Interactive, Evolutionary Search in Upstream Object-Oriented Class Design

Christopher L. Simons, Ian C. Parmee, and Rhys Gwynllyw

Abstract—Although much evidence exists to suggest that early life cycle software engineering design is a difficult task for software engineers to perform, current computational tool support for software engineers is limited. To address this limitation, interactive search-based approaches using evolutionary computation and software agents are investigated in experimental upstream design episodes for two example design domains. Results show that interactive evolutionary search, supported by software agents, appears highly promising. As an open system, search is steered jointly by designer preferences and software agents. Directly traceable to the design problem domain, a mass of useful and interesting class designs is arrived at which may be visualized by the designer with quantitative measures of structural integrity, such as design coupling and class cohesion. The class designs are found to be of equivalent or better coupling and cohesion when compared to a manual class design for the example design domains, and by exploiting concurrent execution, the runtime performance of the software agents is highly favorable.

Index Terms—Software design, evolutionary computation, interactive search.

1 Introduction

There exists much evidence to suggest that early life cycle software engineering design is a nontrivial and demanding task for software engineers to perform. Brooks [1] believes *"the hard part of building software to be the specification, design, and testing of this software construct, not the labour of representing it and the testing of the fidelity of the representation."* Guindon [2] reports that the software design process is complex and rarely sequential in reality, frequently iterating and relying heavily on opportunistic thoughts of the designer. Zannier et al. [3] have observed the complexities of both rational and opportunistic/naturalistic design decision making in software design. Glass [4] goes further to suggest that the complexities of some software designs may be beyond human comprehension. In addition, novice software designers find the skills of concept identification very difficult to grasp [5], suggesting that the process of software design is difficult to learn. Even for experienced software designers, Glass notes that developer performance may vary more than 28:1 from the best engineers to the worst [4]. To help overcome these difficulties, mature heuristics and manual strategies for the discovery of candidate classes from the problem domain are available to the designer, e.g., Wirfs-Brock and McKean [6], Evans [7], and the existence of many live complex software-intensive systems in use in society today suggests that early life cycle software design can be successful. Nevertheless, it also seems likely that poor requirements capture and software designs can have significant deleterious downstream consequences for software development [8]. The impact of upstream software designs of poor structural integrity and poor traceability to the problem domain appears to be far-reaching and hugely detrimental to subsequent development productivity. Given the difficulties of upstream software design, the potential of computational tool support to assist the software designer is considerable. It seems likely that even modest gains in the traceability and structural integrity of early life cycle software designs might be amplified as the development life cycle progresses to yield significant productivity gains.

However, the extent of current computational tool support provided to the upstream software designer is mixed. Some UML modeling tools, e.g., [9], seem to do little to proactively support the early design process; rather, they appear to provide an opportunity for the designer to formally record the output of their design decisions after the cognitive hard work of following manual heuristics and strategies has been done. Other commercially available UML modeling tools, e.g., [10], [11], [12], provide considerable utility to the designer through, for example, dependency analysis, verification for UML 2.n well-formedness rules, automated code generation, reverse engineering, and refactoring suggestions. Nevertheless, such UML modeling tools rarely suggest candidate solution classes derived from the problem domain and then follow up by evaluating them.

Natural Language Processing (NLP) tools, e.g., [13], [14], [15], attempt to automate early life cycle design by arriving at models based on requirements text in a closed system that excludes the human designer from the process. However, while the approach of NLP tools is certainly traceable to the problem domain and proactive in suggesting candidate solution classes, it is essentially convergent in the sense that "optimal" design solutions must be derived from natural language constructs in the design problem statement, thus affording little opportunity to explore possible solution design variants. For any computational design support tool

- *C.L. Simons and I.C. Parmee are with the Department of Computer Science, University of the West of England, Frenchay, Bristol BS16 1QY, UK. E-mail: {chris.simons, ian.parmee}@uwe.ac.uk.*
- *R. Gwynllyw is with the Department of Mathematics and Statistics, University of the West of England, Frenchay, Bristol BS16 1QY, UK. E-mail: rhys.gwynllyw@uwe.ac.uk.*

