Multi-Objective Genetic Algorithms to Support Class Responsibility Assignment

Michael Bowman  Lionel C. Briand  Yvan Labiche

Software Quality Engineering Laboratory
Department of Systems and Computer Engineering
Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
(613) 520 2600 {2471, 5583}

{mgbowman, briand, labiche}@sce.carleton.ca

ABSTRACT

Expertise in Object-Oriented (OO) analysis and design is not an easy skill to acquire. There is ample evidence that this is hard to teach and apply. Though there are many methodologies, they all rely on human judgment and decision making for assigning responsibilities to classes. In this paper, our objective is to provide decision-making help to assign methods and attributes to classes in a class diagram. The technology we use is based on a multi-objective genetic algorithm (MOGA) and uses class coupling and cohesion measurement. Our genetic algorithm takes as input a class diagram to be optimized, typically produced during the analysis phase of software development (i.e., a domain model), and suggests possible improvements to the diagram. The choice of a MOGA stems from the fact that there are typically many evaluation criteria that cannot be easily combined in one objective, and several alternative solutions are acceptable for a given OO design problem. This article presents our approach in details, our decisions regarding the multi-objective genetic algorithm, and reports on a case study. Early results suggest that, under certain simplifying assumptions, the MOGA can help correct suboptimal class assignment decisions.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and techniques – Object-Oriented Design Methods, Computer-aided Software Engineering.

General Terms

Algorithms, Measurement, Design.

Keywords

Genetic Algorithm, Multiple objective, Coupling, Cohesion.

1. INTRODUCTION

Expertise in Object-Oriented analysis and design is not an easy skill to acquire. There is ample evidence that this is hard to teach and apply [26]. Though there are many methodologies, they all rely on human judgment and decision making for assigning responsibilities to classes. In this paper, our objective is to provide decision-making help to assign methods and attributes to classes in a class diagram. We first focus on diagrams exclusively containing domain classes, which are referred to as domain models and which are usually part of the early analysis steps [15]. We do so in order to simplify the problems in the early stage of this research. Future work will explore similar solutions for design class diagrams.

Our approach is based on a multi-objective genetic algorithm [27], uses class coupling and cohesion measurement [4, 5], and aims at providing interactive feedback to designers. The genetic algorithm (GA) takes as input a class diagram to be optimized, specifically information about method and attribute dependencies. 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 to the diagram using these measures as evaluation criteria. The GA 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 of the GA search is therefore to discover optimal assignments of attributes and methods to classes in regards to coupling and cohesion, thus leading to a more maintainable design [4], while accounting for user defined constraints on the class diagram.

Our main motivation for using the more complex multi-objective GAs is practical and is based on the recognition that it is very difficult, in our application domain, 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 design, along with interacting with the GA heuristic itself, the search will be guided towards an optimal class diagram that will be based on both the search and the designers input. The motivation is once again practical as we recognize the fact that, 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.

The rest of the paper is structured as follows. Section 2 describes related work. Sections 3, 4, and 5 provide details about our approach, specifically our measurement of class diagram quality, our operators for changing a model, and our use of a multi-objective GA. A case study is described in Section 6 and conclusions are drawn in Section 7.

2. RELATED WORK

The application of a metaheuristic search technique, such as GAs, to the field of software engineering is discussed in [7]. This paper discusses a wide range of possible applications of search heuristics to problems in software engineering, including the maintenance and re-engineering of software using program
transformation. This idea is expanded upon in [18] where the authors use a simulated annealing algorithm to automatically improve the structure of an existing inheritance hierarchy. The design metrics are expressed as a sum of weighted objectives in order to measure the designs and suggest improvements. This is further expanded in [23], where the authors use a GA to automatically determine potential refactorings of a system, not just an inheritance hierarchy. The paper also uses a sum of weighted objectives that measures the coupling, cohesion, complexity and stability of the system. The algorithm then searches the system for possible refactorings 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 software decay.

These two 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 algorithm (the Strength Pareto approach [28]) we use in this paper is able to present a number of possible tradeoffs to the designer. We think this is very important in our context as it is a priori difficult for any designers to weigh different design properties based on any objective criteria. Another difference of our paper with the techniques presented in [18, 23] is that they focus on the prevention of design decay during an iterative development process whereas we aim at providing decision aid and improving the early design of domain classes.

Although the Strength Pareto (SPEA2) approach has been just recently introduced in [28], there are several applications of the technique [3, 13, 17, 20]. At present, we are not aware of any applications of the SPEA2 technique, or other multi-objective evolutionary algorithms, to problems in software engineering.

