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

Abstract

Class responsibility assignment is not an easy skill to acquire. There is ample evidence that this is hard to teach and apply. Though there are many methodologies for assigning responsibilities to classes, they all rely on human judgment and decision making. In this paper, our objective is to provide decision-making help 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. Our MOGA takes as input a class diagram to be optimized, typically produced during the analysis phase of software development and evolution (i.e., a domain model) in the context of Model-Driven Development, 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 into one objective, and several alternative solutions are acceptable for a given OO domain model. This article presents our approach in details, our decisions regarding the multi-objective genetic algorithm, and reports on a case study. Our results suggest that the MOGA can help correct suboptimal class responsibility assignment decisions.

1. Introduction

Class responsibility assignment 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." [19] There is indeed evidence that this is hard to teach and apply (e.g., [31]). Not only this is 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., [7]), they all rely on human judgment and decision making, primarily based on heuristics. In this paper, our objective is to provide decision-making help for class responsibility assignment in an analysis or early design UML class diagram. Our work takes place in the context of Model Driven Architecture/Development (MDD) [17], 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 in changing models, not code, which is then re-generated. In this paper, we first focus on diagrams exclusively containing domain classes (the PIM), which are often referred to as analysis or domain models and which are usually part of early Analysis steps [19]. Future work will explore similar solutions for lower-level design class diagrams.

Our work bears some similarity to refactoring. Although most of the work in this area has considered source code refactoring, there is a trend to also consider refactorings at higher levels of abstraction, such as refactorings of UML models [22]. 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 [32], uses class coupling and cohesion measurement [5, 6], 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 which can be extracted form other UML diagrams, e.g., Sequence diagram, OCL contracts. 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 various aspects of coupling and cohesion, thus leading to a more maintainable model [5], 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 model, along with interacting with the GA heuristic 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 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 when selecting a specific solution.

The rest of the paper is structured as follows. Section 2 describes related work. Sections 3 to 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

A wide range of possible applications of meta-heuristic search techniques, such as GAs, to the field of software engineering is discussed in [8], e.g., the maintenance and re-engineering of software using program transformations. This idea is expanded upon in [23] 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 [28], 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 [12]: 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 also 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 algorithm we use in this paper [33] 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 designer to weigh different design properties based on any objective criteria. Another difference of our paper with these techniques is that they focus on the prevention of code decay during an iterative development process whereas we aim at providing decision aid and improving early OOAD models.

Refactoring [12] and reengineering [11] 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 [22]) when so-called "bad-smells" (e.g., a god class) have been identified (e.g., using metrics [18]). Although some refactorings [12, 22] and reengineering patterns [11] change class responsibility assignment, this is not the main objective of those activities, 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 of specific anti-patterns, and does so at the model level during early life-cycle phases. It is therefore more general in the sense that it will address a larger number of class responsibility assignment problems.

Although the Strength Pareto (SPEA2) approach has been recently introduced in [33], there are several applications of the technique already reported [4, 16, 21, 25].

3. Quality Measurement

3.1 Basic Definitions

The information we are using to optimize the domain 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 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 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 SIM(m) and PIM(m), respectively, 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 *m'* can be invoked on any 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 LSIM, or local SIM[1]. A "*" appended to the above set names denotes *indirect* accesses, invocations, or dependencies.

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 $m_2 \in PIM(m_1)$, 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, generalization, and usage dependency relationships [19]. 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. Association ends are however only accounted for in cohesion measures (see below). 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:

$(a \in AR \circ LSIM^*(m) \vee a \in AER \circ LSIM^*(m)) \wedge \exists c \in C, m \in M(c) \wedge (a \in A(c) \cup AE(c))$

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 UML models [24], 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 [7]. For example, sequence diagrams tell us what methods can invoke other methods at run-time, OCL operation 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 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 [5, 6]. Using these frameworks we selected three coupling measures based on the dependencies defined previously. 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 4. 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

$$MAI(c_1, c_2) = \bigcup_{m \in M(c_1)} \{(m,a) \mid a \in A(c_2) \wedge DMA(m,a)\}$$

Definition 5. Set of Method–Method Interactions (MMI)

For two classes $c_1 \in C$ and $c_2 \in C$, the set of Method Method Interactions 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

[1] $m' \in LSIM(m) \Leftrightarrow m' \in SIM(m) \wedge \exists c \in C, m \in M(c) \wedge m' \in M(c)$

[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 will translate into more method-method and method-attribute dependencies, and is therefore indirectly accounted for.

classes do not have any common ancestor (denoted *Others(c)* for any class *c*).

Definition 6. Method–Attribute Coupling (MAC)

For a given class $c_1 \in C$, Method–Attribute Coupling $MAC(c_1)$ counts all MAI from class 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 $c_1 \in C$, Method–Method Coupling $MMC(c_1)$ counts all MMI from class 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 $c_1 \in C$, Method–Generalization Coupling $MGC(c_1)$ counts all MMI and MAI from class c_1 to classes that are in the same generalization as c_1 but are not ancestors of c_1:

$$MGC(c_1) = \sum_{c_2 \in OthersGen(c_1)} |MMI(c_1, c_2)| + |MAI(c_1, c_2)| .$$

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, consider the method-based dependencies defined above (DMA, DMM, and LR). 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 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, 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 [5]. 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 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 [5], 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 $c \in C$, the set of cohesive interactions $CI(c)$ is equal to 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

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: $RCI(c) = |CI(c)| / |CI_{max}(c)|$.

