigated in the context of our problem.

The overall algorithm for SPEA2 is very similar to a single objective GA, with a number of variations. First, a population P and an archive $\overline{P}$ evolve (instead of simply one population in a simple GA). During fitness assignment, individuals in both P_t and $\overline{P}_t$, i.e., sets of individuals produced at iteration t, are evaluated (each evaluation results in a vector). Note that SPEA2 accounts for high density areas in the search space and penalizes the individuals that lie in those areas (see [39] for details). During selection, the non-dominated individuals of P_t and $\overline{P}_t$ (i.e., the interesting tradeoffs) are copied into archive $\overline{P}_{t+1}$. SPEA2 has a clustering mechanism that removes some non-dominated solutions (that are too close to others) and therefore maintains the size of the archive constant while preserving diversity in the archive (see [39] for details). Next, the genetic selection operator is applied on the individuals of $\overline{P}_{t+1}$ to fill a mating pool, and recombination and mutation operators are applied to the mating pool to generate P_{t+1}. This continues until a number of generations has been reached.

6.3 Parameters of the MOGA

We have seen that valid changes include moving attributes and methods between classes, as well as adding and removing classes. The chromosome representation must therefore track the attributes and

methods, and the class to which they belong. To do so, each method or attribute in the assessed class diagram is assigned a gene within the chromosome. The length of the chromosome therefore equals the number of class members. Dependency information is represented by a dependency matrix, stored separately since the dependency information does not change during the search, and is used only for computing cohesion and coupling measures. Chromosomes simply consist of integer values, where the position of the gene within the chromosome represents the class member, and the gene's integer value denotes their class assignment: e.g., *(10, 12, …)* denotes that the 1st class member (i.e., the first gene) belongs to the 10th class and the 2nd class member (i.e., the second gene) to the 12th class. Using this representation, it is impossible to have an empty class represented in the chromosome.

A number of parameters of the MOGA have to be set, and they can be set for different purposes. We set our parameters to optimize the quality of the final solutions, rather than to increase the rate of convergence for example.

Determining the ideal population size for a GA is challenging but important [20]: e.g., population size has been found to be one of the main parameters affecting multi-objective (i.e., two objectives) genetic programming [9]. For traditional GAs a variety of adequate population sizes have been suggested [20]: some recommend a range between 30 and 80 [18], while others suggest a smaller population size, around 20 and 30 [31]. For MOGA, authors tend to use larger population sizes than those recommended for single objective GAs (reported values range from 30 to 80), and they also increase the population size proportional to the number of objectives [24, 39]. We follow their suggestions and use a population size of 64 individuals per objective. The effect of the population size on the results of the MOGA is investigated in the third part of our case study, in Section 7.

The archive size also has an important effect on performance. In [24] the authors examine the effect of elitism on the performance of the GA: Elitism is the extent to which the best individuals are not only stored permanently, but also take part in the selection of offspring. They report that strong elitism together with a high mutation rate should be used to achieve best performance. Elitism relates, among other factors, to the size of the archive: If the size of the archive is large, and then filled by individuals from the population, the best individuals have less chances of being part of the selected offspring. No systematic study of the impact of the archive size can currently be found in the literature. Authors

however report on archive sizes in the range of [¼, 4] of the population size [25, 30, 39]. To keep computation time within reasonable bounds, we set the archive size to half the size of the population.

When using an elitism algorithm such as SPEA2 a high mutation rate is desirable to achieve optimum performance [24]. The authors suggest the use of mutation rate based on the length of the chromosome to achieve an average of approximately five mutations per chromosome, or 5 / (*length*) where *length* is the length of the chromosome. These findings are consistent with [33]: mutation rates based on the chromosome length perform significantly better than those that are not. Based on these findings, we first used a mutation rate of 5 / *length*. But, after performing our initial experiment and examining the output, we concluded that this mutation rate was too high and led to unstable results. We obtained much better results with a significantly lower rate of 1 / *length*, which is more suitable for the CRA problem.

The crossover rate also affects performance. A crossover rate that is too high will not allow desirable genes to accumulate within a single chromosome whereas if the rate is too low, then the search space will not be fully explored [20]. De Jong [11] concluded that a desirable crossover rate for a traditional GA should be about 60%. Grefenstette et al. [18] built on De Jong's work and found that the crossover rate should range between 45% and 95%. Consistent with these findings, we used a crossover rate of 70%.

In terms of selection, SPEA2 uses a binary tournament [17] where the fitter individual is selected 90% of the time, and 10% of the time the other individual is chosen. We use a 1-point crossover operator [17] as this is simple and used in SPEA2 [39]. Mutation occurs on a gene by gene basis. Our mutation operator randomly changes the class that a given attribute or method is assigned to. When it is determined that a gene's field should be moved into a different gene (class), it is reassigned to a different class randomly, assuming equal probabilities for all classes and an equal probability to assign it to a newly created or existing class.

7 CASE STUDY

In order to validate our approach, we performed a case study. The goal was threefold: (1) Validate that the approach was an efficient means of identifying meaningful trade-offs for improving a class

responsibility assignment (CRA); (2) Demonstrate that the recommended multi-objective approach brings significant benefits compared to simpler alternatives; and (3) Test the impact of population size on the search effectiveness, as this is one of the most important MOGA parameters for which there is little guidance in the literature (Section 6.3).

To achieve these goals, we broke our case study into three parts. The first part involved making several suboptimal modifications to an existing class diagram with proper CRA (Section 7.1). The goal is to determine if the changes made by the CRA optimization algorithm are meaningful, and are capable of returning to the original, optimal CRA (Goal 1). The second part of the case study compares the selected SPEA2 algorithm to a selected subset of simpler search algorithms in order to assess the benefits of using a MOGA (Section 7.2): random search, random hill climbing, and single objective GA (Goal 2). Finally, the third part of the case study involves varying population size to determine its effect on the identification of useful CRA solutions and their cohesion and coupling metric values (Section 7.3). All algorithms were run on an Intel Core 2 Quad Core 2.4GHz processor with 4GB of RAM.

7.1 Assessing convergence towards optimal CRA

This section first presents the design of the study and then presents the results for different cases of CRA improvements.

7.1.1 Study design

7.1.1.1 Assessing CRA improvements

To assess convergence towards optimal CRA, our strategy is to start with a well-designed class diagram, coming from an authoritative and independent source. A variety of carefully selected sub-optimal changes are then introduced into the class diagram. The MOGA algorithm then takes these sub-optimal CRAs, and attempts to improve upon them. The goal is to assess whether the algorithm converges on the original CRA, which is considered a reference in terms of good design. Our target CRA is thus determined independently from this research though the type of suboptimal changes must be carefully justified in terms of representative categories of changes. Four representative and suboptimal CRA changes were selected for the study. They are discussed in the following sub-sections.

7.1.1.2 Subject selection

For this case study, the ARENA system was used [7, 8] as it is designed independently from our research and its analysis level class diagram and sequence diagrams are available. Furthermore, it was designed by experts and can be therefore considered to be a quality analysis model, a reference model towards which we want to converge. The ARENA case study is a framework for building multi-user, web-based systems to organize and conduct tournaments (e.g., a Tic Tac Toe tournament). The analysis level class diagram of the ARENA system that is used in the case study is shown in Appendix A, and details on model element dependencies (e.g., method-attribute dependencies—recall Section 4.1) are available in [2].

Although of modest size, the domain model is not trivial[4] and consists of 14 classes and 138 class members (methods, attributes and association ends). Of these 138 class members, 49 of them cannot be altered by the GA as they are overridden or implemented through generalization relationships (Section 5.2). Another 42 class members are formed into 14 groups with the association ends that they manipulate (Section 5.2). The search space, assuming the number of classes does not change, is therefore all the possible assignments of movable (grouped) class members to 14 classes (i.e., 138-49-42+14=61) and its size is therefore considerably large: 14^{61}. An analysis of this domain model shows a total of 287 dependencies, specifically 63 method-attribute dependencies, 59 method-association end dependencies, and 165 method-method dependencies. The coupling and cohesion values for the ARENA model are: 99.0 (MMC), 5.0 (MAC), 0.0 (MGC), 0.20 (RCI), and 0.24 (TCC).

7.1.1.3 Filtering proposed CRA solutions

The large number of solutions returned by SPEA2 presents a practical challenge when analyzed by a designer. The number of returned solutions is therefore trimmed using a range of coupling and cohesion measures specified by the designer in order to prune solutions that are outside an acceptable range. For the purposes of this case study, only solutions that had values of at most 15 for MAC, 105 for MMC, 0 for MGC, and at least 0.18 for RCI and 0.2 for TCC were retained for evaluation. These values were chosen since they are close to the original ARENA model metric values, but still allow for

[4] The analysis model was completed by adding class members for a number of use cases that were not initially considered, e.g., we moved from 112 class members to 138 class members.

some lower quality solutions to be included with respect to some of the metrics. These solutions are referred to as *within-range* solutions (Definition 18).

