Multi-objective Genetic Algorithm to Support Class Responsibility Assignment

By:

Michael Bowman

A thesis submitted to the Faculty of Graduate Studies and Research

In partial fulfillment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

July 2007

Your file Votre référence
ISBN: 978-0-494-33639-7
Our file Notre référence
ISBN: 978-0-494-33639-7

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 thesis, 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 thesis presents our approach in detail, 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.

ACKNOWLEDGEMENTS

First off I would like to give my sincere thanks and appreciation to Dr. Briand and Dr. Labiche for all of their support and guidance throughout my research. Their dedication and exemplary practices have been an inspiration and it has been an honour to work with them.

My warmest thanks to my family and friends for the support and encouragement they have given me. Without them, I would not have been able to get this far.

Last, but certainly not least, a thank you to everyone in the SQUALL lab for the mutual support and encouragement throughout my research work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF FIGURES

1 INTRODUCTION

1.1 Objective

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." [33] There is indeed evidence that this is hard to teach and apply (e.g., [51]). 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., [11]), they all rely on human judgment and decision making, primarily based on heuristics. In this thesis, 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) [29], 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.

1.2 Context

As mentioned above, our work takes place in the context of Model Driven Architecture/Development (MDD). In this thesis, 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 [33]. Future work will explore similar solutions for lower-level design class diagrams (Section 10.2). At lower levels of design, the fitness function will be much more complex, and as the purpose of this thesis

is to present the initial application of a multi-objective optimization technique, the simplest case is used.

1.3 Outline of Approach

Our approach is based on a multi-objective genetic algorithm [52], uses class coupling and cohesion measurement [7, 8], 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 from 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 thesis. 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 [7], 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 following figure outlines our approach to the improvement of software class design using a GA heuristic.

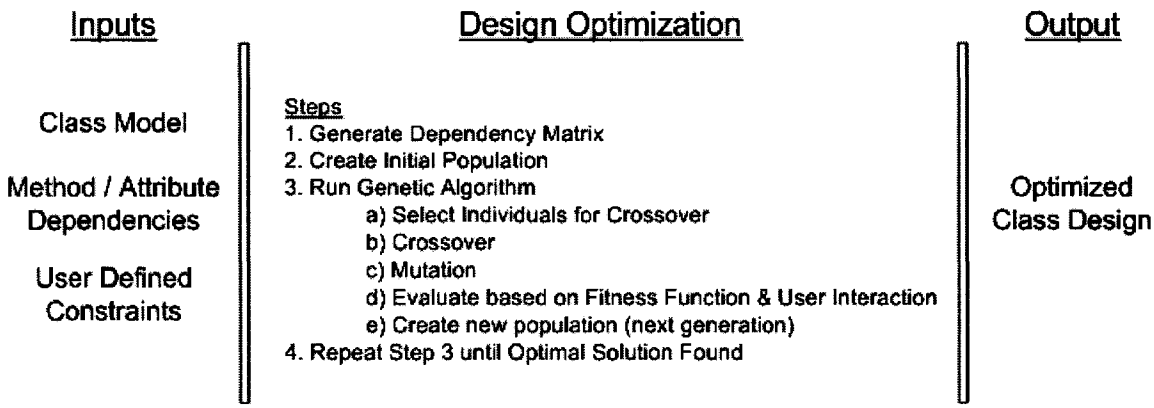


Figure 1 Overview of Design Optimization Process

1.4 Thesis Contributions

The main contributions of this thesis include: 1) Definition of a change model for applying a search-based software engineering technique to the class responsibility assignment problem to the PIM; 2) A formalization of class design optimization as a multi-objective problem; 3) A framework for applying a multi-objective evolutionary algorithm to the problem of class responsibility assignment.

1.5 Overview

The remainder of the thesis is organized as follows: Section 0 presents related works in the field of genetic algorithms and engineering design and optimization, as well as interactive genetic algorithms and creative evolutionary systems; Section 3 outlines the class responsibility assignment problem in more detail; Section 4 introduces the class model that will be used as an example throughout this thesis; Section 5 discusses the inputs required for the optimization, such as the dependency information and user defined constraints; Section 6 presents the change model that is used by the genetic algorithm on the class model in order to perform the search; Section 7 discusses the fitness function measures and the multi objective approach to the design improvement; Section 8 describes the formulation of the genetic algorithm in detail; Section 9 presents the implementation of the Evolutionary Software Design (ESD) tool, along with the results of several case studies; and finally Section 10 discusses future work and Section 11 concludes.

2 RELATED WORKS

2.1 Search Based Software Engineering

The application of a metaheuristic search technique, such as a genetic algorithm, to the field of software engineering is discussed in [13]. This paper discusses a number of possible search techniques (discussed in Section 2.2) as well as 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 [40] 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. The scope of this approach is expanded in [47], where the authors use a genetic algorithm to automatically determine potential refactorings of the system, not just an inheritance hierarchy. The paper uses a sum of weighted objectives once again that measures the coupling, cohesion, complexity and stability of the system. The algorithm in [47] 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 of this technique is to help prevent software decay, also known as design drift, from affecting the quality of the systems structure.

These two approaches both use a sum of weighted objectives to balance the various metrics. While this is clearly helpful, it can only take into account one possible, predetermined trade-off among objectives, whereas the Pareto based multi-objective algorithm (the Strength Pareto approach [57]) we use 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 approach with the techniques presented in [40, 47] 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.

Refactoring [21] and reengineering [19] 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 [36]) when so-called "bad-smells" (e.g., a god class) have been identified (e.g., using metrics [32]). Although some refactorings [21, 36] and reengineering patterns [19] 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.

2.2 Multi-objective Optimization Techniques

[13] discusses the application of a number of metaheuristic search techniques to search based problems in software engineering. Metaheuristic search techniques are a set of generic algorithms which are concerned with searching for optimal or near optimal solutions to a problem within a large multi-modal search space. [13] discusses a number of possible metaheuristics: hill climbing, simulated annealing, tabu search and genetic algorithms [24].

As discussed in [13], hill climbing, simulated annealing and tabu search are all local search heuristics, whereas genetic algorithms sample the global search domain. This makes genetic algorithms more expensive, but effective at search extremely complex fitness landscapes. Also, the local search techniques consider only one solution at a time, whereas a genetic algorithm is a population based approach. It evaluates a large number of possible solutions, in order to better explore the global search domain. Using a population based technique makes genetic algorithms well suited to the domain of multi-objective optimization, as they can evaluate a population of possible tradeoffs. Therefore, our thesis focuses on the application of an evolutionary algorithm to the problem of class design optimization, specifically a multi-objective genetic algorithm.

There are a number of techniques for optimizing multi-objective problems. [14] provides a survey of many evolutionary multi-objective optimization techniques, categorizing them into techniques which use an aggregate fitness function (e.g. weighted sum, goal attainment, ε-constraint), Pareto based approaches (e.g. MOGA, NSGA, NPGA), and non-Pareto based approaches (e.g. VEGA, game theory, weighted min-max). More recently, new Pareto based evolutionary algorithms have been introduced. They are the non-dominated sorting algorithm two (NSGA-II) [18], strength Pareto approach (SPEA) [58], and the strength Pareto approach two (SPEA2) [57]. A number of these multi-objective evolutionary techniques and fitness assignment schemes are considered for the class responsibility assignment problem, and we compare them in detail in Section 7.4.3 and 7.4.4.

2.3 Applications of evolutionary algorithms

The application of genetic algorithms to search based software engineering, and to software design, is one example of applying creative evolutionary systems. Creative evolutionary systems are defined [3] as a computer system that makes use of some aspect of evolutionary computation and is designed to:

- aid our own creative process, and / or
- Generate results to problems that traditionally require creative people to find solutions.

These creative evolutionary systems grew up in the field of evolutionary design, since problems such as software design are complex with large numbers of objectives, constraints and parameters. Design specifications will usually be a moving target as preferences change. [3] presents a number of examples of creative evolutionary systems, in areas such as electrical and civil engineering, artwork, etc. For example, in the area of circuit design, evolutionary algorithms have been shown to be capable of producing human-competitive results [31].

Another aspect of creative evolutionary systems is the idea of interactive genetic algorithms. These genetic algorithms are able, in some way, to receive feedback from the user during the course of the optimization. The algorithm then incorporates this feedback into the determination of the individual's fitness. In this manner the search can be guided towards designs and specific tradeoffs that are desirable to the user. [12] has applied this technique to plane truss design. Their approach, called an Intelligent Genetic Design Tool (IGDT), allows the designer to select from a number of solutions at each generation, and then incorporates the preference of the designer into subsequent generations. In this manner, the user is able to guide the search by influencing the fitness assignment of the algorithm.

Genetic algorithms themselves have been applied to a wide variety of engineering problems [22, 23], although typically using a sum of weighted objectives approach, rather than a Pareto based approach. Although the Strength Pareto (SPEA2) approach has been just recently introduced in [57], there are several applications of the technique [6, 28, 35, 44]. At present, we are not aware of any applications of the SPEA2 technique, or other multi-objective evolutionary algorithms, to problems in search based software engineering.

3 CLASS RESPONSIBILITY ASSIGNMENT PROBLEM

The class responsibility assignment problem is one that is encountered in the early stages of software design, and has a great impact on the overall design of the application [33]. However, the assignment of responsibilities to classes is a very difficult skill both to teach and to master [51]. Also, the quality of the resulting design is subjective, and highly dependent on the skill of the designer. Approaches to class responsibility assignment are subjective, relying upon human decision making, experience and judgment (e.g. [11]). While the assignment of responsibilities to classes is identified as a key aspect of object-oriented analysis and design, there is little to aid a designer at this early stage of design.

However, like a number of problems in software engineering, the class responsibility assignment problem is well suited for applying a search based technique [13] to aid the designer. [13] outlines four characteristics of problems suitable for a search based approach, specifically: a very large search space, no known efficient and complete solution, suitable fitness functions, and finally cheap generation of candidate solutions.

- Large search space: In our context, the size of the search space is the total number of possible class assignments that can be formed from the class members (methods, attributes and association ends, see Section 5.1.8). The search space consists of every valid design involving these classes, methods and attributes.

- No known efficient and complete solution: As discussed, the optimal assignment of responsibility to classes is subjective and based on the designer's judgment. While guidelines exist to aid in the design process, there is no one solution that guarantees the best possible design.

- Suitable fitness functions: While the overall quality of a design is ultimately subjective, there are known metrics that can be used to measure various aspects design quality [54]. These can be used to compare and measure the fitness of proposed designs in the search.

- Cheap generation of candidate solutions: The class responsibility assignment problem can be represented as a list of the classes to which the class members (methods, attributes and association ends) are assigned. This list can then easily be altered to generate new candidate solutions.

Given the above properties, the class responsibility assignment problem is a good candidate for applying a search based heuristic. The possible heuristics to apply are discussed in Section 2.1. Our approach optimizes the assignment of class members (Section 5.1.8) based on the dependencies between them (Section 5.1.10), in the early stages of UML design (the PIM).

One problem of applying the search-based heuristic is measuring the fitness of the candidate solutions. While many possible quality measures exist that could be used [54], most are not comparable with each other, and it is difficult to balance these quality metrics into a single fitness function. In addition to this, the final say in the optimum trade-off between these quality metrics should be left with the designer, not with the heuristic. In order to overcome these difficulties, we formulate the class responsibility assignment problem as a multi-objective problem. The fitness function will consider these objectives, and present the designer with a number of possible candidate solutions, each representing a different trade-off between the objectives. In order to determine the quality of the candidate solutions, complementary coupling and cohesion quality metrics will be used as the multiple objectives for the search.

Figure 2 shows in detail the search based approach that will be presented in this thesis.

In order to determine both the starting initial solution, and the dependencies that will be used to measure the fitness of the candidate solutions, several inputs are required. Our goal is to provide feedback to the designer during the analysis phase of the design. This means that all of the dependency information must be based on information gathered from the design at that stage. OCL contracts and sequence diagrams are used as the main source to determine the dependencies between class members. The current class design is needed to determine the class members and the initial starting point for the search.

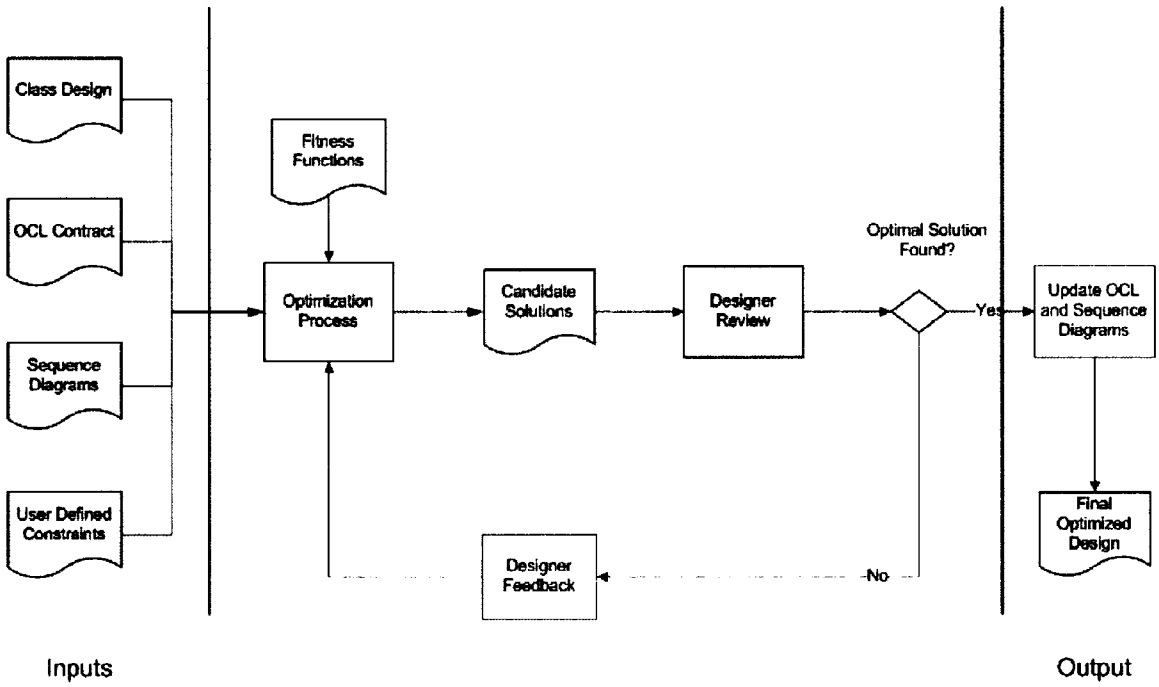


Figure 2 Search based approach for class responsibility assignment

These inputs are entered into the optimization process, and the heuristic takes over. The steps involved in the actual optimization process are outlined in Figure 1 in the introduction. It searches for improvements to the current design using coupling and cohesion design quality metrics. Using a multi-objective fitness evaluation, a number of candidate solutions are generated for the designer. The designer can then review the candidate solutions, and if she decides that an ideal solution has been found, can then stop the search process. Otherwise, the designer provides feedback to the optimization process based on the candidates presented so far. This can be in the form of new constraints on the design changes, or allowing the heuristic to improve the fitness of solutions containing certain properties. The optimization process then continues with these new constraints, and produces another set of candidate solutions. The process continues until an ideal solution is found.

Finally, once a candidate solution is found, the OCL contracts, sequence diagrams and other model information is brought up to date to match the new class diagram. This can be done automatically based on the dependency information and the original OCL contracts, sequence diagram and class design.

This thesis will cover the basics of the approach presented above. For the search, the input model is presented in Section 4, and covers all of the input required by the heuristic. The change model, which allows the heuristic to properly explore the search space, is presented in Section 6. Details regarding the fitness function are presented in Section 7, including the coupling and cohesion quality metrics used, and the evolutionary multi-objective fitness function presented. With the exception of the evolutionary fitness function information presented in Section 7.4, these sections are solution independent, and are not specific to the multi-objective genetic algorithm that we use. Therefore, the information presented in these sections of the thesis could be reused on another technique.

The remaining sections discuss the specific details of the multi-objective genetic algorithm. Section 8 presents the genetic algorithm in greater detail; and Section 9 presents a prototype tool developed, and the case studies that were performed. Thus, the information presented in Section 8, as well as Section 7.4 could only be reused if another type of genetic algorithm was to be applied to the class responsibility assignment problem.

Some of the areas of the optimization presented above are not within the scope of this thesis. These areas are highlighted red in Figure 2. The interactive aspect of the optimization process, where the designer can provide input to the heuristic, is part of the future work. As is the automated parsing and updating of the OCL contracts and sequence diagram information. All of the future work is discussed in Section 10.

4 SAMPLE MODEL

In order to aid in illustrating the various concepts and approaches presented in this thesis, a working example will be used. This section outlines the model, class members and dependencies that will be used to illustrate the techniques and approaches presented.

4.1 Class Diagram

The class diagram for the sample model is shown in Figure 3. This sample design was adapted from the Monopoly game model presented in [33]. The design represents the Monopoly design in [33] after iteration 2. It is a simple model, but includes all of the elements that are necessary for the discussion of the engineering optimization being performed on the design.

The main control class is `MonopolyGame`. It manages each of the rounds of the Monopoly program. The players, represented by the `Player` class, take turns moving around the board represented by the `Board` class. Each square is represented by a `Square` class, which is an abstract class for the actual squares. There are 3 types of squares included in the model, the "Go Square", the "Income Tax Square" and a regular square. Players are represented by Pieces on the Board. So the `Piece` class represents the player on the square. Finally, in order to move around the board, the players roll dice to determine how far that they move each turn. The `Die` class represents the dice used by the Players. The flow of the application is as follows: The `MonopolyGame` class gives each player a turn. The `Player` rolls the dice, and then moves their piece to the next square. The effects of the square are worked out, and then the turn passes to the next player, and the process repeats. The game continues until all of the turns are finished, and then it completes. Since this model is from an early iteration of the development, the model does not completely implement a Monopoly game however there is sufficient functionality to serve as an example in this paper.

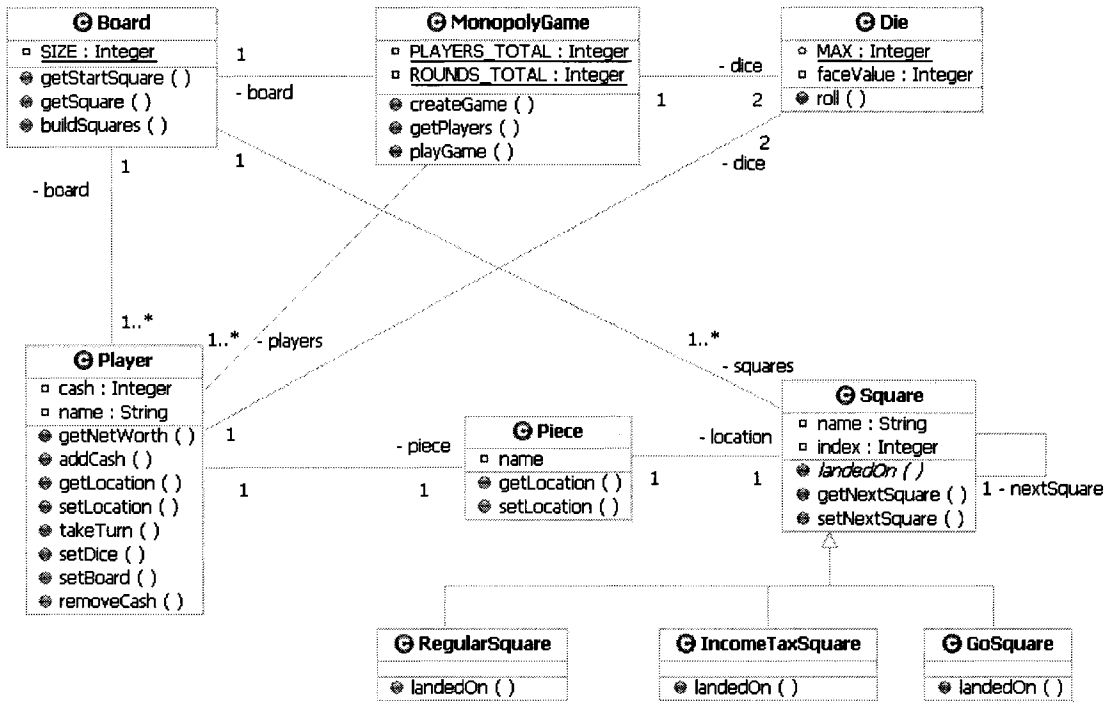


Figure 3 Sample Model: Monopoly Game

4.2 Model Dependencies

Further to the class diagram presented in Section 4.1, the class members and their dependencies must be known. The dependencies are used to determine the quality of the model, and discover possible improvements. Class members are defined formally in Section 5.1.8 and dependencies presented in Section 5.1.10.

The dependencies for the sample design are presented in Appendix A. The first chart (Figure 29), lists the class members that are present in the model, and assigns each an id value. The second chart is a matrix, which lists the id of the class members in the rows (i) and the columns (j). A value of 1 at [i,j] represents a dependency between the class member i and class member j. This matrix, referred to in this paper as the dependency matrix, will be discussed in full detail later in Section 7.1.2. The sample matrix is presented in Figure 30 in Appendix A. These dependencies were determined using the sequence diagrams in [33].

5 INPUT MODEL

This section outlines the input requirements of the class responsibility assignment search heuristic, as discussed in Section 1.3 and Section 3. This section is organized as follows: Section 5.1 will present the terminology and formalism for the class model; Section 5.2 discusses the class model information required from the user in order to perform the design improvement; and finally Section 5.3 will describe the user defined constraints on the model that can be provided.

5.1 Terminology and Formalism

In order to adequately define the model in terms of the dependencies between the class members, and the relationships between the classes, the formal definition of these concepts must first be presented. The following formalism was presented in [7, 8], and uses set theory to describe the system concepts.

5.1.1 System

Definition 1. System, classes and inheritance relationships

An object oriented *system* consists of a set of classes C. There can exist inheritance relationships between the classes such that for each class $c \in C$ let

- $-Parents(c) \subset C$ be the set of parent classes of class c
- $-Children(c) \subset C$ be the set of children classes of class c
- $-Ancestors(c) \subset C$ be the set of ancestor classes of class c
- $-Descendents(c) \subset C$ be the set of descendent classes of class c
- $-Siblings(c) \subset C$ be the set of sibling classes of class c (classes which share a common ancestor class with class c).

5.1.2 Methods

A class has a set of methods

Definition 2. Methods of a class

For each class $c \in C$, let M(c) be the set of methods of class c.

Methods in the system may be abstract or concrete and either inherited, overridden, or newly defined, public or non-public. All of these concepts are important for defining the dependencies between class members.

Definition 3. Declared and implemented methods

For each class $c \in C$ let:

- $M_I(c) \subseteq M(c)$ is the set of all implemented methods
- $M_D(c) \subseteq M(c)$ is the set of all declared (inherited) methods

where $M(c) = M_I(c) \cup M_D(c)$ and $M_I(c) \cap M_D(c) = \phi$

Definition 4. Inherited, overriding and new methods

For each class $c \in C$ let:

- $M_{INH}(c) \subseteq M(c)$ be the set of inherited methods of c.
- $M_{OVR}(c) \subseteq M(c)$ be the set of overriding methods of c.
- $M_{NEW}(c) \subseteq M(c)$ be the set of non-inherited, non-overriding methods of c.

Definition 5. Public and non-public methods

For each class $c \in C$ let

- $M_{pub}(c) \subseteq M(c)$ the set of public methods of c, and

- $M_{npub}(c) \subseteq M(c)$ the set of non-public methods of c.

A public method is a method that is visible outside its own class and inheritance hierarchy, and can be accessed by any other method in the system. A nonpublic method can only be accessed by a subset of methods. A protected method may only be accessed by a method within the same inheritance hierarchy, while a private method may only be accessed by a method in the same class. These restrictions, however, are not important for the dependencies discussion later on, however, and thus only the distinction between public and non-public methods is presented.

Definition 6. Set of all methods in the system

The set M(C) is the set of all methods in the system, and is represented as $M(C) = \bigcup_{c \in C} M(c)$.

Finally, each method has a set of parameters associated with it

Definition 7. Method Parameters

The set of parameters for a method m is denoted as *Par(m)*

5.1.3 Method Invocations

In order to be able to define the dependencies between class members in the system, it is necessary to define the set of methods that $m \in M(C)$ invokes, and the frequency of these invocations. Due to polymorphism, method invocations may be either static or dynamic, so it is necessary to distinguish between the two. For static invocations, the invoked method is determined by the type of the variables that references the object for which the method invocation occurs. For dynamic invocations, the invoked methods are determined by considering all possible types that the object for which the method invocation occurs may have at run-time. For each method $m \in M(C)$ the following sets are defined.

Definition 8. SIM(m): The Set of Statically Invoked Methods of m

Let $c \in C$, $m \in M_I(c)$, and $m' \in M(C)$. Then $m' \in SIM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' is invoked for an object of static type class d.

Definition 9. NSI(m,m'): The number of static invocations of m' by m

Let $c \in C$, $m \in M_I(c)$, and $m' \in SIM(m)$. Then $NSI(m,m')$ is the number of method invocations in m where m' can be invoked for an object of static type class d and $m' \in M(d)$.

Definition 10. PIM(m): The Set of Polymorphically Invoked Methods of m

Let $c \in C$, $m \in M_I(c)$, and $m' \in M(C)$. Then $m' \in PIM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' may, because of polymorphism and dynamic binding, be invoked for an object of dynamic type d.

Definition 11. NPI(m,m'): The number of polymorphic invocations of m' by m

Let $c \in C$, $m \in M_I(c)$, and $m' \in PIM(m)$. Then $NPI(m,m')$ is the number of method invocations in m where m' can be invoked for an object of dynamic type class d and $m' \in M(d)$.

Calls that are made by methods to other methods within the same class are denoted as LSIM, or local SIM. Formally, this is defined as follows:

Definition 12. LSIM(m): The set of local statically invoked methods of m.

Let $c \in C$, $m \in M_I(c)$ and $m' \in M(C)$. Then m'∈LSIM(m) ⇔ m'∈SIM(m) ∧ ∃c∈C, m∈M(c) ∧ m'∈M(c).

5.1.4 Indirect Method Invocations

In addition to the *SIM(m) and PIM(m)*, which are the sets of directly invoked methods by a method $m \in M(C)$, the set of indirectly invoked methods must also be defined. Method m indirectly invokes method m', if there are methods $m_1, m_2, ..., m_n$ such that m directly invokes m_1, m_1 directly invokes m_2, etc., and m_n directly invokes m'. Given this idea, indirect method invocations can now be formally defined.

Definition 13. Indirectly Invoked Methods

$\forall m \in M(C)$, let:

$$SIM^*(m) = \{m' \mid m' \in M(C) \wedge \exists n > 1, \exists m_1, m_2 ..., m_n \in M(C) : m_1 = m \wedge m_n = m' \wedge \forall i, 1 < i \leq n : m_i \in SIM(m_{i-1})\}$$

$$PIM^*(m) = \{m' \mid m' \in M(C) \wedge \exists n > 1, \exists m_1, m_2 ..., m_n \in M(C) : m_1 = m \wedge m_n = m' \wedge \forall i, 1 < i \leq n : m_i \in PIM(m_{i-1})\}$$

5.1.5 Attributes

Classes have attributes. Like methods, the attributes of a class may be inherited, overridden or newly defined. The attributes are modeled using a similar formalism as that of methods.