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 [5], 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:

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

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

Definition 12. 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.)

$$TCC(c) = 2\frac{\left|\left\{(m_1,m_2) \mid m_1,m_2 \in M(c) \wedge m_1 \neq m_2 \wedge cu(m_1,m_2)\right\}\right|}{|M(c)|\left(|M(c)|-1\right)}$$

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 five measures capturing different and complementary aspects of coupling and cohesion.

4. Change Model

As stated above, 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 assignment, it is necessary to define the search space by determining possible changes to the model.

4.1 Methods, Attributes, and Association Ends

Since the search is based on existing class diagram information 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. 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, attributes, and association ends 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. In particular, the 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 grouped together with the association end. 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.

4.2 Classes and Relationships

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 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 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, association ends, 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. (We already mentioned such a constraint in Section 4.1.)

5. Multi-Objective GA (MOGA)

The objective of our search is to optimize the coupling and cohesion of a given class diagram based on five distinct measures (Section 3.2). However, in order to address those five 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) [32]. Although a single objective optimization problem may have a unique optimal solution, MOPs present a possibly large 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 [32] as follows, where in our context fitness functions are coupling and cohesion measures and decision variables correspond to set cardinalities involved in our measures:

Definition 13. Multi-objective Problem (MOP)

A MOP solution minimizes the components of a vector of fitness functions $F(\vec{x})$, where $\vec{x}$ is an n-dimensional decision variable vector $(\vec{x} = x_1,...,x_n)$ for some universe Ω. Formally, a MOP minimizes $F(\vec{x}) = (f_1(\vec{x}),..., f_k(\vec{x}))$ subject to constraints $g_i(\vec{x}) \le 0, i = 1,...,m, \vec{x} \in \Omega$.

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 14. 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., $\forall i \in \{1,...,k\}, u_i \le v_i \wedge \exists i \in \{1,...,k\} : u_i < v_i$.

Definition 15. Pareto Optimal Set

For a given MOP $F(\vec{x})$, the Pareto optimal set (P^*) is

$$P^* = \{\vec{x} \in \Omega \mid \neg\exists \vec{x}' \in \Omega : F(\vec{x}') \prec F(\vec{x})\}$$

Definition 16. Pareto Front

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

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 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.

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, +∞], 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 [3]. 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) [3].

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 not always easy to apply. 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 [26] was the first one proposed but its performance has been shown to be suboptimal compared to more recent alternatives such as NSGA [30]. NSGA itself has been shown to be outperformed by SPEA [34]. More recently, improvements to both NSGA and SPEA have been proposed: NSGA-II [10] and SPEA2 [33], which have been shown to produce similar results on some MOPs [33]. The time complexity of NSGA-II is better than SPEA2, but [33] 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 [33] 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 [33]. In Step 2, each individual *i* in both the external (archive) set $\overline{P}_t$ and the population P_t is assigned a strength value *S(i)*, representing the number of solutions it dominates. Based on *S(i)*, the raw fitness *R(i)* is then calculated for each individual *i* [33]:

$$R(i) = \sum_{j \in P_t \cup \overline{P}_t,\, j \succ i} S(j) \quad \text{where} \quad S(i) = |\{ j \mid j \in P_t \cup \overline{P}_t \wedge i \succ j \}| .$$

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.

```
Inputs: N     (population size)
        N̄     (archive size)
        T     (maximum number of generations)
Output: A     (non-dominated set)

Step 1—Initalization:
Generate an initial population P0, create an empty archive
(external set) P̄0 = ∅. Set t=0.
Step 2—Fitness Assignment:
Calculate fitness values of individuals in Pt and P̄t.
(Described in Section 5.2 as function F())
Step 3—Environmental Selection:
Copy all non-dominated individuals in Pt and P̄t to P̄t+1.
If size of P̄t+1 exceeds N̄ then reduce P̄t+1 by means of
clustering, otherwise if size of P̄t+1 is less than N̄ then
fill P̄t+1 with dominated individuals in Pt and P̄t. (See
description in Section 5.2.)
Step 4—Termination:
If t≥T then set A to the set of decision vectors
represented by the non-dominated individuals in P̄t+1.
Stop.
Step 5—Mating Selection:
Perform the genetic algorithm selection operator on P̄t+1
in order to fill the mating pool.
Step 6—Variation:
Apply recombination and mutation operators to the mating
pool and set Pt+1 to the resulting population. Increment
generation counter (t=t+1) and go to Step 2.
```

Figure 1 SPEA2 overall algorithm (from [33])

SPEA2 uses the kth Nearest Neighbor technique [33]. 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 [21]). 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 σ_i^k, where $k = \sqrt{N + \overline{N}}$. The density value *D(i)* corresponding to individual *i* is defined by: $D(i) = 1/\left(\sigma_i^k + 2\right)$.

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 solutions[3] [34]. 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: $\overline{P}_{t+1} = \{ i \mid i \in P_t \cup \overline{P}_t \wedge F(i) < 1 \}$.

If the external set fits exactly into the archive ($|\overline{P}_{t+1}| = \overline{N}$) then the environmental selection set is complete. Otherwise, either the archive is too large ($|\overline{P}_{t+1}| > \overline{N}$) or too small ($|\overline{P}_{t+1}| < \overline{N}$). If the archive is too small, then the best $\overline{N} - |\overline{P}_{t+1}|$ individuals in the previous

[3] Boundary solutions are solutions on the Pareto front that show extreme values for one or more fitness functions.

