3 PROPOSED EVOLUTIONARY SEARCH

Fundamental components of computational evolutionary search and exploration include the solution representation and its associated genetic operators (i.e. selection, recombination and mutation) that engage with the representation. However, formulating an appropriate and natural representation together with its associated genetic operators is perhaps one of the most difficult parts of designing an effective evolutionary algorithm. In this chapter, a number of established and more recent representations are examined for their natural fit and hence potential application to early lifecycle software design. In addition, novel proposals are put forward and contrasted with other representations previously reported in the research literature.

3.1 Representation

In industrial early lifecycle software design practice, representation of object-oriented software design is provided by the Unified Modelling Language (UML). Since its arrival in 1999 (Booch, *et al.*), UML has become the standard representation of the object-oriented paradigm in industrial practice and is widely adopted. According to Booch *et al.*, *"the UML is a graphical language for visualising, specifying, constructing and documenting the artefacts of a software-intensive system"*. At time of writing, the specification for the UML is at version 2.3 (Object Management Group, 2010c) and contains, among other things, a meta-model to define all legal constructs, their semantics and their visual depiction. With regard to early lifecycle software design, the principal constructs provided by the UML are *use cases* (originally Jacobson, *et al.*, 1992) for behavioural requirements capture in the design problem, and *classes* to denote the structural aspects of the design solution. Visually, a small class diagram showing UML notation of three classes is shown in figure 3.1. Each class, attribute and method is a discrete model element. As can be seen in figure 3.1, the three example classes act as placeholders for the six attributes (i.e. data) and the six methods (i.e. implementations of relevant responsibilities). Each class denotes a concept or abstraction relevant to the design problem domain; couples between classes are denoted as solid arrows with an open arrow-head indicating the direction of couple. It is important to note that the constructs of the software design are discrete, and that there is no ordering among the elements of the notation.

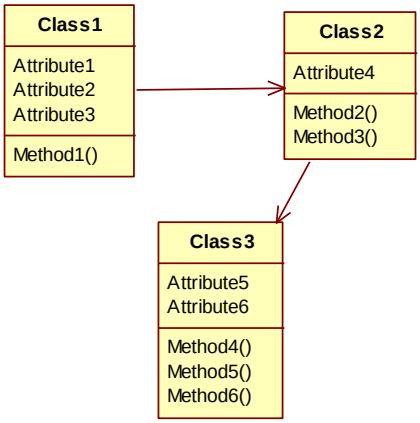

Figure 3.1. Example Class Diagram in UML Representation

For the purposes of this thesis, to be a basis of effective evolutionary search and exploration, any representation is required to:

- provide an appropriate abstraction of the essential characteristics of the software design solution,
- facilitate effective visualisation for the interactive human designer,
- enable efficient genetic operators (i.e. selection, recombination and mutation), and
- provide traceability from design problem to design solution.

To address these requirements, a number of existing representations have been considered in turn as follows. For example, genetic algorithms e.g. Fraser, 1957 (surveyed in Fogel, 2002), and Holland, 1975, adopt a binary scheme whereby solution information is encoded as '1' or '0' bits in a binary string, each bit coding for a portion of relevant solution information. However, it appears likely that a mapping from binary strings to early lifecycle software designs would be unnatural, complex and difficult to visualise. Thus it appears likely that the binary string approach is ill-suited to effectively represent software design solutions.

As an alternative to binary strings, evolutionary strategies (e.g. Rechenberg, 1973) employ real-valued representations to encode solution information. According to Eshelman and Shaffer (1993), the use of real numbers in a chromosome string enables good exploitation of gradual continuities of solution variables i.e. a small change in continuous representation maps results in a small change in solution search space. However, given the discrete, unordered nature of the software design solution, it would

appear that real-valued representations are also ill-suited to effectively represent software design solutions.

Perhaps more relevant to software development are the tree-like representations used in genetic programming (e.g. Cramer, 1985, Forsyth, 1986, Koza, 1992). Initially used to evolve simple computer programs, the application of genetic programming has expanded to, for example, evolvable hardware circuitry. In genetic programming, the tree-like representation consists of functions and terminals, which are traversed recursively. Both crossover and mutation are used as diversity promoting operators which can result in offspring that are very different to the parent individuals, depending on which node in the tree is used for crossover or mutation. Because of this, it is possible for tree representations to grow quickly ('code bloat') and so controlling the consequences of this can become complex. As genetic programming tree-like representations are intended for software programs, it might seem that such representations would be a good candidate to represent early lifecycle software designs. However, there are some fundamental drawbacks. For example, while the tree-like structures map well to program source code functions, they map much less well to discrete, unordered constructs of early life cycle software designs. Secondly, there is a fundamental paradigm mismatch between the functional programs represented by tree structures, and object-oriented modelling such as the UML. In functional programs, the tree must have a 'top' (assuming an inverted tree structure when compared to nature). The tree 'top' is a necessary place for ordered recursion to commence for the computer function or hardware circuit to execute. In contrast, object-oriented software designs have no 'top'. Instead, they have a bounding scope comprised of many unordered objects. System behaviour may initiate in any object with the scope of the design. Because of this philosophical mismatch and the resulting construct incompatibility, it seems that tree-like structures are also ill-suited to representing early life cycle software designs.