Definition 18. Within-Range Solution. A "*within-range*" solution is a CRA, generated by the search, which meets or exceeds a threshold set for each of the five objectives. These thresholds are subjective, chosen by the designer to represent the minimum acceptable trade-offs between the objectives. In practice, they can be determined by not allowing any of the coupling and cohesion measures to get much worse than in the CRA to improve. These thresholds are used to filter the results of the search down to a number of reasonable solutions that can be examined by the designer.

7.1.1.4 Identifying equivalent CRA solutions

While the concept of within-range solutions does limit the number of solutions returned to the designer, it does not necessarily indicate that the solution represents a quality CRA. To ensure that the algorithm can produce meaningful CRAs, the within-range solutions are manually examined and compared to the original CRA. If the new solution fixes the changes made to the original ARENA CRA, and does not contain any other design flaws, then that solution is said to be an *equivalent solution* (Definition 19). In other words, the solution does not have to exactly match the original CRA, but should be of comparable quality, something that we inevitably have to check manually.

Definition 19. Equivalent Solution. An "equivalent" solution is a CRA, generated by the search, that is "within-range" and meets the following criteria: (a) The moved class members are returned to their original class (or an equivalent, child class); (b) Differences between the original CRA and the solution CRA are not considered practically significant. This means that, though the solutions are not identical, differences do not impact the quality of the design, based on object-oriented design principles [8, 23] (we mentioned a few in Section 3). (An example of an equivalent CRA produced by the search for the ARENA system is discussed in [2].)

7.1.1.5 Data collection

For each CRA change, our SPEA2 algorithm was run for a total of 200 generations. Its parameters were set as discussed in Section 6.3. To account for the non-deterministic nature of genetic algorithms,

SPEA2 was run 10 times[5] for each of the four modified designs and the following data was collected: number of within-range and equivalent CRA solutions, model-level cohesion and coupling measurements for all within-range solutions. In practice, we can expect that designers would also run SPEA2 several times and select the run yielding the largest number of within-range solutions, thus increasing the chances of finding equivalent solutions.

7.1.1.6 Data analysis

The analysis will investigate how many generations are necessary, on average, for SPEA2 to fill the archive and identify a stable set of within-range solutions. In addition it will report, for the best run, on the extent to which model-level cohesion and coupling measurements improve over the generations across within-range solutions. This would be expected if our fitness function effectively guides the search. The best run in this case is the one that returns the largest number of within-range and equivalent solutions. The statistical significance ($\alpha = 0.05$) of differences between initial model cohesion and coupling values, and the resulting metric distributions for within-range solutions, will be tested using the non-parametric, Wilcoxon Rank-Sum Test [14], as metric distributions may significantly depart from normality. Distributions of model-level cohesion and coupling measures will therefore be compared between the initial model and subsequent models obtained after 100 and 200 generations, and the statistical significance of their difference tested as described above. The goal of this analysis is to determine the extent to which SPEA2 helps improve cohesion and coupling. Furthermore, the number of within-range and equivalent solutions will be analyzed across the 10 runs in order to determine the number of generations needed to achieve the highest possible number of within-range and equivalent solutions. Standard Analysis of Variance and its non-parametric equivalent (Wilcoxon Rank-Sum Test) will be used to compare solution frequencies across generation levels to determine whether differences are statistically significant ($\alpha = 0.05$) [14]. We will also rely on the Bonferroni adjustments [14] to inform the reader about whether taking into account repeated testing with such adjustments would lead to different conclusions. Note, however, that such adjustments must be interpreted with care and are only briefly discussed in our results as an additional piece of

[5] This was the maximum number of runs we could consider given the fact we had, during our experiments, to manually verify each within-range CRA solution to determine whether it was equivalent to the original one.

information, as their use is controversial[6]. Due to space constraints, we will not report the result of every single statistical test, but provide only the strict minimum information necessary for the reader to form an opinion.

7.1.2 First Representative Change

The first change to the ARENA system involved moving three methods, without their supporting attributes, into a new class (see Appendix A). The negative effect of this change is twofold, it lowers cohesion by introducing a new class, and it raises the coupling by moving the methods away from the attributes they use.

Table 2 shows the model-level coupling and cohesion measurements of all within-range solutions of the best run of the algorithm at 50, 100, 150 and 200 generations. The table includes the time it took (in minutes) to reach that generation (e.g., 4 minutes for generation 50), and the best, worst and average model level measurements across within-range solutions. Values for MGC are not shown as within-range solutions must have a value of 0 for this measure. Table 2 shows that coupling and cohesion improve significantly up to 100 generations but the improvement tends to level off after that (e.g., for the best solution) thus suggesting that 100 generations are enough from a practical standpoint. When performing a non-parametric, Wilcoxon Rank-Sum Test to compare metric distributions across within-range solutions after 100 generations to the initial model values, the improvement (i.e., decrease in coupling and increase in cohesion) is significant (p-value < 0.0001) for all metrics but MAC and a Bonferroni adjustment makes no difference. Figure 2 shows the average archive size and number of within-range solutions returned per generation over the 10 runs. We can see that the archive fills up pretty quickly (around 20 generations). The number of within-range solutions also reaches a plateau around the same time and remains below 20.

[6] Perneger [29], in his well-known paper, notes that " … Bonferroni adjustments are, at best, unnecessary and, at worst, deleterious to sound statistical inference."

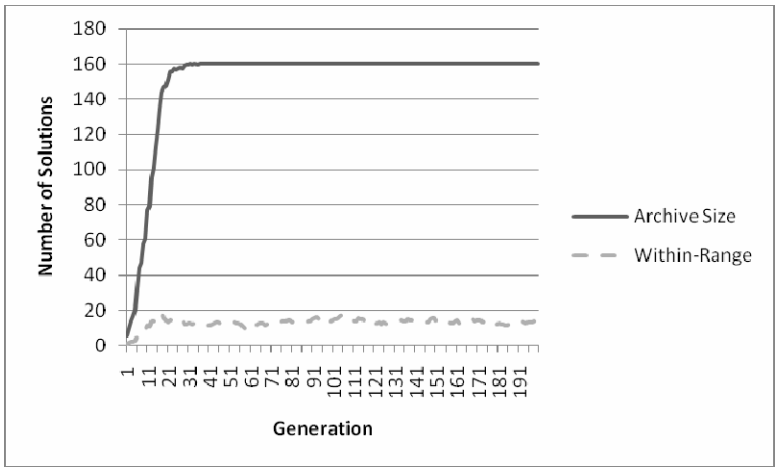


Figure 2 Average Within-range Solutions by Generation for Change #1

	Initial Model				**50 Generations** (4 min)				**100 Generations** (9 min)				**150 Generations** (14 min)				**200 Generations** (19 min)			
	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC
Avg.	11.0	105.0	0.17	0.22	11.5	97.59	0.46	0.47	10.44	93.56	0.43	0.46	10.04	96.68	0.48	0.49	10.75	97.81	0.46	0.50
Best					7	90	0.53	0.59	5	88	0.55	0.59	5	88	0.58	0.60	5	89	0.55	0.62
Worst					15	105	0.34	0.36	15	105	0.30	0.33	14	104	0.34	0.38	15	104	0.30	0.39

Table 2 Summary of Change #1 Results for the Best Run

If we focus on the results at 100 generations, we see that we obtain very good values for all measures, values that are even better than those of the original class diagram for three of the measures (MMC, RCI, TCC). Here, the best solutions are not the original one but equivalent ones. For instance, in the original ARENA analysis (Appendix A), class `Tournament` is associated to class `League`, which is itself associated to `TournamentStyle`, and `Tournament` has to navigate through `League` to access the style of the tournament. An equivalent (a designer could say better) solution that the GA returns is to have class `Tournament` associated to `League` as well as `TournamentStyle` but no association between `League` and `TournamentStyle`. This reduces method-method coupling between `Tournament` and `TournamentStyle` (while maintaining the coupling between `Tournament` and `League`), and increases the cohesion of `Tournament`. This illustrates that our GA can provide good, interesting alternatives to the CRA problem.

7.1.3 Second Representative Change

For the second change, attributes of one class are moved to a closely related class (see Appendix A). The effect on the metrics is not as profound as it is in the first change. Our intent is to try to mask the

change from the algorithm, and determine if the heuristic can still return the model to the original CRA.