Manuscript received 21 Aug. 2008; revised 3 Oct. 2009; accepted 12 Oct. 2009; published online 26 Feb. 2010.
Recommended for acceptance by M. Harman and A. Mansouri.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2008-08-0250.
Digital Object Identifier no. 10.1109/TSE.2010.34.

to be effective, it must support and enable the design process itself. As with other previous research within early life cycle design [16], we continue to conjecture that it is currently impossible to fully automate the software early life cycle design process (excluding and thus replacing the designer) given the richness and complexities of design. It is again suggested that it is possible—and indeed desirable—to support (rather than replace) the human designer during design processes and, for this to be effective, the human designer and the computational support tool must collaborate interactively [17].

Previous work also indicates that interaction between the upstream designer and the computational support tool is at its most effective when it involves deep interaction, i.e., both "parties" adapt and change their behavior in the experience of the interaction [18]. Adaptive computational intelligence, for example, has been frequently deployed as an "intelligent assistant" in software development. For example, in a survey of software engineers by Rech et al. [19], a demand for unobtrusive, fast, and reactive computational assistance in core software engineering phases has been observed. However, such assistance typically manifests itself in small-scale housekeeping jobs, such as design model syntax checking, code completion, and limited programming code refactoring. Such assistance appears somewhat mechanistic rather than insightful. Nevertheless, Rech et al. suggest that in the future *"we might see even more intelligent assistance ... that proactively supports tasks such as software reuse, learning on demand, or automated product variant configuration, and which bridge the gap between phases of software development."* Rech et al.'s vision appears not to extend to upstream design, however.

In reality, effective two-way interaction constitutes an open system, in the sense of a collection of collaborating software, hardware, and human components, working together cooperatively toward a common goal. To underpin understanding of interactive computing in open systems, a logical theory of interaction has been suggested by Milner [20], while the ways in which open systems' behaviors can be expressed by the composition of collaborative components is explained by Arbab [21]. Goldin and Wegner [22] go so far as to suggest that *"...interaction is more powerful than algorithms"* and that interactive adaptive behavior of open systems is the new paradigm for computing in the modern era. While such claims remain to be fully substantiated, they do lend support to the notion that computational tool support is most effective when it is open and interactive.

In other fields, interactive computational support for conceptual design has received significant attention within the evolutionary computation (EC) domain, including applications in conceptual engineering design. For instance, Parmee has reported the successful integration of evolutionary search and exploration with a range of conceptual engineering design projects [23]. Parmee also reports on the iterative improvement of conceptual design representation via a user-centered EC approach [24] that also utilizes agent-based support [25]. He also proposes human-centric intelligent systems for design exploration and knowledge discovery [26] and, within a drug design/discovery domain, has successfully transferred techniques and strategies developed initially for conceptual engineering design into commercial drug design processes [27]. Other recent work addresses user-assessment of the aesthetics of conceptual designs generated by an evolutionary search engine and the introduction of online machine-learning techniques to support the machine-based assimilation of user preferences [28], [29].

Given that the utility of these approaches is proving beneficial across several diverse fields, could user-centered evolutionary search be a basis of computational tool support in early life cycle software design?

Certainly, interactive, evolutionary search affords opportunities for not only synthesizing a design space of candidate classes for exploration, but also their subsequent evaluation by objective fitness functions. As an open system, opportunities for both human and machine to jointly steer the direction of search present themselves. Furthermore, a search-based approach is complimentary to the manual heuristics and strategies of, for example, Wirfs-Brock and McKean [6], who offer manual "search strategies" for the discovery of candidate classes. Thus, in a cross disciplinary transfer of technology, we hypothesize that the benefits of an engineering design EC search-based approach may also be beneficial to software engineering design.