A further candidate representation for software design is a directed graph e.g. evolutionary programming graphs (Fogel, *et al.*, 1966). Parmee *et al.* (2000) report an example of a layered hierarchy representation in the engineering design domain, in essence a hybrid of string and tree structures, which contains a list of discrete design options at the top layer. From each discrete design option emanates a continuous variable set at a lower layer to code for various design properties. A similar layered hierarchy approach can also be seen in the software design representation of Seng *et al.*

(2005) in their attempts to refactor software subsystem decompositions. The top layer of the Seng *at al.* representation specifically codes for candidate software subsystems, while the lower level codes for classes which effectively results in a model of the software source code as a directed graph. Layered directed graphs are also used by Machwe and Parmee (2006a, 2006b, 2009) to represent design solutions for beam bridge design and urban furniture design. Machwe and Parmee describe their representation as 'component based' as it is composed of components (top layer) which relate via a directed graph to component properties (lower layer). For example, in the design of urban furniture (a park bench), discrete top layer components represent seat, leg and backrest. For each top layer component, at the lower layer there are associated continuous or discrete properties such as style, position or dimension.

Layered hierarchical design representations enable rich abstraction of the design solution. However, it is possible for large layered hierarchies to become complex, resulting in complex genetic operators. For example, Parmee *et al.* (2000) report that for such representations, as is also the case for genetic programming, crossover can be disruptive even for simple hierarchies, preventing satisfactory exploration of the search space. In addition, unless controlled, crossover and mutation may produce infeasible design solutions. Furthermore, because of computational complexities in the crossover operator, Seng *et al.* report that for a case study of 'JHotDraw' (a Java graphics library of some 207 classes), evolutionary runs can take in the order of days to execute. Thus although layered hierarchical representations offer rich abstraction through structures of discrete and continuous elements, they are perhaps too rich for early lifecycle software design, and so not a natural representation.

A yet further candidate for early lifecycle software design is integer representation. For example, Grouping Genetic Algorithms (GGAs) (Falkenauer, 1999) are a class of genetic algorithms specifically aimed at problems of allocating discrete resource objects into groups, such as the bin packing problem or the line balancing problem. As described by Falkenauer, the GGA integer representation comprises two components: firstly a component to represent a vector of discrete resource objects to be allocated, and secondly, a component to encode groups. Ordering within the representation is significant as position in the vector denotes resource identity. Falkenauer (1999) provides the following two examples

```
ADBFEB:BEFDA
AAABBB:AB
```

Each discrete resource object to be allocated is denoted on the left hand side of the colon by its position in the vector. The first position denotes discrete resource object one, the second position denotes the second discrete object, etc. Thus in both examples above, there are six discrete objects to be allocated. Groups are denoted on the right hand side of the colon. Thus in the first example above there are five groups, while in the second example there are two. The value at each vector position before the colon simply denotes the group to which the discrete object belongs, which the values after the colon identifies each group. Falkenauer states that the 'names' of the groups are merely 'mute' labels to identify each group. As Falkenauer points out: *"The important point is that the genetic operators will work with the group part of the chromosomes, the standard object part […] merely serving to identify which objects actually form which group"*. Thus recombination, for example, is potentially powerful as groups are subject to variation during evolution, depending on where in the chromosome splicing occurs. Nevertheless, as with layered hierarchies (described previously), crossover can also produce infeasible offspring. Indeed, Falkenauer therefore suggests a number of complex mechanisms for either preventing the creation of infeasible offspring, or the repair of infeasible offspring. To address this crossover complexity, Tucker *et al.* (2005) propose RGFGA (Restricted Growth Function Genetic Algorithm) to specifically provide an efficient representation and crossover for Grouping Genetic Algorithms. In the field of software design, Bowman *et al.* (2010) use a related but different integer representation for UML class models: class members (i.e. attributes and methods) are assigned an integer position in a vector, and the value of the integer represents a class to which that member is assigned. The classes thus also have 'mute' identifiers, as in the GGA representation. In one sense this is useful and relevant because the representation specifically provides for classes as grouping constructs which is consistent with the UML semantic. However, the actual grouping element, as addressed in the Falkenauer GGA representation, is missing in the Bowman *et al.* representation. This is significant as it is not entirely clear from the Bowman paper how the classes are formulated. Furthermore, it is not clear how crossover might brings about changes in classes i.e. class introduction and removal.