Definition 14. Declared and Implemented Attributes

For each class $c \in C$ let *A(c)* be the set of attributes of class *c*. $A(c) = A_D(c) \cup A_I(C)$ where

- $A_D(C)$ is the set of attributes declared in class *c*
- $A_I(C)$ is the set of attributes implemented in class *c*

Definition 15. A(C): The set of all attributes in the system.

A(C) is the set of all attributes in the system and is represented as $A(C) = \bigcup_{c \in C} A(c)$

5.1.6 Attribute Accesses

Methods may access a set of attributes. The set of attributes that are accessed by a method *m* is defined as follows.

Definition 16. AR(m) Attribute access

For each $m \in M(C)$ let *AR(m)* be the set of attributes directly accessed by method *m*.

In addition to directly accessing a given attribute, a method may also indirectly access an attribute, in a similar manner to indirectly invoking a method. A similar idea to the indirectly invoked methods is used to define indirect attribute access.

Definition 17. $AR^*(m)$ Indirect attribute access

For each $m \in M(C)$ let $AR^*(m)$ be the set of attributes indirectly access by method *m*.

$AR^*(m) = \{a' \mid a' \in A(C) \wedge a' \in AR(m') \wedge \exists n \geq 1, \exists m_1, m_2, ..., m_n \in M(C) : m_1 = m \wedge m_n = m' \wedge \forall i, 1 < i \leq n : m_i \in PIM(m_{i-1})\}$

5.1.7 Association Ends

In addition to methods and attributes, we need to consider the three types of relationships that we can expect to be part of the class model. These are association, generalization and usage dependency relationships [33]. Generalization and Usage dependencies will be considered in the general case, and are discussed below in Section 5.1.11. Associations, however, are a special case in our input model. Note that we don't differentiate between associations, aggregation, and composition relationships, since the latter two are specializations of the first. In our input model, association ends will be handled in a similar manner as attributes. This makes sense, as both are typically implemented as

references to instances. In other words, a bidirectional, binary association will translate into two attributes, one in each class at its ends. It is necessary to distinguish association ends from attributes, however, as they are managed differently in the change model (see Section 5) and fitness evaluation (Section 6). *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 18. Association Ends

For each class $c \in C$, let $AE(c)$ be the set of all association ends within c.

Definition 19. Association End Reference

For each method $m \in M(C)$, let $AER(m)$ be the set of all association ends that are directly invoked by m.

For example, the `MonopolyGame` class in the sample model (Section 4) is shown in Figure 4 below.

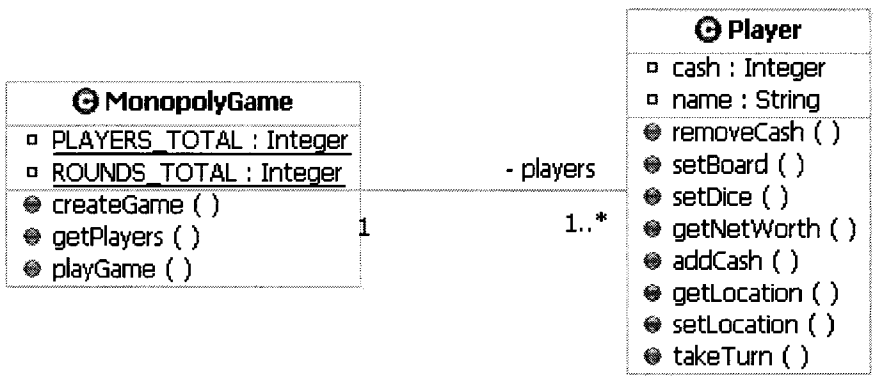


Figure 4 MonopolyGame with Player

`MonopolyGame` has an association with `Player`, called `players`. For our input model, this association would be represented as an association end, similar to an attribute. So for the purposes of the optimization process, the `MonopolyGame` class would appear as shown in Figure 5, with the players association as an attribute.

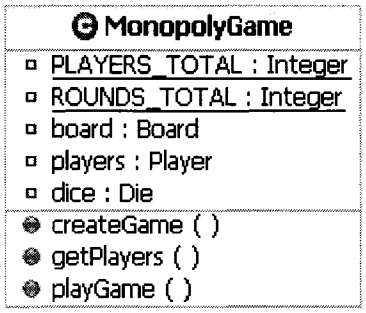


Figure 5 MonopolyGame with players association end as an attribute

5.1.8 Class Member

Class members are the methods, attributes and association ends that make up a given class $c \in C$. Formally, the class members for c are defined as:

Definition 20. Class members

For each class $c \in C$, let $CM(c)$ be the set of class members where $CM(c) = A(c) \cup M(c) \cup AE(c)$

5.1.9 Types

Attributes, parameters and methods all have types which must be considered when determining the dependencies between class members. A programming language provides a basic set of built-in types, and the user can define new class types as well as traditional types.

Definition 21. Basic Types and User-Defined Types

- BT is the set of built-in types provided by the programming language
- UDT is the set of user-defined types (e.g. records, enumerations but not classes)

The type of an attribute, parameter, or method return value is a class, a built-in type or a user defined type. The set of available types T is defined as follows:

Definition 22. The Set of Available Types

The set T of available types in the system is $T = BT \cup UDT \cup C$

The next definition determines how the type of attributes and parameters will be denoted.

Definition 23. Types of attributes and parameters

For each attribute $a \in A(C)$ the type of attribute is denoted by $T(a) \in T$. For each method $m \in M(C)$ and each parameter $v \in Par(m)$ the type of v is denoted by $T(v) \in T$.

Finally, the following definition presents how the return type for a method is represented.

Definition 24. Return types of methods

For each method $m \in M(C)$, the return type is denoted by $T(m) \in T$.

5.1.10 Dependencies

With the formalized definitions presented above, it is now possible to define a dependency between class members. These dependencies form the basis for the relationships between the classes, and the metrics that will be used to evaluate the model. There are seven types of dependencies defined. Both direct and indirect dependencies are considered.

Definition 25. Direct Method – Attribute Dependency

A direct method – attribute (DMA) dependency exists between $m \in M(C)$ and attribute $a \in A(C)$ if $a \in AR(m)$. This is written as *DMA(m, a).*

Definition 26. Direct Method – Method Dependency

A direct method - method (DMM) dependency exists between method $m_1 \in M(C)$ and method $m_2 \in M(C)$ if $m_2 \in PIM(m_1)$. A MM dependency is represented by *DMM(m_1,m_2).*

Definition 27. Indirect Method – Attribute Dependency

An indirect method – attribute (IMA) dependency exists between method $m \in M(C)$ and attribute $a \in A(C)$ if $a \in AR^*(m)$. This is written as *IMA(m, a)*.

Definition 28. Indirect Method – Method Dependency

An indirect method – method (IMM) dependency exists between method $m_1 \in M(C)$ and method $m_2 \in M(C)$ if $m_2 \in PIM^*(m_1)$. A MM dependency is represented by $IMM(m_1,m_2)$.

Definition 29. Class – Attribute Dependency

A class – attribute (CA) dependency exists between class $c \in C$ and attribute $a \in A(C)$ if $c = T(a)$. This dependency is shown as *CA(c, a)*.

Definition 30. Class – Method Dependency

A class – method (CM) dependency exists between class $c \in C$ and method $m \in M(C)$ if $v \in Par(m) \wedge c = T(v) \vee c = T(m)$ (if c is the type of a parameter of m, or c is the return type of m). A CM dependency is represented as *CM(c, m)*.

Definition 31. 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:

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

5.1.11 Class Relationships

Relationships exist between the various classes in the system. These relationships represent how the classes interact with each other. There are three types of relationships considered for the system model. The first is a generalization, or inheritance relationship.

The second is a usage relationship. Finally, the last type of relationship between the classes is an association, or aggregation relationship, which has already been presented in Section 3.1.7. The other two relationships are defined formally below.

A generalization relationship exists when two classes are in a common inheritance hierarchy. Formally, this is defined as

Definition 32. Generalization Relationship

A generalization relationship exists between two classes $c_1 \in C$ and $c_2 \in C$, $c_1 \neq c_2$, if $c_2 \in Ancestors(c_1) \cup Descendants(c_1)$.

A usage relationship, exists between two classes if there is a class – method dependency between a method of the first class and the second class. Formally, this is defined as:

Definition 33. Usage Relationship

A usage relationship exists between two classes $c_1 \in C$ and $c_2 \in C$, $c_1 \neq c_2$ if $\exists m \mid m \in M(c_1) \wedge CM(c_2, m)$.

5.2 Class Model

5.2.1 Class Members

The first piece of information required from the class model is the class members that will be manipulated by the search. These are the private attributes, public methods, and association ends. Association ends are treated as attributes (Section 5.1.7) with one exception. Accessor methods for association ends must be included in the input model, while accessor methods for attributes are not included. This is discussed in more detail in Section 5.2.3.

An example of the class member information needed for the optimization process is shown in Appendix A, Figure 29 for the Monopoly Game sample model.

5.2.2 Dependency Information

As presented in Section 3, the heuristic will search for improvements to the class model based on dependency information between the class members. The classes, class members, and their dependencies are all the information that the heuristic needs from the existing design. The methods used to acquire the dependency information needed are not important for the search heuristic. However, since the heuristic is applied to an analysis model, sequence diagrams are used to determine dependencies between the various methods in the system, while the OCL contracts of the class members are used to determine attribute dependencies for each of the methods.

Although the heuristic makes use of information from dynamic diagrams such as sequence diagrams, its goal is to provide feedback into improvements to the static model and design of the system at the domain class layer. Thus, only the class design is considered, altered and presented by the heuristic. The application of our technique to other levels of class design is discussed in the future work (Section 10.2)

An example of the dependency information is shown in the dependency matrix (Appendix A, Figure 30) for the sample model in Section 4.2.

5.2.3 Constructors and Accessor Methods

When determining the dependency information, it is important to determine how constructors and accessor methods are handled. Accessor methods, also referred to as get and set methods, may have an effect on the dependency determination. Accessor methods may be used to mask method-attribute dependencies since the client method can call the accessor method instead. This can have an effect on the cohesion measurement (discussed later in Section 7.2). Typically, an analysis model will not include accessor methods [33], but this has the drawback of preventing methods in other classes from accessing the private attributes, which may be necessary as class members are moved around. To solve this, it is assumed that each attribute has a get and a set method associated with it. This allows methods in other classes to access the private attributes. These accessor methods are assumed, and any call to an accessor method is considered to

be a method-attribute dependency to the corresponding attribute. This allows methods in other classes to call private attributes, and avoids any possible effect that the accessor method may have on the measurement of cohesion.

While accessor methods for attributes are assumed by the input model, the accessor methods for association ends must be included. Methods that simply use the reference corresponding to the association end (adding or removing objects to a collection, returning the reference, etc.) are included in the input class model. This is done in order to be able to manipulate the association ends and the methods that operate directly upon them in a meaningful way.

For example, the `MonopolyGame` class in the sample model (Section 4) has an association with the `Player` class, called `players`. This association end would be included in the `MonopolyGame` input. The method `MonopolyGame.getPlayers()` is used to access this association. Thus, `getPlayers()` is also included in the input model, since it operates on an association end.

Constructors can also have an impact on the dependencies if the constructors perform method class and modify a number of attributes. However, constructors are not typically a part of the class design at the analysis level [33], and are not considered for optimization.

5.3 User Defined Constraints

Along with the class design being placed into the heuristic, the user can also specify constraints on the search. The constraints will affect the changes that are allowed by the genetic algorithm on the design. The changes permitted during the optimization process are outlined in Section 6. The user is able to limit the type of changes that can be done on the model. The limits that s/he can place on the evolution are as follows.

- Fix class members to their current class so they cannot be altered by the search heuristic

- Group class members together so they must always be in the same class together (although this class may change).

- Allowing / disallowing new classes to be created

Each setting is explained in greater detail in the sections below.

5.3.1 Fix class members to their current class

The first possible constraint is to fix class members to their original class assignment. This means that the optimization algorithm will not change the class assignment of these class members (the change model is discussed in Section 6). These class members will still be used to measure the quality of the design, but the class assignment for fixed members will not change.

5.3.2 Group class members together

The second user defined constraint on the changes that can be made by the algorithm to the design is to group class members together. These class members must be moved together so that they are always a part of the same class. The designer may choose to do this in order to keep class members that are related together without restricting these class members to a specific class.

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. The association end, and the methods that operate upon it, are considered to represent a single concept (the association) within the model, and are grouped and moved together.

For example, the `MonopolyGame` (Figure 3, Section 4) class has the association end players, and the reference method `getPlayers()`. The `getPlayers()` method returns the reference represented by the players association end. Thus, `getPlayers()` and players are grouped together. This allows for the manipulation of players without breaking `getPlayers()`, and allows methods of other classes to access the players association.

5.3.3 New class creation

The final type of constraint that the user can place on the design is allowing or preventing the creation of new classes. If allowed, the optimization will attempt to create new classes by moving methods and attributes into them. However, if the creation of classes is disallowed, then methods and attributes will only be shuffled to the existing classes.

6 CHANGE MODEL

In order to be able to perform a heuristic search for the optimal class member assignment, it is necessary to define changes that the heuristic is capable of performing on the given model. This section outlines the changes to the model that are permitted, and those that are not permitted, on the class design being optimized.

6.1 Class Members

6.1.1 Signature

This search is based on existing model information, the signature of the methods (types, return types, parameters) is already known.

These signatures, along with the OCL contracts for the methods and the interaction diagrams, form the basis for the dependencies between the methods and attributes in the model, and they cannot be subject to change in the heuristic. Thus, once the method-method, method-attribute and method-association end dependencies have been identified (e.g. from UML documents), only the ownership of class members matters and we do not need to know about the attribute types and method signatures, nor will they change.

6.1.2 Addition and removal

The change model will 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 and sequence diagrams. Existing methods and dependencies could be broken up, and new methods added. Likewise, methods could be removed, and their responsibility and dependencies merged into other methods that depended upon them. However, this is outside the scope of the heuristic, and thus it is assumed that methods and attributes cannot be added or removed. This is discussed in more detail in the future work (Section 10.2).

6.1.3 Changing class assignment

The main aspect of the heuristic will be to move class members (methods, attributes and association ends) from one class to another, in order to determine if there exists a class assignment scheme with an improved cohesion and coupling than initially in the model. In order to be able to affect the measures of coupling and cohesion, the heuristic must be capable of manipulating the class members assigned to a given class, by moving them from one class to another. This is the only operation that the heuristic will be allowed to perform on the class members, as only the class they are associated with may change, and the rest of the information (which the dependencies rely upon) remains constant.

As an example, the figure below (Figure 6) shows the `Die` and `Player` class from the sample model included in Section 4.

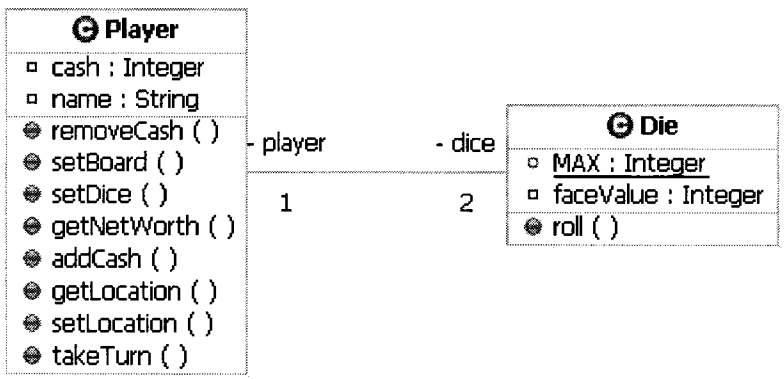


Figure 6 Die and Player

If the method `roll()` is moved from the `Die` class to the `Player` class, we get the design outlined in Figure 7.

Since `roll()` has been moved to `Player`, `takeTurn()` no longer depends on `Die`. However, `faceValue` remains in `Die`, `roll()` will still require a relationship between `Player` and `Die`. This shows what happens to the design when a method is moved from one class to another. The change in relationships between the classes will affect the coupling and cohesion values, as discussed in Section 7.1 and Section 7.2 below.

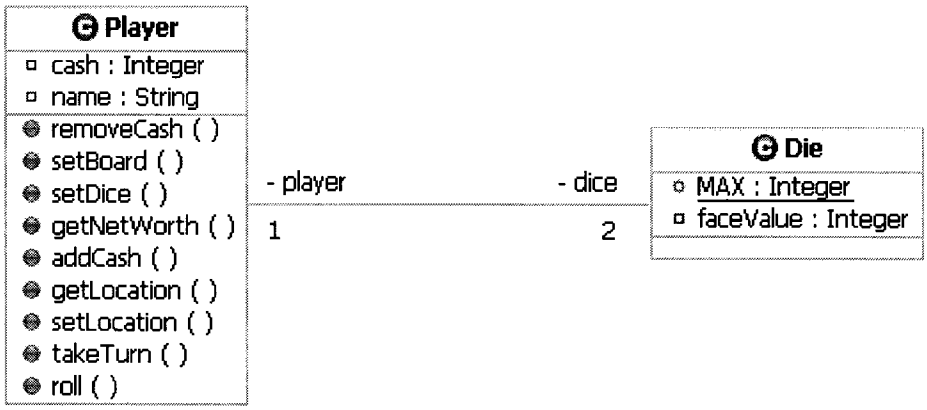


Figure 7 Die and Player with roll() moved

Association ends and their groups are handled in the same manner. Figure 8 shows `MonopolyGame`, `Player` and `Board`.

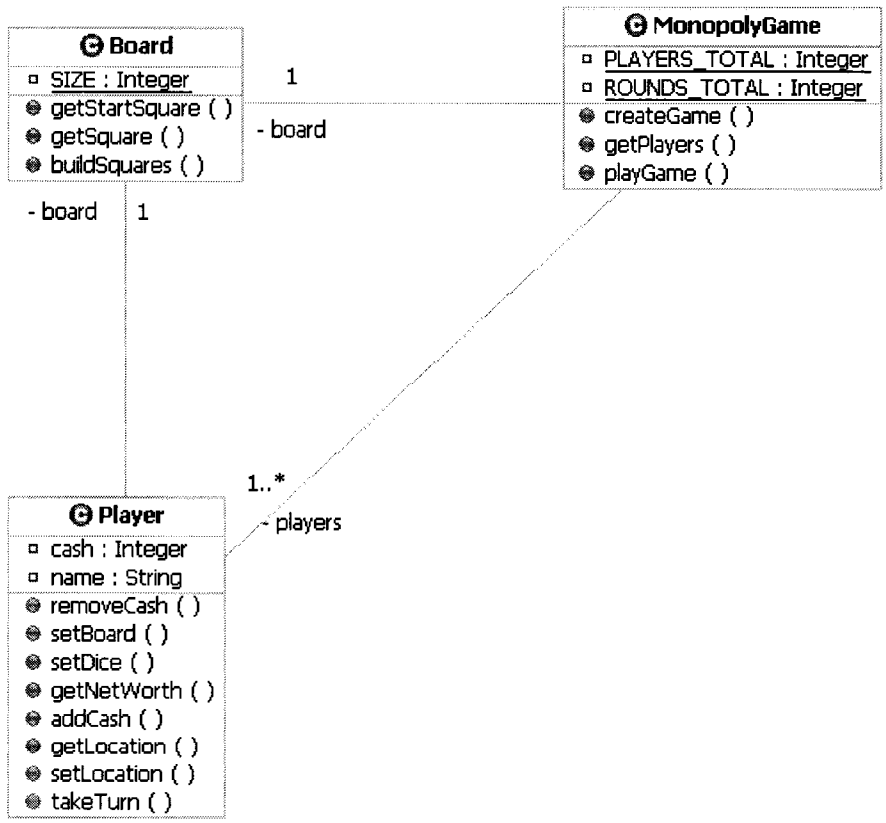


Figure 8 MonopolyGame, Board and Player from the Sample Model

If the association end `players` is moved from `MonopolyGame` to `Board`, it would change the format so now the `Board` has `players` on it, rather than the `MonopolyGame` having `players`. Since the `getPlayers()` accessor method is used to manage the reference to `players` in `MonopolyGame`, it would have been grouped with the association end (Section 5.3.2), and must move with the association. Thus, `players` and `getPlayers()` would move together into `Board`. This is shown in Figure 9 below.

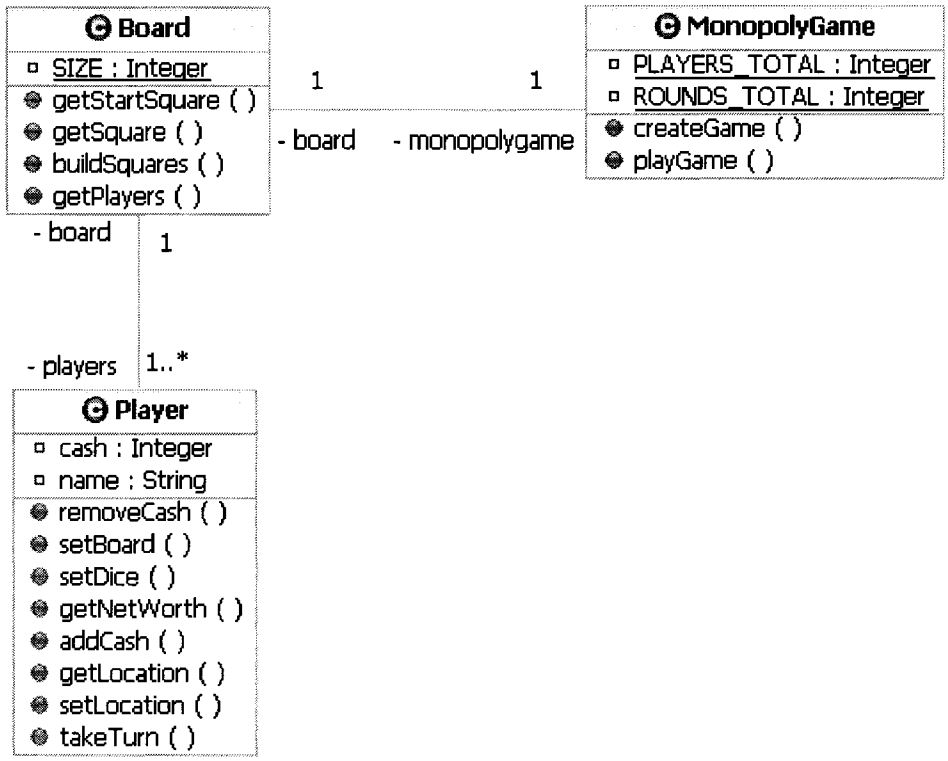


Figure 9 MonopolyGame, Board and Player with the players association end moved

6.2 Classes

6.2.1 Addition and Removal

The classes in the model, unlike the attributes and methods, may be manipulated, added and removed. During the course of the search, this may result in some classes being removed, and new classes being added (assuming this is allowed by the constraints) to the overall design.

More specifically, classes are removed from the design if they are empty when the optimization heuristic is complete, as there is no purpose served by including an empty class in the design itself.

If allowed, classes are added to the system when moving class members in the design and, rather than assigning them to an existing class, placing them in a new class. In this manner, new classes are created and populated by existing class members.

For example, the following figure (Figure 10) once again shows `Player` and `Die`, this time along with `MonopolyGame`.

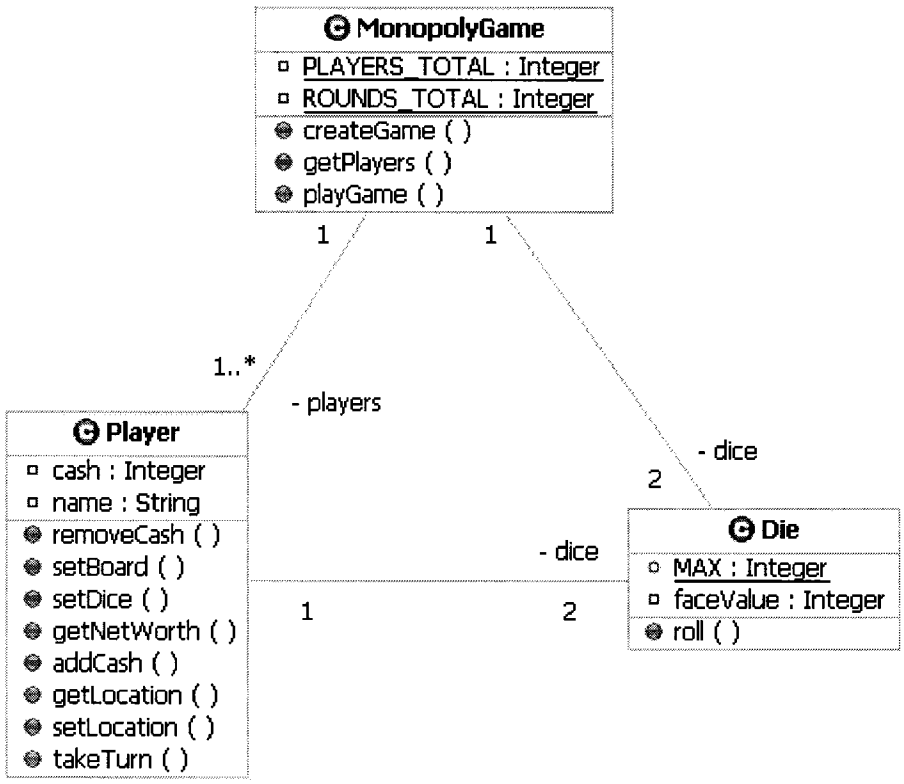


Figure 10 Player, Die and MonopolyGame

Now, if during the optimization process, `roll()`, `getFaceValue()`, `MAX`, and `faceValue` are moved from `Die` over to `Player`, then `Die` will empty (as seen in Figure 11).

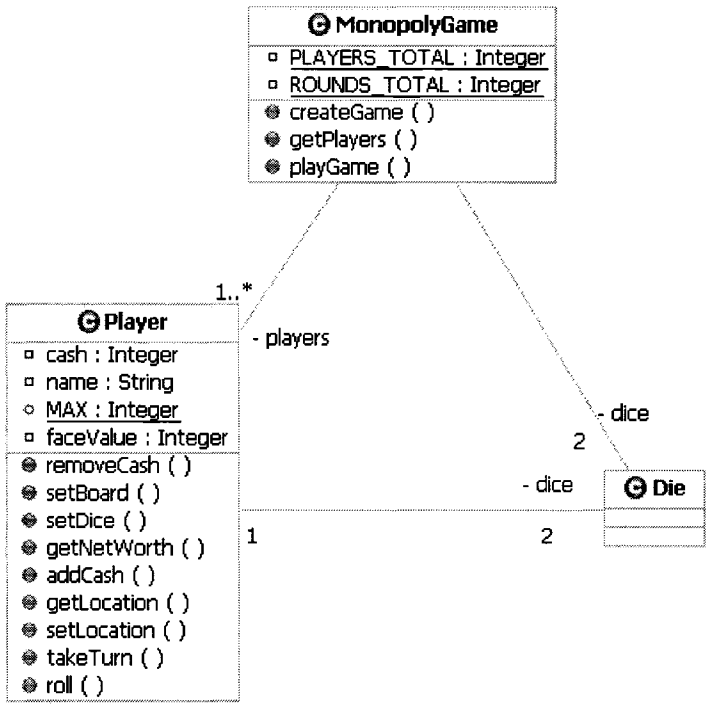


Figure 11 Player, Die and MonopolyGame; with Die now empty.