Table 3 shows the results of the algorithm at 50, 100, 150 and 200 generations for the best run, showing the best, worst and average metric values for the within-range solutions as before (MGC not shown). As for the previous change, most of the improvements are obtained before 100 generations and the coupling and cohesion measures are better than the original values for three of the measures. Like for Change 1, the differences are statistically significant (p-value < 0.0001, the Bonferroni adjustment makes no difference) for all measures but MAC. We observe the same CRA alternatives as those discovered by the GA for Change 1. Since the number of within-range solutions exhibited a pattern similar to that in Change 1 [2], no further details are provided.

	Initial Model				**50 Generations** (4 min)				**100 Generations** (9 min)				**150 Generations** (14 min)				**200 Generations** (19 min)			
	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC
Avg.	10.0	99	0.18	0.24	8.05	98.55	0.45	0.49	10.88	95.88	0.46	0.48	7.92	97.69	0.47	0.49	10.64	95.35	0.45	0.46
Best					5	89	0.50	0.54	6	90	0.55	0.59	5	89	0.56	0.56	5	89	0.57	0.58
Worst					14	105	0.37	0.42	15	105	0.37	0.39	13	103	0.33	0.38	14	101	0.36	0.40

Table 3 Summary of Change #2 Results for the Best Run

7.1.4 Third Representative Change

This third change involved moving an association end and its grouped class members from a single class into a newly created class (Appendix A). The grouped class members are closely related, so the newly added class is perfectly cohesive, with dependencies between its members. This is expected, similarly to the second change, to mask the change from the algorithm.

Table 4 shows the results from the best run for the third change, in the same format as the previous two changes (MGC not shown). The statistical analysis results and conclusions we can draw are very similar to what we observed for the two previous changes and we therefore do not provide further details.

	Initial Model				50 Generations (4 min)				100 Generations (9 min)				150 Generations (14 min)				200 Generations (19 min)			
	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC
Avg.	8.0	104	0.25	0.30	6.91	93.04	0.39	0.44	9.44	95.11	0.47	0.47	8.21	96.34	0.48	0.50	9.28	96.55	0.48	0.49
Best					5	89	0.49	0.58	5	88	0.59	0.56	5	89	0.58	0.59	5	90	0.57	0.58
Worst					11	105	0.33	0.35	15	105	0.35	0.35	14	105	0.39	0.38	14	104	0.35	0.35

Table 4 Summary of Change #3 Results for the Best Run

7.1.5 Fourth Representative Change

The fourth change is not a new change in its own right, but rather the combination of the previous three changes. All of the modifications made to the design in the previous three changes were placed into the model for Change 4 (Appendix A). The goal here is to determine how well the search performs when faced with a large number of sub-optimal changes in a CRA. Table 5 summarizes the results of the search for the best run, in the same format as the previous three changes (MGC not shown), and Figure 3 shows the average archive size and the average number of within-range solutions.

	Initial Model				50 Generations (4 min)				100 Generations (9 min)				150 Generations (14 min)				200 Generations (19 min)			
	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC
Avg.	19.0	110	0.21	0.27	9.14	101.1	0.43	0.47	7.63	97.81	0.48	0.53	9.6	97.8	0.49	0.56	9.16	100.6	0.52	0.61
Best					6	98	0.46	0.49	5	93	0.56	0.60	6	92	0.57	0.64	6	94	0.55	0.65
Worst					13	105	0.40	0.45	11	105	0.38	0.36	15	104	0.40	0.49	14	104	0.44	0.54

Table 5 Summary of Change #4 Results for the Best Run

The conclusions that we can draw are similar to those for the other three changes. One difference though is that, as expected, it takes on average more generations (approximately 150 versus 20) than for the first three changes for the GA to converge towards a stable plateau of within-range solutions in the archive. However, roughly the same within-range solutions are eventually returned to the user. This time, following the same procedure, the differences are statistically significant for all four metrics, including MAC, which starts at a much higher value (19) than for the three previous changes. From Table 5, we can see that though coupling stops improving after 100 generations, cohesion still does increase. This is also different from the previous three changes and might explain why the number of within-range solutions keeps increasing. In practice, we should probably expect to run the GA for more than 150 generations as several CRA mistakes are likely to be present in the initial class diagram.

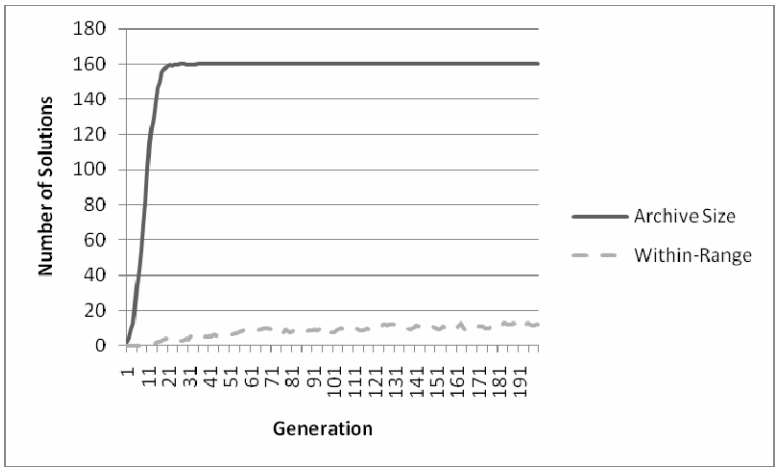


Figure 3 Average Within-range Solutions by Generation for Change #4

7.1.6 Discussion

The same pattern emerged for the four changes regarding the number of within-range solutions returned per generation, as shown in Figure 2 and Figure 3. The reason for the steady number of within-range solutions in the archive is the truncation operation used in the SPEA2 algorithm. As more non-dominated solutions are found, these solutions are added to the archive. Once the archive is full, then the non-dominated solutions begin to be truncated out. The truncation operation favors boundary solutions, so it works to maintain a wide spread of solutions across the search space. Since there is a limited number of within-range solutions, and these solutions represent a small part of the overall search space, the clustering operation ensures that the number of solutions within this limited area is maintained once the archive is full, in order to maintain as diverse an archive as possible, resulting in a steady number of within-range solutions.

Let us now delve into the details of the data. Table 6 shows, for each change and over ten runs, the average number of solutions in the archive that are within-range and the average percentage of those solutions that are equivalent to the original solution. It highlights a number of interesting facts. First, the number of within-range solutions is small, suggesting that in practice the designer could afford to spend some time looking at them all and decide which ones are interesting by taking additional considerations into account, e.g., what changes are expected in the future, which classes will be assigned to different programmers. The differences between the number of within-range solutions across generation levels is only statistically significant for Change 4. As expected from the results in

the previous section, it takes more generations for Change 4 to converge towards a maximum number of within-range and equivalent solutions. Because of this small number of within-range solutions, it is not surprising to observe significant fluctuations in terms of percentage of equivalent solutions (e.g., the percentages for Change 1 across the four generations reported are: 4.7, 14, 15, and 11). Those fluctuations are due to the clustering (Section 6.2) that removes some non-dominated solutions, which may include equivalent solutions, in order to keep the archive at a specific size. Overall, the differences in number of equivalent solutions across generation levels are only statistically significant (ANOVA and Wilcoxon Rank Sums) for Change 2 (p-value = 0.0001) and Change 4 (p-value = 0.0004), once again. It takes 150 generations for Change 2 to reach a plateau in terms of equivalent solutions whereas for Change 4, the results at 200 generations are still significantly better than those at 150 generations. For the two other changes, 50 to 100 generations appear to be enough. These results can be explained by the fact that the changes rank in increasing complexity (to fix) according to the following order: Change 3, Change 1, Change 2, and Change 4.

	Number of Generations			
	50	100	150	200
Change 1	12.5, 4.7%	13.9, 14%	15.7, 15%	14.1, 11%
Change 2	14.4, 6.8%	13.6, 16%	14.9, 28%	15.7, 29%
Change 3	13.8, 22%	15.7, 23%	16.1, 24%	17.1, 21%
Change 4	6.6, 2.5%	7.8, 5.8%	10.9, 16%	12.6, 23%

Table 6 Average # Within-range, Average % Equivalent Solutions

Table 7 shows the minimum, maximum, average, and standard deviation for the number of within-range solutions for each of the four changes at generation 100 over the ten runs. It also shows the minimum and maximum numbers of equivalent solutions and the average and standard deviation percentages of equivalent solutions. Its purpose is to highlight and discuss the variance across SPEA2 executions. The standard deviation for the number of within-range and equivalent solutions is high, and in the worst run the SPEA2 algorithm was unable to find a single equivalent solution for three of the four changes. On average, over the 10 runs, SPEA2 was able to find at least one equivalent solution for each change. In practice, because of its non-deterministic nature, we can expect that it is necessary to run the SPEA2 algorithm a number of times to ensure that satisfactory results be obtained. The number of runs in each particular situation will be determined by the time it takes to perform one run and the time constraints of the designer. Consistent with the results of Table 6, the GA seems to recover from

change 3 more easily than from changes 1 and 2, which are themselves, as expected, easier to recover from than change 4 (higher percentages of identical or equivalent solutions): the class members moved in changes 3 had enough interactions to provide the SPEA2 with sufficient information to guide these class members back to an equivalent CRA, while in change 1 and 2 there are fewer cohesive interactions with the class members from their original class assignment, and thus less information to guide the algorithm back to an equivalent CRA.