Overall, none of the representations discussed above appears a perfect and natural candidate for early lifecycle software design. Binary strings and continuous real-valued representations do not naturally relate to software designs. Tree-based representations (e.g. Koza, 1992) relate to the software domain, but to functional

program source code rather than early lifecycle software designs. Hierarchical layered graphs (e.g. Seng *et al.* 2005, Machwe and Parmee, 2006a, 2006b) afford great richness of abstraction through a combination of discrete and continuous variables, but are too rich to naturally represent software designs. Integer representations (e.g. Falkenauer, 1999, Bowman *et al.*, 2010) support grouping of discrete items and show potential because the UML class semantic is a grouping of discrete methods and attributes. However, ordering is important in such integer representations whereas ordering is not a feature of software design solutions. In addition, it would be necessary to extend integer representation to simultaneously cater for grouping of both attributes and methods. Furthermore, none of the above representations provide direct and natural support for abstraction or support visualisation of early lifecycle software design. Indeed, none of the above representations address software design problem to solution traceability (although this may be not entirely surprising as none of the representations set out to achieve this goal).

Therefore, a novel object-based representation for early lifecycle software design is proposed wherein discrete attributes and methods are grouped in classes. Indeed, as grouping of attributes and methods is crucial to the representation, grouping is explicitly represented by discrete classes. There is no ordering among the classes, and there is no ordering of attributes and methods within a class. Classes are not named and so are 'mute' identifiers to distinguish between groupings. Because the representation is discrete and unordered, fitness gradients among solution individuals are unlikely. This suggests that categorical operators (e.g. random switching of methods and attributes among class groupings) are the most natural and appropriate operators for diversity preservation, rather than any continuous operators that might creep along fitness gradients. In addition, the proposed discrete representation provides a natural abstraction of the software design solution which explicitly facilitates effective visualisation for the interactive human designer. The proposed representation also enables a mechanism for traceability from design problem to design solution. An example of the representation, tracing from design problem to potential design solutions, is described as follows.

Capturing software design problems as use cases (Jacobson, 1992, Cockburn, 2001) has been widely applied in industrial software engineering. Indeed, use cases have been incorporated within the Unified Modelling Language (OMG, 2010c). A use case is essentially a narrative story that describes the interaction between an 'actor'

(typically a human being) and the software system-to-be, by means of chronological sequence of steps. Such steps are recorded as text. Cockburn (2001) refers to a step in the chronological sequence as an 'action step' and suggests that the textual grammar of an action step should be as follows:

> *"Subject…verb…direct object…prepositional phrase.*
>
> *For example:*
>
> *The system…deducts...the amount…from the account balance."*

Indeed, it is useful to capture the required behaviour and capabilities of the software system-to-be in terms of 'action steps', because in manual software design, these 'actions' typically go forward to the design solution as candidate 'responsibilities' to be grouped within early life cycle UML design classes. A complete set of use cases defines all action steps for a system-to-be which accordingly provides the scope and definition of the software design problem. Thus from the textual narrative of the use case action steps, it is possible to identify the actions (or responsibilities) that the software is to perform, together with individual items of data which the software will use (i.e. manipulate) in order to perform its actions. Actions and data are uniquely identified by their textual names in the action step. If an action and a datum are co-located in the same step of the narrative, an action is considered to "use" the datum. Thus, overall, the software design problem can be specified by three sets extracted from the use cases of the design problem, namely:

- a set of data,
- a set of actions, and
- a set of action-data uses.

To promote traceability from the design problem to the design solution, it is proposed that design solutions be derived from the sets of data and actions in the design problem. This is achieved by constructing a set of design solution attributes directly from the design problem data, and constructing a set of design solution methods directly from the design problem actions. Attributes and methods are identified by textual names taken from the design problem data and actions. The design solution search space is thus scoped and defined by these sets of attributes and methods, which are grouped within classes. According to the object-oriented software design paradigm, it is widely considered poor practice for classes to lack either attributes or methods, or for attributes or methods to be duplicated. Because of this, the proposed representation requires that

each class holds at least one attribute and one method, and that each attribute and each method be allocated once to a class. Figure 3.2 provides a simple visual example of the proposed representation and derivation of attributes and methods from data and actions. Figures 3.3 shows the representation of a single example instance of a software design solution, while figure 3.4 shows a UML class diagram of the structure of the proposed solution representation.