Indeed, search-based approaches are not new to software engineering. Applying search space exploration and optimization to software engineering was crystallized in a proposal by Harman and Jones in 2001 [30], and since then, search-based approaches to software engineering are being increasingly researched in all phases of the software development life cycle. A useful repository of publications in the field of search-based software engineering has been recently developed by Zhang [31]. For example, search-based software clustering and refactoring have been widely investigated; examples can be found in [32], [33], [34] and show promising results. Refactoring reorganizes software modules to promote their structural integrity by removing duplicate code to achieve "simpler" designs [35], while guaranteeing to preserve the semantics of the original software. Refactoring may be achieved by applying refactoring patterns [36]. (Of course, refactoring has no effect on the traceability of a design solution to its design problem; a design that does not trace to its problem requirements remains a poor design no matter how much refactoring is applied.) Clustering and refactoring approaches are applied downstream in terms of the software engineering life cycle, i.e., after the human software engineer has produced the necessary design artifacts, developed source code, and built and deployed the executable software. This differs significantly from the approach described in this paper wherein computational search and exploration supports the designer during the interactive process of upstream software design. Specifically, there are a number of significant differences between search-based refactoring approaches and the user-centered, interactive evolutionary search-based design advocated in this paper:

1. In terms of the software engineering development life cycle, search-based refactoring takes place after the human software engineer has designed and built the executable software; user-centered, interactive

search-based design takes place during upstream design before the design is realized in source code.
2. Search-based refactoring involves a representation of the solution space, whereas user-centered, interactive search-based design involves both a representation of the design problem and a representation of the design solution which are inherently traceable.
3. The representation of the solution space used by refactoring is typically derived from and maps to source code, whereas the representation of the solution space of interactive, search-based design is a UML class model.
4. By definition, refactoring must preserve the semantics of software, whereas interactive, search-based design changes the software design through an evolutionary process.

This paper's approach is consistent with other upstream approaches of search-based evolutionary algorithms within software engineering, such as the application of clustering techniques to software component architecture design [37]. A further upstream approach is that of Egyed and Wile [38], who report an interesting approach to support for managing design-time decisions using constraint satisfaction techniques. The designer commences design via a UML modeling tool and computational support reduces ambiguity among design choices by computing all of the legal possibilities using UML "well-formedness" rules, from which the designer may select. Indeed, approaches employing the application of well-formedness rules are also present in commercially available tools, e.g., [10], [11]. However, while this approach is interactive, Egyed and Wile see the approach as a largely manual search of the design space, the designer always having to provide initial models for the computational tool to work with.

Whenever complex interactive, user-centered human-computing activities are undertaken, agent-based support has often proved useful. We suggest, therefore, that the application of agent-based support of evolutionary algorithms for search-based upstream software design results in a more effective and efficient search process, better supporting the designer.

The idea of agent-based support for interactive human-computer activity is becoming increasingly established. Early work by Maes [39] reports the use of agents that reduce work and information overload in office-type applications. Through joint human-computer interaction with adaptive e-mail and meeting scheduling agents, humans increasingly trust the behavior of agents, which, in turn, become increasingly competent. Negroponte [40] suggests the notion of "interface agents" as a means of enhancing human-computer interaction in general. Wooldridge [41] describes a variety of research and industrial multiagent systems that support interactive human-computer activities, including agents acting as expert assistants or interface agents. Yet a variety of challenges remain for agent-based support of such interactive, human-computer activity. Bradshaw [42] points out that *"we want agents to be designed to fit well with how people actually work together,"* to ensure *"...effective and natural coordination, appropriate levels of modality and feedback, and adequate predictability and responsiveness to human control are maintained."* Klein et al. [43] elucidate this sentiment further by laying down 10 challenges for making automation a "team player" in joint human-agent activity. They first point out that all team players of joint activity, be they human or agent, must enter into an agreement, or *"basic compact,"* by which the participants intend to work together. From this agreement, all participants should be mutually predictable and directable in their actions, while maintaining common ground. Among the challenges that Klein et al. identify, perhaps the most relevant to upstream software design are 1) the need for agents to be able to observe and interpret pertinent signals of status and intention, and 2) the need for underlying technologies to support both planning and autonomous behavior of agents to enable a collaborative approach between participants. In the support approach applied in this paper, we attempt to rise to the challenges posed by Bradshaw and Klein et al.

Within the discipline of engineering design, Cvetkovic and Parmee [25] describe agent-based support within an interactive evolutionary design system. They report promising results of agent-based support for all phases of a conceptual design scenario for engineering design, and classify their agents within the specific context of conceptual design as

- *interface agents* that support natural user interaction, optionally hiding low level, uninteresting details from the designer,
- *search agents* that perform both single- and multi-objective search using evolutionary algorithms, and
- *information agents* that filter and mine design scenario information to make decisions about what is relevant to the designer and what is not.