	Min	Max	Avg	Std Dev.
Change 1	8, 0	25, 3	13.9, 14%	5.4, 13%
Change 2	12, 0	17, 5	13.6, 16%	3.9, 13%
Change 3	9, 1	27, 10	15.7, 23%	5.4, 13%
Change 4	2, 0	11, 2	7.8, 5.8%	4.0, 7.7%

Table 7 Within-Range and Equivalent Solutions at Generation 100 over 10 runs

Future work will have to investigate further ways to rank and classify alternative solutions, as well as ways to allow the user to retain good solutions from generation to generation so that they are not lost because of clustering. What the results show is that, after 100 generations, a good percentage of the within-range solutions are likely to be good alternative analysis models in terms of CRAs. This is important as in practice the designer would start browsing alternative solutions and should be able to find a few applicable ones quickly.

7.2 Comparison with other heuristics

Section 7.2.1 discusses the design of this study. Sections 7.2.2 to 7.2.4 discuss the results for the random search, the hill climber and the single objective GA, respectively. A wrap-up discussion is provided in Section 7.2.5.

7.2.1 Design of case study

7.2.1.1 Baselines of comparisons

The goal of the second part of the case study is to determine if there are benefits from using a MOGA when compared to simpler search heuristics. We use as baselines of comparison the following heuristics: a random search algorithm, a random mutation hill climber, and a single objective genetic algorithm. A random search consists in generating a number of solutions at random, without guiding

the search in any way. This helps determine whether the search problem is difficult enough to warrant any further investigation. Furthermore, in [10], the authors recommend that a hill climber algorithm be applied to any search based software engineering problem since hill climber algorithms have been shown to perform well when compared to more complex genetic algorithms, and can produce good results even in the presence of an unfavourable landscape. Further, the comparison to a hill climber algorithm will validate the need for a global search approach to the CRA problem. In our case, we use a Random Mutation Hill Climber, which has been shown to outperform a traditional genetic algorithm on certain problems, and to offer a more competitive comparison to a GA than other hill climber algorithms [15]. The use of a single objective GA will demonstrate that a multi-objective search is required for the CRA problem. Without the need for a complex fitness function, or an archive and truncation operation, the single objective genetic algorithm is a better alternative as it is much faster than SPEA2. The goal of this comparison is to determine if the single objective genetic algorithm is able to discover satisfactory CRAs within a reasonable number of runs, when compared with the SPEA2.

7.2.1.2 Setting the parameters of search algorithms

All the parameters discussed below were set to make the comparisons as fair as possible, and when not possible, make sure SPEA2 is not advantaged in any way possible. Parameters and numbers of runs (below) are summarized in Table 8. The hill climber algorithm we used is called a Random Mutation Hill Climber (RMHC) [15], similar to the algorithm used in [19]. The RMHC is a simple algorithm that makes single, random changes (in our case, assigning a class member to a different class than the original) to a starting solution to generate neighbours, and then compares these neighbour solutions to determine an optimal (best) value. The algorithm repeats with this new starting solution until no superior neighbouring solutions can be found. This is a weakness of hill climbers, since they have no way of knowing if the local optimum achieved is the global optimum. Search spaces with many such local optima are difficult for hill climbers to explore. To find the neighbour that offers the most improvement, 500 random neighbours are generated (similar to [19]), and the best of these 500 is used as the new candidate. If no improvement is found in these 500 solutions, then the current candidate is considered the best solution. This simplifies the evaluation of neighbour solutions and allows the control over the number of neighbours evaluated at any given step (see below). However, the random element means that no run of the RMHC will be the same, and that we need to run it a number of times

to determine if it is able to find any reasonable solutions. We run it a number of times so that RHMC results are comparable to SPEA2, as explained below.

The single objective genetic algorithm uses similar parameters as the SPEA2 to ensure a fair comparison: binary tournament selection, with the best individual selected 90% of the time; single point crossover, at a rate of 70%, 100 generations per run, for 10 runs as for SPEA2 in this section. To maximize the potential of the single objective genetic algorithm, the top solution from each generation is checked, and is recorded if it is within-range. Note that these parameters, as discussed in Section 6.3, were specifically recommended for single objective genetic algorithms and were therefore not specifically tuned for SPEA2. If anything, these parameters should be to the disadvantage of SPEA2.

With a single objective approach, either with the RMHC or the single objective algorithm, it is necessary to normalize the objectives to create a range independent single objective problem [1]. To do so, the coupling measures are normalized over the amount of possible coupling in the system. This is determined as the number of DMA and DMM dependencies in the system (Section 4) for MMC, MAC and MGC (Section 4), respectively. The cohesion metrics are already normalized, and do not need to have their range adjusted. The resulting single objective fitness function is defined in Definition 20.

Definition 20. Single objective fitness function. The single objective fitness function for the nearest neighbour and sum of weighted objectives algorithm for a given class diagram *C*, denoted *F*(*C*), is:

$$F(C) = w_1 RCI(C) + w_2 TCC(C) - w_3\left(\frac{MAC(C)}{|\,DMA(C)\,|}\right) - w_4\left(\frac{MMC(C)}{|\,DMM(C)\,|}\right) - w_5\left(\frac{MGC(C)}{(|\,DMA(C)\,| + |\,DMM(C)\,|)}\right);$$

where: *RCI*, *TCC*, *MAC*, *MMC*, *MGC, DMA*, and *DMM*(*C*) are defined in Section 4 and are applied here to an entire class diagram (summing their values for the classes in the diagram), w_1, w_2, w_3, w_4, w_5 are weights, and *F*(*C*) is to be maximized. The weights are set to equally balance each of the objectives. Given that there are five objectives, each objective is weighted to contribute 20% to the overall fitness ($w_i = 0.2$). We refer to this hill climber as RMHC1. The single objective genetic algorithm using these weights is referred to as SOGA1. We also tried another set of weights to better reflect the criteria for within-range solutions, which are not permitting any method-generalization coupling (Section 7.1). To reflect this, the weight of MGC was increased from 0.2 to 10 (without changing any of the other weights) to heavily penalize solutions that have a MGC above zero. We refer

to this hill climber as RMHC2. The single objective genetic algorithm using these weights is referred to as SOGA2.

7.2.1.3 Setting the number of runs

It is important to ensure that results across techniques be comparable and, in case where this is not straightforward, at least make sure SPEA2 is not advantaged in any way possible so as to obtain conservative results. To do so, the number of overall fitness evaluations is used to ensure that each algorithm has an equal opportunity to explore the search space. We used 100 generations of our SPEA2 algorithm as a baseline of comparison in this part of the study: with a population size of 320 (Section 6.3), we have 32,000 individual fitness evaluations per run, for 10 runs. The single objective genetic algorithm is given the same population size as SPEA2 (320 individuals) and also runs for 100 generations per run, thus leading again to 320,000 fitness evaluations across 10 runs. Similarly, the random search generates 320,000 random individuals. The random mutation hill climber is a more complex case. The number of evaluations is determined by the number of "moves", that is the number of times 500 hundred neighbours are randomly picked and a new best solution identified. We run RMHC just enough times so that the number of fitness evaluations is at least as large as for SPEA2, to ensure comparability. All the above information is summarized in Table 8.

The fourth, most complex change, that includes the three other simpler changes (Section 7.1.5), is used to compare the four algorithms (random, hill climber, single objective GA, our MOGA). This change involves the move of six class members, that is all the class members involved in changes 1-3.

7.2.1.4 Data analysis

In this section we wish to compare the number of within-range and equivalent solutions produced by each search technique. Then for each of the six class members involved, as an attempt to better understand the results and test the statistical significance of the differences across techniques, we analyze and compare the proportions of runs where an acceptable CRA has been achieved for each class member across techniques. A Z-Score test for proportions will be used to this effect [14].