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 $\overline{P}_{t+1}$ until $|\overline{P}_{t+1}| = \overline{N}$. 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 chromosome 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 assessed class diagram is assigned a gene within the chromosome. The length of the chromosome will therefore be equal to the number of class members. 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 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 (represented by the first gene) belongs to the 10th class and the 2nd class member (represented by the second gene) to the 12th class. Using this representation, it is impossible to have an empty class represented in the chromosome.

Determining the ideal population size for a GA is challenging but important [2]. For traditional GAs a variety of adequate population sizes have been suggested [15]: some recommend a range between 30 and 80 [14], while others suggest a smaller population size, around 20 and 30 [27]. 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 [20, 33]. We follow these suggestions and use a population size of 64 individuals per objective.

The archive size also has an important effect on the performance of the MOGA. In [20] 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, then 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 [21, 25, 33]. To keep computation time within reasonable bounds, we therefore set the archive size to half 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 [20]. 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 [29]: 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 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 / *length*.

The crossover rate is another determining factor in the performance of the GA. 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 [15]. De Jong [9] concluded that a desirable crossover rate for a traditional GA should be about 60%. Grefenstette et al. [14] 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 [13] 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 [13] as this is simple and used in both SPEA and SPEA2 where it seems to work fine based on existing case studies [33]. In terms of mutation operator, the most obvious mutation is to randomly change the class that a given attribute or method is assigned to, assuming equal probabilities for all classes. This would be done on a gene by gene basis. When it is determined that a gene's method or attribute should be moved into a different class, it is reassigned to a different class randomly, with an equal probability to introduce a new class as to assign it to any other existing class.

For the case study discussed below, we implemented SPEA2 using the Java Genetic Algorithm Package framework [1], which is available under the LGPL and Mozilla Public License.

6. Case Study

The goal of our case study, at a high level, is to determine whether our GA can help improve an

analysis/domain class diagram from the point of view of class responsibility assignments. To assess this in a objective manner, we need to know what should be the optimal assignment of the model being improved. This is why instead of using an existing, imperfect domain model as a case study, we decided to investigate whether our GA can fix known, sub-optimal responsibility assignments of various types and scale resulting from modifying a known, satisfactory model. Second, in order to get additional insights in terms of the scalability of the approach, it is also important to know whether fixing a complex problem is significantly more expensive (time-wise) than fixing simpler ones. Third, we want to assess whether our implementation of SPEA2 can return a reasonable, manageable number of solutions to the user. These last two aspects will tell us whether our approach can scale up to large, significantly suboptimal models. To the extent possible, these questions have to be answered in a systematic, controlled, and objective manner. To do this, we need to select a domain model which we consider correct (or optimal) with respect to class responsibility assignment. Then we need to devise a variety of changes, of various complexity levels, to be applied to the optimal domain model, thereby generating sub-optimal class responsibility assignments. The goal of the case study is then to evaluate whether the GA can return to the known, original (and optimal) assignment or some other, similar acceptable solution (there is usually no unique optimal analysis model).

We therefore selected the ARENA system [7] 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 (optimal) 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). Although of modest size, the domain model is not trivial[4] and contains 14 classes and 152 class members (methods, attributes, and association ends). Of these 152 class members, 49 cannot be moved by our GA as they are overridden or implemented through generalization relationships. The 42 remaining elements are grouped in 14 groups with the association ends they manipulate. The search space, assuming the number of classes does not change, is therefore all the possible assignments of movable (grouped) class members (i.e., 152-49-(42-14)=75) to 14 classes and its size is therefore considerably large: 14^{75}. An analysis of this 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 measurement values for this model, which represent the baseline on which we want to improve, are: 99.0 (method–method coupling), 5.0 (method–attribute coupling), 0.0 (method generalization coupling), 0.20 (ratio of cohesive interactions), and 0.24 (tight class cohesion).

We devised several suboptimal modifications to the original ARENA analysis model. These modifications were applied to the original model one at a time (representing relatively simple sub-optimal assignments), as well as all together to form a larger, more complex sub-optimal assignment. They were selected to involve attributes, association ends, and methods, and to involve both new and existing classes. Each modified ARENA analysis model was then optimized using our implementation of the SPEA2 algorithm. We present below three representative modification examples, and their combination, and discuss them in detail. For each of those modifications we also analyzed the number of good analysis solutions across generations.

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 most 15 for method–attribute coupling, 105 for method–method coupling, 0 for method generalization coupling, and at least 0.18 for ratio of cohesive interactions, and 0.2 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. The goal was also to forbid coupling within generalization hierarchies: between ancestors and descendants (as this does not make sense) and between siblings (as this is not common and not present in the models to be optimized, there is no reason to introduce any). 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.

[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 152 class members.

Our SPEA2 implementation was run on the modified analysis models for 200 generations for all four 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 analysis model. Running on a Pentium 4 3.0GHz processor with 1GB of RAM, the execution time (for 200 generations) ranged from 46 to 55 minutes[5].

Change 1 consists in moving three methods, without their supporting attributes, into a new class. The negative effect on the analysis model 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. Since these methods only manipulate the reference corresponding to this association, we also indicate to the algorithm to move these class members as a single group as they conceptually belong together (see Section 4.1)[6]. Further details regarding the description of those three changes and the original class diagrams are provided in Appendix. Change 4 is to apply all three abovementioned changes together.