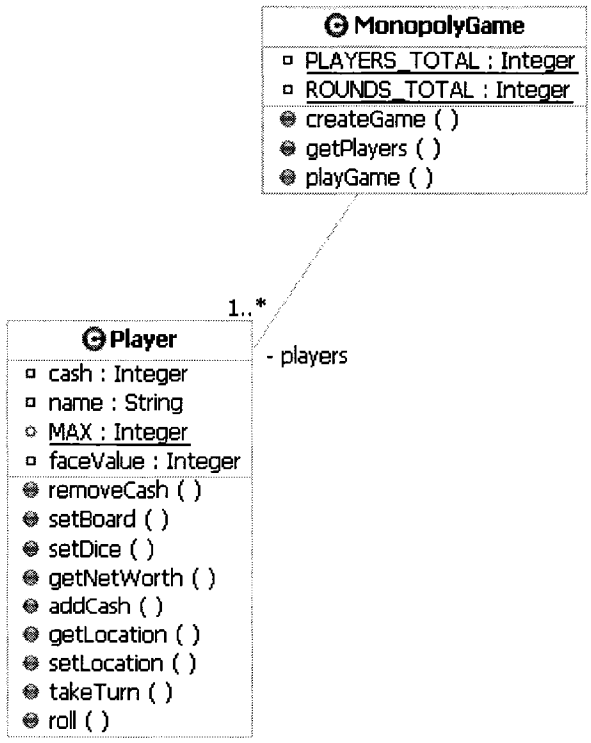


Figure 12 MonopolyGame and Player; Die has been removed

`Die` can now be removed from the design, as shown. However, since some methods in `MonopolyGame` class relied on the former class members of `Die`, a relationship between `Player` and `MonopolyGame` is now required. The final result of removing `Die` is shown in Figure 12.

6.2.2 Addition, removal, and manipulation of relationships

The relationships between the classes are determined by the dependencies between class members, as presented in Section 3.1.10. The dependencies between class members are used to determine where usage relationships between classes must exist. Therefore, the optimization will not consider usage relationships as they can be derived from the class member dependencies. Association relationships, on the other hand, will be handled by treating association ends separately and in a similar manner to attributes, as discussed in Section 5.1.7. In addition to this, association ends are grouped with the methods that manipulate them, as presented in Section 5.3.2. Each association end, and its group, can therefore move from one class to another during the search. Since they are represented as class members in the search heuristic, then the changes to the association end groups have already been accounted for in the manipulation of the methods and attributes (Section 6.1.3). The generalization relationship is a special case in our change model than the other two relationships between classes, and is discussed in more detail in Section 6.3.

So, in summary, usage relationships are determined by the dependencies between class members, and are not considered in the change model. Association ends, and their groups, are handled like attributes in the change model, and follow the same change model as attributes (Section 6.1.3) and finally Generalization relationships are a special case and discussed in Section 6.3 below.

6.3 Generalization

Generalization relationships are treated differently than usage and association relationships within the change model, as they cannot be accounted for by the method-method, method-attribute, and method-association end dependencies. First consider

moving any method that belongs to a generalization hierarchy, including abstract methods. This would have an important impact on the dependencies that must be maintained. 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 in its own (existing or to be created) child classes (i.e. creating new methods). Alternatively, a new inheritance hierarchy could be created that would receive the abstract method and all its concrete implementations in child classes.

At this initial stage, these changes to the class diagram are too complex, and the impact they will have on the search heuristic is not known. The following simplifying assumption is used: changes to the existing generalization hierarchies are limited to class members that are not overridden. Other elements in the hierarchies cannot be moved during the search. The modification of Generalization hierarchies is discussed in the future work section (Section 10).

For example, the inheritance hierarchy in the `MonopolyGame` model (Section 4) involves four classes. Square is the parent, with `GoSquare`, `NormalSquare` and `IncomeTaxSquare` inheriting from `Square` (Figure 13). `Square` defines an abstract method, `landedOn()`, which is implemented in the three child classes. If `landedOn()` was to move from `Square`, it would break the polymorphic invocation of `landedOn()` by `Player.takeTurn()`. So it cannot be moved. This is the same for the concrete implementations in the three child classes. So as a simplifying assumption, the `landedOn()` methods are fixed in place, and will not be considered by the search.

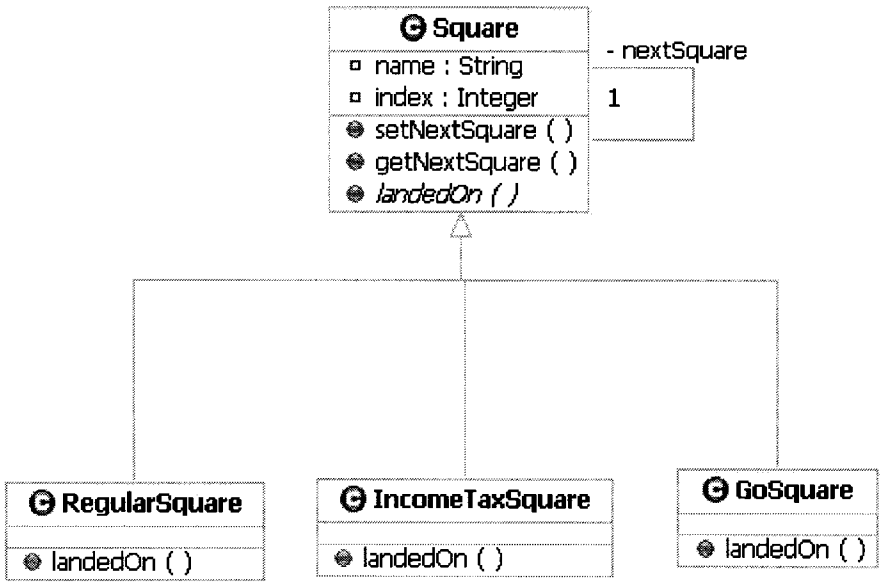


Figure 13 Square Inheritance Hierarchy

6.4 Constraints

In order to limit the search, and to ensure that the resulting model is reasonable, some constraints are necessary. When making changes to the model, the heuristic must be able to account for certain constraints on the model itself. These constraints, and their justification, are as follows.

6.4.1 Empty classes

Classes cannot be empty. A class that becomes empty during the course of the heuristic is removed and not considered for the metric. The reason for this is that there is no purpose including classes that do not contain at least one class member. The empty class is not removed entirely, but the class is left in case it is needed, to keep the heuristic from adding a large number of new classes. If new classes are not permitted, empty classes are kept around so the total number of classes is not reduced unnecessarily. When the heuristic is finished, any class still empty is removed from the resulting design.

6.4.2 Relationships to classes

Given that classes cannot be empty, classes must also have at least one relationship to another domain class. That is, none of the classes in the domain should be standalone classes. In order for class members to be assigned to an optimal location, there must be at least some dependencies to these class members, otherwise it is impossible to determine their optimal class assignment. So not only must classes have at least one other relationship to another domain class, but each class member in the model must have at least one dependency, either as a client or a server. This allows a search based technique to determine the optimal class assignment, as without dependency information this is impossible.

6.4.3 User Defined Constraints

In addition to the two constraints listed above, the user constraints inputted into the algorithm discussed in Section 5.3 must also be taken into account. These constraints limit the changes that can be performed on the model, and the optimization algorithm must be adjusted accordingly. This means preventing class members from being moved by fixing them within the original class (as discussed in Section 5.3.1); grouping class members together into a single group so that the class members are always in the same class together (as discussed in Section 5.3.2), and allowing / preventing the creation of new classes in the design (as discussed in Section 5.3.3). These constraints need to be taken into consideration by the change model for the optimization. For example, if the user has not allowed new classes to be created, then the change discussed in Section 6.2.1 cannot be performed on the design.

One possible technique that may be beneficial is to allow the user to add, modify or remove constraints while the search is progressing. This would allow the user to guide the search towards specific alternatives. This type of interactive user feedback is not a part of the current work, and is discussed in more detail in the future work (Section 10).

7 FITNESS FUNCTION

In the search heuristic, the fitness function is used to evaluate each of the candidate solutions, and allows for the comparison of the different candidate designs. In our goal of improving the domain class models, the objectives are to analyze and recommend improvements based on two properties, cohesion and coupling.

The coupling measures that are used are discussed in Section 7.1, and the cohesion measures used are described in Section 7.2. In addition to the use of coupling and cohesion measures to indicate the fitness of the design, user-defined constraints and interactive user feedback can also play into determining how fit each individual solution in the algorithm is. The effect of user defined constraints on the fitness function is discussed in Section 7.3. Finally, there are many methods used to handle multi objective optimization of genetic algorithms. The implications of a multi objective search, along with some various multi objective genetic algorithm techniques, are discussed in Section 7.4.

7.1 Coupling Measure

Coupling, or the nature and extent of dependencies in a system, measures how interdependent the classes in a design are upon one another. A general definition of coupling is "the measure of the strength of association established by a dependency from one module to another" [50]. The stronger the coupling of classes in the design, the more difficult they are to understand, change, and correct [8]. In order to improve the design, the goal should be to lower the overall coupling value as much as possible. The more relationships in the design, the harder it is to update, maintain and reuse.

In order to determine what measure to use for coupling in the genetic algorithm, we use the Unified framework for coupling measures [8], outlining the criteria for the coupling measure. This allows us to better define what type of coupling measure is needed in our context, and to define the properties required for the selected coupling measures. The

unified framework [8] presents six framework criteria for selecting and comparing coupling measures. These are discussed below.

Type of Dependency: The type of dependency refers to the mechanism that constitutes coupling between two classes. From the formalism presented in Section 5.1, the dependencies that will constitute coupling in our context are the method-attribute and method-method dependencies. Note that method-association end dependencies are not counted towards coupling. However, as outlined in Section 5.2.3, accessor methods for associations are included, and thus any coupling between a method and an association can be accounted for through the method-method dependencies with the association end's accessor method.

Locus of Impact: Locus of impact refers to how the class is considered when determining its coupling, as either a client or a server. With import coupling, the class is considered to be the client and the coupling of the class measures its use of other class members. Export coupling, on the other hand, analyzes the class in the role of the server, measuring the use of the class members by other classes.

Since the goal is to determine the proper class assignment of class members, the coupling value of a class should be determined by how many dependencies it has with other classes. Import coupling, where the class is measured as the client of the other classes in the system, is used. It is important to note that import coupling for the system should be equal to export coupling for the system. Import coupling is simply used as it is a bit more straight-forward to determine (i.e. get the class member, get all of its dependencies and count those contributing to coupling).

Granularity: Granularity indicates the level of detail at which the information is gathered. For the design optimization, the class domain will be used, as the goal of the heuristic is to optimize the class design. In order to count the dependencies, the number of individual dependencies is tallied for each class member, and these values are used to compute the coupling value of the class.

Stability of Server: This criterion allows the coupling metric to differentiate between stable and unstable classes. The idea behind this is that coupling to a stable class is acceptable, as the class is unlikely to change, whereas coupling to an unstable class has a negative effect on the design of the system. However, since the heuristic is considering only domain class models at the initial stages of design, all classes are likely to be subject to change as the design is evolves both via the optimization technique and the subsequent design iterations (creating lower level design and code). Thus, the stability of the server class is not considered in the coupling measure, since all of the classes are likely going to be subject to change.

Direct or Indirect Dependencies: This criterion determines if only direct dependencies should count towards coupling, or if indirect dependencies should count as well. Both direct and indirect method – attribute and method – method dependencies are presented in Section 3.1. Using design by contract (DBC) [37], the client class member should be kept separate from the implementation of the server. The server should hide the details of its implementation away from the client. Thus, the client class member should not be penalized if the server has a large amount of dependencies to other class members. Since we are measuring the coupling of class members as clients, it is only necessary to count the direct coupling to server class member. The details of the server's implementation are hidden from the client, including its dependencies to other class members, and it does not make sense to penalize the clients coupling due to the dependencies of the server. Thus, direct dependencies are used to calculate coupling in our context.

Inheritance: The inheritance criterion in the framework is used to define how the coupling metric should handle an inheritance relationship. The first aspect of this criterion is if the metric should measure coupling between members of an inheritance hierarchy (referred to as inheritance based coupling), or coupling between classes that are not related by inheritance (referred to as non-inheritance based coupling). In our context, the goal is to reduce all coupling as much as

possible, and both inheritance and non-inheritance based coupling are not desirable in the design. Thus it is important to be able to measure both inheritance and non-inheritance based coupling, in order to minimize it.

The second aspect of the inheritance criterion in the framework is the assignment of methods and attributes to the classes i.e. should inherited class members be considered or not? The coupling of the class members of the parent class would be counted when calculating the parent class coupling. If the coupling of these class members is included in the child classes, then they will be counted multiple times in the overall system coupling (once for the parent, and once for each of the child classes.) Thus, inherited methods and attributes are not considered towards coupling, to avoid having the coupling of these class members counted numerous times. Only new and overridden class members will be considered for the coupling metric in our context.

Finally, the third aspect that needs to be considered is if polymorphic invocations need to be considered or only static invocations. It is important to include all of the polymorphic invocations in the coupling metric. To explain why it is important to include polymorphic invocations, consider the following example, taken from the sample model in Section 4.

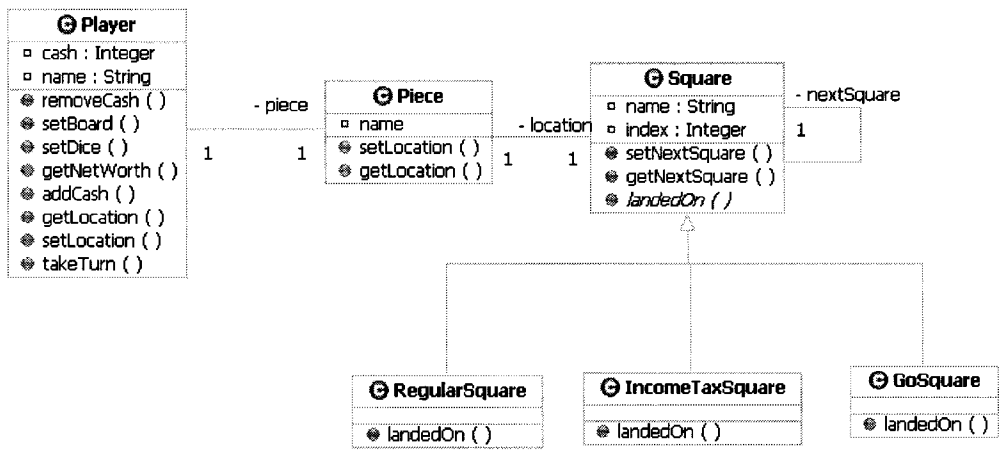


Figure 14 Square Inheritance hierarchy with Player and Piece

The `Player.takeTurn()` method depends upon `Square.landedOn()`, which is overridden in the three child classes. While statically, `takeTurn()` is only dependent upon the `Square.landedOn()` method, it may be invoking the `landedOn()` method of any of the child classes, and is dependent upon their implementation as well. If only static dependencies are considered, the number of class members that `takeTurn()` is dependent upon is reduced, and it would appear as though the `takeTurn()` method is coupled to only one class member in the inheritance hierarchy, while in fact it is coupled to four classes. To accurately reflect the number of class members and classes that a client is coupled to, polymorphic invocations will be counted, and not simply static. This more accurately reflects the number of class members and classes that the class member is dependent upon.

The above six criteria values illustrate the properties required of the coupling metric. The coupling metrics discussed below are based upon the suite of measures presented by Briand et. al [9], adapted to fit the dependencies presented in Section 5.1. There are three coupling metrics used. The first is based on Method – Attribute dependencies, called Method – Attribute Coupling (MAC). The second is based on Method – Method dependencies and is called Method – Method Coupling (MMC). Finally, the third accounts for coupling within an inheritance hierarchy, and is called Method – Generalization Coupling (MGC).

In order to define the coupling metrics, the dependencies must first be expressed as a set of interactions between two classes. There are two types of interaction sets considered, one for attributes and one for methods. Since there may be a number of interactions between any two classes, the interactions must be represented by a set. Formally, the interaction sets are:

Definition 34. Set of Method – Attribute Interactions (MAI)

For two classes $c1 \in C$ and $c2 \in C$ the set of Method-Attribute Interactions between $c1$ and $c2$ is defined as $MAI(c1,c2) = \bigcup_{m \in M(c1)} \{(m,a) \mid a \in A(c2) \wedge DMA(m,a)\}$

Definition 35. Set of Method – Method Interactions (MMI)

For two classes $c1 \in C$ and $c2 \in C$ the set of Method-Method Interactions between $c1$ and $c2$ is defined as $MMI(c1,c2) = \bigcup_{m \in M(c1)} \bigcup_{m' \in M_{NEW}(c2) \cup m' \in M_{OVR}(c2)} \{(m,m') \mid DMM(m,m')\}$

The first two coupling metrics used are a summation of the interactions between classes not in the same inheritance hierarchy. To this end, it is necessary to formally define the set of *Other* classes for a given class $c \in C$ i.e. the classes that are not a part of the same inheritance hierarchy as class c.

Definition 36. Set of Other classes of class c.

$$Others(c) = C - (Ancestors(c) \cup Descendents(c) \cup \{c\})$$

The third coupling measure, Method – Generalization coupling, is a summation of the interactions between classes within the same inheritance hierarchy. Thus, it is necessary to formally define the set of *OtherGen* classes for a given class $c \in C$ i.e. the classes that are a part of the same inheritance hierarchy as a class c. Ancestor classes of c are excluded from this set, as their class members would be visible within c and would not count towards coupling.

Definition 37. Set of OtherGen classes of class c.

$$OthersGen(c) = (Descendents(c) \cup Siblings(c))$$

The coupling metrics used sum the set of interactions between the measured class $c1 \in C$ and the set of other classes. Formally, the three coupling metrics are:

Definition 38. Method – Attribute Coupling (MAC)

For a given class $c1 \in C$, Method – Attribute Coupling $MAC(c1)$ counts all MAI from class $c1$ to classes that are not ancestors or descendents of $c1$.

$$MAC(c1) = \sum_{c2 \in Others(c1)} |MAI(c1,c2)|$$

Definition 39. Method – Method Coupling (MMC)

For a given class $c1 \in C$, Method – Method Coupling $MMC(c1)$ counts all MMI from class $c1$ to classes that are not ancestors or descendents of $c1$.

$$MMC(c1) = \sum_{c2 \in Others(c1)} |\, MMI(c1, c2)\,|$$

Definition 40. Method – Generalization Coupling (MGC)

For a given class $c1 \in C$, Method – Generalization Coupling $MGC(c1)$ counts all MAI and MMI from class $c1$ to classes that are descendants or siblings of $c1$.

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

In order to calculate the coupling value for the optimization, the class diagram coupling measure is obtained by summing the coupling values for all of the classes.

7.2 Cohesion Measure

Cohesion measures how well each class implements one and only one abstraction. Stevens et al. defined cohesion as the degree to which elements of a class belong together [50]. In object-oriented systems, highly cohesive classes are easier to develop, maintain, and reuse [7]. Therefore, the goal of the search heuristic is to maximize the amount of cohesion in the domain layer class design overall.

As was performed previously in Section 7.1, we will once again make use of the Unified Framework presented by Briand et al., this time using the cohesion framework presented in [7], in order to define our requirements for our cohesion measure. This will, in turn, aid in determining the properties that our cohesion measure must have. According to [7], there are five criteria for cohesion measures that must be considered when selecting a metric to use. These are discussed below.

Type of dependency: The first criterion is the type of dependency that is used in order to measure the cohesion within the class. Cohesion measures how well the

class represents a single concept, which is determined by the extent that the class members interact with each other. Three of the types of dependencies discussed in Section 5.1.10; Method-Attribute, Method-Method and Method-Association End, and Local (in-)direct access, are used to determine the cohesion of the class in our context. By determining how much the methods interact with the other members of the class, we will be able to get an idea of how well the class represents a single concept.

The class based dependencies, Class – Method, and Class – Attribute, will not count towards the cohesion measure since our goal is to determine the optimal assignment of class members, and our cohesion measurement will use dependencies that reflect that. Thus, only method based dependencies are used. Also, since the classes will change, the class based dependencies are not as meaningful in the context of our search as the method based dependencies.

Finally, in order to determine the cohesion, only the class members that are in the same class are considered. The local (in-)direct access will be used to determine how the methods are interacting with the attributes and associations ends of the same class, and determine if they are representing a single concept.

Domain of the measure: The domain of the measure criteria specifies the objects to be measured. Since the goal of the heuristic is to optimize the system design at the class model level, the most logical domain for the measure would be the class. Each class can be measured for its cohesion value then the values can be combined to give overall system cohesion.

Direct or Indirect: This framework criterion indicates if the cohesion measure should use only direct dependencies, or if indirect dependencies between class members should also be counted towards the cohesion value of the class. 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.

Inheritance: The inheritance framework criterion outlines how the cohesion metric should account for classes within an inheritance hierarchy. There are two aspects to the inheritance criterion: how are class members assigned to the classes, and is polymorphism taken into consideration? The first aspect indicates if inherited class members should be taken into consideration in the analysis of cohesion or not. 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. This suggests that in order for a class in an inheritance hierarchy to be cohesive, all of the methods and attributes within that class must be considered. So, for the design optimization heuristic, inherited methods and attributes must be included in the cohesion metric used.

The polymorphic aspect of the inheritance framework criteria indicates if polymorphic invocations should be considered. Since it is not possible to polymorphically invoke a method or attribute of the same class, there is no need to consider polymorphic dependencies, and rather static dependencies will be sufficient to measure the cohesion of a class.

Accessor Methods and Constructors: The final framework criterion discusses how the cohesion metric handles accessor methods and constructors. This is an issue, since accessor methods can cause problems for measures which count references to attributes [7]. The reason is that accessor methods can artificially lower the cohesion value by hiding a methods access to an attribute. However, because in our context indirect dependencies are used to measure the cohesion, the use of accessor methods does not hide the attribute reference. Thus, no special treatment is required for accessor methods in the design optimization heuristic. Constructors, as discussed in Section 5.2.3, are not considered since they are not

included in the analysis system design. Thus, the cohesion metric does not need to take accessor methods or constructors into account.

The five criteria presented above provide the properties that cohesion metrics must have in order to be considered for the design optimization heuristic. In order to measure how well the class represents a single abstraction, two types of dependencies are considered for the cohesion metric. The first measures the attributes that each method in the class is indirectly dependent upon, and the second measures common attribute usage between the methods of the class. The first measure is based upon the approach by Briand et al. in [10] and the second measure is based upon the tight class cohesion measure by Bieman and Kang [5]. Both measures are normalized, as all cohesion measures should be [7], and the two cohesion measures are complementary (See Section 7.2.3).

7.2.1 Cohesive Interactions metric

The cohesive interactions metric measures the number of cohesive interactions in a class, taken as a percentage of all possible cohesive interactions in the same class. A cohesive interaction is defined as an indirect dependency between a single method and attribute of class *c*. Formally, this is expressed as:

Definition 41. 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 dependencies between the methods $m \in M(c)$ and the attributes $a \in A(c)$ or association ends $a \in AE(c)$.

$$CI(c) = \bigcup_{m \in M(ac)} \{(m,a) \mid a \in A(c) \cup AE(c) \wedge LR(m,a)\}$$

Let $CI_{max}(c)$ be the set of all possible cohesive interactions in the class interface. It is then possible to define the ratio of cohesive interactions as

Definition 42. Ratio of Cohesive Interactions

The ratio of cohesive interactions (*RCI*) for all classes $c \in C$ is the number of cohesive interactions in class c, over the number of possible cohesive interactions in the class interface of c.

$$RCI(c) = \frac{|CI(c)|}{|CI_{\max}(c)|}$$

The $RCI(c)$ ranges from 0 to 1, where values 0 and 1 indicate minimum and maximum cohesion, respectively. 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 represent the cohesion value across the entire system, the RCI values for the classes are averaged. This gives a cohesion value in the range of 0 and 1 for the system.

Definition 43. Ratio of Cohesive Interactions for the System

For the system C the ratio of cohesive interactions $RCI(C)$ is given as the average of the ratios of cohesive interactions for all of the classes.

$$RCI(C) = \frac{\sum_{c \in C} RCI(c)}{|C|}$$

7.2.2 Tight Class Cohesion

The tight class cohesion metric presented in [5] is based on the concept of common usage. The idea is that methods which invoke common attributes (or association ends) should be together in the same class, and represent a single abstraction. We extend this to include methods which invoke common attributes, association ends or each other. In order to represent tight class cohesion, first the concept of common usage must be defined. Common usage represents the case when two methods either directly or indirectly access a common attribute or association end of a given class, or invoke each other.

Definition 44. Common usage

The predicate $cu(m1, m2)$ (common attribute usage) which is true if $m1, m2 \in M(c)$ directly or indirectly use an attribute of class c in common, or invoke each other:

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

The tight class cohesion metric can then be defined as the percentage of pairs of public methods of the class with common usage.

Definition 45. Tight class cohesion

Tight class cohesion (TCC) is the pairs of public methods of a class $c \in C$ with common usage.

$$TCC(c) = 2\frac{|\{\{m_1, m_2\} \mid m_1, m_2 \in M_I(c) \cap M_{pub}(c) \wedge m_1 \neq m_2 \wedge cu(m_1, m_2)\}|}{|M_I(c) \cap M_{pub}(c)|(|M_I(c) \cap M_{pub}(c)| - 1)}$$

When a class contains less than two methods, TCC is undefined. As before, the TCC value is currently given for only the class. In order to measure the cohesion metric across the system, the TCC value for each of the classes is summed, and then averaged over all of the classes in the system. This gives the average percentage of TCC present in the system.

Definition 46. Tight class cohesion across the system

For the system C, the tight class cohesion value is given as the average TCC value for each class in the system.

$$TCC(C) = \frac{\sum_{c \in C} TCC(c)}{|C|}$$

7.2.3 Complementary measures

In the case of coupling (Section 7.1), it is fairly obvious to see that the three coupling measures used measure distinctly different aspects of coupling, coupling to attributes, coupling to methods and coupling within inheritance hierarchies. In the case of cohesion, both of the cohesion measures presented measure the interaction of methods within the class. The reason for having two cohesion measures is that the way they measure the method interactions is slightly different, and thus can complement each other.

In order to understand how these two measures complement each other, it is important to understand the differences between them. First, the Ratio of Cohesive interactions is a count of the number of methods accessing each of the attributes or association ends. While methods that have attributes in common will likely raise the RCI value, it is not necessary for methods to share common attributes to have a reasonable RCI value. When no attributes are present, the RCI cohesion measure cannot be calculated. Tight Class Cohesion measures how much the methods of the class have in common. Methods must access common attributes / association ends in order to be measured. So while RCI measures how much a method accesses the attributes of the class (and how much the methods belong with those attributes), TCC measures how much the methods have in common (or how much the methods belong in the same class together). TCC doesn't need attributes to be present in order to be measured, but does require more than one method. So a class with many attributes but one method cannot be measured by TCC, but can by RCI, whereas a class with many methods but no attributes has not RCI measure, but has a TCC value.

By measuring the cohesion based on the method's interactions with each other with TCC, and the cohesion based on the method's interaction with the attributes / association ends with RCI, the two cohesion measures complement each other, and can work to present different alternatives for the designer to consider.

7.3 Constraints

In our optimization, several constraints are placed upon the optimization search, as discussed in Section 6.4. There are two general ways to incorporate constraints into a genetic algorithm heuristic [55]. The first is to introduce the constraint into the representation itself. This is known as a "hard constraint", and it prevents individuals that would violate the constraint from being created. The second way to represent constraints in the genetic algorithm heuristic is to allow constraints to be violated by individual solutions, but to penalize the individuals fitness based on the constraints that they violate. This is known as a "soft constraint". This has the benefit of preventing innovative solutions from being removed due to violating the constraints, but it makes the fitness function more complex. More recently, a third method introduced for handling constraints is to use co-evolution [38]. Co-evolution consists of using two or more separate populations, each solving a slightly different aspect of the problem. By having each population run separately then interact and trade strong individuals, they can explore more of the search space, with each population following a different constraint.