Technique	Parameters	# of Runs and fitness evaluations
SPEA2	**Population Size:** 64 individuals per objective, 320 total **Archive Size:** 160 (1/2 population size) **Selection:** Binary Tournament (90% best individual, 10% other) **Crossover Rate:** 70% **Crossover operation:** 1-point crossover **Mutation Rate:** 1 / length of chromosome	10 runs of 100 generations 32,000 fitness evaluations per run 320,000 fitness evaluations
Random Search	None	320,000 random solutions Note: This is equivalent to the number of fitness evaluations for 10 runs with SPEA2
RMHC	**# of random neighbours:** 500 **Weights:** RMHC1: equal weights (0.2) RMHC2: weight of 10.0 for MGC	35 and 34 runs for weight options 1 & 2 RMHC1: 325,500 fitness evaluations RMHC2: 321,500 fitness evaluations
Single Objective GA	**Population Size:** 320 **Selection:** Binary Tournament (90% best individual, 10% other) **Crossover Rate:** 70% **Crossover operation:** 1-point **Mutation rate:** 1 / length of chromosome	10 runs 100 generations per run 32,000 fitness evaluations per run 320,000 fitness evaluations

Table 8 Parameter Settings, Runs, and Evaluations

7.2.2 Random Search Results

Randomly generating 320,000 candidate solutions failed to produce any within-range solution, using the same ranges discussed in the first part of the case study (Section 7.1). In an attempt to get within-range solutions with the random algorithm, the number of solutions generated was even increased to 640,000 without success. It is not necessary to perform a statistical test here to assess the significance of the difference with SPEA2 as it is obvious that random search is not a viable option.

Given the size of the search space, the fact that the random algorithm was unable to discover a within-range solution is not surprising, though size is not the only factor as the landscape also affects the complexity of the search. The fact that the random search is not capable of producing meaningful results however indicates that our case study involves a search that is far from trivial.

7.2.3 Random Mutation Hill Climber

Recall that the best run of the SPEA2 algorithm was able to produce 11 within-range solutions by generation 100 and that overall all ten runs 78 within-range solutions were identified. After the 35 and 34 runs of RMHC1 and RMHC2, respectively, only 8 and 6 within-range solutions were found, that is much lower numbers than with SPEA2. Because these numbers are small to enable statistical comparisons, we decided to increase the number of runs to 160 in order to increase the number of within-range solutions, even though this is leading to a very large number of fitness evaluations (> 1.2 Million) and is therefore providing an unfair advantage to RMHC. After 160 runs, RMHC1 and RMCH2 yield 35 and 26 within-range solutions, respectively. Figure 4 compares the class members repaired by RMHC with 160 runs and SPEA2: The horizontal axis lists the 6 class members moved, while the vertical axis shows the percentage of within-range solutions that were able to move that class member to the same or an equivalent position in the class diagram. As shown in Figure 4, all of the algorithms were able to return the six class members to their original location in one or more within-range solutions. The hill climbers had slightly more success than the SPEA2 at fixing the fourth class member, though the difference is not statistically significant. However, for all other class members, the SPEA2 performed better, especially for class members 1, 2 and 6 where the difference is substantial and significant. Table 9 shows the Z-Score test results for the difference between two population proportions [14]. The scores must be above 1.63[7] for the difference to be significant ($\alpha = 0.05$). This is true for all comparisons except SPEA2 vs. RMHC1 for class member 6, which is just below this threshold. The fact that we have relatively small numbers of within-range solutions limits the statistical power of our tests, hence explaining this latter result.

[7] Underlined scores are significant. If one wants to follow a Bonferroni adjustment, then $\alpha = 0.05/6 = 0.0083$ and then Z must be superior to 2.4. However, such an adjustment is very conservative as tests are not independent. This would mean that with the current data set, we cannot assert a statistically significant difference for Change 6.

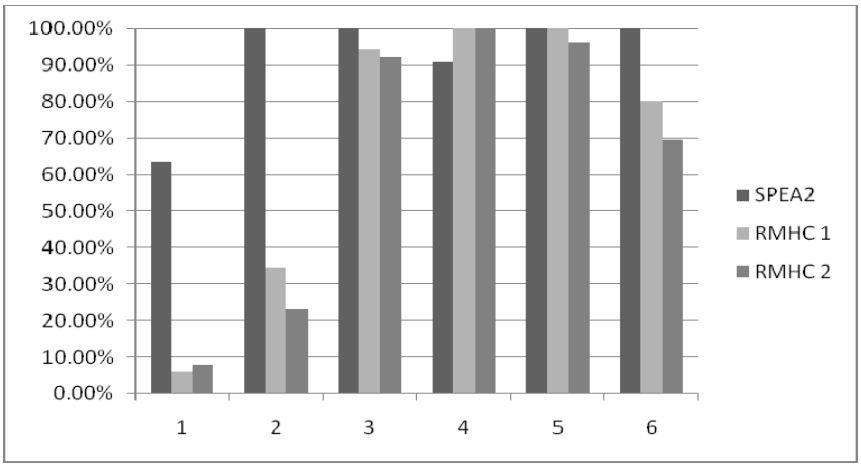


Figure 4 Comparison of class members repaired

	SPEA2 vs. RMHC1	SPEA2 vs. RMHC2
Class Member 1	4.22484498	3.62565915
Class Member 2	3.802007	4.29123367
Class Member 6	1.61086058	2.07808569

Table 9 Z-Score test for class members 1, 2, and 6: Z-Score > 1.63 for $\alpha = 0.05$

While the number of within-range solutions found by RMHC is significant, a within-range solution does not necessarily represent a quality solution. For a solution to be useful, it must be identical or equivalent to the original ARENA design. In the best run of the SPEA2 for Change 4 (most complex one), two equivalent solutions were found out of the 11 within-range solutions. When accounting for all ten SPEA2 runs, 78 within-range solutions were found, including six equivalent solutions. After investigating the solutions generated by the 160 runs of hill climber, we found that *none* of the within-range solutions were equivalent. Indeed, not one of the hill climber solutions was able to fix all six class members, let alone be an equivalent solution. This suggests that a multi-objective, guided search such as the SPEA2 is required.

The main reason that the SPEA2 is able to discover and maintain an equivalent solution is the diversity granted by using a multi-objective algorithm. The archive management performed by the SPEA2 ensures that a wide range of Pareto optimal solutions are maintained. As equivalent solutions represent but a small part of the vast search space, this diversity maximizes the chance of finding meaningful results, and thus the SPEA2 has outperformed the two hill climbers.

The above explanation is clearly demonstrated by looking at the distribution of coupling and cohesion values formed by within-range solutions of RMHC and SPEA2 (Figure 5). The differences in the spread of values are stunning, though much less so for MAC, thus suggesting a much greater variety of solutions for SPEA2.

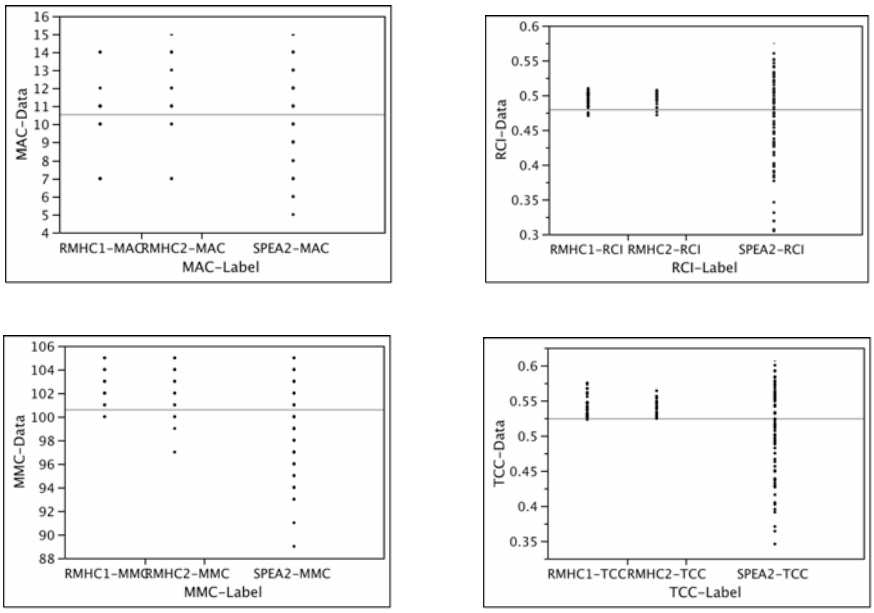


Figure 5 Scatterplots of Within-Range solutions for RMHC and SPEA2

7.2.4 Single Objective Genetic Algorithm

Using the same setting as SPEA2, with again 100 generations per run and 10 runs, using the first set of weights, the single objective genetic algorithm (SOGA1) was unable to find any within-range solution. We also tried the second set of weights (SOGA2). This time we additionally increased the elitism intensity of the algorithm by preserving the best solution from each generation, ensuring that the best individual is always selected for the next generation. However, after another 10 runs, the genetic algorithm was still unable to find a within-range solution.