### 3. DESIGN QUALITY MEASUREMENT

#### 3.1 Basic Definitions

The information we are using to optimize the design 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 3.2.

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 set 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 $c$, $M(c) = M_{NEW}(c) \cup M_{OVR}(c)$, the sets of newly defined and overridden methods in $c$, respectively. $AR(m)$ 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 $m$. The former is denoted $SIM(m)$ and the latter $PIM(m)$, with $SIM(m) \subseteq PIM(m)$. A method $m'$ in class $c'$ is statically invoked by $m$ when $m$ invokes $m'$ on an instance of type $c'$. In addition, assuming it is not overridden, $m'$ can be invoked on any instance of any subclass of $c'$ and these invocations are referred to as polymorphic. A "**" appended to the above set names denotes *indirect* accesses, invocations, or dependencies. $T()$ either refers to the type of an attribute or the return type of a method. $Par(m)$ denotes the set of parameter types of method $m$.

**Definition 1.** Method – Attribute Dependency (DMA, IMA)

A direct method–attribute dependency exists between $m \in M(C)$ and $a \in A(C)$ if $a \in AR(m)$. This is denoted $DMA(m,a)$. Similarly, if $a \in AR'(m) \rightarrow AR(m)$, we have an indirect method–attribute dependency denoted $IMA(m,a)$.

**Definition 2.** Method – Method Dependency (DMM, IMM)

A direct method–method dependency exists between $m_1 \in M(C)$ and $m_2 \in M(C)$ if $m_1 \in PIM(m_2)$, and is denoted $DMM(m_1,m_2)$. Similarly, if $m_2 \in PIM'(m_2) \rightarrow PIM(m_1)$, we have an indirect method–method dependency $IMM(m_1,m_2)$.

In addition to methods, attributes, and dependencies, we need to consider three types of relationships that we expect to be part of a typical class model: generalization, association, and usage dependency relationships [15]. 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. Note that we do not differentiate between associations, aggregation, and composition relationship, the two latter ones being a specialization of the first.

Generalization relationships will not change during the GA search but are nevertheless accounted for when we modify the class model. This complex issue is discussed in Section 4.2. Usage dependencies among classes are already accounted for when we consider Method-Attribute and Method-Method dependencies as the former normally imply the latter.

The dependency information on which we rely can be retrieved from Unified Modeling Language (UML) design models [19], and in particular interaction diagrams and operation contracts which can be expressed in the Object Constraint Language (OCL). These are typical components of an analysis or design model expressed with the UML [6]. For example, sequence diagrams tell us what methods can invoke other methods at run-time, OCL operation contracts suggest what attributes can be accessed by which methods, and class diagrams tell us about $M()$ and $A()$ sets.

To summarize, although our goal is to improve class responsibility assignment, as modeled by a class diagram, we still need to rely on information provided by other components of a UML model.

#### 3.2 Coupling and Cohesion Measurement

Many measures for cohesion and coupling have been proposed in literature. Frameworks to support the selection of appropriate measures in specific application contexts have been proposed [4, 5]. Based on these frameworks we selected two coupling measures based on attribute and method – method dependencies, as defined above. Our coupling measures are 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 one inherited being accounted for in the context of the parent class.)
Definition 3. Set of Method – Attribute Interactions (MAI)
For two classes \(c_1 \in C\) and \(c_2 \in C\), the set of Method Attribute Interactions between \(c_1\) and \(c_2\) is defined as
\[
\text{MAI}(c_1, c_2) = \bigcup_{m \in M(c_1)} (\text{MAI}(m, a) | a \in A(c_2) \land \text{DMM}(m, a))
\]

Definition 4. Set of Method – Interaction Interactions (MMI)
For two classes \(c_1 \in C\) and \(c_2 \in C\), the set of Method Interaction Interactions between \(c_1\) and \(c_2\) is defined as
\[
\text{MMI}(c_1, c_2) = \bigcup_{m_1 \in M(c_1)} \bigcup_{m_2 \in M(c_2)} \text{DMM}(m_1, m_2)
\]

The coupling measures below are based on the summation of the interactions between classes which are not an ancestor/descendant of one another (denoted \(\text{Others}(c)\) for any class \(c\)).