Cvetkovic and Parmee's notion of interactive agency appears consistent with that expressed by Maes [39], Negroponte [40], and Wooldridge [41].

This paper reports the findings of two experimental early life cycle design episodes wherein a human designer and software agents interact and collaborate to jointly steer an evolutionary, multi-objective search toward useful and interesting class designs. Section 2 describes the necessary components of evolutionary search, namely, the search space representation, the objective fitness functions, and the genetic operators used to promote population diversity. Section 3 details the two example design problems, while Section 4 outlines the flow of the experimental design episode. Section 5 reveals the results obtained, and some limitations of the approach are stated in Section 6 before the paper concludes in Section 7.

2 Evolutionary Search

2.1 Representation

The search approach described below represents both the design problem and design solution. Indeed, the design solution representation is derived directly from the design problem representation, providing traceability from the design problem to the design solution.

The design problem domain representation is derived from use cases [44], which describe scenarios of interaction between user and the software system-to-be. The designer records the steps of the scenarios and, in particular, the

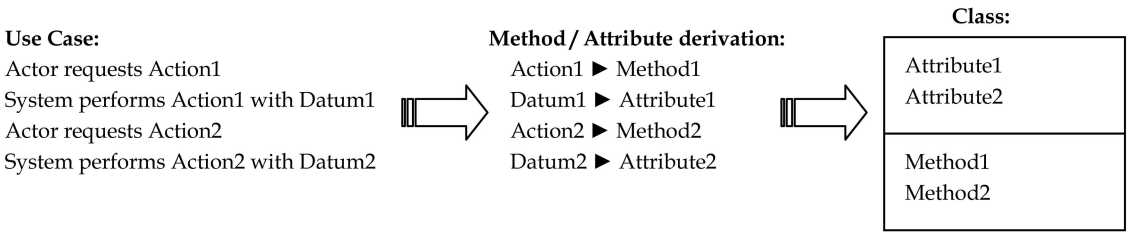

Fig. 1. Example of representation of design problem and corresponding possible design solution.

actions (verbs) and data (nouns) contained in each step, e.g., "The system deducts *(verb)* an amount *(noun)* from the account *(noun)*." If an action and datum are colocated in the same step of the narrative text, the action is said to "use" the datum. This representation of the design problem has been described previously by Simons and Parmee and further details can be found in [45]. The sets of 1) actions, 2) data, and 3) uses thus together constitute the scope of the design problem.

The representation of the design solution search space is object-oriented and comprises classes, methods, and attributes. Attributes are derived directly from the members of the set of data specified in the design problem, while methods are derived from members of the set of actions. The design search space thus comprises a set of attributes and a set of methods, derived from, and thus highly traceable to, the problem domain. Classes are represented as groupings of methods and attributes although, of course, there are many ways in which methods and attributes may be grouped into classes. Specifically, the notion of a class as a grouping in this representation requires that each class holds at least one attribute and one method, and that each method and each attribute are only allocated once to a class. The resulting search space is a highly scattered landscape of discrete candidate class designs with no fitness gradients immediately apparent. Fig. 1 provides a simple visualization of the representation.

Formally, the search 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 $M = \{\mathrm{m}_1, \mathrm{m}_2, \mathrm{m}_3\}$ and $A = \{\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3, \mathrm{a}_4\}$ with $c = 2$. An example of a partition of M into two parts is $\{\{\mathrm{m}_2\}, \{\mathrm{m}_1, \mathrm{m}_3\}\}$ and this partition identifies the two classes. An example of a surjection, f, from A to this partition is as follows:

$$f(\mathrm{a}_1) = \{\mathrm{m}_2\}, \quad f(\mathrm{a}_2) = \{\mathrm{m}_1, \mathrm{m}_3\}$$
$$f(\mathrm{a}_3) = \{\mathrm{m}_1, \mathrm{m}_3\}, \quad f(\mathrm{a}_4) = \{\mathrm{m}_2\}.$$