In order to handle the constraints outlined in Section 6.4 in our context, hard constraints are used. The optimization heuristic itself will be responsible for ensuring that the solutions returned are valid, and thus there is no need to modify the fitness function to account for invalid individuals. The techniques used to ensure that the solutions remain valid are discussed in Section 8.

7.4 Multi objective fitness ranking

The objective of our search is to optimize the coupling and cohesion metrics in the class model. However, in order to satisfy all of these objectives at the same time, it may be necessary to discover multiple tradeoffs between them to find the best design. This type of problem is referred to as a multi-objective problem (MOP) [52]. 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 (DM) is thus

required in order to choose an acceptable solution (or solutions) by selecting one or more of the solution vectors. MOPs are mathematically defined in [52] as follows:

Definition 47. Multiobjective Problem (MOP)

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

A multiobjective problem consists of *n* decision variables, *m* constraints, and *k* objective of which any or all of the objective functions may be linear or nonlinear [52]. The evaluation function maps the decision variables $\vec{x} = x_1,...,x_n$ to vectors $\vec{y} = a_1,...,a_k$, which may or may not be onto some part of the objective space, depending on function and constraints of the particular MOP.

MOPs themselves are characterized by the objectives that may be (in)dependent and / or incommensurable. The objectives being optimized will often conflict, which places a partial ordering on the search space. This partial ordering 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 MOP, 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, but represent different tradeoffs between the various objectives. Since an MOP consists of a number of objectives, it is necessary to be able to compare the solutions in terms of these objectives. To do so, two key concepts are used, the first being *Pareto optimality,* and the second is *range independence.*

7.4.1 Pareto optimality

Pareto optimality is a concept that was introduced by Vilfredo Pareto in [41, 53]. Using the formalism from [52], the key Pareto concepts can be mathematically defined as follows:

Definition 48. Pareto Dominance

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

Definition 49. Pareto Optimality

A solution $x \in \Omega$ is said to be Pareto optimal with respect to Ω if and only if there is no $x' \in \Omega$ for which $\vec{v} = F(x') = (f_1(x'),..., f_k(x'))$ dominates $\vec{u} = F(x) = (f_1(x),..., f_k(x))$.

Definition 50. Pareto Optimal Set

For a given MOP $F(x)$, the Pareto optimal set (P^*)

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

Definition 51. Pareto Front

For a given MOP $F(x)$ and Pareto optimal set P^*, the Pareto optimal front (PF^*) is defined as:

$$PF^* := \{\vec{u} = F(x) = (f_1(x),..., f_k(x)) \mid x \in P^*\}$$

Pareto optimal solutions are also termed *non-inferior, admissible,* or *efficient* solutions; their corresponding vectors are termed *nondominated* [52]. These solutions form the set of all solutions whose corresponding vectors are nondominated with respect to all other comparison vectors. When plotted in the objective space, the nondominated vectors are collectively known as the *Pareto front*. Dominated solutions are also termed *covered.*

To illustrate Pareto optimality, take the following multi-objective function, from [45]:

$$F_1(x) = x^2; F_2(x) = (x - 2)^2$$

Plotting the two functions gives the graph shown in the figure below:

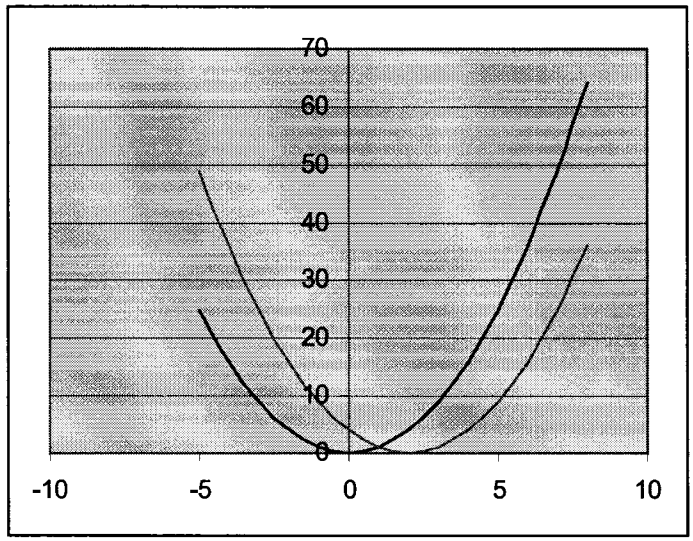


Figure 15 Multi objective function

Assuming that our goal is to minimize both F_1 and F_2, it is obvious in the above graph that the optimal value for F_1 is $x = 0$, and the optimal value for F_2 is $x = 2$. If we let $A(x)$ be a vector, with $A(x) = (F_1(x), F_2(x))$, then we can find the Pareto-optimal values for the multi objective function. Consider the following points:

X	$F_1(x)$	$F_2(x)$
-2	4	16
0	0	4
1	1	1
2	4	0
3	9	1

Table 1 Sample objective values

From the above, we know that A(0) dominates A(-2) since $F_1(0) < F_1(-2)$ and $F_2(0) < F_2(-1)$. Likewise, A(0) dominates A(3), since $F_1(0) < F_1(3)$ and $F_2(0) < F_2(3)$. Similarly, both A(1) and A(2) dominate both A(-2) and A(3). However, A(0) does not dominate A(1) since $F_2(0)$ is not less than $F_2(1)$. A(1) also does not dominate A(0) however, since $F_1(1)$ is not less than $F_1(0)$. So both A(1) and A(0) are in the non-dominated set for this multi objective function. Similarly, A(2) is non-dominated by either A(1) or A(0), so it is also part of the non dominated set.

In the multi objective function presented above, any value $0 \leq x \leq 2$ is contained in the non dominated set. This range of values is known as the Pareto optimal front, since every A(x) in the range is non-dominated. In terms of optimizing the multi objective function, any point $0 \leq x \leq 2$ would be an optimal solution, representing a different tradeoff between the two objectives. Note that in this simple two objective problem, the Pareto optimal front of $0 \leq x \leq 2$ is the global optimal. In real multiobjective problems, this global optimal is not known, and finding it is an NP-Complete problem.

This illustrates why population based algorithms such as genetic algorithms are a good choice for multi objective optimization. Because a range of individual solutions are used 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 in the solution.

The final decision on which objective to use is left with a decision maker, rather than the optimization algorithm or heuristic. The decision maker is left to choose from the solutions found by the heuristic that are in the Pareto optimal set the final solution.

7.4.2 Range independence

Another important concept when performing optimization on multi objective functions is the idea of range independence. Range independence is presented in [4]. The authors state there are two categories for methods for comparing objectives in a multi objective function, the first being range dependent methods, and the other being range independent methods.

In order to discuss range independence, it is first necessary to define the effective range of an objective function. The effective range is the range of values returned by a particular objective in a multi objective problem. The effective range is determined not only by the objective function itself, but also by the domain of the input values into the algorithm, and also the representation of the individual genes themselves. According to [4], the definition of the effective range is as follows:

Definition 52. Effective range

> The effective range of $F(x)$ is the range from $\min(F(x))$ to $\max(F(x))$ for all values of x that are actually generated by the algorithm, and for no other values of x.

The effective range of all of the objectives is occasionally the same, but it is possible that each objective in the problem has a different effective range. Consider coupling and cohesion metrics. Cohesion is usually normalized, and measured in the range [0, 1] whereas coupling is measured in the range $[0, +\infty]$. Thus, the effective range of the two measures is very different.

The only way to ensure that all objectives in a multi objective problem are treated equally by the algorithm is to ensure that all the effective ranges of the objective functions are the same, or to ensure that the objectives are not combined or compared to one another [4]. So the choice is to make the effective ranges of all the objectives equal, and then use a range-dependent ranking method, or use a range-independent ranking method. The full definition of range dependent and range independent ranking methods in [4] is:

Definition 53. Range dependent ranking method.

> Given the objective functions of a problem: $F_{1 \ldots n}(x)$ and a set of solution vectors $\{\vec{u}_1, \vec{u}_2, \ldots, \vec{u}_m\}$. A multi objective ranking method is range-dependent if the fitness ranking of $\{\vec{u}_1, \vec{u}_2, \ldots, \vec{u}_m\}$ defined by the method may change when the effective ranges of $F_{1 \ldots n}(x)$ change.

Definition 54. Range independent ranking method.

> Given the objective functions of a problem: $F_1 \ldots_n$ (x) and a set of solution vectors $\{\vec{u}_1, \vec{u}_2, ..., \vec{u}_m\}$. A multi objective ranking method is range-independent if the fitness ranking of $\{\vec{u}_1, \vec{u}_2, ..., \vec{u}_m\}$ defined by the method do not change when the effective ranges of $F_1 \ldots_n$ (x) change.

Range-independent methods are more widely applicable; as they don't rely on changing the ranges to fine tune the algorithm. Range-dependent ranking 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).

7.4.3 Categories of multi-objective fitness functions

In order to evaluate the MOP of optimizing the class design, it is necessary to use a multi-objective fitness evaluation in the algorithm. There are two general categories of multi-objective fitness functions that can be used for the optimization technique. Those that combine the objective values into a single fitness value, referred to in this paper as *summation approaches*, and those that evaluate the fitness as a vector of individual objective values using Pareto optimality, referred to in this paper as *vector based approaches*. Summation approaches, because they must combine the objectives in order to reach a single fitness value, are range dependent, and 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 approaches. The objectives are compared on an individual basis. However, the comparison of the individuals tends to be more complex in a vector based fitness rank. Vector based approaches also tend not to converge on single solutions, but rather a range of Pareto optimal solutions. This presents the designer with a wider range of optimal solutions to choose from, but these solutions may be inferior to the single solution developed in the summation approach. The following table summarizes the various approaches to the multi-objective optimization search.

Approach	Summation or Vector based	Strengths	Weaknesses
Sum of Weighted Objectives	Summation	Fast. Simple. Discovers strong solutions.	Use of weights is subjective and problem specific. Cannot explore the Pareto Optimal front
VEGA	Vector	Fast. Simple.	Only explores a limit section of the Pareto optimal space
Non dominated sorting	Vector	Uses clustering to ensure a wide range of Pareto optimal solutions found.	Requires a subjective sharing distance.
Strength Pareto Approach	Vector	Uses clustering to ensure wide range of optimal solutions returned. Does not require any input parameters.	Complex algorithm.
Pareto Archived Evolutionary Strategy	Vector	Uses small population.	Complex to implement. Not a genetic algorithm
Non dominated sorting II	Vector	Fast. Discovers strong non-dominated solutions. Uses clustering to ensure a wide range of optimal solutions	Does not perform as well in problems with greater than 3 objectives.
Strength Pareto Approach 2	Vector	Discovers strong non-dominated solutions. Uses clustering. Performs well in problems with many objectives.	Very complex

Table 2 Summary of multi-objective search techniques

7.4.3.1 Sum of weighted objectives

The first method discussed in this thesis for ranking the fitness values of multi objective functions is the sum of weighted objectives. This is a fairly common method of assigning a weight to each objective and then combining them into a single objective function. The basic idea of combining the objectives using weights was first proposed by Zadeh [56], and it is frequently used in [22, 23] for multi objective functions in engineering design and optimization.

The sum of weighted objectives approach is a range dependent approach, and it relies entirely on proper weights being chosen for the objectives in order to determine the area of the search space that will be explored. If the ranges of the objectives are not the same, setting the weights becomes more difficult, as the ranges must be made equal in order to

be combined. Because setting these weights is subjective, the solution becomes problem specific.

However, if the ranges can be normalized, the sum of weights method has some great benefit over the other methods discussed. By combining the objectives into a single fitness function through the use of weights, the sum of weighted objectives approach can find very strong solutions, dominating the solutions returned by some of the other approaches mentioned. However, the weights used limit the area of the search space explored, and thus the sum of weighted objectives is only capable of demonstrating one possible tradeoff between the objectives in each run. To produce different tradeoffs, the algorithm must be executed again with the new weights.

7.4.3.2 Vector evaluated genetic algorithms

Vector evaluated genetic algorithms, or VEGA, is a range-independent method for evaluating multiobjective functions. VEGA was introduced by Schaffer in [45]. The idea was to preserve good solutions in the Pareto-optimal set using a vector fitness representation. This is done by modifying the selection process of a traditional genetic algorithm. Rather than assigning a fitness value, individuals are selected from the population in stages. In each stage, the individuals are ranked according to one objective, and a number of the highest ranked individuals are selected for the next generation for this stage. Figure 16 illustrates the selection process.

The process of shuffling the entire population in Figure 16 refers to randomly ordering the individuals that were selected when creating the sub-populations, so that the resulting population is unordered. Non-dominated solutions would have a high ranking regardless of the objective, and so these individuals will be selected numerous times for the next generation, allowing the algorithm to converge towards these non-dominated solutions.

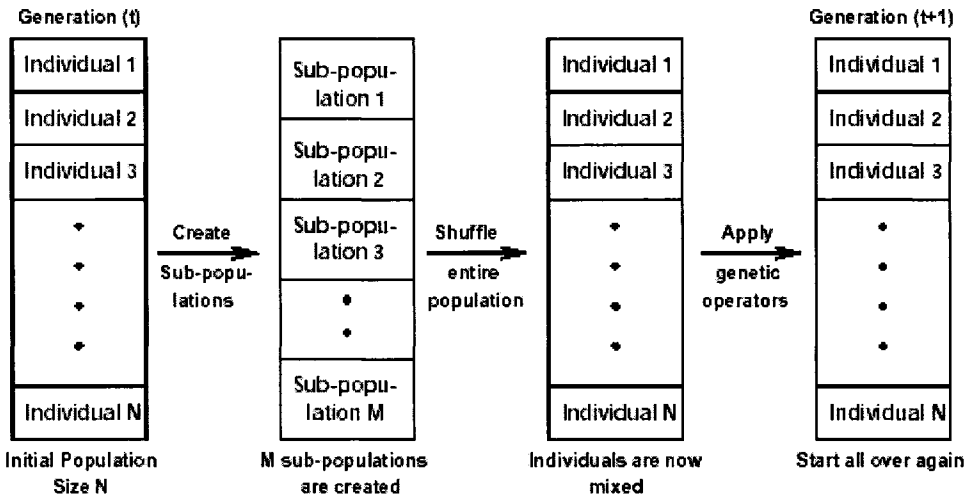


Figure 16 Schematic of VEGA selection [14]. It is assumed that the population size is *N* and that there are *M* objective functions.

In [45], a discovered problem with the VEGA algorithm is that it tends to converge very quickly on a sub-optimal, non-dominated solution. If one or two individuals become non-dominated early in the search, they have the potential of dominating the entire search with an overwhelming selection advantage when they are ranked against the dominated solutions. To overcome this problem, the authors in [45] recommend a random breeding heuristic among the selected individual solutions. This prevents the first non-dominated solutions from taking over the entire population.

Even though in [45], the authors report good performance with VEGA, the main advantage of this approach is its simplicity. However, Coello [14] notes that the shuffling and merging of all the sub-populations corresponds to averaging the fitness components associated with each of the objectives. Since Schaffer used proportional fitness these fitness components were in turn proportional to the objectives themselves. So the resulting fitness evaluation corresponds to a linear combination of the objectives where the weights depended on the distribution of the population at each generation. VEGA is thus a special case of the sum of weighted objectives approach, where each objective is taken equally.

7.4.3.3 Non dominated sorting

The bias of VEGA towards some Pareto optimal solutions, as discussed in Section 7.4.3.2 above, led Goldberg in [24] to recommend a technique known as non dominated sorting. One implementation of this idea of non dominated sorting is the Non-dominated Sorting Genetic Algorithm (NSGA) of Srinivas and Deb [49]. The idea behind the algorithm is to use a ranking selection method to emphasize good points and then apply a niche method to maintain stable subpopulations of these good points.

As with the VEGA, the NSGA is identical to a regular GA save for the selection of the individuals. The algorithm performs the selection of individuals as follows [49]: The nondominated individuals are first identified and are assigned a large, arbitrary fitness value. The same fitness value is applied to give equal reproduction opportunity to all of the nondominated individuals. Then the fitness values of the nondominated individuals are shared based on the number of surrounding individuals. Sharing is done by degrading the fitness of an individual by dividing the original fitness by a quantity proportional to the number of solutions around it. Once the fitness values have been shared, then these nondominated individuals are temporarily removed from the population, and a second nondominated group is found from the remaining individuals. They are assigned a second, arbitrary fitness value that is smaller than the minimum shared fitness value of the previous group. The fitness values of this second group are shared, and the process continues until the entire population has been assigned a fitness value. Selection is performed using these fitness values, and rest of the GA performs as usual.

The NGSA works as shown in Algorithm 1. In the algorithm, σ_{share} is the sharing parameter and represents the maximum distance between any two individuals to become a member of the same niche.

```
Initialize Population
Generation = 0
while(generation < maxGen) do
      front = 1
      while (!populationClassified) do
            identify nondominated individuals
            assign dummy fitness values (see below)
            share in the current front (see below)
            front = front + 1
      reproduction according to dummy fitness values
      apply genetic operators
      generation = generation + 1

assign dummy fitness values:
      if (front == 1)
            fitness = large, arbitrary value
      else
            fitness = minimum fitness of previous front

share in current front:
      for each individual i in population do
            for each individual j in the population do
                  d_{i,j} = distance(i, j)
                  if (d_{i,j} < σ_share)
                        sharingFunction = 1 - (d_{i,j} / σ_share)^2
                  else
                        sharingFuntion = 0
                  nicheCount = nicheCount + sharingFunction
            fitness_i = fitness_i / nicheCount
```

Algorithm 1: NSGA [49]

The NSGA was tested against the VEGA method discussed in Section 7.4.3.2 to compare the two methods. In [49], the authors report that while both VEGA and NSGA found nondominated solutions, VEGA would cluster its individuals around specific points in the nondominated set, whereas NSGA would spread the population out across the Pareto optimal front. This gives a much more diverse range of individual, nondominated solutions, representing a wide range of possible tradeoffs between the included objectives. So while the performance is comparable, the spread of returned solutions is much better in NSGA rather than VEGA.

In addition to providing more solutions, the NSGA approach offers other advantages. There is very little need to modify the standard GA in order to apply the technique, other than the selection method, and the fitness objectives can be used directly rather than having to be combined in order to determine a ranking. Also, because of an even spread of individuals across the nondominated front, a wider range of tradeoffs is present to the user at the end of the optimization.

The main drawback of the NSGA is that in order to implement the sharing technique used in [49] and discussed in [25], it is necessary to determine the sharing distance parameter in order to determine how to share the fitness values. This parameter is subjective, in a similar manner to weights, and can impact the performance of the algorithm.

An improvement to the NSGA, the NSGA-II, was introduced by Deb et. al [18], which eliminates the need for the sharing parameter, and also improves the run time complexity of the algorithm. The NSGA-II is discussed in Section 7.4.3.6.

7.4.3.4 Strength Pareto Approach

The Strength Pareto Evolutionary Algorithm (SPEA) was introduced by Zitzler and Thiele in [58]. The authors perform a comparison of several multi-objective genetic algorithms, including both VEGA and NSGA, and found that when compared to an algorithm using a weighted sum approach, the weighted sum genetic algorithm would produce solutions that dominate 90% of the non-dominated solutions presented by VEGA and NSGA, albeit with around 20 times the computational effort [58]. This prompted them to introduce the Strength Pareto Approach.

The Strength Pareto approach has the following in common with other multi objective genetic algorithms:

- Stores the set of non-dominated solutions that are found so far by the algorithm externally to the genetic algorithms population.

- Uses Pareto optimality to assign scalar fitness values to individuals

- Performs *clustering* (discussed in the algorithm below) to reduce the number of nondominated solutions stored in the external set without destroying the characteristics of the current Pareto front.

According to [58] the Strength Pareto approach is unique in the following ways:

- It is the first algorithm to combine the above three points into a single algorithm.
- The fitness of an individual solution is determined based on the solutions stored in the external nondominated set.
- All solutions in the external nondominated set participate in the selection of the next generation for the genetic algorithm.
- A new *niching* method is provided that is Pareto-based, and does not require a distance parameter.

Niching is a technique used by the Strength Pareto algorithm (among others) in order to create and maintain stable sub-populations (niches) within the population of the genetic algorithm. The aim of the niching technique is to maintain diversity in the algorithms population. The SPEA uses a niching technique that is based on Pareto dominance and is performed when the fitness of each individual is evaluated. The technique itself is presented in the discussion of the fitness assignment below. The overall SPEA algorithm is shown in Algorithm 2.

Step 1: Generate the initial population P and create the empty, external nondominated set P'.

Step 2: Copy nondominated members of P to P'

Step 3: Remove solutions within P' that are dominated by any other member of P'

Step 4: If the number of externally stored nondominated solutions exceeds a given maximum N' (set by the algorithm user), prune P' by means of clustering (presented below).

Step 5: Calculate the fitness of each individual in P as well as in P'.

Step 6: Perform the genetic algorithm selection operation on individuals from P + P' (multiset union).

Step 7: Apply crossover and mutation operators as usual

Step 8: If maximum number of generations is reached, then stop, otherwise go to step two.

Algorithm 2: SPEA [58]

The fitness assignment is done for both the P' and P population sets. The fitness of each individual is based off of its Pareto dominance, the lower the value, the better the individual.

1) Each solution in the P' set is assigned real value [0,1] called its *strength*; The strength is proportional to the number of population members in P which the individual dominates.

2) The fitness of an individual in the P set is calculated by summing the strengths of all external nondominated solutions in P' that dominate the individual. One is added to this value to guarantee that members of P' have better fitness than members of P.

This fitness assignment provides niching based upon Pareto dominance, rather than the fitness sharing based on distance as in NSGA. This is a much simpler technique, eliminating the need for a distance measurement. This is great advantage of the SPEA.

The Pareto optimal set for a given problem can be extremely large or even contain an infinite number of solutions. From the point of view of the decision maker, having to choose the final solution from the set of Pareto optimal solutions becomes useless once the set exceeds reasonable bounds. Also, the size of the external, nondominated set affects the behaviour of the SPEA. The algorithm relies on a uniform spread of individuals in the external population in order to assign fitness values; otherwise it might be bias towards a specific region in the search space. To address this problem, the SPEA uses a method called clustering.

Definition55. Clustering

In general, cluster analysis partitions a collection of m individuals into n groups of relatively homogeneous elements, where $n < m$.

The exact method used by the SPEA is called the *average linkage method* and is presented in [39]. Algorithm 3 shows the method implemented in the SPEA.

Step 1: Initialize the cluster set C; Each external nondominated point i in P' constitutes a distinct cluster.

Step 2: If the size of C is less than a user defined limit, then go to 5.

Step 3: Calculate the distance of all possible pairs of clusters. The distance of two clusters is given as the average distance between pairs of individuals across the two clusters. In [58] the Euclidean metric on the object space is used for distance.

Step 4: Determine two clusters with minimal distance; the chosen clusters amalgamate into a larger cluster. Return to step 2.

Step 5: Compute the reduced nondominated set by selecting a representative individual per cluster. In [58], the centroid individual (the individual with the minimal average distance to all other points in the cluster) is used as the representative solution.

Algorithm 3: SPEA Clustering [58]

The results of the SPEA were favorable when compared to the VEGA, NSGA and the sum of weighted objectives approaches. [58] reports that SPEA is able to find solutions that cover 100% of the nondominated solutions of the other multiobjective GAs for the test algorithms, while the solutions of the other multiobjective GAs cover less than 5% of the nondominated solutions of SPEA. So its performance on the test problems (including the problem introduced in Section 7.4.1) is stronger than the other multiobjective genetic algorithms, as it covers more of the Pareto optimal front.

SPEA is the strongest multi objective genetic algorithm that has been presented so far, but is also the most complex. One of the advantages of the SPEA is that it provides an external population of nondominated solutions, maintained at a size set by the user. This could be used in conjunction with user interaction and feedback to encourage certain design directions. Since the nondominated set will always contain the optimal solutions found thus far, and since these individuals remain in the breeding pool, the user would be able to influence the fitness of the top solutions, which will more likely result in better guidance towards the desired solution. The population can be large enough to maintain diversity while only presenting a small, optimal subset for user evaluation. Since this is already a part of the SPEA, this gives it an advantage over the other techniques seen so far. Also, as already discussed, SPEA provides a better front coverage than VEGA or NSGA, leading to a wider range of Pareto optimal solutions presented.

The SPEA algorithm has three major weaknesses identified in [57]. The first is the fitness assignment. Individuals that are dominated by the same archive members will have the

same fitness values. This becomes a problem when the archive contains a single individual, as all of the population will have the same rank independent of whether they dominate each other or not. As a result, selection pressure is greatly reduced in this case. The second is that the individuals in the current generation are indifferent and don't dominate each other, very little information can be obtained on the basis of the partial order defined in the dominance relation. This situation is likely to occur in the presence of two or more objectives, and density information has to be used to guide the search when it occurs. Clustering makes use of density information, but applies only to the archive and not to the population as a whole. Finally, the algorithm used in the SPEA to maintain the size of the archive tends to remove boundary solutions, which should be kept in the archive for a good spread of nondominated individuals.

In order to address these issues, Zitzler et. al. proposed a improvement to the SPEA, called the SPEA2. The SPEA2 is discussed in more detail in Section 7.4.3.7.

7.4.3.5 Pareto Archived Evolution Strategy

In a break from the multi objective genetic algorithms presented in the above sections, the Pareto Archived Evolution Strategy, or PAES, presented by Knowles and Corne in [30], is not a genetic algorithm at all, but rather an *Evolution Strategy* (ES).

Evolution Strategies are another type of evolutionary algorithm, and were first reported by Rechenberg in [42]. They rely more heavily on mutation of solutions and less on a population of individuals in order to explore the search space. The most basic form, the (1 + 1)-ES, has a population of one individual, and mutates that individual to form the next candidate generation. The superior individual (the original or the mutant) is kept, and the process repeats. This simple algorithm is good at exploring the search space, but finds optimal values much slower than a traditional GA.

The PAES uses an evolutionary strategy combined with an external archive of nondominated solutions. The basic form of the PAES is the (1 + 1)-PAES, which executes in a similar manner to the (1 + 1)-ES. The stages of the PAES are shown in Algorithm 4.

```
Step 1: Generate initial random solution c and add it to archive
Step 2: Mutate c to produce m and evaluate m
      If c dominates m then discard m
      Else if m dominates c replace c with m, and add m to archive
      Else if m is dominated by any member of the archive, discard m
      Else apply test(c,m,archive) to determine new current solution
(discussed below)
Step 3: Repeat Step 2 until termination criterion reached
```

Algorithm 4: PAES [30]

In a similar fashion to the SPEA discussed in Section 7.4.3.4, the PAES uses a clustering technique on its externally stored archive in order to determine which nondominated solutions should be stored for the final result. This allows for a reasonable number of nondominated solutions to be kept while ensuring that a wide range of solutions on the Pareto optimal front are maintained. The test(*c*,*m*,archive) method is used to determine if a solution is accepted and archived or not. The pseudocode for the test(*c*,*m*,archive) method is presented in Algorithm 5.

```
if the archive is not full
      add m to the archive
      if (m is in a less crowded region of the archive then c)
            accept m as the new current solution
      else maintain c as the current solution
else
      if (m is in a less crowded region of the archive then x for some
member x in the archive)
            add m to the archive, and remove a member of the archive
            from the most crowded region
            if (m is in a less crowded region of the archive then c)
            then accept m as the current solution, otherwise maintain c
            as the current solution.
      else
            if (m is in a less crowded region of the archive than c)
            accept m as the new current solution, otherwise maintain c
            as the current solution.
```

Algorithm 5: PAES Clustering Algorithm

As evident in Algorithm 5, the PAES relies fairly heavily on its crowding algorithm for the archive. The crowding procedure is based on recursively dividing up the *d*-dimensional phenotype objective space [30]. It has a low computational cost and is adaptive, so it does not rely on a niching parameter like the NSGA does. When each solution is generated, its grid location is found and noted using a tree encoding. A map of the grid is also maintained, noting the number of solutions residing in any given grid,

referred to as the grids population. The assignment of grid locations is adaptive, adjusting the grid to only cover the range of current solutions in the archive.