Definition 5. Method – Attribute Coupling (MAC)
For a given class \(c \in C\), Method–Attribute Coupling \(\text{MAC}(c)\) counts all MAI from \(c\) to classes that are not ancestors or descendents of \(c\):
\[
\text{MAC}(c) = \sum_{c \in \text{Others}(c)} \text{MAI}(c, c)
\]

Definition 6. Method – Method Coupling (MMC)
For a given class \(c \in C\), Method–Method Coupling \(\text{MMC}(c)\) counts all MMI from \(c\) to classes that are not ancestors or descendents of \(c\):
\[
\text{MMC}(c) = \sum_{c \in \text{Others}(c)} \text{MMI}(c, c)
\]

For the fitness value of our GA, we obtain a class diagram coupling measure by summing coupling values for all the classes. Similarly, for cohesion measurement, we measure cohesion at the class level, considering the method-based dependencies defined above, both direct and indirect (DMA, DMM, IM, and IMM). It is not always meaningful to expect every class member to be directly related to another. By considering indirect dependencies between class members, the 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 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, in order 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.

Last, we should determine how to handle accessor methods and constructors. This is an issue, since 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 methods 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 dependency in a class (IMA) 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 7. Cohesive Interaction (CI)
For a given class \(c \in C\), the set of cohesive interactions \(\text{CI}(c)\) is equal to the set of all indirect method attribute dependencies between the methods \(m \in \text{E}(c)\) and the attributes \(a \in \text{A}(c)\):
\[
\text{CI}(c) = \bigcup_{m \in \text{E}(c), a \in \text{A}(c)} \text{DMM}(m, a)
\]

Assuming \(\text{CI}_{\text{max}}(c)\) is the set of all possible cohesive interactions in the class, accounting for (in)direct method–attribute dependencies, we define the ratio of cohesive interactions:

Definition 8. Ratio of Cohesive Interactions
The ratio of cohesive interactions (RCI) for class \(c \in C\) is the number of cohesive interactions in class \(c\), over the number of possible such interactions:
\[
\text{RCI}(c) = \frac{|\text{CI}(c)|}{|\text{CI}_{\text{max}}(c)|}
\]

Note that when no method is present in a class, its RCI measure is undefined. In order to compute the cohesion value across the entire class diagram, which will be needed for our GA fitness function, the RCI values for all the classes in \(C\) are averaged.

We also use a complementary measure, tight class cohesion [4], which is based on the concept of common attribute usage. The idea is that methods which use common attributes 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 (in)directly use a common attribute of a given class, or when one method (in)directly invokes the other.

Definition 9. Common usage (cu)
The predicate \(\text{cu}(m_1, m_2)\) is true if \(m_1, m_2 \in \text{E}(c)\) (in)directly use an attribute of class \(c\), or if \(m_1\) (in)directly invokes \(m_2\):
\[
\text{cu}(m_1, m_2) \iff \begin{cases} \bigcup_{m_1 \in \text{E}(c)} \bigcup_{m_2 \in \text{E}(c)} \text{AR}(m_1) \land \text{AR}(m_2) \land \text{A}(c) \land \text{SIM}(m_1, m_2) \lor m_1 \in \text{SIM}(m_2) \end{cases}
\]

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

Definition 10. Tight class cohesion (TCC)
Tight class cohesion (TCC) is the pairs of methods of a class \(c \in C\) with common usage. (It is normalized.)
\[
\text{TCC}(c) = \frac{2 \times \left| \left\{ (m_1, m_2) | m_1, m_2 \in \text{E}(c) \land \text{SIM}(m_1, m_2) \right\} \right|}{|E(c)| \times (|E(c)| - 1)}
\]

When a class contains less 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.

To summarize, our GA fitness function is based on four measures capturing different and complementary aspects of coupling and cohesion.

4. CHANGE MODEL
As stated above, the goal of our GA is to optimize a domain model (class diagram) by finding an optimal assignment of methods and attributes to classes. In order to be able to perform a
search for such assignment, it is necessary to define the search space by determining possible changes to the diagram.

4.1 Methods and Attributes
Since the search is based on existing class diagram information on method-method and method-attribute 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 or attributes. It would be conceivably possible to add and remove methods, based on OCL contract information. Existing methods and dependencies could be broken up, and new methods added. Likewise, methods could be removed, and their responsibility and dependencies merged into the other methods. However, this is outside the scope of the current paper.

The main mechanism for our search of better domain models is therefore to move methods and attributes from one class to another, thus affecting the measures of coupling and cohesion. Our implementation also allows the user to specify that certain class members conceptually belong together and can only be moved together, thus representing related concepts. Note that, once method-method and method-attribute dependencies have been identified (e.g., from UML documents), only the ownership of methods and attributes matters and we do not need to maintain (update) types (e.g., attributes) and method signatures.