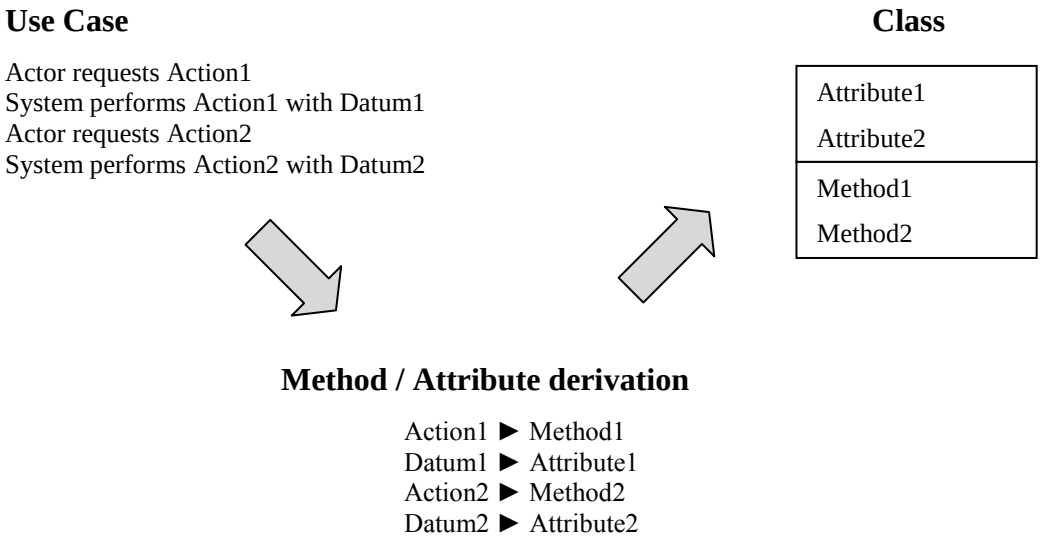

Figure 3.2. Example of Design Problem and Design Solution Representation

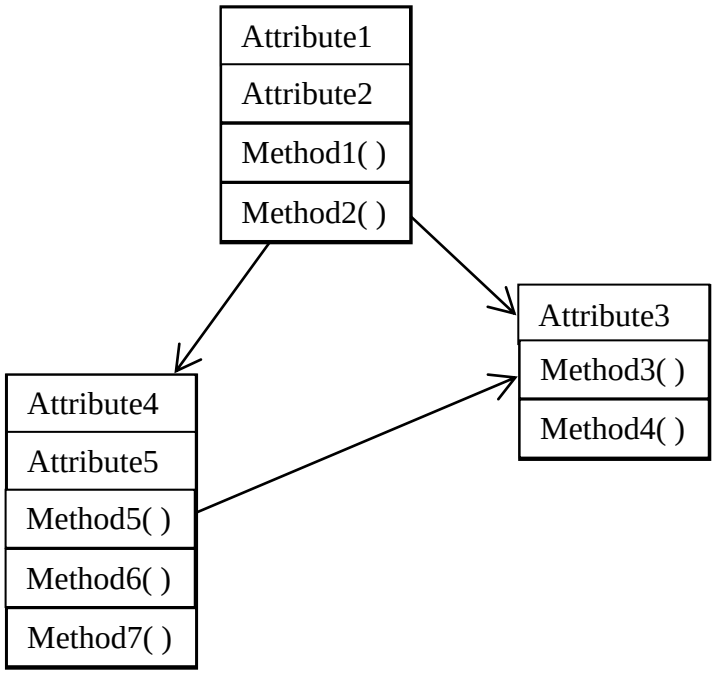

Figure 3.3. Example of a Design Solution Representation

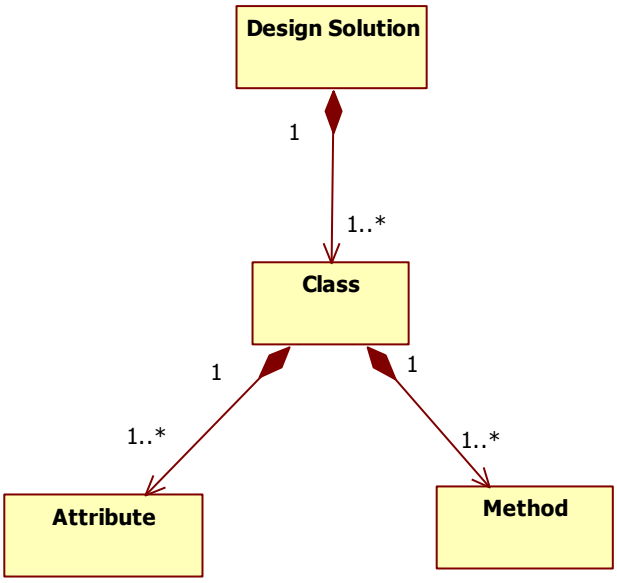

Figure 3.4. UML Diagram of Design Solution Representation Structure

The proposed object-based representation of the software design search space is straightforward in that is does not explicitly cater for coupling relations between classes. However, coupling relations between classes are arrived at by applying the set of action/datum uses from the design problem. For each action/datum use, the corresponding solution method and attribute are located. With respect to the class containing the method, the used attribute must either be located in the same class or a different class. If the used attribute is in the same class as the method, a design couple is considered to be 'internal' to the class and is taken to be an indication of cohesion within the class. However, if the used attribute is located in a different class, a design couple is considered to be 'external' to the class. Of course, the set of 'uses' of the design problem is static. Given that attributes and methods can be allocated among classes in many ways, the degree of 'external' coupling will vary among individual software design solutions. Therefore, application of the set of uses to a design solution thus reveals the extent of not only cohesion within classes ('internal' coupling) but also the 'external' coupling between classes. Object-oriented software design principles suggest that software designs should exhibit high cohesion and low coupling. Thus measures of cohesion and coupling can be used to assess the structural integrity of a software design, and therefore provide a quantitative measure of design fitness.