6.1 Initial Results

Table 1 shows, for each change, the number of solutions in the archive that are within-range (e.g., 7 for Change 1 after 100 generations) and the percentage of those solutions that are identical or equivalent to the original solution (e.g., 14% for Change 1 after 100 generations)[7].

	Number of Generations			
	50	100	150	200
Change 1	12, 0%	7, 14%	10, 0%	12, 33%
Change 2	12, 50%	13, 88%	13, 100%	18, 33%
Change 3	19, 37%	17, 76%	7, 100%	4, 75%
Change 4	2, 0%	11, 27%	10, 40%	13, 15%

Table 1 # Within Range, % Equivalent Solutions

Table 1 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. Because of this small number of within range solutions, it is not surprising to observe significant fluctuation in terms of percentage (e.g., the percentages for Change 1 across the four generations reported are: 0, 14, 0, and 33). Those fluctuations are due to the clustering (Section 5.2) that removes some non-dominated solutions to keep the archive at a specific size. Future work will have to investigate ways to further 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 (e.g., from generation 150 to 200 for Change 4). After 100 generations, a good percentage of the within range solutions are likely to be good alternative analysis models 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. Another interesting result is that the GA seems to recover from Changes 2 and 3 more easily than from changes 1 and 4 (higher percentages of identical or equivalent solution). This is further illustrated by Table 2 that shows, for Change 4, the number of equivalent solutions that recover from the individual changes at different generations: At generation 100, among the 11 within range solutions (Table 1), only 3 and 2 contain a fix to Change 1 and 4, respectively, whereas all of them fix Change 2.

	Number of Generations			
	50	100	150	200
Change 1	0	3	5	2
Change 2	1	11	10	13
Change 3	1	10	9	13
Change 4	0	2	4	2

Table 2 Recovering from individual changes

Let us now look in more details at the evolution process across generations.

6.2 Detailed results for Change 1

For Change 1, Table 3 shows the results of the algorithm at 50, 100, 150 and 200 generations as well as the original coupling and cohesion measures for the modified model. It also indicates the time it took to reach these generations (14 minutes for 50 generations, 55 minutes for 200 generations), and the best, worst, and average coupling and cohesion values for the solutions that were within-range in the archive (we do not show those values for MGC as within range solutions all have value 0 for this measure). From Table 3, we can see 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 thus confirming the results in Table 1 with respect to coupling and cohesion.

[5] We identified that this performance is mostly driven by the fitness evaluation, specifically the time taken to determine local indirect dependencies, which must be performed each time the cohesion metrics are evaluated.

[6] This is an example of user input that was discussed in Section 4.3.

[7] Note that we tried other population sizes than 64 per objective: with a size of 80 per objective, we obtained numbers of within range solutions equivalent to what we report here (though at a higher cost, time-wise); with a size of 30 per objective, we obtained too few within range solutions to draw any useful conclusion.

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, 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 the GA can provide good, interesting alternatives to the class responsibility assignment problem.

Initial Model			
Method - Attribute Coupling:	11.0		
Method - Method Coupling:	105.0		
Ratio of Cohesive Interactions:	0.17		
Tight Class Cohesion	0.22		
	Average	**Best**	**Worst**
50 Generations (14 min)			
Method - Attribute Coupling:	11.16	8.00	15.00
Method - Method Coupling:	98.58	91.00	105.00
Ratio of Cohesive Interactions:	0.41	0.51	0.30
Tight Class Cohesion:	0.40	0.48	0.29
100 Generations (28 min)			
Method - Attribute Coupling:	5.57	5.00	7.00
Method - Method Coupling:	101.00	94.00	105.00
Ratio of Cohesive Interactions:	0.52	0.56	0.46
Tight Class Cohesion:	0.55	0.59	0.51
150 Generations (43 min)			
Method - Attribute Coupling:	9.8	5.00	15.00
Method - Method Coupling:	98.00	93.00	103.00
Ratio of Cohesive Interactions:	0.47	0.55	0.42
Tight Class Cohesion:	0.52	0.63	0.44
200 Generations (55 min)			
Method - Attribute Coupling:	8.00	5.00	12.00
Method - Method Coupling:	101.25	95.00	105.00
Ratio of Cohesive Interactions:	0.51	0.60	0.40
Tight Class Cohesion:	0.58	0.64	0.51

Table 3 Summary of Change #1 Results

Figure 2 shows the number of within range solutions returned per generation for Change 1. The number of within range solutions in the archive first rises fairly quickly, as the archives fills up, then plateaus at around generation 33 (when the archive is full). Afterwards, the number of within range solutions in the archive stabilizes and Table 3 shows that the average metrics for the solutions within the archive remain close to one another. The search of within range solutions stabilizes.

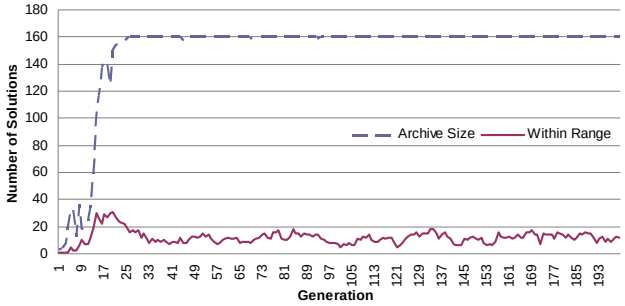

Figure 2 Within Range Solutions by Generation for Change #1

6.3 Detailed results for Change 2

Table 4 shows the results for Change 2 in the same form as for Change 1. 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. We observe the same class responsibility assignment alternatives as those discovered by the GA for Change 1.