While [30] provides the detailed algorithm for the ES and the *test* method, it is very vague on the details of its crowding algorithm. Rather than an algorithm or formula the authors only provide a written description of the adaptive clustering algorithm. In order to make use of the technique, it will be necessary to determine precisely how the adaptive clustering algorithm worked, which may not be possible.

The advantage of the PAES over the other presented algorithms is that it is computationally less expensive than NSGA [30] and similar niching techniques. In addition, since the PAES is an Evolutionary Strategy rather than a Genetic Algorithm, it does not use a population to generate solutions. The archive would only store optimal solutions found so far, rather than contribute to the evolution itself. This simplifies the implementation of the ES. In [30] the performance of the PAES compared favourably to the performance of NSGA on a number of problems.

The fact that the PAES is not a genetic algorithm but rather an evolutionary strategy is a mixed blessing. While in this type of ES (a 1+1-ES) the population is kept at one, and there is no need for crossover, selection or mutation probabilities (or selection and crossover entirely), its reliance upon mutation only may not be the best method for this optimization problem. Without much experience in ES over more traditional genetic algorithms, it is difficult to say whether or not the approach is beneficial for this particular problem or not.

Another major disadvantage of the PAES is its clustering method. Because this method is not described in sufficient detail to duplicate in [30], and the effectiveness of the PAES is dependent upon its clustering method, it is impossible to say how effective the PAES would be given our problem. Since it is necessary to either work to duplicate the clustering method used, or substitute our own, much more work is required to make use of the PAES than other strategies, and it is impossible to predict how well the PAES will perform on the problem.

7.4.3.6 Non Dominated Sorting Algorithm II

The main criticisms for the NSGA (discussed in Section 7.4.3.3) and other nondominated sorting algorithms is their high computational complexity, non-elitism approach and the need for a sharing parameter. In response to these criticism, Deb et. al. proposed a nondominated sorting based multi-objective genetic algorithm called the NSGA-II [18]. This approach addresses all three of the issues with nondominated sorting multi-objective genetic algorithms.

As with the previous multi-objective genetic algorithms, the NSGA-II differs from a traditional genetic algorithm in fitness assignment only. The NSGA-II algorithm can be divided into three parts, the nondominated sort, the density estimation and the crowded comparison operator.

The nondominated sorting aspect of the NSGA-II was designed specifically to have a lower computational complexity than the NSGA, namely $O(mN^2)$. It works as follows: first two entities n_i (number of solutions which dominate solution i) and S_i (a set of solutions which i dominates) is calculated. All points which have $n_i = 0$ are placed in the first front F_1, called the current front. For each solution i in the current front, each member j in the set S_i has its n_j count reduced by one. If the count becomes zero, it is placed in a separate list H. Once all members of the current front have been checked, then the members in the list F_1 make up the first front, and the process is continued by using the newly identified front H as the current front. Algorithm 6 presents the procedure when applied on a population P returns a list of the non-dominated fronts F.

```
Fast-nondominated-sort(P)
for each p ∈ P
      for each q ∈ P
            if (q ≺ p) then
                  S_p = S_p ∪ {q}
            else if (p ≺ q) then
                  n_p = n_p + 1
      if n_p = 0 then
            F_1 = F_1 ∪ {p}
i = 1
while F_i ≠ ∅
      H = ∅
      for each p ∈ F_i
            for each q ∈ S_p
                  n_q = n_q - 1
                  if n_q = 0 then H = H ∪ {q}
      i = i + 1
      F_i = H
```

Algorithm 6 NSGA-II Fast Nondominated Sort [18]

In order to estimate the density of solutions surrounding a particular point, the NSGA-II uses the average distance of the two points on either side of this point along each objective. This quantity, called $i_{distance}$ servers as an estimate of the size of the largest cuboid enclosing the point i without including any other point in the population, and is referred to as the *crowding distance*. Algorithm 7 is used to calculate the crowding distance of each point in the set I.

```
Crowding-distance-assignment( I )
l = | I |
for each i, set I[i]_distance = 0
for each objective m
    I = sort(I, m)
    I[1]_distance = I[l]_distance = ∞
    for i = 2 to (l − 1)
        I[i]_distance = I[i]_distance + (I[i+1].m − I[i−1].m)
```

Algorithm 7 NSGA-II Crowding [18]

$I[i].m$ refers to the m-th objective function value of the i-th individual in the set I.

The crowding comparison operator for the NSGA-II, denoted $\geq_n$ is used to guide the selection process at the various stages of the algorithm towards a uniformly spread out Pareto-optimal front. Assuming that each individual i in the population has two attributes: it's not dominated rank i_{rank} and its local crowding distance $i_{distance}$. The partial order $\geq_n$ is defined as:

$i \geq_n j$ if $(i_{rank} < j_{rank})$ or $((i_{rank} = j_{rank})$ and $(i_{distance} > j_{distance}))$

Between two solutions with differing nondominated ranks the point with the lower rank is preferred. Otherwise, if both points belong to the same front then the point which is located in the region with lesser number of points is preferred.

The NSGA-II main algorithm, which makes use of the three key concepts defined above, works as follows. First the initial parent population P_0 is created. This population is sorted based on the nondomination. Each solution is assigned a fitness equal to its nondominated level (1 is the best level), and the fitness is to be minimized. Genetic operators (selection, crossover, mutation) are applied to create a child population Q_0 of size N. From the first generation onwards, the procedure for the NSGA-II is shown in Algorithm 8.

$R_t = P_t \cup Q_t$

F = fast-nondominated-sort(R_t)

until $|P_{t+1}| < N$

crowding-distance-assignment(F_i)

$P_{t+1} = P_{t+1} \cup F_i$

Sort($P_{t+1}, \geq_n$)

$P_{t+1} = P_{t+1}[0:N]$

Q_{t+1} = make-new-pop(P_{t+1})

$t = t+1$

Algorithm 8: NSGA-II [18]

First, a combined population is formed. The population R_t will be of size $2N$. Then the population is sorted according to nondominance. The new parent population is formed by adding solutions from the first until the size exceeds N. Therefore, the solutions of the last accepted front are sorted according to $\geq_n$ and the first N are picked. This population is then used for the genetic operators are then applied to the population and the generation is advanced.

The overall complexity of the NSGA-II algorithm is $O(mN^2)$, which is better than the NSGA, SPEA, or SPEA2. In addition, NSGA-II is an elitist algorithm, which has been shown by Zitzler, Laumanns and Theile [34] to be greatly beneficial in aiding better convergence in multi-objective genetic algorithms. In [18], Deb et. al. compared the performance of the NSGA-II to the PAES, and showed that the NSGA-II was able to outperform the PAES on a number of problems. Similarly, Zitzler et al. [57] compared the NSGA-II with the SPEA, SPEA2 and PEAS algorithms. [57] shows that the performance of the NSGA-II is better than the SPEA and the PEAS, and provides similar solutions to the SPEA2 algorithm (discussed in Section 7.4.3.7). The only drawback to the NSGA-II algorithm is that it was shown by [57] to perform slightly poorer in problems with a large number of objectives than the SPEA2 algorithm. But the NSGA-II has a much better worst case complexity than the SPEA2 algorithm, and is able to produce comparable results.

7.4.3.7 The Strength Pareto Evolutionary Algorithm 2

After the SPEA technique (discussed in Section 7.4.3.4) was introduced, several potential weaknesses in the approach were identified. In response to these potential drawbacks, as well as the introduction of several new techniques (NSGA-II by [18]; PESA by [16]), the SPEA2 technique was introduced [57]. The SPEA2 algorithm differs from the SPEA technique in the following ways:

- An improved fitness assignment scheme, which takes for each individual into account how many individuals it dominates and it is dominated by.
- A nearest neighbor density estimation technique is incorporated which allows a more precise guidance of the search process.
- A new archive truncation (clustering) method guarantees the preservation of boundary solutions.

The overall algorithm for the SPEA2 is shown in Algorithm 9.

The fitness assignment in the SPEA2 has changed from the assignment in SPEA. The goal is to avoid the situation that individuals dominated by the same archive members have identical fitness values. Each individual i in both the external set $\overline{P}_t$ and the population P_t is assigned a strength value $S(i)$, representing the number of solutions it dominates:

$$S(i) = |\{j \mid j \in P_t \cup \overline{P}_t \wedge i \succ j\}|$$

On the basis of the strength value, the raw fitness $R(i)$ is then calculated for each individual i.

$$R(i) = \sum_{j \in P_t + \overline{P}_t, j \succ i} S(j)$$

Inputs:	N	(population size)
	$\overline{N}$	(archive size)
	T	(maximum number of generations)
Output:	A	(nondominated set)

Step 1: Initalization: Generate an initial population P_0 and create the empty archive (external set) $\overline{P}_0 = \varnothing$. Set t = 0

Step 2: Fitness Assignment: Calculate fitness values of individuals in P_t and $\overline{P}_t$. (Described below)

Step 3: Environmental Selection: Copy all nondominated individuals in P_t and $\overline{P}_t$ to $\overline{P}_{t+1}$. If size of $\overline{P}_{t+1}$ exceeds $\overline{N}$ then reduce $\overline{P}_{t+1}$ by means of clustering, otherwise if size of $\overline{P}_{t+1}$ is less than $\overline{N}$ then fill $\overline{P}_{t+1}$ with dominated individuals in P_t and $\overline{P}_t$. (Described below)

Step 4: Termination: If $t \geq T$ or another stopping criteria is satisfied then set A to the set of decision vectors represented by the nondominated individuals in $\overline{P}_{t+1}$. Stop.

Step 5: Mating Selection: Perform the genetic algorithm selection operator on $\overline{P}_{t+1}$ in order to fill the mating pool.

Step 6: Variation: Apply recombination and mutation operators to the mating pool and set P_t to the resulting population. Increment generation counter ($t = t+1$) and go to Step 2.

Algorithm 9: SPEA2 [57]

This fitness value is to be minimized, where $R(i) = 0$ is a nondominated individual, whereas a high $R(i)$ value means i is dominated by many individuals. In order to ensure that the fitness value is meaningful when most individuals don't dominate each other, additional density information is needed to discriminate between individuals with the same raw fitness values. For each individual I the distances (in objective space) to all individuals j in both the external set and the population are calculated and stored in a list. This list is then sorted in increasing order, and a density estimate is given by 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:

$$D(i) = \frac{1}{\sigma_i^k + 2}$$

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 addition to the new fitness value assignment, the clustering method used by the SPEA has also been changed in the SPEA2. In the original SPEA, the clustering algorithm tended to remove boundary individuals. This has been corrected in the SPEA2 algorithm.

The clustering method is performed during the environmental selection step (Step 3 of the algorithm). In this part, the first step is to copy all nondominated individuals from the external set and the population to the external set of the next generation.

$$\overline{P}_{t+1} = \{i \mid i \in P_t + \overline{P}_t \wedge F(i) < 1\}$$

If the external set fits exactly into the archive (i.e. $|\overline{P}_{t+1}| = \overline{N}$) 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 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 each iteration, the individual chosen for removal for which $i \leq_d j$ for all $j \in \overline{P}_{t+1}$ with

$$i \leq_d j :\Leftrightarrow \forall 0 < k < |\overline{P}_{t+1}| : \sigma_i^k = \sigma_j^k \vee \exists 0 < k < |\overline{P}_{t+1}| : \left[\left(\forall 0 < l < k : \sigma_i^l = \sigma_j^l\right) \wedge \sigma_i^k < \sigma_j^k\right]$$

where σ_i^k denotes the distance of i to its k-th nearest neighbour in $\overline{P}_{t+1}$. In words, the individual which has the minimum distance to another individual is chosen at each stage; if there are several individuals with minimum distance then the tie is broken by considering the second smallest distance, and so forth. How this truncation method works is illustrated in the following figure, from [57].

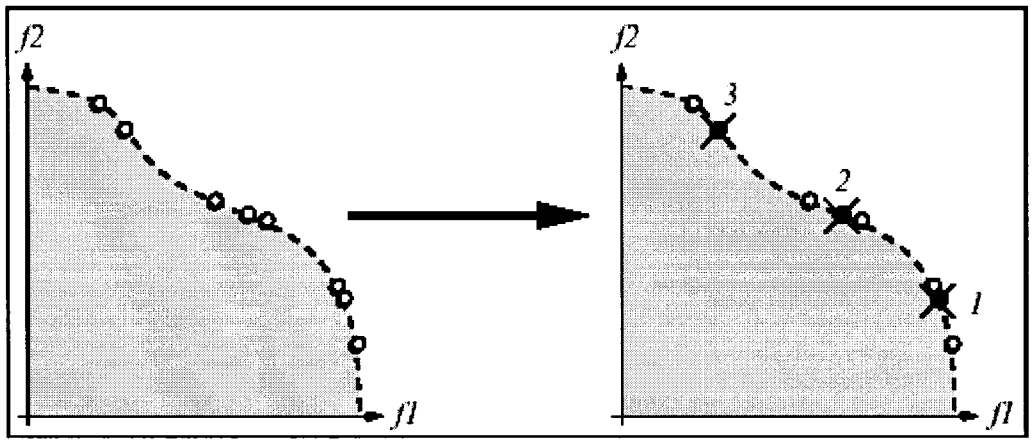


Figure 17 Illustration of archive clustering method used in SPEA2

[57] also includes a discussion of the performance of the SPEA2 algorithm when compared to the SPEA algorithm, the NSGA-II algorithm, and the Pareto Envelope Selection Algorithm (PESA) [15]. The authors show that the SPEA2 performs better than both the SPEA and PESA techniques, and comparably to the NSGA-II technique. The NSGA-II and SPEA2 both have high performance in the test problems in [57], however the authors report that the SPEA2 technique performs slightly better than the NSGA-II algorithm in problems with a high number of objectives.

7.4.4 Comparison of approaches

It is necessary to define several criteria in order to compare the techniques that are presented above. These criteria are as follows:

Time complexity and Scalability: The time complexity of each algorithm can be given by the time complexity of its fitness evaluations. Since all of the algorithms use the genetic algorithm operators, and differ only in how fitness is determined, the complexity of the genetic algorithm is omitted, and the complexity of the fitness evaluation is used to determine the time complexity and scalability.

Parameters needed: Lists the necessary parameters for the algorithm. The inputs for each algorithm are not always trivial, and may have a drastic effect on the performance of the

algorithm. So algorithms that require fewer inputs will be favoured over those that require many or subjective inputs.

Performance in empirical studies: This is a key comparison point of the various algorithms. This criterion indicates how well the algorithm has been shown to perform in empirical studies, assuming that these studies are available.

The following table summarizes the approaches presented in the section above with regards to the criteria discussed.

Technique	Time Complexity	Parameters Needed	Empirical Studies
Sum of Weighted Objectives	$O(N)$	- Objective weights	May outperform VEGA, NSGA
VEGA	$O(N^{\|F\|})$	None	May outperform Sum of Weighted Objectives
NSGA	$O(N^2 - N)$	- Sharing distance	Outperforms VEGA and may outperform Sum of Weighted Objectives
SPEA	$O(N + \overline{N})^2$	- Size of archive	Outperforms NSGA, VEGA and Sum of Weighted Objectives
PAES	$O(N)$	- Size of archive	Outperforms NSGA, VEGA
SPEA2	$O((N + \overline{N})^3)$	- Size of archive	Outperforms SPEA, PAES
NSGA-II	$O(mN^2)$	None	Outperforms SPEA, PAES

Table 3 Summary of fitness assignment approaches

Sum of weighted objectives: The first technique discussed is the sum of weighted objectives. This is the only summation technique used. Although the time complexity of the sum of weighted objectives is good, its overall performance is driven by the proper selection of weights for each objective. Because this technique relies upon weights to make the objective ranges comparable the sum of weighted objectives approach is very problem specific. Also, since the weights determine the tradeoff made between the objectives, the solution returned by the algorithm is heavily dependent on the weights. Finally, these weights are subjective, and difficult to set.

The sum of weighted objectives approach was shown to perform well compared to the VEGA and NSGA techniques in [58], as it was able to find solution which dominated the solutions found by these techniques. [14] reports that the sum of weighted objectives approach is efficient, and can find strongly nondominated solutions, however these solutions require the weights to be set appropriately, which can be a problem if there is insufficient information to determine the weights beforehand. Another drawback of the sum of weighted objectives approach is that it cannot generate Pareto optimal solutions in the presence of non-convex search spaces, which can be a major problem for real world applications. For these reasons, the sum of weighted objectives approach is not suited for the optimization of class designs.

Vector Evaluated Genetic Algorithm (VEGA): VEGA does not have the drawback of requiring weights set by the designer for each objective. However the performance of VEGA has been shown to be poor when compared to the NSGA [18], SPEA [58], and PAES [30]. These techniques are able to produce results that will dominate the results generated by the VEGA. Also, as noted in [43], the shuffling and merging of all the sub-populations corresponds to averaging the fitness components associated with each of the objectives. This means that VEGA, like the sum of weighted objectives approach, is also unable to handle non-convex search spaces. Since other algorithms have been shown to outperform VEGA, it will not be considered for the optimization.

Nondominated Sorting: The NSGA is the first Pareto based algorithm considered. It has been shown to outperform VEGA [49], and has a relatively low computational complexity. Its main strength, according to [14], is its ability to handle multiple objectives, and that it does its sharing in the parameter space, rather than the objective space, which leads to a better distribution of individuals. Its main weakness, however, is its dependency upon the sharing parameter. This parameter is difficult to choose, and has a large impact on the performance of the NSGA. Also, the SPEA has been shown to produce superior results to the NSGA for the 0/1 knapsack problem [58].

Strength Pareto Approach: The SPEA technique is the first algorithm discussed that uses two populations, maintaining an archive of the nondominated individuals. The size of this

archive is the only parameter for the technique, and it has been show to perform well when compared to the sum of weighted objectives, VEGA, and NSGA approaches [58]. The weakness of the Strength Pareto approach is that it is more complex than the other techniques, in that it has a greater time complexity and is less scalable due to the maintenance of two populations. Also, there are several known problems with the SPEA algorithm that affect its performance [57]. The clustering technique used tends to remove boundary solutions, the fitness assignment fails if a single individual dominates many others in the population, and finally if there are many individuals in the population that do not dominate each other then very little information can be gathered based on the dominance relation between these individuals.

With the problems stated above, and due to the fact that a more modern version of the SPEA technique is presented in [57], the SPEA approach will not be used for the optimization problem.

Pareto Archived Evolution Strategy: The PAES is the only technique discussed that is not a genetic algorithm but rather an evolutionary strategy. The PAES is shown to perform favorably when compared to VEGA and NSGA while having a low computational complexity [30]. Even though the PAES maintains an archive of non-dominated solutions, its complexity is comparable to a single population genetic algorithm, since an evolutionary strategy does not use a population. The PAES also uses a clustering technique that does not need a sharing parameter and is computationally inexpensive. Disadvantages to the PAES are that as an evolutionary strategy it relies completely upon mutation to move through the search space, which may result in it getting stuck in local optima unable to escape with only mutation to guide it. The PAES is also unable to move from a good Pareto result to a worse one, since individuals are only considered if they are an improvement. Note that these drawbacks apply only to the (1 + 1)-PAES, but [30] reports that the (1 + 1)-PAES was the most reliable strategy for its test problem, and thus it is reasonable to consider only it and not the ($\mu + \lambda$) variants.

Non dominated Sorting II: The NSGA-II algorithm is the modern version of the NSGA technique. It has a low computational complexity, does not require a sharing parameter,

and is an elitist algorithm similar to the SPEA(2) and the PAES. It has been shown to outperform the PAES and SPEA algorithms, and perform comparably to the SPEA2 algorithm. As the number of objectives increases, however, the SPEA2 is able to perform slightly better than the NSGA-II.

Strength Pareto Approach 2: The SPEA2 algorithm is more modern version of the SPEA technique discussed earlier. The SPEA2 algorithm uses an improved fitness assignment and clustering method than the SPEA, removing the problems associated with population density, fitness assignment when there is only one dominate individual, and the tendency to remove boundary solutions present in the original technique [57]. Of all of the techniques discussed, the SPEA2 is the most computationally complex and the least scalable. The worst case time complexity of the fitness calculation is high, and the fact that the SPEA2 maintains two populations makes it less scalable then some of the other techniques.

The SPEA2 technique is shown in [57] to outperform the SPEA technique on all test problems, and to have comparable performance to the NSGA-II algorithm [18]. However, the results shown in [57] state that the SPEA2 algorithm has advantages over the NSGA-II as the number of objectives increases. The density estimation included in the SPEA2 algorithm improves its performance when faced with a larger number of objectives. So while the performance of the SPEA2 is comparable to that of the NSGA-II technique, in the presence of many objectives, the SPEA2 algorithm appears to be superior.

Given the advantages and disadvantages of each technique discussed above, the algorithm that will be used for the optimization of class designs is the Strength Pareto Approach 2 algorithm (SPEA2). Although it has a relatively high computation complexity, it is not unreasonable. The results from [57] show that the SPEA2 performs better than the other techniques presented, especially when faced with a large number of objectives, as is the case with the class design optimization problem.

8 GENETIC ALGORITHM

As stated in Section 7.4, the multi-objective genetic algorithms vary from a traditional genetic algorithm in terms of how the fitness value is assigned to the individual solutions. The selection, crossover and mutation operators of the traditional genetic algorithm all remain. Having introduced the SPEA2 algorithm in Section 7.4.3.7, which will be used for the class design optimization search, it is now necessary to discuss the remaining aspects of the genetic algorithm in more detail.

To formulate the class responsibility assignment problem as a genetic algorithm, this section will first provide some background into the genetic algorithms themselves. Secondly, this section will discuss the representation of the class design as an individual solution within the genetic algorithm. Finally, a discussion of the genetic algorithm operations is provided.

8.1 Design Representation

In order to perform the optimization search on the class design using a genetic algorithm, it is necessary to encode the design into a format that can be used by the algorithm in order to perform fitness evaluation and genetic operators. This encoding can be divided into two parts, the encoding of the class members and their class assignment, and the encoding of the dependency information. The first is used to form the chromosomes within the genetic algorithm, and the second to form the basis for the objectives used to calculate the individual fitness.

8.1.1 Representing class members

The class members: attributes, methods and association ends, are the basis for the class design. The goal of the optimization is to find the optimal assignment of these class members to classes, given the dependencies (Section 5.1.10) between them. Thus, the individual solutions must represent both the class members, and their class assignment.

This is done by assigning each gene in the chromosome to a specific class member. The class members are assigned an integer identifier value, which corresponds to their position within the chromosome. As an example, assume we have five class members: attribute a1, attribute a2, attribute a3, method m1 and method m2. The id of the class members are a1 = 0, a2 = 1, a3 = 2, m1 = 3, and m2 = 4. This identifier indicates the position of the class member in the chromosome. So in this case, the chromosome would be five genes long, the first gene representing a1, the second a2, the third a3, the fourth m1 and the last representing m2.

In order to encode the class assignment, each gene in the chromosome is given a positive integer value, representing the current class. Genes with the same integer value indicate that the class members the genes represent are assigned to the same class. By using this representation, it isn't possible to have an empty class in the design, as the number of classes within the design is determined by the assignment of the class members. Thus, the constraint that the solution cannot contain empty classes (discussed in Section 6.4.1) will always be satisfied.

To continue the example, the following table shows class members and their id values:

Class Member	Id value	Initial Class
a1	0	Class one
a2	1	Class one
a3	2	Class one
a4	3	Class two
a5	4	Class two
m1	5	Class one
m2	6	Class one
m3	7	Class two
m4	8	Class two
m5	9	Class two

Table 4 Class member example

Given the id values and class assignment shown above, a chromosome representing this solution would be:

{1, 1, 1, 2, 2, 1, 1, 2, 2, 2}

Using the class member list for the Monopoly game model in Appendix A, Figure 29, the initial gene for the Monopoly game would be:

{1,1,1,1,1,1,1,1,2,2,2,2,2,3,3,3,4,4,4,4,4,4,4,4,4,4,4,4,4,5,5,5,5,5,6,6,6,6,6}

Note that the abstract method Square.landedOn(), and the concrete implementations of landedOn(), are not included in the gene above as per the discussion in Section 6.3.

8.1.2 Representing dependency information

Dependency information must be recorded for the design, along with the assignment of the class members. However, unlike the class assignment and class member information, the dependency information does not change during the course of the genetic algorithm. Indeed, the dependency information forms the basis of the coupling and cohesion metrics used, and thus the basis for the fitness function. In order for the search to work, the dependency information cannot change during the course of the optimization. Since the dependency information is static, it can be stored independent of the individual solutions. To track the dependency information, a dependency matrix is constructed.

Each column of the matrix represents a class member, and there is one row per method (since the dependencies tracked only consider methods as clients). A dependency of the method row would be represented by a 1 in the corresponding column.

In other words, each row i would represent a given method, m_i, and each column j would represent an attribute, method, or association end d_j. An entry in the matrix at (i,j) would mean that the method m_i has a dependency with d_j. Note that the dependency matrix only records direct dependencies. From this information, indirect dependencies can be derived for each method.

To illustrate, using the class design presented in Section 4.1 and the method and attribute dependencies in Section 4.2 the dependency matrix illustrated in Figure 30 in Appendix A is generated. A more simple dependency matrix, one involving only 5 attributes and 4 methods, is shown in the table below.

	A1	A2	A3	A4	A5	M1	M2	M3	M4
M1	1	1	0	0	0	0	0	1	1
M2	1	1	0	0	0	0	0	0	1
M3	0	0	1	1	0	0	0	0	1
M4	0	0	0	0	1	0	0	0	0

Table 5 Sample Dependency Matrix

The matrix contains one row for each method, and a column for each attribute, method, and association end.

8.2 Population

An important factor in the genetic algorithm is the size and maintenance of the population of solutions. A genetic algorithm search consists of many individuals, and in the Strength Pareto genetic algorithm an archive of elite individuals is maintained alongside the normal population. The first generation population is randomly created, and the subsequent generations are a result of applying the genetic operations, selection, crossover and mutation, to the first generation. The two factors in population that can affect the results of the genetic algorithm are population size and population maintenance (or replacement).

8.2.1 Population Size

The size of the population has a drastic affect on the efficiency and usefulness of the genetic algorithm. Too small, and there are not enough individuals to adequately explore the search space. Too large, and the cost of running the genetic algorithm becomes much greater, as too much time is spent generating the populations [2]. In addition to this, the SPEA2 algorithm used will have an additional effect on the population size. The SPEA2

algorithm maintains an archive of elite individuals, as well as the population, and the size of this algorithm will have an effect on the effectiveness of the algorithm.

The size of the population in the algorithm, along with the size of the archive, will remain fixed throughout the heuristic. Determining the ideal population size is challenging [2], and for traditional genetic algorithms a variety of population sizes has been suggested [27]. Some recommend a range of 30 to 80 individuals [26], while others suggest a smaller population size, around 20 and 30 [46]. For the multi-objective genetic algorithm, authors use a larger population size than those recommended for single objective GAs, and they increase the population size proportional to the number of objectives [57]. Following these suggestions, the population size for the genetic algorithm will be 64 individuals per objective. The effect of the population size on the results is investigated in case study #3 (Section 8.4).