Formally, the solution space is defined by all possible allocations of the methods and attributes to a specific number of classes, and the classes are identified by these allocations. Let M and A denote the set of methods and attributes, respectively, and let c denote the number of classes. For the purposes of evaluating the cardinality of the search space S, for a given number of classes, every element of the search space can be considered to be a partition of M into c parts combined with a surjection from the set A to this partition. The partition of M into c parts determines the classes by determining the methods contained in the classes. The surjection then assigns each of the attributes in A to a class (a part of the partition). This function is a surjection since every class must have at least one attribute. For example, consider the sets $M = \{m_1, m_2, m_3\}$ and $A = \{a_1, a_2, a_3, a_4\}$ with $c = 2$. An example of a partition of M into two parts is $\{\{m_2\},\{ m_1, m_3\}\}$ and this partition identifies the two classes. An example of a surjection, f, from A to this partition is as follows:

$$f(a_1) = \{m_2\}, \quad f(a_2) = \{ m_1, m_3\}, \qquad f(a_3) = \{ m_1, m_3\}, \qquad f(a_4) = \{m_2\}.$$

The combination of the partition and the surjection results in two classes, these being

$$\{ m_2, a_1, a_4 \} \quad \text{and} \quad \{ m_1, m_3, a_2, a_3 \}.$$

For a given M, A and c, the partition of M into c parts followed by a surjection from A to this partition uniquely identifies an element of the search space. Furthermore, all elements of the search space can be identified in this manner. Let the cardinalities of M and A be represented by m and a, respectively. The number of different partitions of set M into c parts is given by $S(m,c)$ where S is the Stirling number of the second kind. The Stirling number of the second kind is defined recursively by:

$$S(n,1) = 1$$

$$S(n,n) = 1$$

$$S(n,r) = S(n\text{-}1,r\text{-}1) + r\, S(n\text{-}1,r)$$

The number of different surjections from set A to a set of cardinality c (this being the partition of M into c parts) is given by $c!S(a,c)$. Hence, by the product rule, the cardinality of the search space is given by:

$$|S| = c!\, S(a,c)\, S(m,c) \qquad (1)$$

Table 3.1 shows the cardinality of the search space, $|S|$, as a function of a, m and c where, for convenience, we have taken $a = m$. For a fixed number of classes, that is, for a fixed column in the table, the cardinality of the search space displays an exponential growth with respect to the values of a, m. It is interesting to note that quantifying the cardinality of the design solution search space suggests that the number of possible

designs for even small scale class designs is beyond human comprehension. For example, eight attributes and eight methods allocated into five classes results in a search space of cardinality 132300000. It is conjectured that this exponential growth in the cardinality of the search space might be one of the many causative factors behind the difficulties of early lifecycle software design.

Table 3.1.

$|S|$ as a function of the number of attributes, a, the number of methods, m, and the number of classes, c. For a convenient illustration we have taken $a = m$.

$a, m \setminus c$	1	2	3	4	5	6	7	8
1,1	1							
2,2	1	2						
3,3	1	18	6					
4,4	1	98	216	24				
5,5	1	450	3750	2400	120			
6,6	1	1922	48600	101400	27000	720		
7,7	1	7938	543606	2940000	2352000	317520	5040	
8,8	1	32238	559836	69441622	132300000	50944320	3931360	40320

3.2 Quantitative Fitness Functions

The software engineering community has widely applied quantitative metrics to software development artefacts in an attempt to quantify various properties of software, including structural integrity. Indeed, a number of structural design properties of software have been widely investigated and within the object-oriented design paradigm, it is generally accepted that good indicators of superior software design structural integrity are (1) high cohesion within classes and (2) low coupling among classes. Many cohesion and coupling metrics have been suggested and as a result, a number of surveys of the use of cohesion and coupling metrics have been conducted and frameworks proposed (e.g. Chidamber and Kemerer, 1994, Briand *et al.*, 1999). For the purposes of early lifecycle software design search however, it is necessary to select metrics that enable evolutionary search. In selecting such metrics, it is important that the metric (1) can be applied to early lifecycle software design models rather than downstream programming language source code, and (2) is efficient to compute. Based on this, the Cohesiveness of Methods (COM) metric (Harrison *et al.*, 1998) has thus been selected

as the basis of measuring cohesion, and Coupling Between Objects (CBO) (Briand, *et al.*, 1999) has been selected as the basis of measuring coupling.

Generally, within a class, the more uses of attributes in the class by methods in the class, the more cohesive the class. According to Harrison *et al.* (1998), Cohesiveness of Methods (COM) is: *"for each attribute, the sum of all the methods using an attribute divided by the total number of methods, all divided by the number of attributes in the class".* Values of COM thus range from 1.0 (totally cohesive, i.e. all methods use all attributes) to 0.0 (not at all cohesive, i.e. no methods use any attributes). Harrison *et al.*'s COM metric is defined as follows. For a class *C*, let A_c and M_c be the set of attributes and methods, respectively, that are contained in class *C*. Then the COM fitness for the class C, denoted by $f(C)$, is given by:

$$f(C) = \frac{1}{|A_C||M_C|} \sum_{i \in A_C, j \in M_C} \delta_{ij} \qquad (2)$$

where $\delta_{ij} = \begin{cases} 1 & \text{if method } j \text{ uses attribute } i \\ 0 & \text{otherwise} \end{cases}$

For an early lifecycle software design, it is straightforward and convenient to compute the average value of COM from all classes in a design.