Initial Model			
Method - Attribute Coupling:	10.0		
Method - Method Coupling:	99.0		
Ratio of Cohesive Interactions:	0.18		
Tight Class Cohesion	0.24		
	Average	**Best**	**Worst**
50 Generations (11 min)			
Method - Attribute Coupling:	7.66	5.00	10.00
Method - Method Coupling:	97.33	91.00	102.00
Ratio of Cohesive Interactions:	0.43	0.50	0.35
Tight Class Cohesion:	0.47	0.55	0.37
100 Generations (23 min)			
Method - Attribute Coupling:	8.87	5.00	14.00
Method - Method Coupling:	97.00	89.00	104.00
Ratio of Cohesive Interactions:	0.42	0.52	0.27
Tight Class Cohesion:	0.50	0.59	0.40
150 Generations (36 min)			
Method - Attribute Coupling:	10.27	5.00	14.00
Method - Method Coupling:	94.88	86.00	103.00
Ratio of Cohesive Interactions:	0.42	0.56	0.27
Tight Class Cohesion:	0.49	0.59	0.39
200 Generations (48 min)			
Method - Attribute Coupling:	9.58	5.00	15.00
Method - Method Coupling:	95.91	91.00	104.00
Ratio of Cohesive Interactions:	0.44	0.55	0.28
Tight Class Cohesion:	0.49	0.58	0.39

Table 4 Summary Change #2 Results

Figure 2 shows a pattern similar to Figure 3 and for the same reasons. One difference though is that at generations 9 – 10, the number of solutions in the archive reaches a peak and then drops in generation 11. (This also happened in Figure 2, although to a lesser extent, around generation 20.) This drastic drop is due to a specific individual, which is better than a large number of archive individuals, that is discovered by the GA and therefore results in these individuals being removed from the archive as they become dominated.

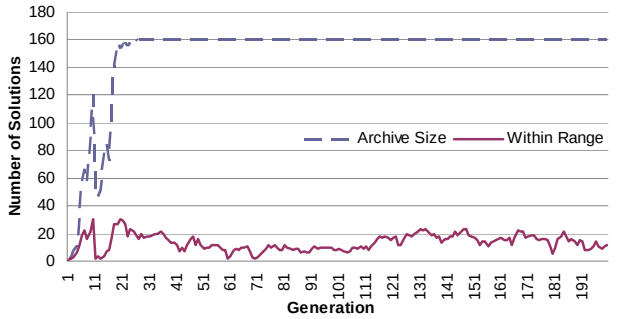

Figure 3 Within Range Solutions by Generation for Change #2

6.4 Detailed results for Change 3

Table 5 shows the results for the third change, in the same format as the previous two changes, and Figure 4 shows the number of within range solutions by generation for this change. The conclusions we can draw are similar to what we observed for the two previous changes.

Initial Model			
Method - Attribute Coupling:	8.0		
Method - Method Coupling:	104.0		
Ratio of Cohesive Interactions:	0.25		
Tight Class Cohesion:	0.30		
	Average	**Best**	**Worst**
50 Generations (11 min)			
Method - Attribute Coupling:	7.78	5.00	12.00
Method - Method Coupling:	97.05	90.00	102.00
Ratio of Cohesive Interactions:	0.46	0.51	0.35
Tight Class Cohesion:	0.51	0.59	0.41
100 Generations (23 min)			
Method - Attribute Coupling:	7.17	5.00	15.00
Method - Method Coupling:	99.52	87.00	105.00
Ratio of Cohesive Interactions:	0.49	0.56	0.35
Tight Class Cohesion:	0.56	0.60	0.47
150 Generations (36 min)			
Method - Attribute Coupling:	7.00	6.00	9.00
Method - Method Coupling:	99.28	92.00	105.00
Ratio of Cohesive Interactions:	0.47	0.56	0.38
Tight Class Cohesion:	0.51	0.58	0.46
200 Generations (48 min)			
Method - Attribute Coupling:	10.00	6.00	15.000
Method - Method Coupling:	98.75	94.00	102.00
Ratio of Cohesive Interactions:	0.44	0.50	0.41
Tight Class Cohesion:	0.55	0.58	0.49

Table 5 Summary of Change #3 Results

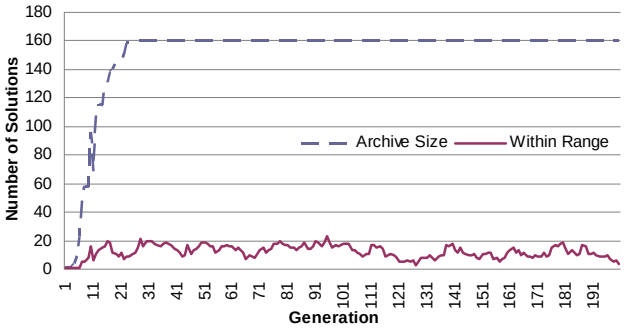

Figure 4 Within Range Solutions by Generation for Change #3

6.5 Detailed results for Change 4 (all three changes together)

Table 6 shows the results for the fourth change, where the first three changes are used together, in the same form as the previous three changes, and Figure 5 shows the number of within range solutions per generation for this change. The conclusions we can draw are similar to what we observed for the previous changes. One difference though is that, as expected, it takes more generations (approximately 80 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, the same within range solutions are eventually returned to the user.