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

$$\{\mathrm{m}_2, \mathrm{a}_1, \mathrm{a}_4\} \quad \text{and} \quad \{\mathrm{m}_1, \mathrm{m}_3, \mathrm{a}_2, \mathrm{a}_3\}.$$

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

$$S(n, 1) = 1,$$
$$S(n, n) = 1,$$
$$S(n, r) = S(n-1, r-1) + rS(n-1, r).$$

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 $c!\mathrm{S}(a, c)$. Hence, by the product rule, the cardinality of the search space is given by

$$|S| = c!\,\mathrm{S}(a, c)S(m, c). \tag{1}$$

Table 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 and 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 class designs is beyond human comprehension. For example, eight attributes and eight methods allocated into five classes results in a search space of cardinality 132,300,000. We speculate that this exponential growth in the cardinality of the search space might be one of the many causative factors behind the difficulties of early stage class design.

2.2 Objective Fitness Functions

Objective fitness functions available to the designer relate to the structural integrity of the solution class designs and include cohesion of classes, coupling between classes, and number of classes in a design. Cohesion of classes is calculated according to an enhanced version of the Cohesion of Methods (COM) metric of Harrison et al. [46]. Generally, within a class, the more uses of attributes in the class by methods in the class, the more cohesive the class. (Conversely, should a method of a class use an attribute of another class, a couple between the two classes exists.) Appropriate

TABLE 1
$|S|$ as a Function of the Number of Attributes, a, the Number of Methods, m, and the Number of Classes, c

$A,m \setminus c$	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

For a convenient illustration, we have taken $a = m$.

"uses" are taken from the set of uses specified in the problem domain. Values of COM 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 $\mathrm{A_c}$ and $\mathrm{M_c}$ be the set of attributes and methods, respectively, that are contained in class C. Then, the COM fitness for a class C, denoted by $f(C)$, is given by

$$f(C) = \frac{1}{|A_c||M_c|} \sum_{i \in A_c, j \in M_c} \delta_{ij}, \tag{2}$$

where

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

As reported in [45], however, there is a limitation in COM: It fails to take account of class size. Thus, a class with a single attribute and a single method may score maximum cohesion (1.0), whereas a class with, for example, three attributes and three methods may also score maximum cohesion (1.0). To address this, the cohesion metric used in this study reflects class size by multiplying COM by the number of methods and attributes in a class. The modified cohesion fitness of a class C is given by

$$cohesion = (|A_c|\ |M_c|) * f(C). \tag{3}$$

Coupling between classes is calculated based on the Coupling Between Objects (CBO) metric taken from Briand et al.'s framework of coupling measurements [47]. Briand et al. define the CBO metric as follows:

$$CBO(c) = |\{d \in C - \{c\} | uses(c,d) \vee uses(d,c)\}|, \tag{4}$$

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 to 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 x) is not. The approach in this study 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 normalize the value of coupling of 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. More 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" to a class, is defined by

$$E = \{(m,a) \in U \mid \ \forall c \in C((m \notin c) \vee (a \notin c))\}, \tag{5}$$

and so the coupling value of a class design then becomes

$$\frac{|E|}{|U|}. \tag{6}$$

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.

2.3 Diversity Promoting Operators

The genetic operators used to promote diversity are recombination and mutation. Recombination is achieved by means of crossover, whereby two individual design solutions are chosen at random from the population and their attributes and methods are swapped between the two individuals based upon their position in the sequence of classes inside the two individuals. Mutation, on the other hand, is performed on a single design solution. Attributes and methods are selected at random and moved to a different class from the individual, again selected at random. With both the recombination and mutation operators, an important constraint of the representation is preserved, i.e., the operators do not result in a class lacking either attributes or methods. Recombination and mutation operators are illustrated in Figs. 3 and 4, respectively.