The archive size also has an important effect on the performance of the multi-objective genetic algorithm. The size of the archive determines how elitist the algorithm is. In [34] the authors examine the effect of elitism on the performance of the genetic algorithm, and report that strong elitism together with a high mutation rate should be used to achieve best performance. In [34] they investigated elitism as the percentage change that fit individuals from the previous generation are carried over into subsequent generations. In the SPEA2 algorithm, the archive is first filled with non-dominated individuals. If any space remains, strong dominated individuals from the population are used to fill the archive. In terms of elitism, when the archive is filled with nondominated individuals, then the algorithm is purely elitist (only the strongest individuals survive to the next generation). However, when the archive is filled with both nondominated and dominated individuals, then the elitism is lower, as dominated individuals from the population move over to the next generation. The smaller the archive, the higher the elitism, as the archive would quickly fill with the nondominated solutions.

Unfortunately, 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 [35, 44, 57]. To keep computation time within reasonable bounds, we therefore set the archive size to half the size of the population.

8.2.2 Population Maintenance

Given that the population size (and the archive size) will be fixed within the genetic algorithm, the individuals in the population and the archive must be replaced as new individuals are introduced by the genetic operators. This is known as population maintenance. However, the maintenance of both the population and of the archive is already handled by the SPEA2 algorithm (as discussed in Section 7.4.3.7), and thus no additional work is needed by the genetic algorithm in order to perform the population maintenance.

8.3 GA Operations

The following section outlines the genetic algorithm operations that will be used for the SPEA2 multi-objective genetic algorithm heuristic. The three operations are selection of the chromosomes for breeding, crossover of the selected individuals to create candidate solutions for the next generation, and finally mutation of the candidate individuals to introduce innovation into population.

8.3.1 Selection

The selection of two candidate chromosomes for breeding is performed based on the fitness of the individuals. As discussed in Section 7.4.3.7, individuals are selected from the archive, and not the population itself, for breeding. By selecting only from the archive, only the elite individuals in the population are selected for breeding. There are many methods of selecting individuals for crossover. The various methods differ in how they select the parents based on the fitness value. In the SPEA2 algorithm, the selection method used is a binary tournament selection [57].

Tournament selection is based around taking number of tournaments equal to the population size, and selecting a number (2 or more) of individuals for each tournament. The individuals for each tournament are chosen randomly from the population. Once in

the tournament, the individuals "compete" against each other in order to be selected. The fittest individual is selected 90% of the time, and 10% of the time one of the remaining individuals is chosen at random [24].

This is good because there is no sorting since the tournament participants are selected at random, and there are a number of independent trials equal to the populations' size. The type of tournament that is used in the SPEA2 algorithm is called a binary tournament [57], where each tournament involves only two individuals, selected at random from the population. In the case of the SPEA2 algorithm, the individuals are selected from the archive.

8.3.2 Crossover

Crossover is the act of combining two individuals selected from the population in order to create two new child individuals. This combination must be done in such a way to preserve the validity of the chromosome, which in the case of our representation (presented in Section 8.1) is not a concern. Because the representation does not require any form of maintenance in order to produce valid individuals, a simple and effective crossover method to use is called one-point crossover [24]. This is the crossover method used in both the SPEA and SPEA2 algorithm where it seems to work fine for the existing case studies.

One point crossover involves selecting a single gene, at random, within the chromosome, and the two chromosomes swap genes based on the pivot point selected, in order to produce two new individuals. Figure 18 outlines how one-point crossover works.

The crossover rate is a determining factor in the performance of the genetic algorithm. 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 [27]. De Jong [17] concluded that a desirable crossover rate for a traditional GA should be around 60%. Grefenstette et al. [26] built on De Jong's work and found that the crossover rate should range between 45% and 95%. Consistent with these findings, we use a crossover rate of 70%.

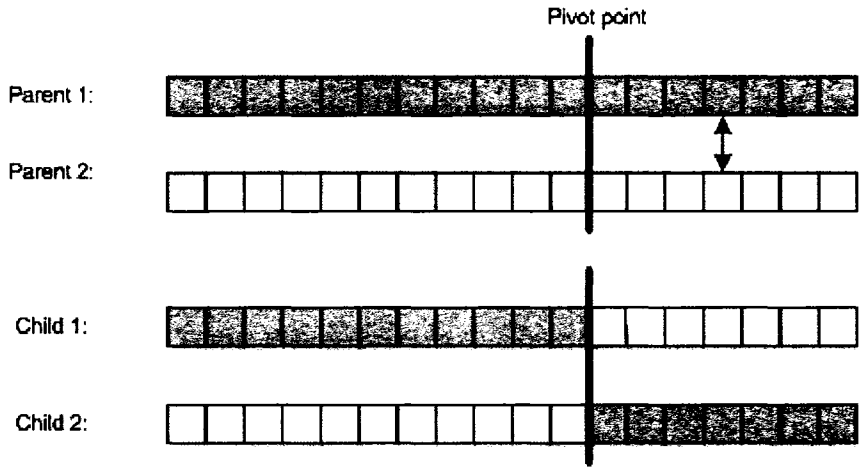


Figure 18 One Point Crossover

8.3.3 Mutation

Mutation is the final genetic operation that is performed during the creation of the new population in the genetic algorithm heuristic. Each new individual, when being introduced into the population, has a chance of being mutated, or having one or more of its genes changed at random. The mutation operation is how innovation is introduced into the genetic algorithm, and how it is able to search a wider area of the search space.

As stated previously, when using an elitist algorithm such as the SPEA2 a high mutation rate is desirable to achieve optimum performance [34]. The authors suggest a mutation rate based on the length of the chromosome to achieve an average of five mutations per chromosome, or 5 / (*length*), where *length* is the length of the chromosome. These findings are consistent with [48]: mutation rates based on the chromosome length perform significantly better than those that are not. Based on these findings, a mutation rate of 5 / *length* is used at first. However, after some experimentation, we found that the mutation rate was too high, and lead to unstable results. This prevented to the SPEA2 from finding good solutions quickly. Thus, a lower mutation rate of 1 / (*length*) was used instead, which seems to lead to more stable results. However, this thesis does not investigate fully the effect that varying the mutation rate has on the effectiveness on the results of the algorithm. Further case studies are needed to determine what the effect the

mutation rate has on the results of the search, and are discussed in Section 10.1 (future work). With this mutation rate, each chromosome is mutated once on average.

The valid changes that can be performed on the design are discussed in the change model in Section 6. From these changes, the possible mutations can be deduced for the individuals. Each of these changes focuses on the assignment of the class members to the classes (discussed in Section 6.1.3). Since the main focus of the search is to find the optimal assignment of class members to classes based on the coupling and cohesion metrics, it makes sense the mutation operations focus on the class assignment.

The mutation used changes the class that a given member is assigned to. This is done on a gene by gene basis. Each gene in the chromosome has a 1 / *length* percent chance of being selected for mutation, where *length* is the length of the chromosome. When it is determined that the gene (class member) should be mutated and moved into a different class, it is reassigned to another class at random. A new class can be introduced into the solution at this point (as per Section 6.2.1). Each class in the model has an equal chance at having the class member assigned to it, including the new class if the user defined constraints allow it (see Section 5.3).

As an example, assume that the following individual was selected for mutation (based on the first sample chromosome shown in Section 8.1.1).

{1, 1, 1, 1, 2, 2, 2, 2, 2}

If the second gene was selected for mutation, then the attribute a2 would have an equal probability of being assigned to class 1, class 2 or a new class. For the sake of this example, assume that the new class (class 3) is chosen, so the resulting chromosome would be:

{1, 3, 1, 1, 2, 2, 2, 2, 2}

9 CASE STUDY

In order to validate the use of a multi-objective genetic algorithm approach to the problem of automated design improvement, a series of case studies were performed. The goals of the case studies were to:

1. Validate that the approach was an efficient means of determining the best possible trade-offs for improving a design.

2. Ensure that not only are the metrics of the system improved, but that the changes made to the model are meaningful, and the final trade-off presented to the design contains these meaningful changes.

3. Test the effect of some of the parameters used (Section 8) in order to determine the impact of these parameters on the search results.

To achieve these goals, three separate case studies were performed. The first case study involved making several suboptimal modifications to an existing, well designed system. The goal is to determine if the changes made by the design optimization algorithm are meaningful, and are capable of returning to the original, optimal design. The second case study involved comparing the selected SPEA2 algorithm to other search algorithms. [13] recommends that in order to apply a complex metaheuristic such as a genetic algorithm to search based software engineering problem, a simple hill climbing technique should be applied. If this hill climber is able to perform somewhat adequately, then a more complex search can be considered. Finally, the third case study involves varying the parameters discussed in Section 8, to try and determine their effect on the results of the search.

This section describes the case studies that were performed in order to evaluate the use of a multi-objective genetic algorithm on the problem of class design improvement. The section is organized as follows. Section 8.1 describes the implementation of the SPEA2 algorithm, known as the Evolutionary Software Design (ESD) tool. Section 8.2, 8.3 and

8.4 describe the first, second, and third case studies respectively, and finally Section 8.5 concludes the section.

9.1 Tool Implementation

The SPEA2 algorithm presented in Section 6 and Section 7 was implemented in a tool in order to determine the effectiveness of using a multi-objective genetic algorithm. The tool implements a genetic algorithm which uses the SPEA2 fitness assignment and archive, as discussed in Section 7.4.3.7 and Section 7.4.4. The genetic algorithm itself was configured using the chromosomes, population, and selection, crossover and mutation operators discussed in Section 8.

The tool, called the Evolutionary Software Design (ESD) tool, was written in Java. In order to simplify the implementation, an existing genetic algorithm framework was extended to include the SPEA2 algorithm. The framework used is the Java Genetic Algorithm Package [1], or JGAP, and is an open source framework available under the LGPL and Mozilla Public license.

The framework included the binary tournament selector and one-point crossover operations, along with the population management used for a traditional genetic algorithm. In order to implement the SPEA2 algorithm, it was necessary to extend the framework to include a new genotype, chromosome, gene implementation and mutation operation. In addition to these aspects, a fitness function evaluator and comparator were added to apply the JGAP framework within our context. These extensions were done as generic as possible, to keep the SPEA2 implementation within the JGAP framework from being problem specific. This allows the implementation to be reused on other multi-objective problems, and not only of the class design optimization. The UML class diagram for the JGAP extension is shown in Figure 31 in Appendix B and this aspect of the implementation is discussed in full detail in Section 9.1.1.

The problem specific aspect of the domain class model design improvement, specifically input model (class members, dependency information, user defined constraints) shown in

Section 5 were also implemented. The UML class design for these domain specific concepts is shown in Figure 32 in Appendix B.

The ESD tool works by reading in the existing model information (class members, class assignment and dependency information) from an XML file, and then using a singleton factory to translate the model information and configure the MOGA. The genetic algorithm is then executed using the parameters specified within a configuration file, and the results of the evolution are decoded from the MOGA back into the model information, and outputted as an XML file containing the non-dominated results from the evolution.

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. In the ESD tool, the designer would do that by specifying a range for the cohesion and coupling measures in order to prune extreme solutions clearly favouring a specific coupling or cohesion measure at the expense of 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 necessarily sacrifice one for gains in the other. It is possible for the designer to specify the range of acceptable values, in order to remove extreme solutions and to obtain a reasonable number of returned solutions for designer evaluation.

9.1.1 JGAP Extensions

In the spirit of possible reuse of the SPEA2 implementation used in the ESD tool, the tool was designed to be as generic as possible when including the SPEA2 archive and fitness evaluation. Not only does this allow the use of the existing genetic operators present in the JGAP framework, but it also means that the resulting framework could be reused for other multi-objective genetic algorithm problems. Thus, one of the goals of the implementation was to extend the framework to include the SPEA2 multi-objective genetic algorithm. The class diagram for these extensions is shown in Figure 31 of

Appendix B, and this Section briefly describes the extensions and how to apply them to other MOGA problems.

The core of the JGAP framework is the `Genotype` class. This is the class responsible for the evolution of the genetic algorithm. This class was extended to include the archive. The `SPEAGenotype` is the driver for the genetic algorithm. It will only work with a bulk fitness function (that evaluates the entire population at one time), and the `SPEAFitnessFunction` should always be used. The `SPEAGenotype` is responsible for managing the archive, and handles the environmental selection aspect of the SPEA2 algorithm (Section 7.4.3.7).

The `SPEAFitnessFunction` extends the `BulkFitnessFunction` of the JGAP framework, and is responsible for the SPEA2 fitness assignment. The `SPEAFitnessFunction` uses the `Objective` interface in order to calculate the raw fitness value and assign the SPEA2 fitness.

To implement a multi-objective genetic algorithm using the SPEA2 and JGAP framework, simply implement the `Objective` interface to represent each objective in the problem, create the `SPEAFitnessFunction` using these Objective implementations, and then use the `SPEAFitnessFunction` and the `SPEAFitnessEvaluator` in the JGAP configuration. Also, rather than using the `Genotype` provided in JGAP, the `SPEAGenotype` must be used. The remainder of the implementation of the genetic algorithm (Chromosomes, genes, crossover, mutation, etc.) remains as it was in JGAP, and more information can be found on implementing a genetic algorithm within JGAP on the project website [1].

9.2 Case study one: Converge towards optimal design

The first case study analyzes the application of the SPEA2 multi-objective genetic algorithm to the problem of domain class design in a qualitative manner. Determining if the ESD tool is able to make meaningful improvement to the quality of a design is the goal of this first case study.

In this case study, a well designed system is used to analyze the changes made by the algorithm. In order to determine if these changes can indeed lead to better design quality, a number of sub-optimal changes are introduced into the design of the system. The algorithm then takes these sub-optimal designs, and attempts to improve upon them. The goal of the case study is to have the algorithm converge on the original, optimal design.

For this case study, the ARENA system was used [11] 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 the algorithm should 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[1] and consists of 14 classes and 138 class members (methods, attributes and association ends). Of these 138 class members, 49 cannot be altered by the genetic algorithm as they are overridden or implemented through generalization relationships (Section 6.3). Another 42 class members are formed into 14 groups with the association ends that they manipulate (Section 5.3.2). The search space, assuming the number of classes does not change, is therefore all the possible assignments of movable (grouped) class members (i.e., 138-49-(42+14)=61) to 14 classes and its size is therefore considerably large: 14^{61}. An analysis of this domain model shows a total of 287 dependencies, specifically 63 method-attribute dependencies, 59 method-association end dependencies, and 165 method-method dependencies. The coupling and cohesion measurement values for this domain model, which represent the baseline on which the algorithm is 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). The analysis level class diagram of the ARENA system that is used in the case study is shown in Figure 20 of Appendix C.

---

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

As mentioned previously in Section 9.1 the large number of solutions returned present a problem when analyzed by a designer. The number of returned solutions is therefore trimmed using a range of coupling and cohesion measures specified by the designer in order to prune extreme solutions. In this case study, the goal was to determine if the optimization could return to the original, optimal design, and thus a range was placed on the measures accordingly. For the purposes of this case study, only solutions that had values of at least 15 for method-attribute coupling, 105 for method-method coupling, 0 for method-generalization coupling, 0.18 for ratio of cohesive interactions and 0.2 for tight class cohesion were retained for evaluation. These solutions are referred as *within-range* solutions. The goal was to avoid solutions that optimize cohesion at the expense of large coupling increases.

The modified designs are discussed below. For each of the modified designs, the ESD tool was run for a total of 200 generations, with a population of 64 individuals per objective (320) and an archive size of half the population. The parameters for the genetic algorithm were configured as discussed in Section 7. Each of the case studies was run on a Pentium 4 3.0GHz processor with 1GB of RAM. The following sub-sections outline the changes made to the original design and the results obtained from the execution of the ESD tool. A discussion of the trends from each of the results concludes this section.

9.2.1 First Change

The first change to the ARENA system involved moving three methods, without their supporting attributes, into a new class. The negative effect of this change is twofold, it lowers cohesion by introducing a new class, and it raises the coupling by moving the methods away from the attributes they use. A class diagram showing this change is shown in Figure 19.

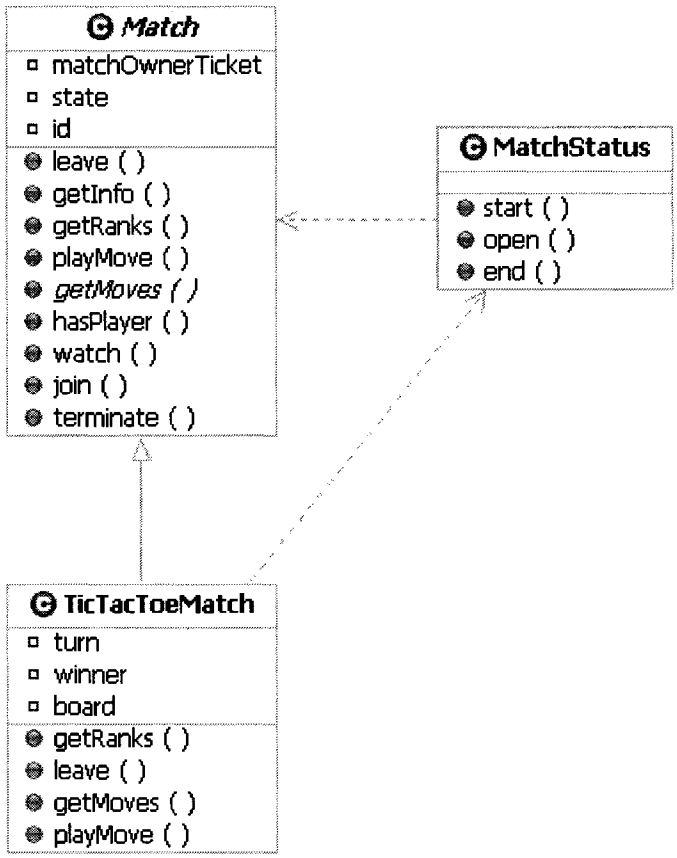


Figure 19 Change #1 to the ARENA design

The three class members of the `Match` class were moved from `Match` over to a new class, called `MatchStatus`. Table 6 below shows results of the algorithm at 50, 100, 150 and 200 generations. The table includes the time it took to reach that generation (14 minutes for generation 50, 55 minutes for generation 200), and the best, worst and average metrics for the within range solutions. Values for MGC are not shown as within range solutions must have a value of 0 for this measure. Figure 20 shows the number of within-range solutions returned per generation.

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 6 Summary of Change #1 Results

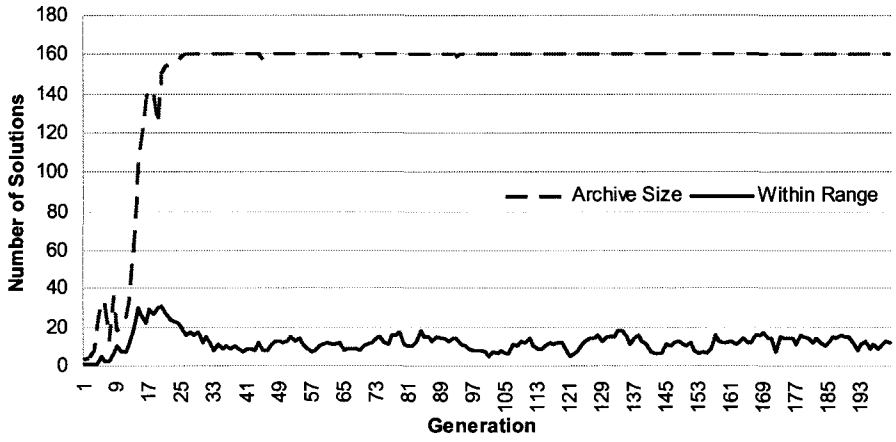


Figure 20 Within Range Solutions by Generation for Change #1

From Table 6, 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.

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.

9.2.2 Second Change

For the second change, attributes of the `Round` class were moved from `Round` to `Match`. Round and Match are closely related, and the attributes moved are used by methods in both classes. The effect of the moved class members on the metrics 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 heuristic can return the model to the original design. A diagram illustrating this change is shown in Figure 21.

Table 7 shows the results of the algorithm at 50, 100, 150 and 200 generations, showing the best, worst and average metric values for the within range solutions as before (MGC not shown). As for the previous change, most of the improvements are obtained before 100 generations and the coupling and cohesion measures are better than the original values for three of the measures. We observe the same class responsibility assignment alternatives as those discovered by the GA for Change 1. Figure 22 shows the number of within range solutions in the archive per generation.

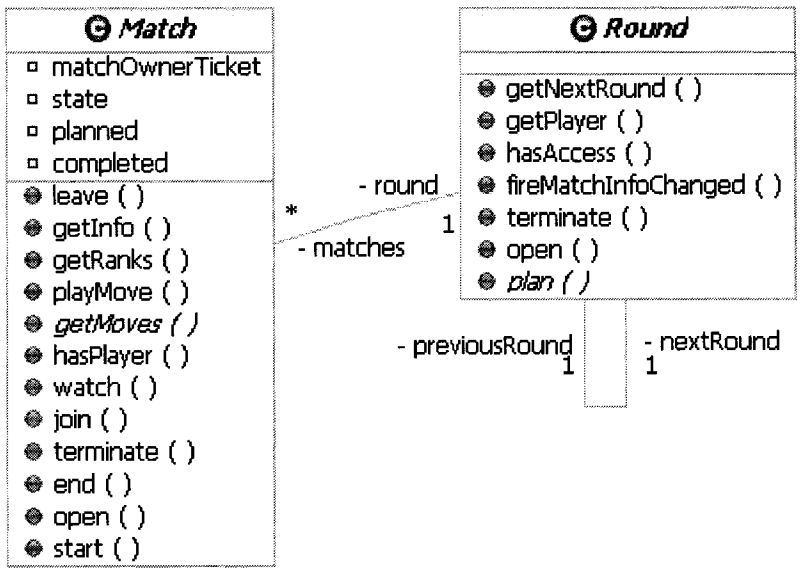


Figure 21 Change #2 to the ARENA design

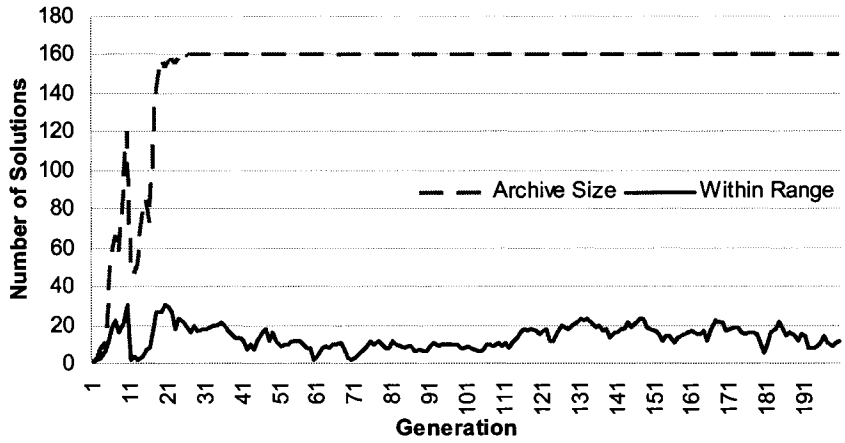


Figure 22 Within Range Solutions by Generation for Change #2

Figure 22 shows a pattern similar to Figure 20 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 20, 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.

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 7 Summary of change #2 results

9.2.3 Third Change

This third change involved moving an association end and its grouped class members from a single class into a newly created class. This group was moved from Tournament, and into a newly created class called TournamentPlayers. The grouped class members are closely 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 23 shows the change to the original arena design.

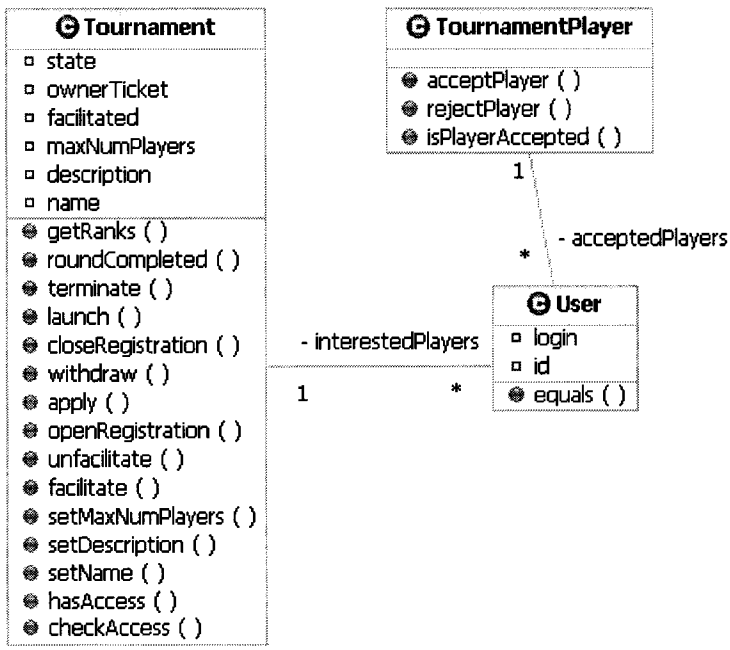


Figure 23 Change #3 to the ARENA design

Table 8 shows the results from the third change, in the same format as the previous two changes (MGC not shown), and Figure 24 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.

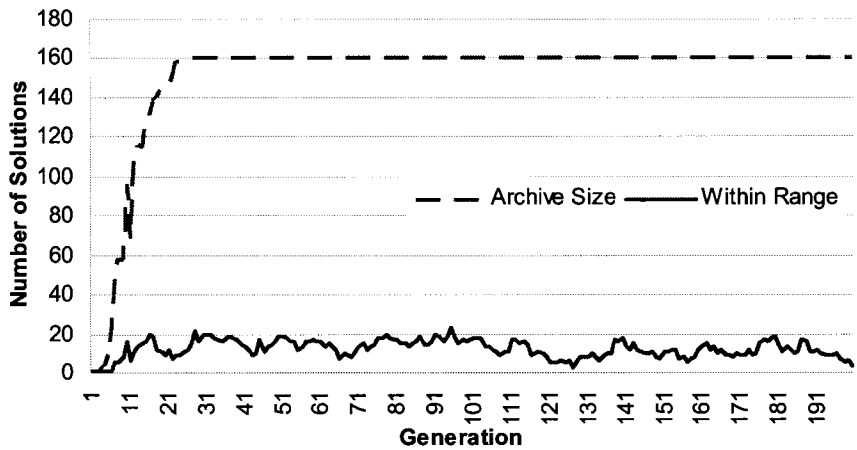


Figure 24 Within Range Solutions By Generation for Change #3

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 8 Summary of change #3 results

9.2.4 Fourth Change

The fourth change is not a new change in its own right, but rather a combination of the previous three changes. All of the modifications made to the design in the previous three changes were placed into the model for change 4. The goal is to determine how well the search performs when faced with a large number of sub-optimal changes in a design. Table 9 summarizes the results of the search, in the same format as the previous three changes (MGC not shown), and Figure 25 shows the archive and the number of within range solutions.