As another attempt to discover a within-range solution, we increased the mutation rate to 5 / length of the chromosome, or an average of five changes per solution. This was done to try and prevent the genetic algorithm from converging on a single solution. We also allowed the genetic algorithm to run much longer, for 1000 generations. The other changes, improving the fitness function and preserving the best individual from each generation, were left in place. Even with all these changes, the single

objective GA was unable to find a single within-range solution in 10 runs (remember that we check the solution after each generation, or a total of 10000 potential solutions). After these changes, we concluded that the single objective genetic algorithm is not able to fix the fourth change, or find a single, within-range solution.

There are a number of potential reasons why the genetic algorithm is not able to find any within-range solution. The most significant factor is the elitism intensity of the genetic algorithm. A standard genetic algorithm is not very elitist, and a good solution may be lost due to mutation or crossover. This can be beneficial, but with such a complicated landscape, requiring a number of subtle changes to the solution, an elitist algorithm is necessary. By not maintaining and leveraging a list of strong solutions, the single objective genetic algorithm is not able to make proper progress towards within-range solutions. We attempted to fix this by preserving the best individual from each generation, but it didn't provide enough help to get to a within-range solution. It seems that a set of optimal individuals is necessary in order to provide enough elitism and guidance towards an optimal solution. The SPEA2, on the other hand, maintains a full archive of optimal solutions, and thus is much more likely to find a good solution.

A second possible reason is that, because the changes required to achieve a within-range solution are very subtle, the fitness function may not provide enough guidance to a single objective algorithm to get it to a solution. Without strong guidance towards better solutions, the genetic algorithm is not able to make sufficient improvement in order to discover a within-range solution. Again, this problem is avoided by SPEA2 as it uses Pareto optimality and a clustering algorithm to ensure that progress is made and maintained for each of the objectives.

7.2.5 Conclusions

Sections 7.2.2 to 7.2.4 investigated three algorithms, a random search, a random mutation hill climber, and a single objective GA, that we compared with the SPEA2 algorithm. As summarized in Table 10, none of them were able to produce satisfactory solutions (i.e., equivalent solutions) for the CRA problems submitted to them. The random search and single object genetic algorithm were not even able to find within-range solutions. During its best run, SPEA2 produced a small number of within-range

solutions to inspect (11) leading to two viable CRA solutions. These results suggest that, for the CRA problem, a multi-objective approach such as SPEA2 is beneficial.

	SPEA2	Random	SOGA	RMHC1	RMHC2
Within-range	78	0	0	8	6
Equivalent	6	0	0	0	0

Table 10 Summary of results

7.3 Effects of population size

7.3.1 Design of the study

The performance of a genetic algorithm is affected by its many parameters, such as crossover rate, mutation rate, elitism intensity, and population size (Section 6.3). But population size is a particularly difficult parameter to set for a multi-objective genetic algorithm. It has been shown to strongly affect the results for single objective genetic algorithms, for which investigations and recommendations have been reported [18, 31]. However, no such studies on the effect of population sizes of MOGAs exist that we know of. So it is important to investigate the effect of population sizes on the results in our application context to make practical recommendations. In Section 6.3, a population size of 64 individuals per objective was chosen. To study the effect of population size on the SPEA2 algorithm, we follow the recommendations for single objective genetic algorithms made in [18], suggesting a population size ranging from 30 to 80. But as recommended in [39], we increased the number of individuals according to the number of objectives and will therefore investigate the effect of having a population size of 30 individuals and 80 individuals per objective, and compare these with the results with those of the population size of 64 used in Section 7.1.

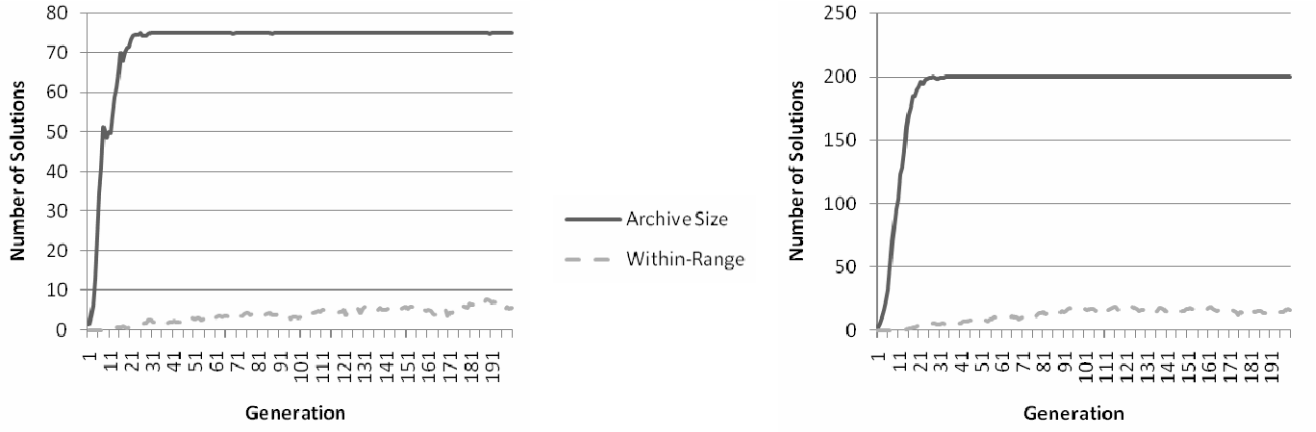
We will keep the rest of the parameters the same as in Section 7.1 to facilitate comparisons. In the future, large-scale simulations should be performed to empirically assess the interactions of all parameters affecting SPEA2 using experimental designs, as this was done for multi-objective genetic programming [9]. This is, however, due in part to space constraints, out of the scope of this paper. Of particular note, the size of the archive will remain proportional to (i.e., half) the size of the population in order to keep the elitism intensity the same. As in Section 7.2, we consider here the fourth and most complex change of our case study (Section 7.1.5). Again, to account for random variations, for each of the population sizes, we ran the SPEA2 ten times.

7.3.2 Data analysis

The analysis will compare different population sizes in terms of their number of within-range solutions, equivalent solutions, and execution times. In addition, the distributions of within-range solutions in terms of their coupling and cohesion values will also be compared. To do so, as we have done in previous sections, we will use standard parametric (ANOVA for the overall analysis, t-test for paired comparisons) and non-parametric (Wilcoxon Rank-Sum Test) statistical tests to compare distributions of numbers of solutions found, execution times, and coupling and cohesion metric values. As above, our α-level = 0.05 and the impact of a Bonferroni adjustment will be evaluated.

7.3.3 Results

Figure 6 shows the average number of within-range solutions for 30 individuals per objective (a) and for 80 individuals per objective (b). Table 11 summarizes the coupling and cohesion results of the best run for the three population sizes we tested (30, 64 and 80 individuals per objective) for the fourth change at 100 generations, using as similar format as Table 5 (MGC not shown). Again, the best run is the run that returns the largest number of within-range and equivalent solutions. Table 12 shows the number of within-range solutions (on average over ten runs, with the standard deviation in brackets) and the percentage of equivalent solutions among them at 50, 100, 150 and 200 generations (on average, with the standard deviation in brackets) for 30, 64, and 80 individuals per objective. The average execution time for each generation is also provided.



(a) (b)

Figure 6 Average Number of Within-range Solutions for Change #4: (a) 30 individuals per Objective, (b) 80 individuals per Objective

As shown in Figure 6 (a) for population size 30, the pace of increase in within-range solutions is slower than for population size 64 (Figure 3). Also, the overall number of within-range solutions obtained when reaching the plateau is lower, as expected since the archive is smaller for the 30 individual run. In Table 11, after 100 generations, the metrics don't show the same improvement as the 64 population size run, though the difference cannot be confirmed to be statistically significant as the number of within-range solutions (2, 11, and 18, respectively) is too small to achieve acceptable statistical power. If we now turn to Table 12, results show that, for a population size equal to 30, only 6% of the few solutions found are on average equivalent CRAs at generation 100, and that there is a high variance in the number of within-range and equivalent solutions found. This shows that, with a small population size of 30, SPEA2 is not able to consistently find equivalent CRAs, and that a larger population size is needed.

	30 Individuals (3 mins)				**64 Individuals** (7 mins)				**80 Individuals** (9 mins)			
	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC	MAC	MMC	RCI	TCC
Avg.	12.5	103	0.42	0.46	7.63	97.81	0.48	0.53	8.94	94.89	0.43	0.49
Best	10	101	0.47	0.48	5	93	0.56	0.60	5	86	0.54	0.58
Worst	15	105	0.38	0.45	11	105	0.38	0.36	14	104	0.30	0.37

Table 11 Impact of population size on metrics: best run results

With 80 individuals per objective, except for MAC that is slightly higher, the coupling and cohesion metric values only shows a slight improvement compared to those for size 64 (Table 11). There seems, therefore, to be no significant gain with that respect. The number of individuals in the archive, when comparing Figure 6 (b) and Figure 3, are similar, with roughly the same overall pattern, a somewhat higher plateau of within-range solutions (200 vs. 160) and higher numbers of equivalent solutions. With the larger archive size, the search with a population size of 80 individuals does, on average, produce a greater number of within-range solutions. Regardless of population size, the variance over ten runs is very high for both population sizes, so the user will still need to perform a number of runs in order to ensure that a good solution is found.