4.2 Classes and Relationships
Since our goal is to determine the optimal assignment of methods and attributes 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 classes that were not identified in the first place. Our strategy is, when a method or attribute 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 either provide an implementation of the method (which would then become concrete) or have concrete implementations of the abstract method in its own (existing or to be created) child classes (i.e., we would create methods). Alternatively, we could create a new generalization hierarchy that would receive the abstract method and all its concrete implementations in child classes.

However, at this initial 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 for this paper, that will be addressed in future work, is that we limit modifications to generalization hierarchies to attributes and concrete methods that are not overridden. Other class elements in hierarchies cannot be moved during the search.

4.3 Constraints
Classes cannot be empty. We also require that classes be involved in at least one dependency, either as a client or server. None of the classes in the domain model should be stand alone classes.

In addition to the two constraints listed above, user constraints must also be taken into account. These constraints limit the changes that can be performed on the model by preventing methods and attributes from being moved. For instance, the user may indicate that some methods and attributes are conceptually related (though not necessarily dependent on each other) and should therefore 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.

5. MULTI-OBJECTIVE GA (MOGA)
The objective of our search is to optimize the coupling and cohesion of a given class diagram based on four distinct measures (Section 3.2). However, in order to address those four objectives at the same time, it may be necessary to consider tradeoffs between them to find the best design. This type of problem is referred to as a multi-objective problem (MOP) [27]. Although a single objective optimization problem may have a unique optimal solution, MOPs present a possibly uncountable set of solutions that, when evaluated, produces vectors whose components represent tradeoffs in the objective space. A decision maker is thus required to choose an acceptable solution (or solutions) by selecting one or more of the solution vectors.

5.1 Basic Principles
MOPs are mathematically defined in [27] as follows, where in our context fitness functions are coupling and (lack of) cohesion measures and decision variables correspond to set cardinalities involved in our measures:

Definition 11. Multi-objective Problem (MOP)
An MOP solution minimizes the components of a vector of fitness functions \( F(\tilde{x}) \), where \( \tilde{x} \) is an n-dimensional decision variable vector \( (\tilde{x} = x_1, \ldots, x_n) \) for some universe \( \Omega \). Formally, an MOP minimizes \( F(\tilde{x}) = (f_1(\tilde{x}), \ldots, f_k(\tilde{x})) \) subject to constraints \( g_i(\tilde{x}) \leq 0, i = 1, \ldots, m, \tilde{x} \in \Omega \).

The objectives being optimized will often conflict, which places a partial ordering on the search space. This makes the problem of finding a global optimum in a MOP an NP-Complete problem.

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.

Key concepts related to MOPs are Pareto optimality, and range independence.
Definition 12. Pareto Dominance
A vector \( \mathbf{u} = (u_1, \ldots, u_k) \) is said to dominate \( \mathbf{v} = (v_1, \ldots, v_k) \) (denoted by \( \mathbf{u} \prec \mathbf{v} \)) if and only if \( u_i \) is partially less than \( v_i \), i.e., \( \forall i \in \{1, \ldots, k\}, u_i \leq v_i \wedge \exists i \in \{1, \ldots, k\} : u_i < v_i \).

Definition 13. Pareto Optimal Set
For a given MOP \( F(x) \), the Pareto optimal set \( (P^*) \) is
\[
P^* = \{ \mathbf{x} \in \Omega | \nexists \mathbf{x}' \in \Omega : F(\mathbf{x}') < F(\mathbf{x}) \}
\]

Definition 14. Pareto Front
For a given MOP \( F(\mathbf{x}) \) and Pareto optimal set \( P^* \), the Pareto optimal front \( (PF^*) \) is defined as:
\[
PF^* = \{ u = F(\mathbf{x}) | \mathbf{x} \in P^* \}
\]

In other words, the Pareto (optimal) front refers to optimal solutions whose corresponding vectors are nondominated 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 the present 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.

There are two categories of methods for comparing objectives in a multi-objective function: range-dependent methods and range-independent methods. The effective range of an objective function is the range of values it can return, which is determined by the objective function itself, the possible domain of input values, and the representation of the individual genes.

Our cohesion and coupling measures have very different ranges: \([0, 1]\) and \([0, +\infty]\), respectively. In such cases, the only way to ensure that all objectives in a MOP are treated equally by the GA is to ensure that the ranges of the objective functions are the same, or to ensure that the objectives are not combined or compared to one another [2]. So the choice is to make the effective ranges of all the objectives equal, and then use a range-dependent method to rank solutions, or a range-independent method must be used. Range-independent methods are more widely applicable and range-dependent methods tend to be more solution specific since the range of the objective must be altered in order to make them comparable (e.g., normalized) [2].