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 9 Summary of change #4 results

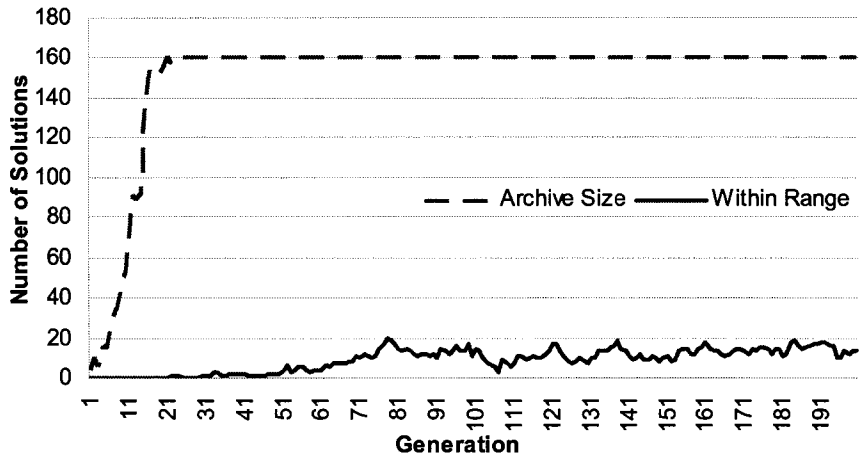


Figure 25 Within Range Solutions by Generation for Change #4

The conclusions that we can draw are similar as those for the other three 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.

9.2.5 General Discussion

The same pattern emerged for the three changes regarding the number of within range solutions returned per generation, as shown in Figure 20, Figure 22, Figure 24 and Figure 25. 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 favours 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 five objectives.

Table 10 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).

	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 10 # Within Range, % Equivalent Solutions

Table 10 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 7.4.3.7) 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 solutions). This is further illustrated by Table 11 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 10), 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 11 Recovering from individual changes

9.3 Case study two: Comparison with other heuristics

The goal of the second case study is to determine if the results returned by the MOGA are worth the use of such a complex heuristic, and to show if comparable results can be achieved using a more simplistic algorithm. To this end, the MOGA heuristic was compared to two other algorithms: a random search algorithm and a random mutation hill climber.

In [13], the authors recommend that a hill climber algorithm be applied to any search based software engineering problem. The motivation behind this is that hill climber algorithms have been shown to perform well when compared to more complex genetic algorithm, and can produce good results even in the presence of an unfavourable landscape. If the simple hill climber does not improve upon the results of a random search, then the problem may not be suited to a search based approach. Thus, the random search is included as a sanity check, to compare with the other algorithms to determine the suitability of a search based approach. The comparison to a hill climber algorithm

will further validate the application of a search based approach to the responsibility assignment problem, and determine if a complex multi-objective genetic algorithm is required.

In order to compare the three algorithms, the fourth change from the first case study, detailed in Section 9.2.4, will be reused. As discussed previously, the size of the search space for this problem is 14^{75}.

The random search algorithm consists of generating a number of solutions at random, without guiding the search in any way. This is used as a sanity check to ensure that the MOGA is producing superior results with its guided search.

The hill climber algorithm used is called a Random Mutation Hill Climber (RMHC) [20]. The RMHC is a simple algorithm that makes single, random changes to a starting solution to generate neighbours, and then compares these neighbour solutions in order to determine an optimal value. The algorithm starts with the original class assignment. It then moves one class member (chosen at random) from its current class to another, randomly selected class. If moving that class member resulted in an improvement in the design metrics, then the new solution is taken as the current best, and the algorithm repeats until a stopping condition is met.

Of importance is that the nearest neighbour hill climbing algorithm is a single objective algorithm. It is necessary to normalize the objectives, and apply weights, in order to create a range independent single objective problem. In order to create the single objective fitness function, the coupling measures are normalized over the amount of possible coupling in the system. This is determined as the number of DMA and DMM dependencies in the system (Section 5.1.10) for MMC, MAC and MGC (Section 7.1) respectively. The cohesion metric is already normalized, and does not need to have its range adjusted. The resulting single objective fitness function is defined in Definition 56.

Definition 56. Single objective fitness function

The single objective fitness function for the nearest neighbour and sum of weighted objectives algorithm for a given system S, denoted $F(S)$, is given by:

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

where: $RCI(S)$ is the Ratio of Cohesive Interaction for the system S

$TCC(S)$ is the Tight Class Cohesion of the system S.

$MAC(S)$ is the Method Attribute Coupling of the system S

$MMC(S)$ is the Method Method Coupling of the system S

$MGC(S)$ is the Method Generalization Coupling of system S

$DMA(S)$ is the set of all direct method attribute dependencies in the system S.

$DMM(S)$ is the set of all direct method method dependencies in the system S.

w_1, w_2, w_3, w_4, w_5 are weights

$F(S)$ is to be maximized.

In order to be comparable, each algorithm will be executed as equally as possible. To do so, the number of overall fitness evaluations will be used to ensure that each algorithm has an equal opportunity to explore the search space. In the SPEA2 algorithm, the results for the algorithm at 100 generations will be used, with an overall population size 320 (as per the configuration outlined in Section 7). This gives a total of 32,000 individual fitness evaluations. To be equal, the random search will generate 32,000 random individuals, and the random mutation hill climber will perform 32,000 exploratory changes. The random

search algorithm and its results are discussed in detail in Section 9.3.1, and the hill climber algorithm in Section 9.3.2.

9.3.1 Random Search

The random search algorithm was an extremely simplistic algorithm. It randomly generates a class assignment for the 61 class members, records the metrics and the solution, and then generates a new solution. This is repeated until the maximum number of fitness evaluations is completed.

Initially, the random search algorithm was used to generate 32,000 candidate solutions, the same number of fitness evaluations as generation 100 of the SPEA2 algorithm. This failed to produce any within range solutions, using the same ranges discussed in the first case study (Section 9.2). In an attempt to get a within range solutions from the random algorithm, the number of solutions generated was increased first to 10 times the number in the SPEA2 algorithm, or 320,000, and then to 20 times the number of fitness evaluations in the SPEA2 algorithm, or 640,000 random solutions. Even with this large number of random solutions, not one solution was within range.

Given the size of the search space, the fact that the random algorithm was unable to discover a within range solution is not surprising. The use of a random search algorithm is simply a sanity check to show that the search space is large and complex enough that a guided search is required, and a simplistic solution such as a random search is not capable of producing meaningful results.

9.3.2 Random Mutation Hill Climber

Where the random search algorithm was used as a sanity check to prove the size and complexity warrants the use of a guided search, the random mutation hill climber [20] will be used to compare the complex and costly SPEA2 algorithm to a more simplistic, faster guided search technique. This case study will examine if the complexity and cost associated with the use of the SPEA2 technique is justified for this problem.

Hill climber algorithms, in general terms, are simplistic greedy algorithm that explores a problem by examining neighbouring solutions. From a starting point, the hill climber explores nearby solutions, and then chooses the neighbouring solution that gives the most improvement. This moves the search in the direction of greatest improvement; hence it is a greedy algorithm. Hill climber algorithms typically stop when they reach a local maximum, and no neighbouring solutions offer any further improvement. This is one weakness of hill climbers, since they have no way of knowing if the local optimum achieved is the global optimum. Search spaces with many such local optima are difficult for hill climbers to explore.

Another challenge of hill climber algorithms is determining what a "neighbouring solution" is. In the case of class member assignment problem, a neighbouring solution is one that has one class member assigned to a different class than the original. To discover these neighbours, it would be possible to change one class member at a time, assigning it to each other class (along with a new class), and measure for any improvement. Another possibility is to randomly select a class member to change and the class to assign it to, similar to the mutation operator used in the genetic algorithm (Section 8.3.3). This is known as a random mutation hill climber [20]. The Random Mutation Hill Climber has been shown to outperform a traditional genetic algorithm on certain problems, and to offer a more competitive comparison to a GA than other hill climber algorithms [20].

A random mutation hill climber generates a neighbour solution by randomly choosing a class member then assigning it to a class at random. To find the neighbour that offers the most improvement, 500 random neighbours are generated, and the best of these 500 is used as the new candidate. If no improvement is found in these 500 solutions, then the current candidate is considered the best solution. This simplifies the evaluation of neighbour solutions and allows the control over the number of neighbours evaluated at any given step. However, the random element means that no run of the hill climber will be the same, and results of the hill climber will vary each time. This means that the hill climber will need to be run a number of times in order to determine if it is able to find any reasonable solutions.

To summarize, the algorithm for the Random Mutation Hill Climber (RMHC) [20] is presented in Algorithm 10.

```
currentBest[] = initialSolution[]
while (!stoppingCondition) {
      for (i = 0; i < 500) {
            neighbour[] = generateRandomNeighbour()
            if (neighbour.fitness > currentBest.fitness)
                  currentBest = neighbour
      }
}

generateRandomNeighbour() {
      position = select random class member
      new_class = select random class assignment
      neighbour[] = currentBest[]
      neighbour[position] = new_class
      return neighbour
}
```

Algorithm 10: RMHC [20]

The fitness function used is the single objective fitness function presented in Section 9.3, Definition 56. The weights are set in order to equally balance each of the objectives. Given 5 objectives, each objective is weighted to contribute 20% to the overall fitness ($w_i = 0.2$).

The goal of the RMHC is to compare it with the SPEA2 algorithm at generation 100 when attempting to solve the fourth change of case study 1 (Section 9.2.4). To do so, the hill climber case study was run three separate times. After each experiment, the hill climber algorithm was improved to attempt to give the hill climber the best possible chance of success. The sections below outline the changes made to the algorithm during each case study, and finally the results of the three experiments are compared with the SPEA2 algorithm. The overall goal of each experiment is to generate an equal number of within-range solutions as the SPEA2 at generation 100.

9.3.2.1 Experiment One

In the first execution of the RMHC, the stopping condition was fixed at 32,000 fitness evaluations, rather than stopping when a local maxima is reached. This would, in theory, allow the algorithm to move away from local optimum positions, and increase the chances of the algorithm finding a global optimal value. 32,000 fitness evaluations were chosen since this is the number of evaluations performed by generation 100 in the SPEA2 algorithm.

The random element of the hill climber means that each run is different, and several runs are required to determine how well the algorithm performs. However, after 40 runs, with 32,000 fitness evaluations per run, the hill climber was unable to produce any within-range solutions.

9.3.2.2 Experiment Two

In the second experiment, the stopping condition was changed back to the usual stopping condition for hill climbers i.e. stop when a local optimum is reached. To ensure that the hill climber will still be comparable to the GA, 32,000 fitness evaluations were still included as a maximum number of possible evaluations. So the algorithm stops if no improvement is found in the 500 neighbour solutions, or if 32,000 fitness evaluations have been reached.

Once again, the hill climber was executed a number of times, in order to attempt to generate the 11 within range solutions. This time, after 40 executions of the RMHC, sufficient within range solutions had been found in order to compare with the SPEA2.

9.3.2.3 Experiment Three

A final change that was made to the RMHC was to adjust the fitness function to better reflect the criteria for within range solutions. Specifically, within range solutions are not permitted any method-generalization coupling. In order to reflect this in the single objective fitness function, the weight of the MGC metric was increased to heavily penalize solutions that have MGC above zero. The weight for MGC (w_5) was increased

from 0.2 to 10.0, without changing any of the other weights. Thus, any MGC would dramatically decrease the fitness of the solution. The stopping condition for the RMHC was unchanged from the second experiment.

The new RMHC was run 40 times, in order to be comparable with both the second experiment and the SPEA2 algorithm. Once again, the RMHC was able to find a number of within-range solutions.

9.3.2.4 Results and Discussion

While the first experiment with the RMHC was not able to produce any within range solutions, the second and third experiments were both able to produce sufficient number of within range solutions to compare with the SPEA2 algorithm.

The SPEA2 algorithm was able to produce 11 within-range solutions by generation 100; experiment one of the RMHC produced no within range solutions; experiment 2 generated 11 within range solutions in 40 runs; finally experiment 3 of the RMHC produced 15 within range solutions in 40 runs. Table 12 shows for each algorithm, the number of class members that were fixed in each solution. Figure 26 compares the overall number of class members fixed for each of the algorithms.

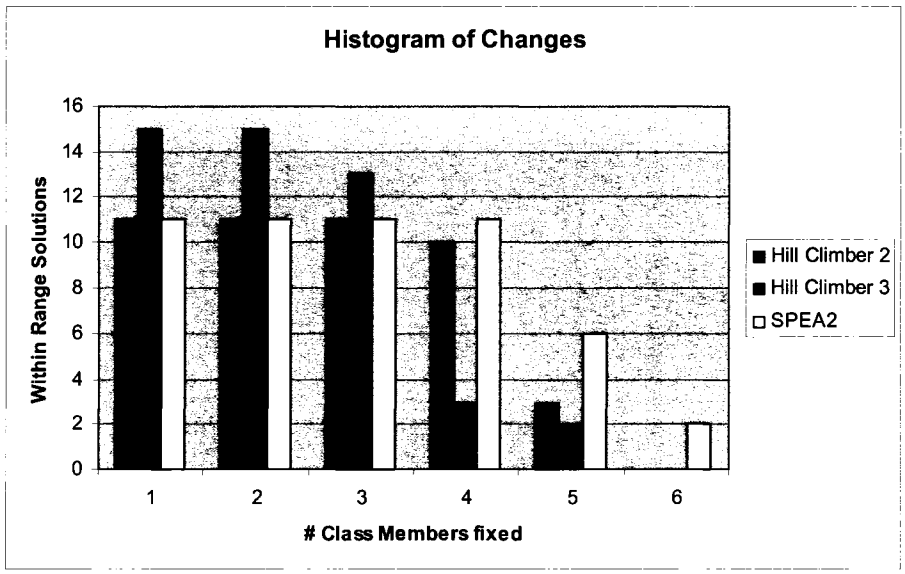


Figure 26 Comparison of class members fixed

Solution #	Hill Climber 1 Class Members Fixed	Hill Climber 2 Class Members Fixed	Hill Climber 3 Class Members Fixed	SPEA2 Algorithm Class Members Fixed
1	0	5	2	6
2	0	3	3	6
3	0	4	3	4
4	0	4	3	5
5	0	4	3	4
6	0	5	3	5
7	0	4	5	5
8	0	4	4	5
9	0	4	2	4
10	0	5	4	4
11	0	4	3	4
12			5	
13			4	
14			3	
15			3	
Average:		4.1818	3.3333	4.7272
St.Deviation:		0.6030	0.8997	0.7862
Variance:		0.3636	0.8095	0.6181

Table 12 Class members fixed by Solution

As shown in Figure 26, the SPEA2 algorithm was the only algorithm able to return all 6 class members to their original location, and all of the within range solutions for the SPEA2 algorithm fixed four or more class members. The RMHC from experiment two performed better than the other two RMHC algorithms, fixing 5 class members on three occasions, and averaging 4 class members fixed. The third RMHC algorithm, with its altered fitness function, was able to find more within range solutions, but tended to fix only 2 -3 class members, so it didn't perform as well as the second RMHC algorithm or the SPEA2 algorithm.

It is important to note also the time it takes to generate these solutions. Each execution of the RMHC produces only a single solution, and it took 40 executions of the hill climber to generate an equal number of within range solutions as the SPEA2 algorithm found in just one run. Thus, while the more simplistic and faster RMHC is able to find within range solutions (albeit with only a one in four chance), it takes a significant number of runs to get the same results as a single run of the SPEA2 algorithm. Considering that, on

average, the RMHC algorithm requires two minutes to find a solution; it takes roughly 80 minutes to generate an equal number of within range solutions as the SPEA2. From Table 9, we know that it took only 21 minutes for the SPEA2 algorithm to get the eleven within range solutions.

9.3.3 Results

Sections 9.3.1and 9.3.2 show the two algorithms, a random search and a random mutation hill climber, that were compared with the SPEA2 algorithm. The random algorithm was unable to produce any results, but given the size of the search space, it was expected that a guided search would be necessary in order to discover any reasonable solution. The random search was more of a sanity check, to be sure that the complexity of the search space was what we expected.

The random mutation hill climber was able to produce some meaningful results, but when compared to the SPEA2, especially considering it took 40 runs in order to create an equal number of within range solutions. So while a basic, single objective guided search is able to find some results, the SPEA2 algorithm is able to find more results in less time than it would take the hill climber to find an equal number.

Overall, the fact that the random search algorithm was unable to produce any results, and that the random mutation hill climber could only find an acceptable solution one out of every four attempts shows that that search space is sufficiently large and complex that the use of a complex multi-objective optimization technique is warranted. The SPEA2 is able to perform much better on the class member assignment problem then these simpler algorithms.

9.4 Case study three: Effect of population size

The performance of a genetic algorithm is affected by the many parameters that are necessary, such as crossover rate, mutation rate, elitism intensity, and population size. Specifically, population size is a difficult parameter to set for a multi-objective genetic algorithm. While for traditional genetic algorithms recommendations have been made by

[26] and [46], no such studies on the effect of population sizes of MOGAs exist that we know of. So it is important to investigate the effect of population sizes on the results of the algorithm.

In Section 8.2.1, a population size of 64 individuals per objective is chosen. Based on the work of [57], we increased the number of individuals based on the number of objectives. In order to determine the effect of the population size on the SPEA2 algorithm, the extremes of the recommended range will be used. [26] recommends that a population size between 30 and 80 be used. This case study will determine the effect of having a population size of 30 individuals per objective and 80 individuals per objective, and compare these with the results for the population size of 64 used in the first case study.

In order to determine the effect of the population size on the results, the rest of the parameters discussed in Section 8 will remain the same. Of particular note, the size of the archive will remain proportional to the size of the population in order to keep the elitism intensity the same. So the archive will be kept at half the size of the population.

As before in the second case study, the base results we will compare to are the results for the fourth part of case study one (Section 9.2.4). This will be used to represent the results obtained for a population size of 64. Table 9 and Figure 25 show the results for the population size of 64 individuals.

Table 13 summarizes the results for 30 individuals per objective (150 total population size and an archive size of 75) for the fourth change, using the same format as Table 9 (MGC not shown). Figure 27 shows the within range solutions for this population size.

As shown in Figure 27, the number of within range solutions is lower for the initial generations than in the case of the 64 population size results. Also, the overall number of within range solutions found is lower, but this is expected as the archive is smaller for the 30 individual run. However, the most telling comparison is in Table 13 and the quality of the within range solutions for the 30 individual population size.

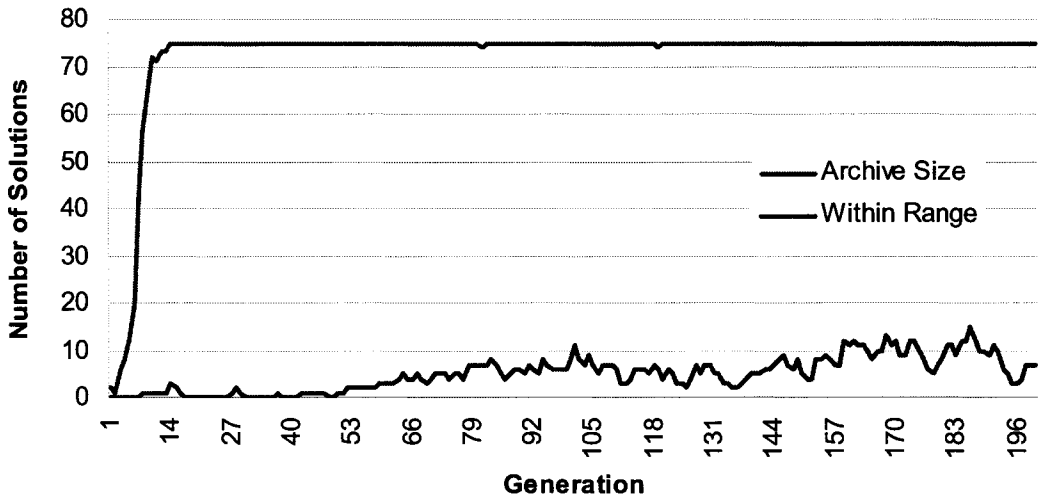


Figure 27 Within Range Solutions for Change #4, 30 individuals per objective

50 Generations (4 mins)	**Average**	**Best**	**Worst**
Method - Attribute Coupling	13.0000	13.0000	13.0000
Method – Method Coupling	105.0000	105.0000	105.0000
Ratio of Cohesive Interactions	0.2939	0.2939	0.2939
Tight Class Cohesion	0.2939	0.2843	0.2843
100 Generations (8 mins)			
Method - Attribute Coupling	11.0000	8.0000	13.0000
Method – Method Coupling	100.8750	94.0000	105.0000
Ratio of Cohesive Interactions	0.4351	0.4795	0.3331
Tight Class Cohesion	0.4351	0.5310	0.3452
150 Generations (14 mins)			
Method - Attribute Coupling	8.4000	6.0000	10.0000
Method - Method Coupling	96.4000	92.0000	100.0000
Ratio of Cohesive Interactions	0.4643	0.5455	0.4194
Tight Class Cohesion	0.4643	0.6010	0.4197
200 Generations (22 mins)			
Method - Attribute Coupling	10.5714	6.0000	15.0000
Method - Method Coupling	97.4286	94.0000	103.0000
Ratio of Cohesive Interactions	0.4821	0.5344	0.3736
Tight Class Cohesion	0.4821	0.5973	0.3798

Table 13 Summary of results for change #4, 30 individuals per objective

The metrics don't show the same improvement as the 64 population size run and no within range solution found in the 200 generations is able fix the fourth change. So 30 individuals is too small for the algorithm to be able to sufficiently explore the search space, and discover the within range solutions that fix the fourth change.

Table 14 and Figure 28 below show the results for the 80 individuals per objective population size (400 total population size and an archive size of 200), in the same format used above (MGC not shown).

50 Generations (15 mins)	**Average**	**Best**	**Worst**
Method - Attribute Coupling	9.409091	6	15
Method - Method Coupling	96.59091	91	105
Ratio of Cohesive Interactions	0.393723	0.520587	0.292397
Tight Class Cohesion	0.393723	0.571016	0.317145
100 Generations (31 mins)			
Method - Attribute Coupling	7.666667	5	14
Method - Method Coupling	97.61905	91	105
Ratio of Cohesive Interactions	0.47766	0.554421	0.370711
Tight Class Cohesion	0.47766	0.615363	0.441827
150 Generations (45 mins)			
Method - Attribute Coupling	9.785714	5	15
Method - Method Coupling	97.64286	92	104
Ratio of Cohesive Interactions	0.479173	0.57025	0.408401
Tight Class Cohesion	0.479173	0.604968	0.502087
200 Generations (60 mins)			
Method - Attribute Coupling	9.461538	6	15
Method - Method Coupling	101.8462	98	105
Ratio of Cohesive Interactions	0.532166	0.577359	0.474875
Tight Class Cohesion	0.532166	0.653192	0.571704

Table 14 Summary of Results for Change #4, 80 individuals per objective

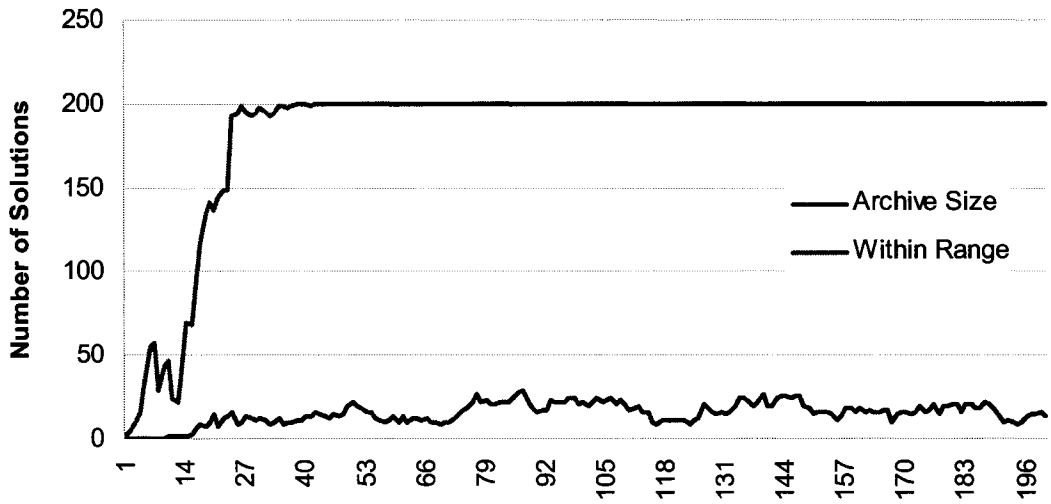


Figure 28 Within Range Solutions for Change #4, 80 individuals per objective

Unlike the 30 individuals per objective results, the metric values shown in Table 14 are similar to those shown in Table 9, making the improvement performed by the 80 population size algorithm similar to those for the 64 population size. Likewise, the number of individuals in the archive, shown in Figure 19 and Figure 16, are comparable, with the same overall pattern and number. Even though the archive size is larger in the 80 individual run, the algorithm does not a much greater number of within range solutions. Also, the increased population size has, as expected, lengthened the amount of time required to execute the search, from 46 minutes to an hour. Finally, and most unusual, the 80 individuals per objective population size was not able to find any within range solutions that are equivalent to the original ARENA design. Thus, the 80 population size algorithm was not able to fix change #4.

The 80 individuals per objective population size took an extra 15 minutes, produced roughly the same number of within range solutions, and it could not revert the fourth change back to the original ARENA design. Having a large population size increased the cost of the SPEA2 algorithm, without providing any significant benefit. Indeed, the quality of the solutions went down.

The results of this case study match the results reported by [2], that is too small of a population and the search space cannot be fully explored, but too large a population size

and the cost becomes too great. Thus, our use of 64 individuals per objective, leading to a population of 320 individuals and an archive size of 160 individuals was the correct choice, giving a good trade-off between the exploration of the search space and the cost of the algorithm.

10 FUTURE WORK

This thesis presents our initial work on applying a multi-objective evolutionary algorithm to the class responsibility assignment problem. While we provide a first step in formalizing and applying an evolutionary algorithm to design optimization, and making use of a multi-objective algorithm for search based software engineering, there are several areas for improvement to the current technique that are outside the scope of this thesis. Some of the possible paths for future work are presented below. First, Section 10.1 discusses next steps in terms of further validating the approach presented in the thesis; Section 10.2 discusses possible improvements to the scope of the optimization; and finally Section 10.3 presents some improvements to the multi-objective genetic algorithm.

10.1 Validation of current approach

Ideally, the next validation step is to apply this 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 multi-objective genetic algorithm would then be very subjective. This makes the model selection difficult.

Another possible validation step of interest is to compare the SPEA2 with other algorithms beyond the simplistic hill climber and random search presented in the second case study (Section 9.3). Specifically, the sum of weighted objectives genetic algorithm is known to be capable of producing strong Pareto-optimal results, assuming the fitness function is properly balanced. Also, the NSGA-II [18] algorithm has been shown to have comparable performance to the SPEA2 [57] as discussed in Section 7.4.4 and it would be beneficial to perform a comparison of the two techniques.

Finally, while case study three investigated the effect of population size on the effectiveness of the genetic algorithm, other important parameters such as elitism

intensity and mutation rate may also have an impact on the performance of the search have been left unexplored. The effect of these parameters on the genetic algorithm should be investigated.

10.2 Scope of the optimization

An important research topic is related to how to handle inheritance hierarchies, as the simplifying assumption to not consider overridden class members was used in this approach. The ability to optimize inheritance hierarchies is extremely important when considering the domain class design, and for this technique to be applicable, the assumptions made restricting the optimization to class members that are not overridden must be addressed.