3 Example Design Problem Domains

3.1 Cinema Booking System (CBS)

The first example design problem domain used in this study is a generalized abstraction of a cinema booking system, derived from a number of established Internet-based cinema

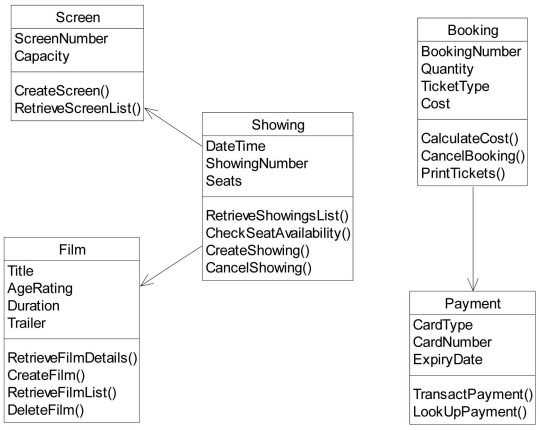

Fig. 2. Cinema booking system manual design.

booking systems existing in the United Kingdom. The design problem domain addresses, for example, making an advance booking for a showing of a film, and payment for tickets on attending the cinema auditorium. A specification of the use cases of the Cinema Booking System design problem is available from [48], and comprises 15 actions, 16 data, and 39 uses. Adhering to the use case specification, the authors' manual class design is shown in UML notation in Fig. 2; the direction of coupling is shown with arrowheads.

Using the cohesion metric described in the previous section, cohesion values of the manual design classes for the Cinema Booking System are shown in Table 2. With regard to the Coupling between Objects metric, the authors' manual design in Fig. 2 has three "external" uses. The total number of uses in the design problem specification is 39, thus the coupling value of the manual design is 3/39, or 0.077.

Applying (1), the cardinality of the resulting design solution search space, derived from this example problem domain, is 2.77248×10^{19}.

3.2 Graduate Development Program (GDP)

The second example design problem domain is an extension to a student administration system performed by the in-house Information Systems Department at the University of the West of England, United Kingdom. Over recent years, this university has sought to record and manage outcomes relating to personal student development during their studies. A strategic decision was made to extend the capabilities of the existing student administration system to be able to record and track a student's personal development in parallel with their academic achievement. As part of this extension, rules for validation of completion of personal development outcomes are required, as is the ability to extract various reports. The extension was implemented and deployed in 2008. A specification of the use cases used in the development is available from [49], and comprises 12 actions, 43 data, and 121 uses. Adhering to the use case specification, the authors' manual class design is shown in UML notation in Fig. 5.

Using the cohesion metric described in the previous section, cohesion values of the manual design classes for the Graduate Development Program are shown in Table 3. With regard to the coupling metric, the authors' manual design in Fig. 5 reveals four external couples. However, there are six attributes of "Development" used by "Rule" and one attribute of "Student" is used by four methods of "Development." This suggests a high degree of coupling between the classes "Student," "Development," and "Rule." This coupling results in a total of 12 external couples. The total number of uses in this design problem specification is 121, thus the coupling value of the manual design is 12/121, or 0.099.

Application of (1) yields a cardinality of the resultant design solution search space derived from this example problem domain of 1.31668×10^{17}.

4 Experiment: A Software Design Episode

The notion of an episode of upstream software design is useful for framing a session of interactive design. In detailing