Certain fitness functions combine several objective fitness values into a single fitness value and are referred to as summation approaches. Others evaluate the fitness as a vector of individual objective values using Pareto dominance and are referred to as vector based approaches. Summation approaches must rely on weights to adjust the objectives in order to make the fitness value meaningful. These weights are typically subjective and problem dependent, making the summation based approach fairly problem specific and subjective. Vector based approaches, on the other hand, are range independent. 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, because there is no meaningful way of normalizing or setting weights for our various cohesion and coupling measures, we choose to adopt a vector-based approach.

5.2 The Strength Pareto Approach
There are many vector-based approaches in literature. VEGA [21] was the first one proposed but its performance has been shown to be suboptimal compared to more recent alternatives such as NSGA [25]. NSGA itself has been shown to be outperformed by SPEA [29]. More recently, improvements to both NSGA and SPEA have been proposed: NSGA-II [9] and SPEA2 [28], which have been shown to produce similar results on some MOPs [28]. The time complexity of NSGA-II is better than SPEA2, but [28] reports that SPEA2 has advantages over NSGA-II in higher dimensional objective spaces (with four objectives and more). 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 [28] and its scalability will therefore need to be investigated in the context of our problem.

The overall algorithm for SPEA2 (Figure 1) can be briefly described as follows [28]. In Step 2, each individual \( i \) in both the external (archive) set \( P_e \) and the population \( P_i \) is assigned a strength value \( S(i) \), representing the number of solutions it dominates. (Note that we omit the \( \rightarrow \) symbol for chromosomes \( i \) and \( j \), though chromosomes are vectors.) Based on \( S(i) \), the raw fitness \( R(i) \) is then calculated for each individual \( i \) [28]:
\[
R(i) = \sum_{j \in P_i \cup P_e \wedge i \not\rightarrow j} S(j)
\]

This fitness value is to be minimized: \( R(i) = 0 \) is a non-dominated individual, whereas \( R(i) \) is high when \( i \) is dominated by many individuals. Summing strengths instead of simply counting dominating individuals is a way to penalize individuals which are in high density areas of the search space and thus preserve diversity in the population.

When most solutions still do not dominate each other, additional density information is needed to discriminate between individuals with the same raw fitness values. SPEA2 uses the kth Nearest Neighbor technique [28]. For each individual \( i \) the distances (in the objective space) to all individuals \( j \) in both the external set and the population are calculated: distances are normalized using the maximum possible values (that can be computed for the system being analyzed, as suggested in [17]). Distances are stored in a list sorted in increasing order, and a density estimate is computed based on the distance to the \( k \)-th element in the list, denoted by \( \sigma_i^k \), where \( k = \sqrt{N + \bar{N}} \). The density value \( D(i) \) corresponding to individual \( i \) is defined by:
\[
D(i) = \lceil \sigma_i^k + 2 \rceil
\]

Finally, the density is added to the raw fitness value to give the fitness \( F(i) \) for an individual \( i \): \( F(i) = D(i) + R(i) \).

In the original SPEA, the clustering algorithm tended to remove boundary individuals [29]. This has been corrected in SPEA2 by using a clustering method during the environmental selection (Step 3). The first step is to copy all non-dominated individuals \( (F(i) < 1) \) from the external set and the population to the external set of the next generation: \( P_e_{i+1} = \{ i | i \in P_i \cup P_e \wedge F(i) < 1 \} \).

If the external set fits exactly into the archive \( (|P_e_{i+1}| = \bar{N}) \) then the environmental selection set is complete. Otherwise, either the archive is too large \( (|P_e_{i+1}| > \bar{N}) \) or too small \( (|P_e_{i+1}| < \bar{N}) \). If the archive is too small, then the best \( \bar{N} - |P_e_{i+1}| \) individuals in the
previous archive and population are copied into the new archive. If the new archive is too large, on the other hand, then the clustering method needs to be applied, which iteratively removes individuals from \( P_{t} \) until \( |P_{t}| = \overline{N} \). At any given point in the iteration, the individual that has a minimal cumulative distance to all other individuals in the archive is removed. In other words, we remove individuals so as to preserve diversity in the archive.