The scope of the optimization is currently limited to domain / analysis models (the PIM). Obviously, the approach of applying a multi-objective optimization technique to software design optimization is not necessarily limited to only the domain class diagram. However, other types of models have different requirements and objectives. For example, it is common that control classes don't have any attributes, and have chiefly methods that manipulate domain level objects. Optimizing the design for classes outside the domain class model will have different requirements and objectives. While the technique may remain the same, the metrics have to be changed to reflect the goals at each level of the design. The formalism provided in this thesis forms the basis for measuring domain class models, and the scope can be expanded using the multi-objective optimization technique, with objectives balanced to each level of the design. For example, our fitness function should account for design pattern at lower level design. The measures of coupling and cohesion are not as efficient when measuring lower level design.

Another important improvement for the scope of the optimization is to include the automated updating of sequence diagrams and OCL contracts once a final, optimal solution is discovered and chosen. The heuristic should be able to use the input information (class members, dependency information, sequence diagrams and OCL contracts) to automatically determine what changes are required based on the class members moved.

In regards to the scope of the work, another possible area for improvement is the assumption that the current class members are sufficient to describe the domain. That is, class members are not added or removed from the design, only modified. It is not necessarily the case that all of the necessary class members are present in the design, and new class members may be necessary. Currently, our approach cannot determine if a class member is missing from the design, or if an extra class member can be removed from the design. However, since the dependencies are determined from the OCL contracts and sequence diagrams (Section 3) it may be possible to use these to introduce and remove class members. The OCL contracts could be adjusted to either include or exclude the modified class member, where appropriate, as could the sequence diagrams. This would expand the scope of the search technique, and remove the assumption that the current class members and dependencies are sufficient to represent the domain fully.

10.3 Improvements to the MOGA

There are several areas of interest regarding possible improvements to the multi-objective genetic algorithm. One is to devise ways for the designer to efficiently interact with the genetic algorithm, in order to guide the algorithm towards desirable solutions. This is known as a user-interactive genetic algorithm, and it has been applied to other areas of engineering in order to allow designers to guide the search towards a desirable solution (Section 2.3). However, an interactive approach to genetic algorithms such as this has not been yet been applied to a multi-objective genetic algorithm.

A second area of interest is how to avoid the loss of possibly good solutions when the archive is truncated during the SPEA2 environmental selection process. It is possible that optimal solutions are being removed to the clustering algorithm applied. By maintaining these solutions, the performance of the multi-objective algorithm could possibly be improved.

11 CONCLUSION

The assignment of responsibilities to classes is a difficult skill to teach and to acquire in practice. While many methodologies exist for assigning responsibilities to classes, they all rely upon human judgement and decision making. This is partially driven by the difficultly in balancing the many factors that make up a good design, and the subjective nature of software design in general.

The goal of this thesis is to provide decision making help to the re-assign class members to classes in a design. The approach is based on carefully selected coupling and cohesion measures (Section 7) based on a number of different types of dependencies (Section 5.1), and makes use of a multi-objective genetic algorithm (Section 8). The cohesion and coupling measures form the building blocks of the fitness function used by the MOGA (Section 7). In this context, there is no meaningful way to combine the selected measurements characterizing the quality of a model. However, recent proposals have suggested a number of approaches for dealing with multiple objectives in a Pareto-optimal based manner in the context of genetic algorithms. Based on careful analysis and comparison of these alternatives (Section 7.4.4), the SPEA2 algorithm was used. The SPEA2 algorithm is used to generate a set of domain models, representing Pareto optimal trade-offs between the coupling and cohesion measures. The designer is then in a position to select an appropriate solutions from among the design alternatives that the SPEA2 puts forward.

A number of case studies were performed to validate the effectiveness of applying a multi-objective genetic algorithm to the problem of class responsibility assignment. The first case study (Section 9.2) demonstrated that when mistakes of a varying spread and magnitude were introduced into a correct domain model, they could be corrected and a variety of acceptable solutions could be optioned within a reasonable number of generations and time. This demonstrates that the multi-objective genetic algorithm is able to fix a variety of artificially seeded assignment problems, which is a significant step towards validating the approach. One issue though is to help prioritize or select from the

alternative trade-offs proposed by the genetic algorithm in a cost effective manner. The current solution is to allow the designer to define an acceptable range for coupling and cohesion measurement which seems, in the case study, to be constraining enough to limit the number of solutions proposed to the user. A significant number of these solutions then turned out to be equivalent to the target optimal model.

The second case study (Section 9.3) examined the size and complexity of the search space, and compared the multi-objective genetic algorithm to other, more simplistic algorithms. A sanity check was performed using a random search algorithm, which after a significant amount of time and effort was not able to discover any optimal solutions. This shows that the search space is sufficiently large that a guided search is necessary and a simplistic algorithm will not suffice. The multi-objective genetic algorithm was compared to a random mutation hill climber, which was based on a single objective formulation of the coupling and cohesion measures. In this case, the genetic algorithm was shown to outperform the hill climber, both in the quality of the results returned and in the time taken to generate a reasonable number of within range solutions. This case study shows that the use of a complex multi-objective genetic algorithm is justified given the size of the search space. The genetic algorithm is able to produce the best results within a reasonable number of generations and time.

In summary, our thesis has presented an initial multi objective genetic algorithm designed to aid in the assignment of responsibility to classes in the analysis level of class design (the PIM). It does so by determining the optimal assignment of class members (methods, attributes and association ends) based on complimentary measures of coupling and cohesion. A prototype tool, implementing our approach, was developed, and numerous case studies were performed. These case studies have shown that a complex multi-objective genetic algorithm is suitable and beneficial for use in the class responsibility assignment problem. These are initial results, and future work will further examine the validity of the multi-objective genetic algorithm, while expanding the scope of the optimization search and the capabilities of the genetic algorithm itself.

REFERENCES

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

[2] Atallah M. J., *Handbook of Algorithms and Theory of Computation*, CRC (Chemical Rubber Company) Press, 1999.

[3] Bentley P. J. and Corne D. W., "An Introduction to Creative Evolutionary Systems," in P. J. Bentley and D. W. Corne, Eds., *Creative Evolutionary Systems*, Morgan Kaufmann, pp. 1-77, 2002.

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

[5] Bieman J. M. and Kang B.-K., "Cohesion and reuse in an object-oriented system," *Proc. ACM Symp. Software Reusability (SSR'95)*, pp. 259 - 262, 1995.

[6] Bleuler S., Braek M., Thiele L. and Zitzler E., "Multiobjective Genetic Programming: Reducing Bloat Using SPEA2," *Proc. Proceedings of the Congress on Evolutionary Computation 2001 (CEC'2001)*, 1, pp. 536-543, 2001.

[7] Briand L., Daly J. and Wust J., "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Engineering*, vol. 3, pp. 65 - 117, 1998.

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

[9] Briand L., Devanbu O. and Melo W., "An Investigation into Coupling Measures for C++," *Proc. 19th International Conference on Software Engineering (ICSE '97)*, pp. 412 - 421, 1997.

[10] Briand L., Morasca S. and Basili V., "Measuring and assessing maintainability at the end of high-level design," *Proc. IEEE Conference on Software Maintenance*, 1993.

[11] Bruegge B. and Dutoit A. H., *Object-Oriented Software Engineering Using UML, Patterns, and Java*, Prentice Hall, 2nd Edition, 2004.

[12] Buelow P. v., "Using Evolutionary Algorithms to Aid Designers of Architectural Structures," in P. J. Bentley and D. W. Corne, Eds., *Creative Evolutionary Systems*, Morgan Kaufmann, 2002.

[13] Clark J., Dolado J. J., Harman M., Hierons R., Jones B., Lumkin M., Mitchell B. S., Mancoridis S., Rees K., Roper M. and Shepperd M., "Reformulating software engineering as a search problem," *Journal of IEE Proceedings - Software*, 2003.

[14] Coello C. A. C., "A Comprehensive Survey of Evolutionary Based Multiobjective Optimization Techniques," *Knowledge and Information Systems. An International Journal*, vol. 1 (3), pp. 269-308, 1999.

[15] Corne D. W., Knowles J. D. and Oates M. J., "The Pareto Envelope-Based Selection Algorithm for Multi-objective Optimisation," *Proc. 6th International Conference on Parallel Problem Solving from Nature*, pp. 839-848, 2000.

[16] Corne D. W., Knowles J. D. and Oates M. J., "The pareto envelope-based selection algorithm for multi-objective optimization," *Proc. Parallel Problem Solving from Nature - PPSN VI*, pp. 839 - 848, 2000.

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

[18] 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 - PPSN VI*, pp. 849 - 858, 2000.

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

[20] Forrest S. and Mitchell M., "Relative building-block fitness and the building-block hypothesis," in D. Whitley, Ed., *Foundations of Genetic Algorithms 2*, Morgan Kaufmann, pp. 109-126, 1993.

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

[22] Gen M. and Cheng R., *Genetic Algorithms and Engineering Design*, John Wiley & Sons, 1997.

[23] Gen M. and Cheng R., *Genetic Algorithms and Engineering Optimization*, John Wiley & Sons, 2000.

[24] Goldberg D. E., *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley, 1989.

[25] Goldberg D. E. and Richardson J., "Genetic Algorithms with sharing for multimodal function optimization," *Proc. Second International Conference on Genetic Algorithms*, pp. 41 - 49, 1987.

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

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

[28] 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 (CEC'2005)*, 1, pp. 236-242, 2005.

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

[30] Knowles J. D. and Corne D. W., "Approximating the nondominated front using the Pareto Archived Evolution Strategy," *Evolutionary Computation*, vol. 8 (2), pp. 148 - 172, 2000.

[31] Koza J. R., Bennett F. H., Andre D. and Keane M. A., "Genetic Programming: Biologically Inspired Compuation That Exhibits Creativity in Producing Human-Competitive Results," in P. J. Bentley and D. W. Corne, Eds., *Creative Evolutionary Systems*, Morgan Kaufmann, pp. 275-298, 2002.

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

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

[34] Laumanns M., Zitzler E. and Thiele L., "On The Effects of Archiving, Elitism, an Density Based Selection in Evolutionary Multi-objective Optimization," *Proc. First International Conference on Evolutionary Multi-Criterion Optimization (EMO 2001)*, Volume 1993 of Lecture Notes in Computer Science, pp. 181 - 196, 2001.

[35] 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 (CEC'2005)*, 1, pp. 435-442, 2005.

[36] Mens T. and Tourwe T., "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30 (2), pp. 126-139, 2004.

[37] Meyer B., "Applying "Design by Contract"," *Computer*, vol. 25 (10), pp. 40 - 51, 1992.

[38] Morrison J., "A Co-evolutionary Framework," *Workshop on Evolutionary Computation: Twelfth Canadian Conf. on Artificial Intelligence*, 1998.

[39] Morse J. N., "Reducing the size of the nondominated set: Pruning by clustering," *Computers Operational Research*, vol. 7 (1-2), 1980.

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

[41] Pareto V., *Cours d'Economie Politque*, Rouge, 1896.

[42] Rechenberg I., "Cybernetic solution path of an experimental problem," Royal Aircraft Establishment, 1965.

[43] Richardson J. T., Palmer M. R., Liepins G. and Hilliard M., "Some guidelines for genetic algorithms with penalty functions," *Proc. Third International Conference on Genetic Algorithms*, pp. 191 - 197, 1989.

[44] 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, EMO 2005*, 3410, 2005.

[45] Schaffer J. D., "Multiple Objective Optimization with Vector Evaluated Genetic Algorithms," *Proc. First International Conference on Genetic Algorithms and their Applications*, 1988.

[46] Schaffer J. D., Caruana R. A., Eschelman L. J. and Das R., "A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization," *Proc. International Conference on Genetic Algorithms*, pp. 51 - 60, 1989.

[47] Seng I., Stammel J. and Burkhard D., "Search-based determination of refactorings for improving the class structure of object-oriented systems," *Proc. 8th Annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, 2006.

[48] Smith J. E. and Fogarty T. C., "Adaptively Parameterized Evolutionary Systems: Self Adaptive Recombination and Mutation in a Genetic Algorithm," *Proc. International Conference on Parallel Problem Solving From Nature*, pp. 441 - 450, 1996.

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

[50] Stevens W., Myers G. and Constantine L., "Structured Design," *IBM Systems Journel*, vol. 13 (2), pp. 115 - 139, 1974.

[51] Svetinovic D., Berry M. and Godfrey M., "Concept identification in object-oriented domain analysis: Why some students just don't get it," *Proc. IEEE International Conference on Requirements Engineering RE'05*, pp. 189 - 198, 2005.

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

[53] Vincent T. L. and Grantham W. J., *Optimality in Parametric Systems*, John Wiley & Sons, 1981.

[54] Whitmire S. A., *Object Oriented Design Measurement*, John Wiley and Sons, 1997.

[55] Yu T. and Bentley P. J., "Methods To Evolve Legal Phenotypes," *Proc. Fifth International Conference on Parallel Problem Solving from Nature*, 1498, pp. 280-291, 1998.

[56] Zadeh L., "Optimality and non-scalar valued performance criteria," *IEEE Transactions on Automatic Control*, vol. 8 (59), 1963.

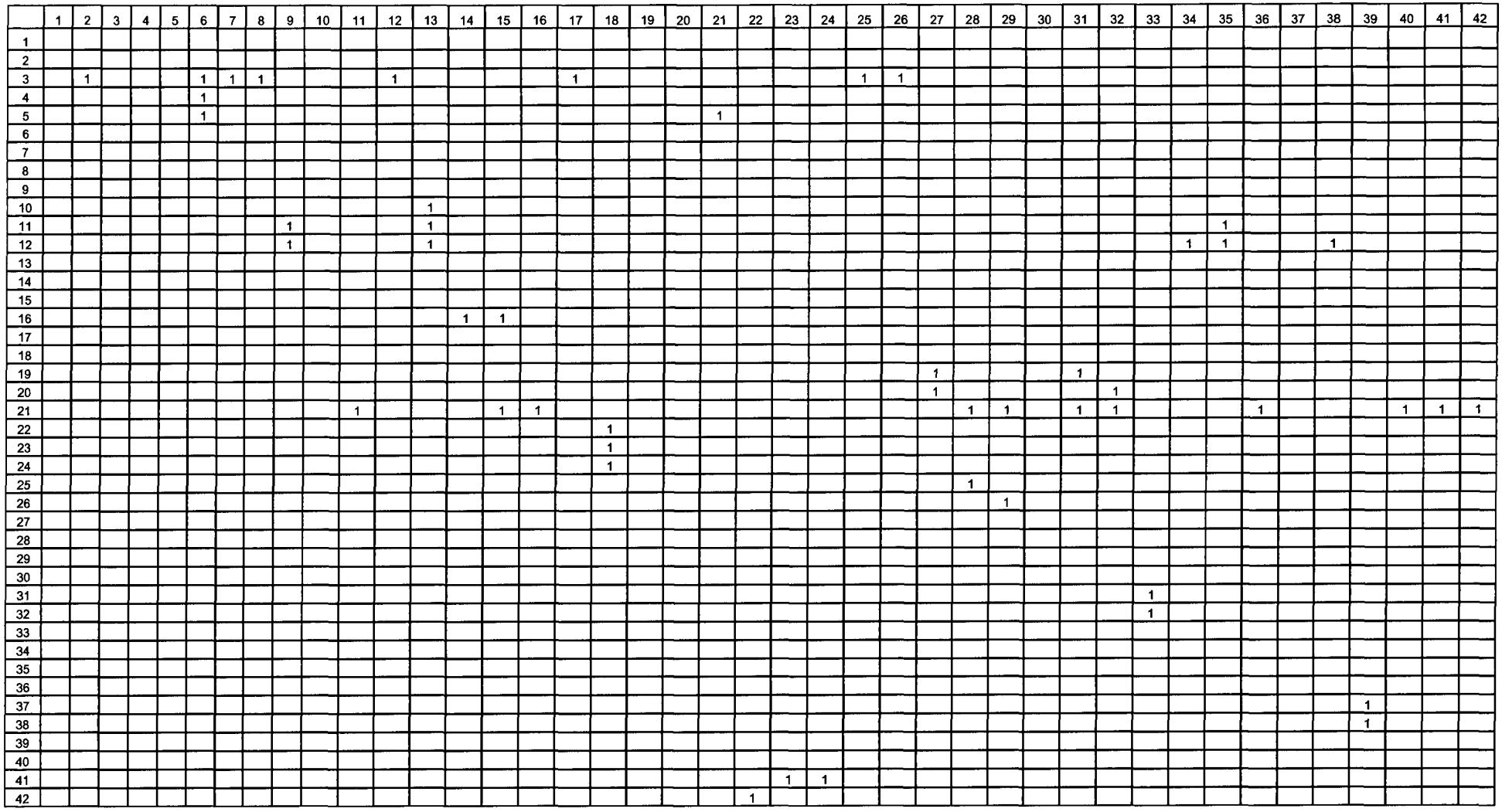
[57] Zitzler E., Laumanss M. and Thiele L., "SPEA2: Improving the Strength Pareto Evolutionary Algorithm," Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Technical Report 103, May 2001, 2001.

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

APPENDIX A: SAMPLE MODEL DEPENDENCIES

Id	Class	Name	Type	Visibility
1	MonopolyGame	ROUNDS_TOTAL	Constant	private
2	MonopolyGame	PLAYERS_TOTAL	Constant	private
3	MonopolyGame	createGame()	Method	public
4	MonopolyGame	getPlayers()	Method	public
5	MonopolyGame	playGame()	Method	public
6	MonopolyGame	players	Association End	private
7	MonopolyGame	dice	Association End	private
8	MonopolyGame	board	Association End	private
9	Board	SIZE	Constant	private
10	Board	getStartSquare()	Method	public
11	Board	getSquare()	Method	public
12	Board	buildSquares()	Method	public
13	Board	squares	Association End	private
14	Die	MAX	Constant	public
15	Die	faceValue	Attribute	private
16	Die	roll()	Method	public
17	Player	name	Attribute	private
18	Player	cash	Attribute	private
19	Player	getLocation()	Method	public
20	Player	setLocation()	Method	public
21	Player	takeTurn()	Method	public
22	Player	addCash()	Method	public
23	Player	getNetWorth()	Method	public
24	Player	removeCash()	Method	public
25	Player	setDice()	Method	public
26	Player	setBoard()	Method	public
27	Player	Piece	Association End	private
28	Player	Dice	Association End	private
29	Player	Board	Association End	private
30	Piece	Name	Attribute	private
31	Piece	getLocation()	Method	public
32	Piece	setLocation()	Method	public
33	Piece	Location	Association End	private
34	Square	Name	Attribute	private
35	Square	Index	Attribute	private
36	Square	landedOn()	Abstract Method	public
37	Square	getNextSquare()	Method	public
38	Square	setNextSquare()	Method	public
39	Square	nextSquare	Association End	private
40	RegularSquare	landedOn()	Method	public
41	IncomeTaxSquare	landedOn()	Method	public
42	GoSquare	landedOn()	Method	public

Figure 29 Monopoly Game Class Member List



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
1																																									
2																																										
3		1				1	1	1				1					1								1	1																
4						1																																				
5						1															1																				
6																																									
7																																									
8																																									
9																																									
10													1																												
11									1				1																						1							
12									1				1																					1	1			1				
13																																									
14																																									
15																																									
16														1	1																										
17																																									
18																																									
19																											1				1										
20																											1					1									
21											1				1	1												1	1		1	1				1				1	1	1
22																		1																							
23																		1																							
24																		1																							
25																												1													
26																													1												
27																																									
28																																									
29																																									
30																																									
31																																	1								
32																																	1								
33																																									
34																																									
35																																									
36																																									
37																																							1		
38																																							1		
39																																									
40																																									
41																							1	1																	
42																						1																			

Figure 30 Monopoly Game Dependency Matrix

APPENDIX B: UML DIAGRAMS FOR THE ESD TOOL

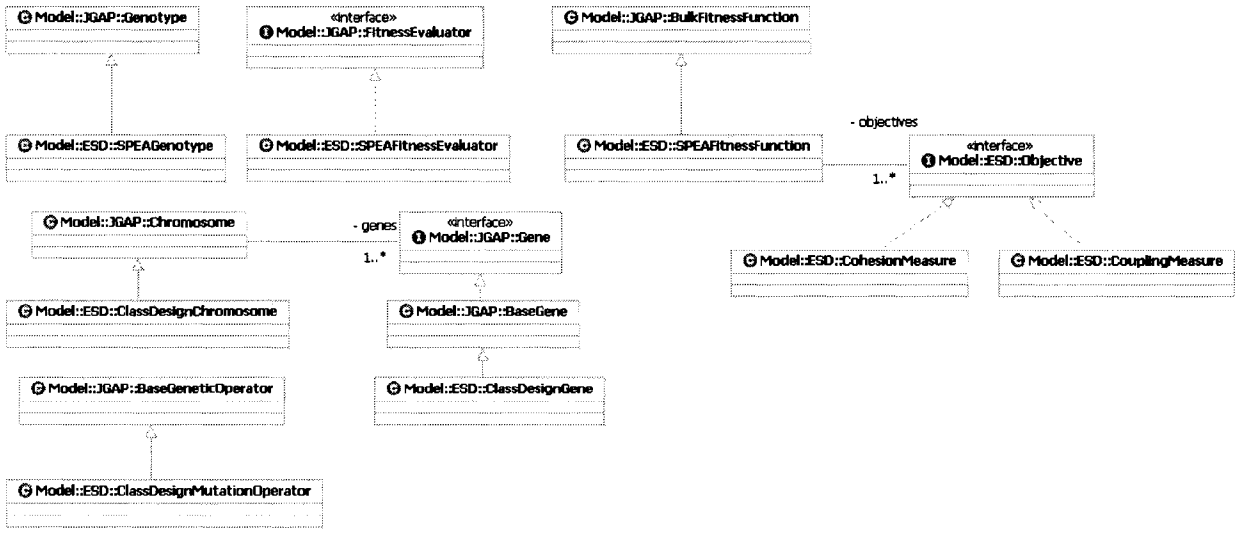


Figure 31 Extensions to the JGAP Framework

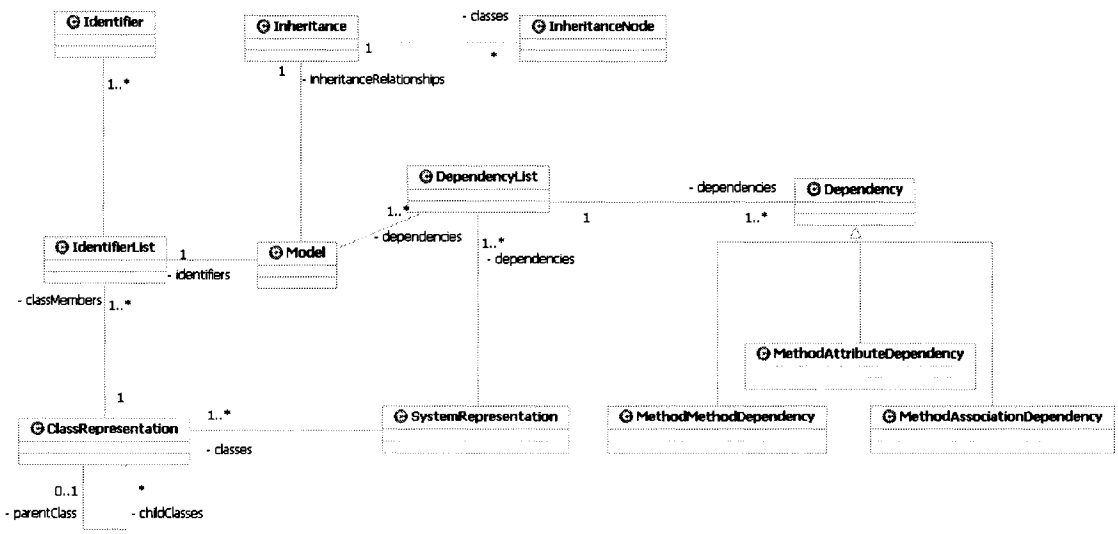


Figure 32 ESD Domain Classes

APPENDIX C: ARENA SYSTEM

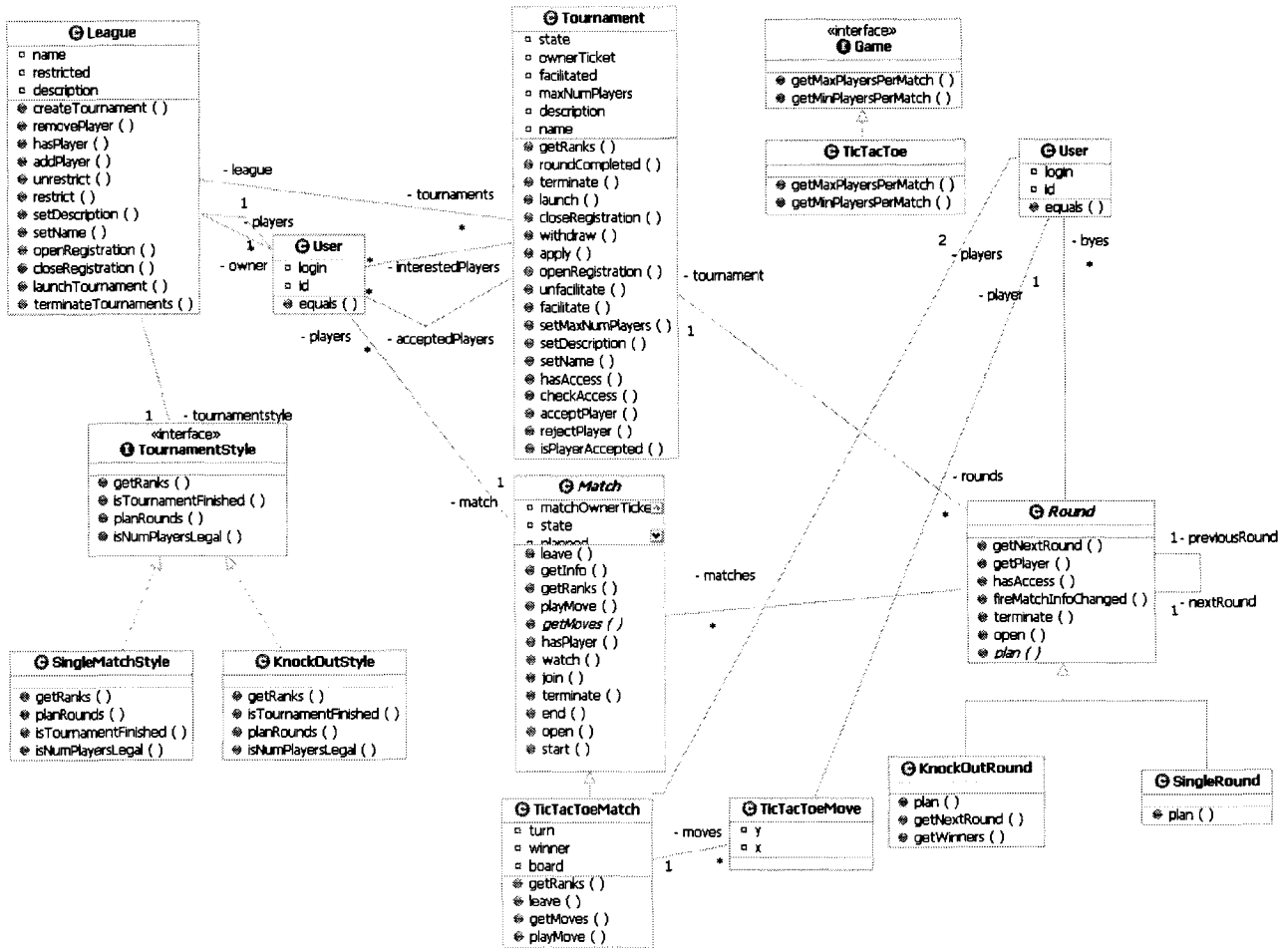


Figure 33 ARENA Class Diagram