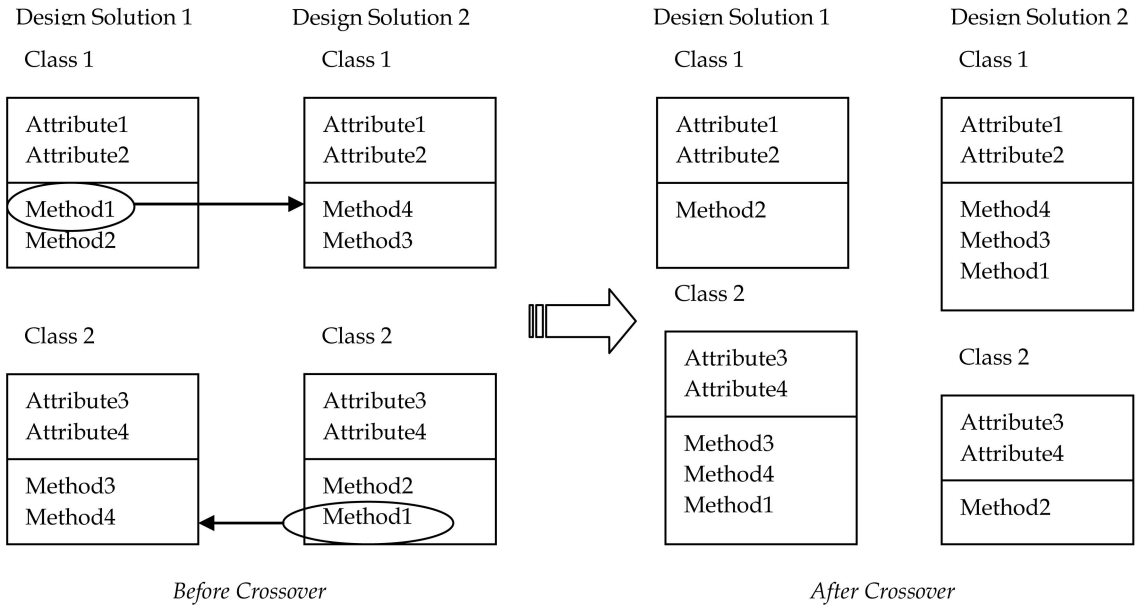

Fig. 3. Example of crossover operator.

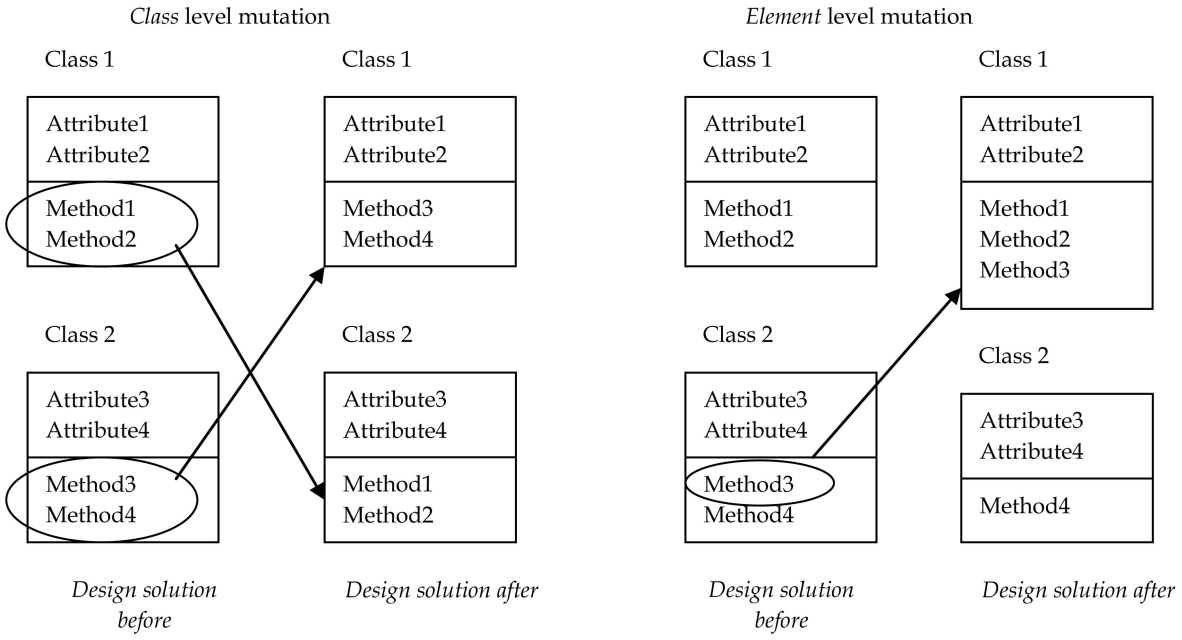

Fig. 4. Example of mutation operator mechanism at both class and element level.