Coupling between classes is calculated based on the Coupling Between Objects (CBO) metric taken from Briand *et al.*'s framework of coupling measurement (1999). Briand *et al.* define the CBO metric as follows:

$$CBO(c) = |\{d \in C - \{c\} \mid uses(c,d) \vee uses(d,c)\}| \qquad (3)$$

where *C* is the set of all classes in the design, *c* and *d* represent two classes in the set *C*, and *uses(x,y)* is a predicate that holds true if a method in class *x* uses an attribute in class *y*. However, as an objective fitness function, there is a drawback with this definition. While the existence of a couple is defined by means of a predicate, the strength of the couple (in terms of the number of methods in class *x* using attributes in class *y*) is not. The approach in this thesis has been to take the strength of coupling between individual classes into account. Furthermore, the total number of uses is known from the problem domain. This fact can be exploited to calculate a value of coupling for a class design within the range 0.0 to 1.0 by dividing the sum of all couples existing between classes by the total number of uses in the problem domain. Formally:

- let *M* represent the set of all methods and *A* represent the set of all attributes.

- let *U* represent the set of all uses, expressed as a subset of $M \times A$.
- let *C* represent the set of all classes, each class having been allocated methods and attributes.

Each individual class is expressed as a subset of $M \cup A$, and *C* can be considered to be a partition of $M \cup A$. The set *E*, i.e. the set of all uses 'external' in a class design, is defined by:

$$E = \left\{ (m,a) \in U \mid \forall c \in C \left((m \notin c) \vee (a \notin c) \right) \right\} \tag{4}$$

Indeed, it is a constraint of the representation that an external use cannot exist when a method 'uses' an attribute within the same class. Therefore, the coupling value of a class design then becomes:

$$\frac{|E|}{|U|} \tag{5}$$

Thus a totally coupled class design has a value of 1.0 while a class design with no couples has a value of zero. This expression of coupling ensures that class designs are comparable for values of coupling, regardless of the number of classes within a class design.

3.3 Selection

Selection has been performed by two techniques, namely tournament selection and fitness proportionate selection, to determine the appropriateness of each for interactive evolutionary search and exploration. Both are described in turn as follows.

According to Bäck (1996), the tournament selection method selects a single individual by choosing some number *q* of individuals randomly from the population and selecting the best individual from the group to survive to the next generation. The process is repeated as often as is required to fill the new population size. Back reports that a common tournament size is $q = 2$, i.e. binary tournaments. Deb (2001) describes binary tournament selection in a broadly similar fashion, but differs slightly in the choice of individuals for tournament. Deb suggests that firstly, two individuals are chosen at random from the population, and the better individual is placed in the mating pool. Thereafter, however, two further different solutions are then picked repeatedly

from the population and so further slots in the mating pool are filled, such that each solution in the population participates in exactly two tournaments. Because tournament selection relies on the relative fitness of individuals (rather than absolute), it lends itself to computationally effective selection. Indeed, Deb cites Goldberg and Deb (1991) who report that such a binary tournament selection *"has better or equivalent convergence and computational time complexity properties when compared to any other reproduction operator that exists in the literature"*. Perhaps because of this, Eiben and Smith (2003) report that *"tournament selection is perhaps the most widely used selection operator in modern applications of genetic algorithms"*. Based on these positive reports, the tournament selection approach implemented in this thesis is inspired by the Deb (2001) proposal.

To compare with tournament selection, a fitness proportionate selection mechanism has been also implemented. In fitness proportionate selection, individuals are assigned copies to the mating pool in proportion to their fitness values. Thus if the average fitness of all population members is f_{avg}, an individual of fitness f_i might expect to achieve f_i / f_{avg} number of copies in the mating pool. In other words, an individual's selection probability depends on its absolute probability value compared to the absolute fitness values of the rest of the population. According to Deb (2001), a straightforward implementation of this selection operator can be thought of as a roulette-wheel selection (RWS) mechanism in which the roulette-wheel is divided into N (population size) divisions, where the size of each is marked in proportion to the fitness of each population member. To perform selection, the wheel is spun N times. After each spin, the solution pointed to by the proportion mark is placed in the mating pool. Thus for the purposes of comparison with tournament selection in evolutionary search, roulette-wheel selection (RWS) has also been implemented.[1]

3.4 Crossover

The crossover operator used with the object-based software design representation has been inspired by the simple binary string genetic algorithm (GA) example presented in Goldberg (1989) and reiterated in Deb (2001). As crossover employs sexual reproduction, two parent individuals breed to produce a single offspring. However, it is