Overall, execution time seems to increase, on average, linearly with population size. This is rather good news as the use of larger population sizes then comes at a reasonable price. It also shows the reasonable quality of our SPEA2 implementation (as SPEA2 has a worst case computational complexity of $O(N^3)$, where N is the sum of the archive size and population size [39]). Differences in within-range and equivalent solutions between population sizes 30 and 80 are always statistically

significant (e.g., p-values are 0.0001 and 0.0073 for within-range and equivalent solutions with 100 generations), even after we adjust our α-level using the Bonferroni procedure. However, this is not always the case when comparing population sizes 30 and 64, and 64 and 80, respectively. This latter result is probably due to the fact that we have—due to practical reasons mentioned above—ten runs for each population size, which limits statistical power. In any case, this does not affect our overall conclusion that, within the explored range (30-80 individuals per objective), population size matters and should be as large as possible given the user's time constraints. Our use of 64 individuals per objective in the previous sections, leading to a population of 320 individuals and an archive size of 160 individuals was a reasonable choice for the CRA problem, giving a good trade-off between the exploration of the search space and the cost of the algorithm.

Overall, the results above match what is stated in [20], that is too small of a population and the search space cannot be fully explored, but too large a population size and the cost may become impractical.

	Number of Generations			
	50	100	150	200
30 individuals	3.1 (3.0), 0% (0%) 1 min	3.1 (3.2), 6% (13%) 3 min	6 (6.8), 3% (10%) 5 min	5.6 (3.2), 14% (25%) 6 min
64 individuals	6.6 (4.3), 3% (6%) 3 min	7.8 (4.0), 8% (11%) 7 min	10.9 (2.6), 17% (15%) 11 min	12.6 (7.4), 22% (15%) 15 min
80 individuals	7.6 (3.8), 4% (9%) 4 min	15.7 (5.6), 12% (14%) 9 min	17.2 (4.7), 17% (14%) 14 min	16 (6.18), 20% (13%) 19 min

Table 12 Average # of Within-range Solutions,% of Equivalent Solutions, and excecution time for 30, 64, and 80 individuals per objective

8 CONCLUSIONS AND FUTURE WORK

The assignment of responsibilities to classes is a difficult skill to teach and to acquire in practice in the context of object-oriented analysis and design. While principles exist for assigning responsibilities to classes, their application relies upon human judgment and decision making. This is partially driven by the difficultly in balancing the many factors that make up a good design, and the subjective nature of software design in general.

This paper investigates the tailoring and application of a multi-objective genetic algorithm (SPEA2) to the Class Responsibility Assignment (CRA) problem. It does so in the context of domain class diagrams that are usually produced during the analysis stage of most methodologies. Our approach is

based on the definition of fitness functions based on carefully selected coupling and cohesion measures that can be retrieved from analysis UML models for example.

A case study was performed to validate the effectiveness of applying SPEA2 to the CRA problem. Because it is inherently subjective to assess the output of such an algorithm in terms of the quality of its CRA, we adopted a validation strategy that would give us a reference model against which to compare SPEA2 outputs. CRA defects of varying types, spread, and magnitude were introduced into a correct domain model, defined by a third party, that then acted as this reference model. Results showed that these defects could be corrected and a variety of acceptable solutions could be optioned within a reasonable number of generations and time.

This demonstrates that the multi-objective genetic algorithm is able to fix a variety of artificially seeded assignment problems, which is a significant step towards validating the approach. One issue though is to help prioritize or select from the alternative trade-offs proposed by the genetic algorithm in a cost effective manner. The current solution is to allow the designer to define an acceptable range for coupling and cohesion measurement. In the case study, this strategy seemed to be constraining enough to limit the number of solutions proposed to the user, while ensuring that a significant number of these solutions were identical or equivalent to the reference model.

The second part of the case study compared the SPEA2 algorithm to other, simpler algorithms. A random search algorithm was not able to discover any acceptable CRA solutions and therefore showed that a guided search is necessary. We also used a random mutation hill climber, based on a single objective formulation of the coupling and cohesion measures. The SPEA2 was shown to outperform the hill climber, both in the quality of the results returned and in the time taken to generate a reasonable number of within-range solutions. A single objective genetic algorithm was also evaluated and showed not to return any acceptable solution. Our conclusion is that the use of a complex, multi-objective genetic algorithm is justified and beneficial for the CRA problem in domain class diagrams, especially given the fact that domain models are likely to be at least as complex as our case study in practice.

There are several areas for improvement that should be targeted by future research. First our strategy should be assessed in the context of industrial case studies with human designers. The challenge in this

case is of course to find a way to assess objectively the impact of our methodology, a problem we have solved here by introducing defects in a correct reference model.

While the case study investigated the effect of population size on the effectiveness of the multi-objective genetic algorithm, the impact of other important parameters such as elitism intensity and mutation rate has been left unexplored.

An important research topic is related to how to handle inheritance hierarchies, as the simplifying assumption to not consider overridden class members was used in this approach. The ability to optimize inheritance hierarchies is extremely important when considering CRA in practice and for our approach to be fully applicable.

In terms of scope, our solution is currently limited to domain / analysis models (the PIM in the MDA framework). Other types of models have different requirements and objectives. For example, design patterns play an important role in design class diagrams and follow different constraints in terms of CRA. As a result, our fitness function should account for other factors than coupling and cohesion in lower level design models.

9 ACKNOWLEDGMENTS

This research was supported by CITO (Ontario) and IBM Rational. M. Bowman received an Ontario Graduate Scholarship.

10 REFERENCES

[1] Bentley P. J. and Wakefield J. P., "Finding acceptable solutions in the Pareto optimal range using multi-objective genetic algorithms," *Proc. Soft Computing in Engineering Design and Manufacturing*, pp. 231-240, 1997.

[2] Bowman M., *Multi-Objective Genetic Algorithm to Support Class Responsibility Assignment*, M.A.Sc. Thesis, Carleton University, Department of Systems and Computer Engineering, 2007

[3] Bowman M., Briand L. C. and Labiche Y., "Multi-Objective Genetic Algorithms to Support Class Responsibility Assignment," Carleton University, Technical Report SCE-07-02 version 3, http://squall.sce.carleton.ca, 2008.

[4] Briand L. C., Daly J. and Wuest J., "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Engineering - An International Journal*, vol. 3 (1), pp. 65-117, 1998.

[5] Briand L. C., Daly J. and Wuest J., "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25 (1), pp. 91-121, 1999.

[6] Briand L. C., Devanbu P. and Melo W., "An Investigation into Coupling Measures for C++," *Proc. IEEE International Conference on Software Engineering*, pp. 412-421, 1997.