Initial Model			
Method - Attribute Coupling:	19.0		
Method - Method Coupling:	110.0		
Ratio of Cohesive Interactions:	0.21		
Tight Class Cohesion:	0.27		
	Average	**Best**	**Worst**
50 Generations (10 min)			
Method - Attribute Coupling:	13.00	12.00	14.00
Method - Method Coupling:	101.00	100.00	102.00
Ratio of Cohesive Interactions:	0.39	0.41	0.36
Tight Class Cohesion:	0.42	0.44	0.40
100 Generations (21min)			
Method - Attribute Coupling:	11.18	7.00	15.00
Method - Method Coupling:	98.90	93.00	104.00
Ratio of Cohesive Interactions:	0.49	0.54	0.40
Tight Class Cohesion:	0.49	0.59	0.39
150 Generations (34min)			
Method - Attribute Coupling:	9.60	6.00	12.00
Method - Method Coupling:	99.10	92.00	104.00
Ratio of Cohesive Interactions:	0.48	0.59	0.39
Tight Class Cohesion:	0.53	0.59	0.48
200 Generations (46min)			
Method - Attribute Coupling:	9.92	6.00	13.00
Method - Method Coupling:	96.46	91.00	104.00
Ratio of Cohesive Interactions:	0.45	0.56	0.37
Tight Class Cohesion:	0.47	0.56	0.37

Table 6 Summary of Change #4 Results

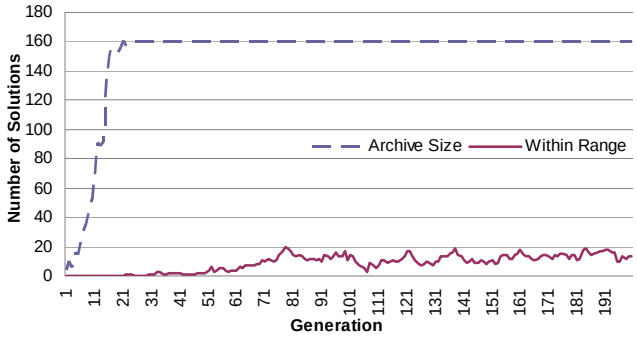

Figure 5 Within Range Solutions by Generation for Change #4

As a basis of comparison, we attempted to fix Change 4 by using a random search. We run the search ten times, using the same number of generations as in our GA, and thus leading to the generation of 640,000 solutions. None of these solutions contained a fix to Change 4. If there

was any doubt given the size of the search space, we can therefore safely conclude that our search problem is not trivial and cannot be simply addressed by a random search.

6.6 Discussion

We have seen above that similar patterns emerged for the four changes regarding the number of within range solutions returned per generation and the change in coupling and cohesion across generations. 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 using the clustering method. This method favors boundary individuals, so it works to maintain a wide spread of solutions across the search space to maintain diversity. This however has a cost as some (perhaps good) solutions are discarded. Then, by restricting the range of values for the returned solutions, we only sample a small section of the overall search space thus sometimes resulting in a decrease of within range solutions after reaching a maximum.

Many of the solutions obtained after 100 generations, while containing the original assignment for the class members that were moved, also included other changes to the original class diagram resulting into better coupling and cohesion values for three of the measures: recall the discussion regarding classes `Tournament`, `League`, and `TournamentStyle`. Our approach can therefore recover from changes of varying spread and magnitude (three simple changes and one larger one), and can also suggest good alternatives to the original model. In our case study running 100 generations takes between 21 and 34 minutes, which is reasonable from a practical standpoint, and that the number of within range solutions is manageable (between 7 and 17), allowing a designer to consider them all. Last, optimizing a domain model that is far from being optimal (Change 4) takes more generation to converge towards a set of acceptable solutions but eventually returns the same set of solutions as for models that are closer to the optimal one (Changes 1 to 3). This suggests our GA can cope with complex fixes though it will take more generations to find them.

7. Conclusion

This paper presents an approach to aid with class responsibility assignment in object-oriented analysis / domain models, 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 measures 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 selected measurements characterizing the quality of a model, 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 domain models representing optimal trade-offs. 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 mistakes of varying spread and magnitude were introduced in a correct domain model, they could be corrected and a variety of acceptable solutions could be obtained within a reasonable number of generations and time. This demonstrates that the GA is able to fix a variety of artificially seeded class assignment problems, which is a significant step towards validating our approach. One open issue though is to help prioritize or select the alternative solutions proposed by the GA in a cost effective manner. The current solution is to define an acceptable value range for coupling and cohesion measurement which seems, on our case study, to be constraining enough to limit the number of solutions proposed to the user. A significant number of these solutions then turn out to be equivalent to the target optimal model.

Ideally, the next validation step is to apply our strategy in the context of a real and imperfect domain model. The difficulty with such a validation approach is that we would not have any objective basis of comparison as the optimal model would be undetermined at the time of the experiment and assessing the improvements proposed by the GA would then be very subjective. 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 devise ways for the designer to efficiently interact with the GA. Last, we want to investigate ways to avoid the loss of possibly good solutions when the archive is truncated during clustering.