### 5.3 Parameters of the Genetic Algorithm

We have seen above that valid changes include moving attributes and methods between classes, as well as adding and removing classes. The representation must therefore track the attributes and methods, and the class to which they belong. In order to do so, each method or attribute in the assigned class diagram is assigned a unique identifying string. Because all of the dependency information is represented by a dependency matrix, used for computing cohesion and coupling measures, the chromosome representation does not need to contain this information. Chromosomes simply consist of an unordered set of method and attribute identifiers denoting their class assignment by prefixing them with the class they are assigned to: e.g., \( [c_1.a_1, c_2.m_1] \) denotes that attribute \( a_1 \) belongs to class \( c_1 \) and method \( m_2 \) to class \( c_2 \). There is, however, a restriction that must be accounted for when designing our operators on such a representation: each method / attribute may only appear once in the chromosome. In addition, consistent with our earlier decisions, it is impossible to have an empty class represented in the chromosome.

Determining the ideal population size for a GA is challenging but important [1]. For traditional GAs a variety of adequate population sizes have been suggested [12]: some recommend a range between 30 and 80 [11], while others suggest a smaller population size, around 20 and 30 [22]. For MOGA, authors use larger population sizes than those recommended for single objective GAs, and they also increase the population size proportional to the number of objectives [16, 28]. We follow these suggestions and use a population size of 80 individuals per objective.

The archive size also has an important effect on the performance of the MOGA. In [16] the authors examine the effect of elitism on the performance of the GA, and report that strong elitism together with a high mutation rate should be used to achieve best performance. They also show that the performance of SPEA improves as the archive size increases relative to the population size: Increasing the archive size implicitly raises the elitism intensity of the technique. We therefore set the archive size to the size of the population.

As stated previously, when using an elitism algorithm such as SPEA2 a high mutation rate is desirable to achieve optimum performance [16]. 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 \( 5 / \text{length} \) where length is the length of the chromosome. These findings are consistent with [24]: 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 \( 5 / \text{length} \). But, after experimenting, we concluded that it was too high and led to unstable results. We obtained much better results with a significantly lower rate of \( 1 / \text{length} \).
6. CASE STUDY

For our case study, we selected the ARENA system [6] since it is designed independently from our research and its domain model (analysis level class diagram) and other related information (e.g., sequence diagrams to determine method-method dependencies) are available. Furthermore, it was designed by experts and can be therefore considered a good design, 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). Although of modest size, the domain model is not trivial and contains 14 classes and 112 class members (methods and attributes). Of these 112 class members, 29 cannot be moved by our GA as they are overridden or implemented through generalization relationships. An analysis of this domain model shows a total of 215 dependencies, specifically 124 method–attribute dependencies and 91 method–method dependencies. The coupling and cohesion measurement values for this domain model, which represent the baseline on which we want to improve, are: 63.0 (method–method coupling), 25.0 (method–attribute coupling), 0.3324 (ratio of cohesive interactions), and 0.3531 (tight class cohesion).

In order to perform our case study in a systematic and controlled manner, several suboptimal modifications were made, one at a time, to the original ARENA design, which was assumed to be a good design, and was thus the target of our optimization. Each modified ARENA design was then optimized using our implementation of the SPEA2 algorithm, our goal being to return to the original ARENA design or some other similar good design. (There is usually no unique optimal design.) The modifications were selected to involve both attributes and methods, and to involve both new and existing classes. We present below three representative examples and discuss them in detail. For each of them we also analyzed the number of generations required to achieve a large proportion of good design solutions in the final archive set proposed to the designer.

One practical issue, both for our experimentation and in practice, is that MOGAs such as SPEA2 provide a large number of alternative, non-dominated solutions. It is therefore necessary to find a way to automatically trim these solutions in order to obtain a reasonably sized set of solutions for the designer to further consider. A designer would do that by specifying a range for cohesion and coupling measures in order to prune extreme solutions clearly favoring a specific coupling or cohesion measure at the expense of the others. That range would be specified as an acceptable percentage of increase over the starting coupling value and a similar percentage of decrease for cohesion. This makes sense as, after all, the goal is to improve cohesion and coupling, and not to sacrifice one for gains in the other. For the purposes of this case study, only solutions that had values of at least 30 for method–attribute coupling, 70 for method–method coupling, 0.29 for ratio of cohesive interactions, and 0.3 for tight class cohesion were retained for evaluation and are referred to as “within-range” solutions. The goal was to avoid solutions that optimize cohesion at the expense of large coupling increases. Admittedly, this is a heuristic and many unacceptable solutions can still be part of that selected subset. However, the case study will allow us to evaluate how effective this is.