[7] Bruegge B. and Dutoit A. H., ARENA, http://sysiphus.informatik.tu-muenchen.de/arena/, 2004 (Last accessed July 2006)
[8] Bruegge B. and Dutoit A. H., *Object-Oriented Software Engineering Using UML, Patterns, and Java*, Prentice Hall, 2nd Edition, 2004.
[9] Castillo F., Kordon A. and Smits G., "Robust Pareto Front Genetic Programming Parameter Selection Based on Design of Experiments and Industrial Data," in R. Riolo, T. Soule, and B. Worzel, Eds., *Genetic Programming Theory and Practice*, vol. IV, *Genetic and Evolutionary Computation*, Springer, pp. 149-166, 2005.
[10] Clarke J., Dolado J. J., Harman M., Hierons R., Jones B. F., Lumkin M., Mitchell B. S., Mancoridis S., Rees K., Roper M. and Shepperd M., "Reformulating software engineering as a search problem," *Journal of IEE Proc. - Software*, 2003.
[11] De Jong K. A., "Learning with Genetic Algorithms: An Overview," *Machine Learning*, vol. 3 (3), pp. 121-138, 1988.
[12] Deb K., Agrawal S., Pratap A. and Meyarivan T., "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II," *Proc. Parallel Problem Solving from Nature*, pp. 849-858, 2000.
[13] Demeyer S., Ducasse S. and Nierstrasz O., *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2003.
[14] Devore J. L., *Probability and Statistics for Engineering and the Sciences*, Duxbury Press, 5th Edition, 1999.
[15] Forrest S. and Mitchell M., "Towards a stronger building-blocks hypothesis: Effects of relative building-block fitness on GA performance," *Proc. Workshop on Foundations of Genetic Algorithms*, 1993.
[16] Fowler M., *Refactoring - Improving the Design of Existing Code*, Addison Wesley, 1999.
[17] Goldberg D. E., *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison Wesley, 1989.
[18] Grefenstette J. J. and Cobb H. G., "Genetic Algorithms for Tracking Changing Environments," *Proc. International Conference on Genetic Algorithms*, pp. 523-530, 1993.
[19] Harman M. and L. T., "Pareto Optimal Search Based Refactoring at the Design Level," *Proc. ACM/IEEE Annual Conference on Genetic and Evolutionary Computation*, pp. 1106-1113, 2007.
[20] Haupt R. L. and Haupt S. E., *Practical Genetic Algorithms*, Wiley-Interscience, 1998.
[21] Kleppe A., Warmer J. and Bast W., *MDA Explained - The Model Driven Architecture: Practice and Promise*, Addison-Wesley, 2003.
[22] Lanza M. and Marinescu R., *Object-Oriented Metrics in Practice*, Springer, 2006.
[23] Larman C., *Applying UML and Patterns*, Prentice-Hall, 3rd Edition, 2004.
[24] Laumanns M., Zitzler E. and Thiele L., "On The Effects of Archiving, Elitism, an Density Based Selection in Evolutionary Multi-objective Optimization," *Proc. International Conference on Evolutionary Multi-Criterion Optimization*, pp. 181-196, 2001.
[25] López-Ibáñez M., Prasad T. D. and Paechter B., "Multi-Objective Optimization of the Pump Scheduling Problem using SPEA2," *Proc. IEEE Congress on Evolutionary Computation*, 1, pp. 435-442, 2005.
[26] Mens T. and Tourwe T., "A Survey of Software refactoring," *IEEE Transaction on Software Engineering*, vol. 30 (2), pp. 126-139, 2004.
[27] O'Keeffe M. and O Cinneide M., "Towards automated design improvement through combinatorial optimization," *Proc. Workshop on Directions in Software Engineering Environments*, 2004.

[28] Pender T., *UML Bible*, Wiley, 2003.
[29] Perneger T. V., "What's wrong with Bonferroni adjustments," *British Medical Journal*, vol. 316 (7139), pp. 1236-1236, 1998.
[30] Rivas-Dávalos F. and Irving M. R., "An Approach Based on the Strength Pareto Evolutionary Algorithm 2 for Power Distribution System Planning," *Proc. Evolutionary Multi-criterion Optimization*, 2005.
[31] Schaffer J. D., Caruna R. A., Eshelman L. J. and Das R., "A study of control parameters affecting online performance of genetic algorithms for function optimization," *Proc. International Conference on Genetic Algorithms and Their Applications*, pp. 51-60, 1989.
[32] Seng I., Stammel J. and Burkhard D., "Search-based determination of refactorings for improving the class structure of object-oriented systems," *Proc. Conference on Genetic and Evolutionary Computation*, 2006.
[33] Smith J. E. and Fogarty T. C., "Adaptively Parameterized Evolutionary Systems: Self Adaptive Recombination and Mutation in a Genetic Algorithm," in Voigt, Ebeling, Rechenberg, and Schwefel, Eds., *Parallel Problem Solving From Nature 4*, pp. 441-450, 1996.
[34] Srinivas N. and Deb K., "Multiobjective optimization using nondominated sorting in genetic algorithms," *Journal of Evolutionary Computation*, vol. 2 (3), pp. 221-248, 1995.
[35] Svetinovic D., Berry D. M. and Godfrey M., "Concept identification in object-oriented domain analysis: Why some students just don't get it," *Proc. International Conference on Requirements Engineering*, pp. 189-198, 2005.
[36] Van Veldhuizen D. A. and Lamont G. B., "Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art," *Evolutionary Computation*, vol. 8 (2), pp. 125-147, 2000.
[37] Whitmire S. A., *Object Oriented Design Measurement*, John Wiley and Sons, 1997.
[38] Yoo S. and Harman M., "Pareto Efficient Multi-Objective Test Case Selection," *Proc. ACM International Symposium on Software Testing and Analysis*, pp. 140-150, 2007.
[39] Zitzler E., Laumanss M. and Thiele L., "SPEA2: Improving the Strength Pareto Evolutionary Algorithm," Swiss Federal Institute of Technology, Computer Engineering and Networks Laboratory, Technical Report 103, 2001.
[40] Zitzler E. and Thiele L., "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach," *IEEE Transactions on Evolutionary Computation*, vol. 3 (4), 1999.

Appendix A The ARENA Case Study

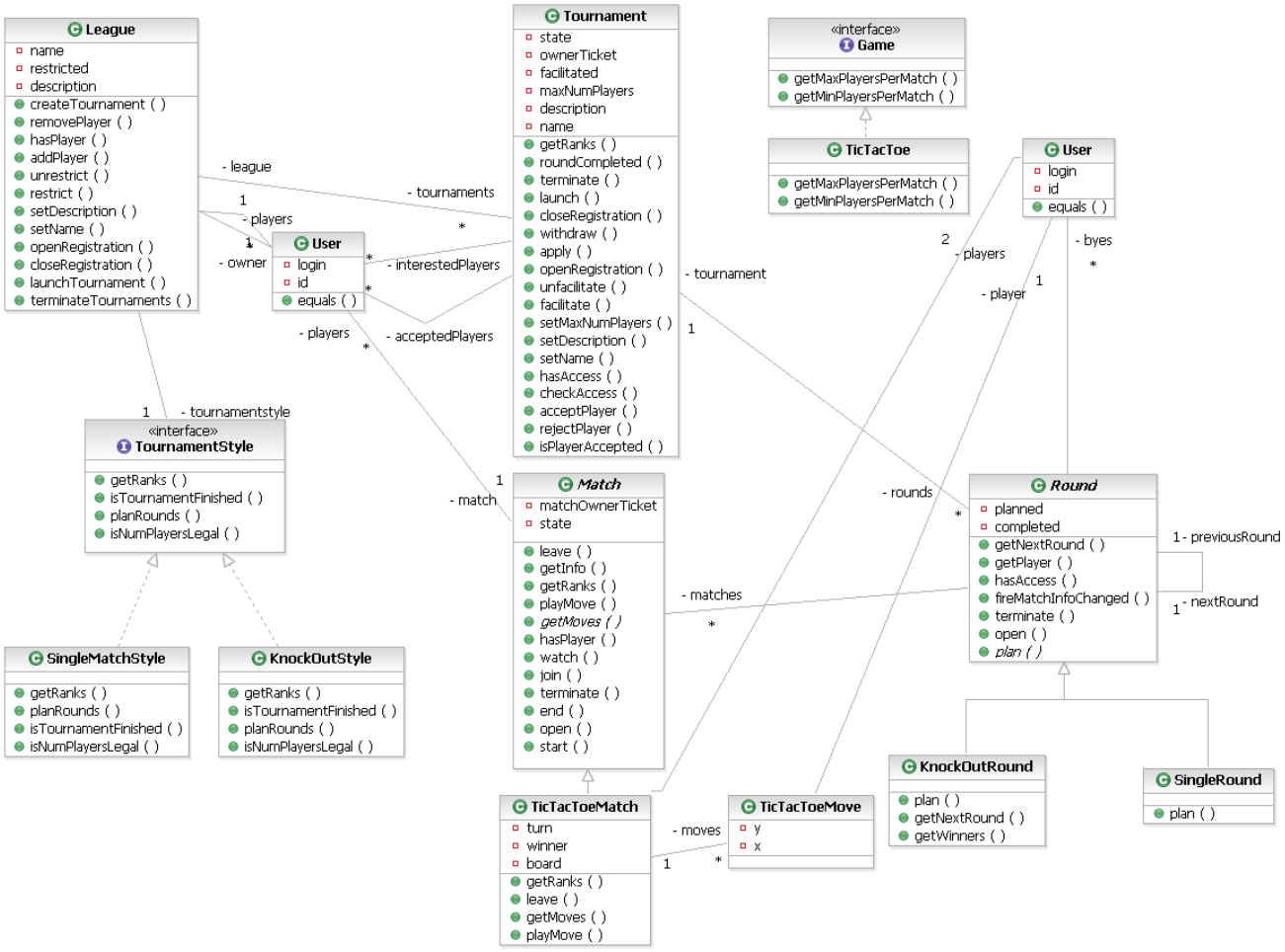


Figure 7 The ARENA case study class diagram

In Change #1, three operations of the `Match` class are moved from `Match` over to a new class, called `MatchStatus`. In Change #2, attributes of the `Round` class are moved from `Round` to `Match`. `Round` and `Match` are closely related, and the attributes moved are used by methods in both classes. This third change involved moving an association end and its grouped class members from a single class into a newly created class. This group was moved from `Tournament`, and into a newly created class called `TournamentPlayers`. Class diagrams resulting from these changes can be found in [3].