8. Acknowledgments

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

9. References

[1] JGAP: Java Genetic Algorithms Package, 2006. http://jgap.sourceforge.net/.

[2] Atallah M. J., *Algorithms and Theory of Computation Handbook*, CRC Press, 1999.

[3] 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.

[4] Bleuler S., Braek M., Thiele L. and Zitzler E., "Multiobjective Genetic Programming: Reducing Bloat Using SPEA2," *Proc. IEEE Congress on Evolutionary Computation*, 1, pp. 536-543, 2001.

[5] 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.

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

[7] Bruegge B. and Dutoit A. H., *Object-Oriented Software Engineering*, Prentice Hall, 2nd Edition, 2004.

[8] 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.

[9] De Jong K. A., "Learning with Genetic Algorithms: An Overview," *Machine Learning*, 3 (3), pp. 121-138, 1988.

[10] 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.

[11] Demeyer S., Ducasse S. and Nierstrasz O., *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2003.

[12] Fowler M., *Refactoring - Improving the Design of Existing Code*, Addison Wesley, 1999.

[13] Goldberg D. E., *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison Wesley, 1989.

[14] Grefenstette J. J. and Cobb H. G., "Genetic Algorithms for Tracking Changing Environments," *Proc. International Conference on Genetic Algorithms*, pp. 523-530, 1993.

[15] Haupt R. L. and Haupt S. E., *Practical Genetic Algorithms*, Wiley-Interscience, 1998.

[16] Hiroyasu T., Nakayama S. and Miki M., "Comparison Study of SPEA2+, SPEA2 and NSGA-II in Diesel Engine Emissions and Fuel Economy Problem," *Proc. IEEE Congress on Evolutionary Computation*, 1, pp. 236-242, 2005.

[17] Kleppe A., Warmer J. and Bast W., *MDA Explained*, Addison-Wesley, 2003.

[18] Lanza M. and Marinescu R., *Object-Oriented Metrics in Practice*, Springer, 2006.

[19] Larman C., *Applying UML and Patterns*, Prentice-Hall, 3rd Edition, 2004.

[20] Laumanns M., Zitzler E. and Thiele L., "On The Effects of Archiving, Elitism, an Density Based Selection in Evolutionary Multi-objective Optimization," *Proc. Int. Conf. on Evolutionary Multi-Criterion Optimization*, 2001.

[21] 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.

[22] Mens T. and Tourwe T., "A Survey of Software refactoring," *IEEE TSE*, 30 (2), pp. 126-139, 2004.

[23] O'Keeffe M. and O Cinneide M., "Towards automated design improvement through combinatorial optimization," *Proc. Workshop on Directions in Software Engineering Environments*, 2004.

[24] Pender T., *UML Bible*, Wiley, 2003.

[25] 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.

[26] Schaffer J. D., "Multiple Objective Optimization with Vector Evaluated Genetic Algorithms," *Proc. Int. Conf. on Genetic Algorithms and their Applications*, 1988.

[27] 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. Int. Conf. on Genetic Algorithms and Their Applications*, 1989.

[28] Seng I., Stammel J. and Burkhard D., "Search-based determination of refactorings for improving the class structure of object-oriented systems," *Proc. Conf. on Genetic and Evolutionary Computation*, 2006.

[29] 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.

[30] Srinivas N. and Deb K., "Multiobjective optimization using nondominated sorting in genetic algorithms," *Journal of Evolutionary Computation*, 2 (3), pp. 221-248, 1995.

[31] Svetinovic D., Berry D. M. and Godfrey M., "Concept identification in object-oriented domain analysis: Why some students just don't get it," *Proc. Int. Conf. on Requirements Engineering*, 2005.

[32] Van Veldhuizen D. A. and Lamont G. B., "Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art," *Evolutionary Computation*, 8 (2), pp. 125-147, 2000.

[33] 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.

[34] Zitzler E. and Thiele L., "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach," *IEEE Trans. on Evolutionary Computation*, 3 (4), 1999.

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 6.

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 7. 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.