Our SPEA2 implementation was run on the modified design for 200 generations for all three reported examples. It was able to quickly recover from the changes, and bring coupling and cohesion back to values similar or identical to those of the original ARENA design. Running on a Pentium 4 3.0GHz processor with 1GB of RAM, the execution time ranged from two to three hours. Change 1 consists in moving three methods, without their supporting attributes, into a new class. The negative effect on the design is twofold: it lowers the cohesion by introducing a new class, and it raises the coupling by moving the methods away from the attributes they use. Change 2 involves moving two attributes from one class to another one related by an association, which results in a smaller change effect on the coupling and cohesion of the system overall. Change 3 involves moving three methods and one association from a single class into a newly created class. In this case, we also indicate to the algorithm to move these class members as a single group as they conceptually belong together. Table 1 shows, for each change, the number of solutions in the archive that are within-range in terms of cohesion and coupling (e.g., 79 for Change 1 after 100 generations) and then the size of the subset of those solutions that are identical or equivalent to the original ARENA solution (e.g., 40 for Change 1 after 100 generations).

<table>
<thead>
<tr>
<th>Change</th>
<th>100 generations</th>
<th>200 generations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change 1</td>
<td>79, 40</td>
<td>79, 79</td>
</tr>
<tr>
<td>Change 2</td>
<td>150, 150</td>
<td>103, 103</td>
</tr>
<tr>
<td>Change 3</td>
<td>337, 245</td>
<td>349, 229</td>
</tr>
</tbody>
</table>

Table 1 Results

The above examples highlight a number of interesting facts. With less than 100 generations, a large percentage of the proposed set of non-dominated, within range solutions are likely to be good alternative designs in terms of class responsibility assignments. This is important as in practice the designer would start browsing alternative solutions and should be able to find a few applicable ones quickly. The results of the three changes are discussed in more detail in the Appendix. Future work will have to investigate ways to further rank and classify alternative solutions.

7. CONCLUSION

This paper presents an approach to aid with class responsibility assignment in object-oriented designs, a skill that has been shown to be difficult to teach and acquire in practice. It is based both on carefully selected coupling and cohesion measurement but also makes use of a multi-objective Genetic Algorithm (GA). Cohesion and coupling form the building blocks of the fitness function used by the GA. Because there is in our context no meaningful way to combine the various measurements characterizing the quality of a design, we resorted to recent proposals for dealing with multiple objectives in the context of GAs. Based on a careful analysis of alternatives, we selected the SPEA2 algorithm which yields an archive set of non-dominated design solutions. The user is then in a position to select an appropriate solution among the alternatives SPEA2 puts forward.

Our case study has shown that, when design mistakes were introduced in a good design, the problem could be corrected and

---

1 We identified that this performance is mainly due to our use of JGAP [14]: each time a chromosome is evaluated, the framework decodes it as many times as there are objectives (i.e., four times in our case).
archive sets with large proportions of acceptable design solutions could be obtained within a reasonable number of generations. This demonstrates that the GA is able to fix artificially seeded problems and is therefore a first step towards validating our approach. One issue though is to help prioritize or select the alternative solutions proposed by the GA in a cost effective manner. The current, partial solution is to define an acceptable value range for coupling and cohesion measurement and since the archive set contains a large proportion of useful, valid solutions, a user would be able to find a satisfactory solution by inspecting a few of the archive set solutions.

The next step is to apply our strategy in the context of a real and imperfect system design. Since determining what an optimal design would be might be difficult in such a context, we would then focus on identifying improvements proposed by the GA. A second important research topic is related to how to handle inheritance hierarchies as we made the simplifying assumption that overridden class members were not considered movable in our study. Yet another topic of interest is to design ways for the designer to efficiently interact with the GA. Last, we intend to improve the performance of our implementation of SPEA2.

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

9. REFERENCES
10. APPENDIX: CASE STUDY
The following section presents the ARENA case study, along with the three representative changes made to the original system, in more detail. The results obtained from each of the changes are presented in sections 10.1, 10.2, and 10.3, respectively. A class diagram of the original ARENA system is provided in Figure 2.

10.1 First representative change
For the first change, three class members of the Match class were moved from the Match class over to a new class, called MatchState. This change is shown in Figure 3. The effect that this change has on the system is twofold. The coupling is increased, as the three class members were moved away from their supporting attributes, and the cohesion was lowered because a non-cohesive class was added into the system.