[1] Other possibly more efficient implementations of proportionate selection are also available. For example, Stochastic Universal Sampling (SUS) (Baker, 1985) requires the generation of only one random number for the whole selection process.

important to recall that there are two important constraints of the representation: (1) each class must contain at least one attribute and one method, and (2) each attribute can only be grouped to one class. Any crossover operator must respect these constraints. To achieve this, it is therefore not always possible to splice each parent individual into two portions and recombine to produce two feasible offspring. Rather, to respect the constraints of the representation, recombination is achieved by means of a trans-positional crossover (TPX). In TPX, two parent individuals are firstly chosen at random from the parent population to breed. Then, secondly, attributes and methods are chosen at random and swapped between the two based on their class position in the individuals. For example, if an attribute was found to be in the first class of the first individual and the last class of the second, the attribute was relocated to the last class for the first individual, and the first class for the second. An example of TPX is shown in figure 3.5. However, to further ensure that the constraints of the search space are not violated, attributes and methods are only selected for positional swap where swapping an attribute or method to another class would not leave the class lacking attributes or methods.

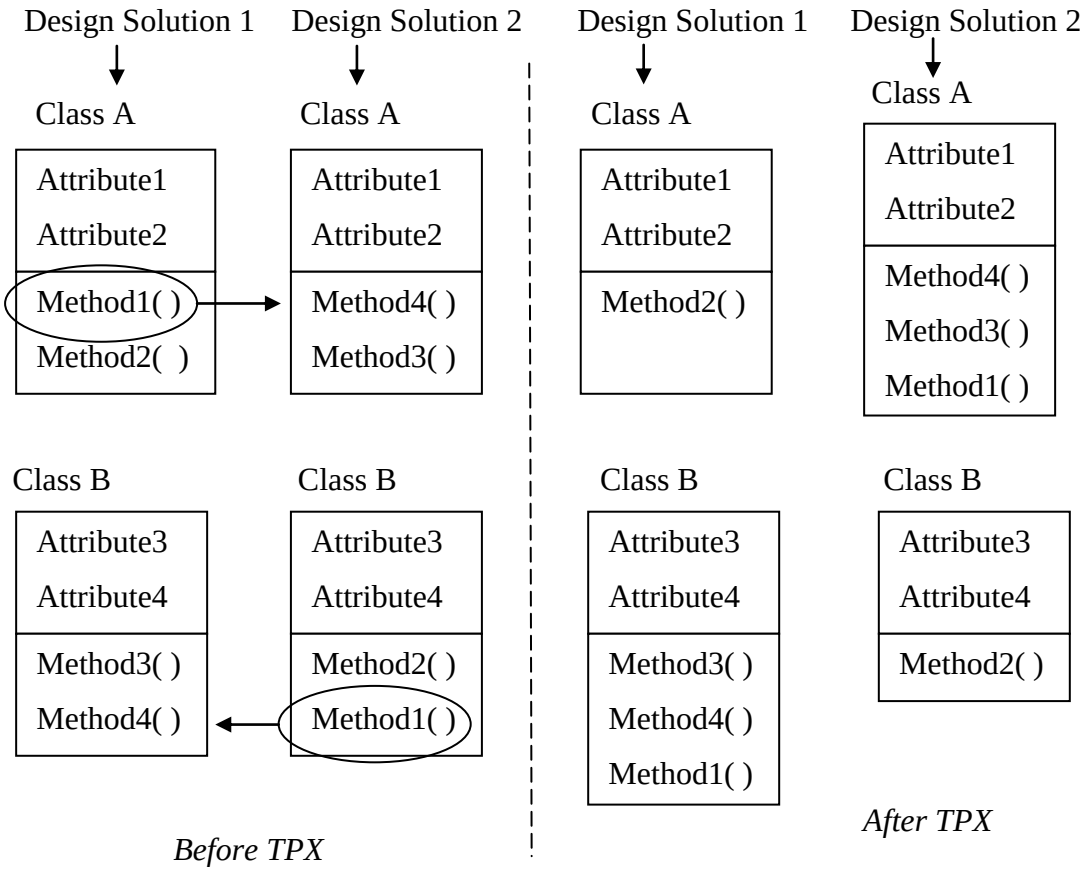

Figure 3.5. Example of Trans-Positional Crossover (TPX)

3.5 Mutation

The mutation operator used with the object-based software design representation has been inspired by the original Evolutionary Programming approach put forward by L.J. Fogel *et al.* (1966). Fogel *et al.* describe a mutation-based evolutionary algorithm applied to a discrete search space in which finite state machines (FSMs) are thought of as "organisms" as the basis of artificial intelligence. Such organisms have the ability to predict their environment coupled *"with a translation of each prediction into a suitable response in the light of a given goal"*. The FSM is defined in terms of finite alphabets of discrete input signals, output signals, and a number of possible different internal states. To specify such a machine, each of these states must be described in terms of symbols that would emerge from the machine when the machine is in that state and receiving each of the possible input symbols. It also requires an initial state to be specified. Fogel *et al.* state their goal as: *"to devise an algorithm which will operate on the sequence of symbols far observed, in order to produce an output symbol that is likely to agree with the next symbol to emerge from the sensed environment."* In essence, the better the FSM can evolve to reflect the sequence of the environment, the better it will be able to predict the next output symbol. It is interesting to note that Fogel *et al.* use only mutation for diversity preservation, possibly because mutation was found to be more natural and computationally efficient for an FSM represented by sets of discrete input signals, output signals and internal states.