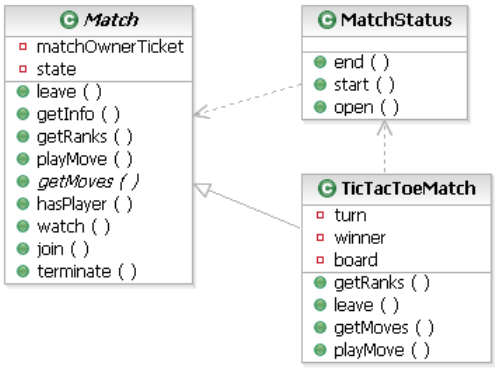

Figure 7 Change #1 to ARENA Design

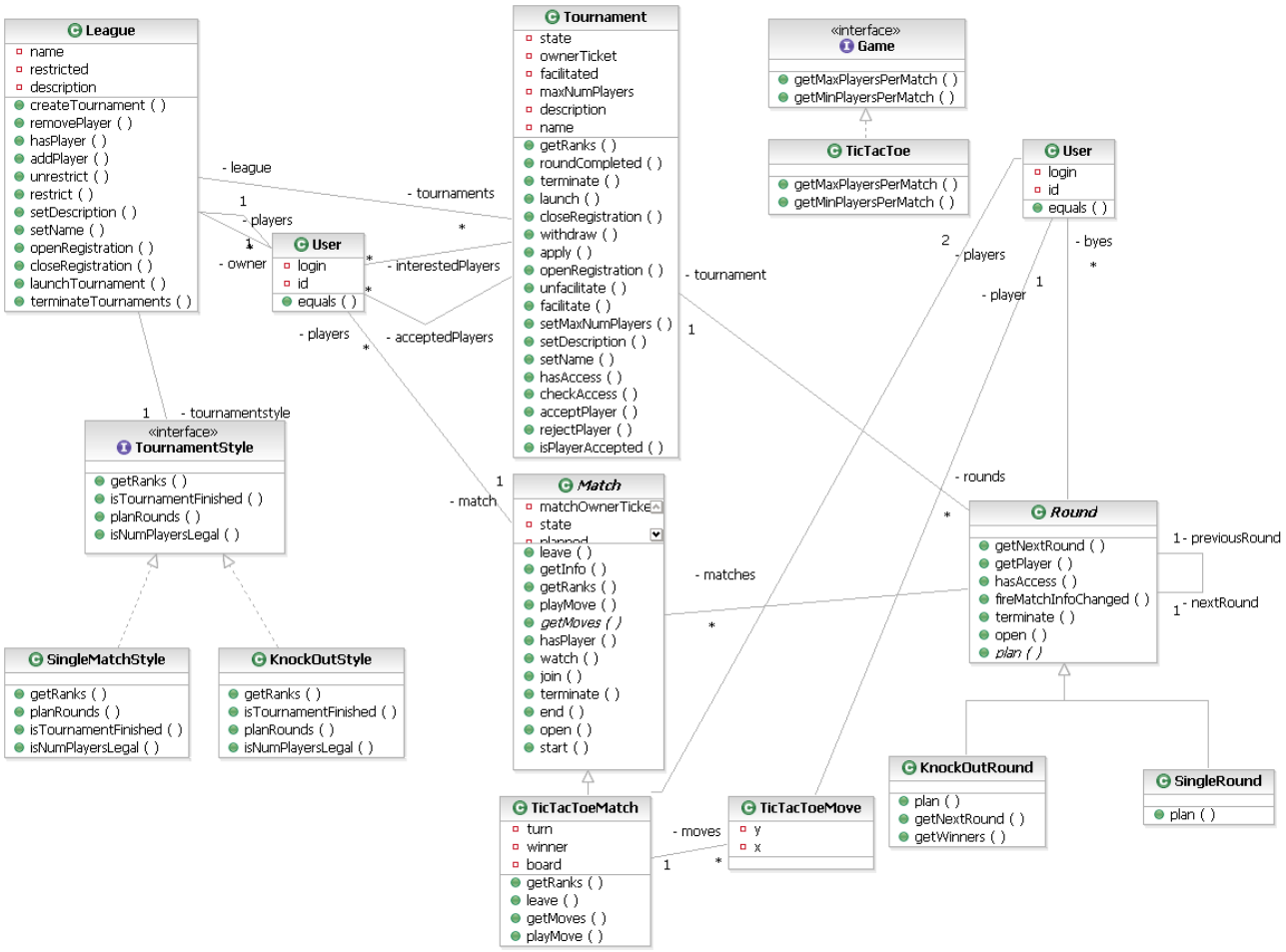

Figure 6 Original ARENA System (class User appears twice for layout purposes)

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. A diagram illustrating the change is shown in Figure 8.

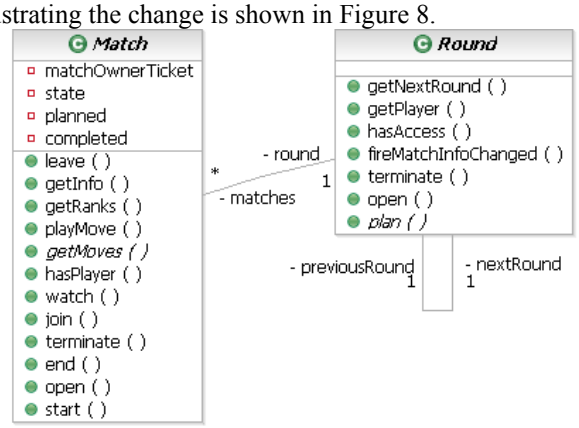

Figure 8 Change #2 to ARENA Design

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. Figure 9 shows the change to the original ARENA design.

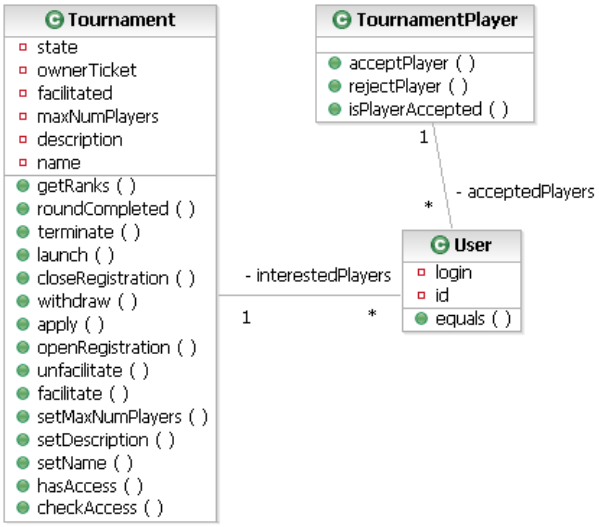

Figure 9 Change #3 to ARENA Design