Table 2 shows the results of the algorithm at 50, 100, 150 and 200 generations. The table includes the time it took to reach that generation, and the best, worst and average metrics for the solutions that were within-range. Figure 4 shows the number of within range solutions returned per generation.
As shown in Table 2, although the allowed range enabled solutions to have metric values that were worse than the original ARENA system (e.g., worst coupling values of 30 and 68 after 50 generations, instead of 25 and 63 for the original design), after 50 generations the algorithm has found solutions that are, on average, better than the original model for three of the four metrics (i.e., all the metrics except method–attribute coupling). In the case of the ARENA system, the most important metric for determining if a solution matches the original system seems to be the method–attribute coupling metric. Considering the number of method–attribute dependencies present in the system (large compared to method-method dependencies—recall Section 6), this metric value is relatively low in the original ARENA design. There are therefore few possible solutions that would improve this metric without adversely affecting the other values. This is represented in Table 2, as each of the three other metrics, MMC, RCI and TCC, improves as the algorithm advances, while method–attribute coupling shows little improvement on average.

Figure 4 shows that the number of solutions that are within range in the archive rises fairly quickly, then plateaus at around generation 50. Afterwards there is little change in the number of within range solutions in the archive. However, the results in Table 2 show that the average metrics for the solutions within the archive is constantly improving. So even though the number of within range solutions holds relatively constant, there is still improvement in the metrics. This pattern holds true for the other changes made in the case study.

### 10.2 Second representative change

For the second change, attributes of the Round class were moved from Round to the Match class. The Round and Match classes are closely related, and the attributes are used by both classes. The effect of the moved class members on the metrics of the system was not as profound as it was in the first change. Our intent is to (try to) mask the change from the algorithm, and determine if the algorithm can return the model to the original design. A diagram illustrating the change is shown in Figure 5.

Figure 5 Change #2 to ARENA Design

Table 3 shows the results of the algorithm at 50, 100, 150 and 200 generations. As before, the table shows the best, worst and average metric values for the within range solutions at each generation, along with the time it took to create that generation. Figure 6 shows the number of within range solutions in the archive per generation.

![Figure 6 Within Range Solutions by Generation for Change #2](image)
As before, the number of within range solutions plateaus after a small number of generations. This shows roughly when the archive filled with non-dominated solutions, and afterwards the spread of solutions remains relatively constant due to the truncation operation of the SPEA2. Also, as before, the metric values were, on average, improved for each successive generation. However, the method–attribute coupling was not improved, as it is, once again, the limiting metric for the model.

### 10.3 Third representative change

The third change involved moving a group of three methods and an association out of the Tournament class and into a new class called TournamentPlayers. These three methods and the association were grouped into a single group, in order to prevent the algorithm from breaking them up. These four class members are related, so the newly added class is perfectly cohesive, with coupling between the members. This will, as before in the second change, work to mask the change from the algorithm. Figure 7 shows the change to the original ARENA design.

Table 4 shows the results for the third change, in the same format as the previous two changes, and Figure 8 shows the number of within range solutions by generation for this change. Figure 8 shows the same pattern as the previous two changes, with the number of within range solutions rising fairly quickly, and then leveling out once the archive has filled with non-dominated solutions. Also, once again the metric values improve for three of the four metrics, TCC, RCI, and MMC, on average while the MAC value stays steady. This shows once again that method–attribute coupling metric is the driving factor for determining the quality of the returned solutions.
10.4 General discussion
The same pattern emerged for the three changes regarding the number of within range solutions returned per generation, as shown in Figure 3, Figure 6, and Figure 8. The reason for the steady number of solutions within range 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 individuals, so it works to maintain a wide spread of solutions across the search space. By restricting the range of values for the returned solutions, we sample a small section of the overall search space. The truncation 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. This explains why the number of within range solutions rises sharply in the initial generations (under generation 50) while the non-dominated individuals are found, and then plateaus, as the spread of non-dominated individuals is maintained by the truncation operation while the algorithm searches for more improved solutions across the four objectives.

Finally, by examining the within range solutions returned after generation 150, another pattern is found. Many of the solutions, while containing the original class member assignment for the three class members moved, also included other changes to the original ARENA system that were not improvements over the original. This is another effect of the SPEA2 implementation. Because the implementation focuses on boundary solutions, it will attempt to improve one measure as much as possible, regardless of the effect on the other metrics. This leads, after a large number of generations have passed, to the algorithm over-optimizing the metric values. The changes made become more radical, and design actually degrades away from the original ARENA system. This type of over-optimization could be expected of any system, in which there is no globally optimal tradeoff between the objectives. Without one overwhelmingly strong solution to converge to, the algorithm will attempt to optimize each of the objectives regardless of its effect on the others. This leads in many cases to the solutions being over-optimized. As more of these solutions are found, they are maintained as they represent boundary solutions. So it is important to not only monitor the number of within range solutions returned, but the quality of these within range solutions, and stopping the algorithm early to prevent the affect of over-optimization.