As mutation is a means of asexual reproduction, one parent individual produces a single offspring by moving or swapping attributes or methods at random between classes. As a single software design individual acts as parent, moving or swapping attributes or methods between classes in a single design more inherently respects the integrity of the sets of attributes and methods, unlike crossover. Swapping an attribute or method from one class in a design to another does not violate the attribute and method sets; swapping also leaves the number of attributes and methods in the classes unchanged. Although moving also does not violate the attribute and method sets, the operation may be constrained when a class contains only one attribute or method. Examples of mutation by swapping and moving an attribute or method are shown in figure 3.6.

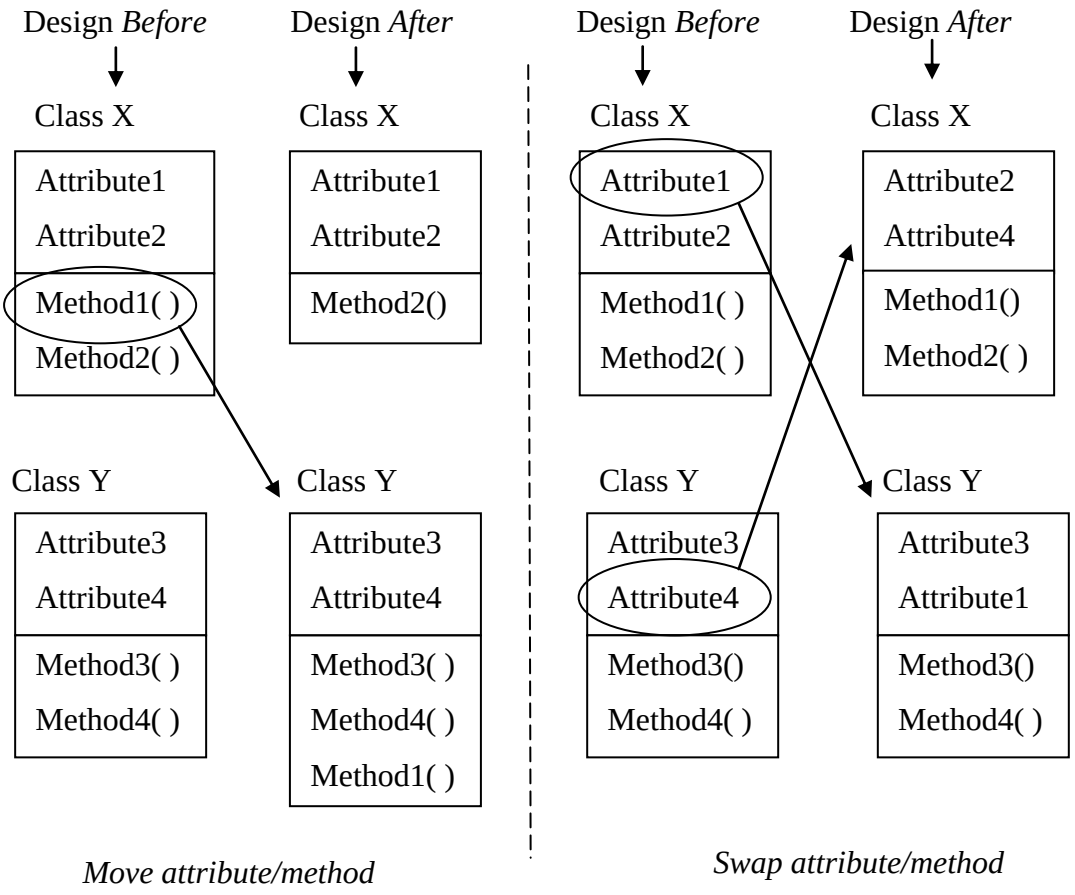

Figure 3.6. Examples of Mutation

3.6 Conclusions

This chapter describes the proposed object-based representation and genetic operators for evolutionary search and exploration of an early lifecycle software design search space. The representation is natural and appropriate for discrete and unordered early lifecycle UML class designs. The representation also facilitates effective visualisation for human designer interaction, and provides a mechanism for traceability from the design problem to the design solution. Using the representation to quantify the design solution search space, an exponential growth in cardinality is observed. It is conjectured that this exponential growth in cardinality of the search space may be one of the many causative factors behind the difficulties of early lifecycle software design.

Having proposed an object-based representation, the following chapter describes the necessary methodology to evaluate the effectiveness of the representation with respect to the research aims of the thesis.