the components of design thought, Lawson describes design events and design episodes. According to Lawson, design events may be the *"physical actions, drawing, modelling, gesturing, acting"* or *"verbalisations ... or entirely internal mental operations."* [50]. He goes on to provide examples of various types of design events: *"... a structuring of a problem, a proposition about a possible solution characteristic, a representation of a solution characteristic, an evaluation of a solution characteristic."* Lawson suggests that design events happen at a point in time and thus are atomistic, with an indivisible nature with respect to design. Lawson further proposes that, as events are usually not unconnected, they often exist as part of some larger purpose. A group of events is carried out to move the design forward in some way. Lawson refers to a group of events as an episode, thus: *"In a dramatic sense, they consist of a series of transactions that deal with a particular theme or themes that can be used to punctuate a larger narrative into the 'scenes' or 'acts' in plays or operas, or the 'chapters' in books, or the 'episodes' in longer running serials on television or radio. It is not the case that they are entirely discrete and separate from the rest of the narrative but that they seem reasonably self-contained"* [50]. This notion of a design episode appears useful and flexible and thus is used, in this paper, to frame a session of interactive evolutionary upstream software design. In outline, an early life cycle software design episode might run from start to finish as follows: First, a design episode requires a design problem.

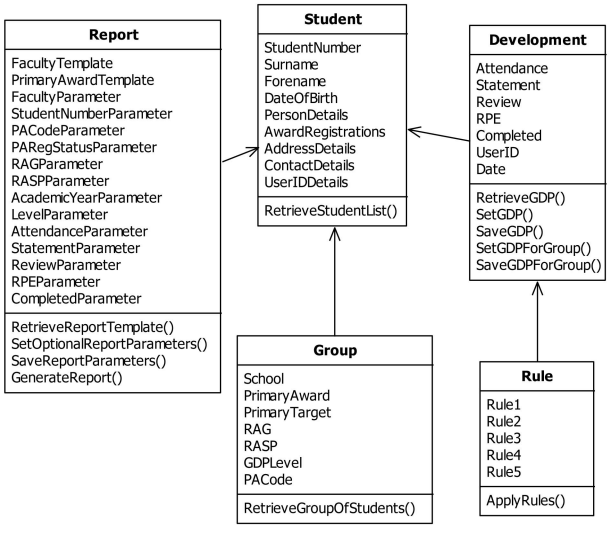

Fig. 5. Graduate development program manual design.

TABLE 2
Cohesion Values of Manual Design for CBS

Class	Cohesion value
Film	6.5
Showing	5.25
Screen	4
Booking	4.6662
Payment	3.125
Average	4.7082

TABLE 3
Cohesion Values of Manual Design for GDP

Class	Cohesion value
Student	10
Development	10.628
Rule	6
Group	8
Report	13.5
Average	9.6256

Representing design problems as use cases is widely applied in software engineering, and so the narrative text of the use case is used to identify the actions that the software is to perform, together with the data that the software will manipulate. From this, an initial population of candidate classes is directly derived. The designer having provided their search preferences (e.g., population size and number of generations) and chosen from the desired objective fitness functions available (e.g., values of class design coupling, cohesion of classes in a design, and number of classes in a design), a software agent performs global multiobjective evolutionary search of the design solution space. To efficiently guide the designer through the design space, further software agents may be used to isolate discrete zones mapping to specific design concepts of superior fitness for subsequent local search. At any point in the episode, the designer may select a class design to add to the portfolio should it appear useful or interesting.

The above outline is described in more detail below. For the purposes of illustration, a numbered sequence is shown, although, in practice, a more iterative approach may be adopted by the designer. Typically, the first event is to commence logging of design events.

4.1 Logging of Design Events

Logging of design events is facilitated by an Event Logger software agent, which has knowledge of all possible design events, together with the date and time within the episode at which the design event occurs. The Event Logger Agent provides the designer with a design event notification service and a chronological trace of all design events as they occur during the episode, including any preferences and choices of the designer to steer the search.

4.2 Expressing Search Preferences

With the design problem recorded, the designer provides their search preferences for both global and local search. Preferences such as population size, number of generations, and choice of fitness functions (and where applicable, mutation and crossover rates) are supplied by the designer, although the designer may opt to select preset default values provided.

4.3 Initializing the Design Solution Space

The design search space is initialized with a population of candidate class designs. The initial population of candidate class design solutions is created by first creating each individual class design with a random number of classes. The minimum number of classes is one, and the maximum number of classes is the smaller of the cardinalities of the set of attributes and the set of methods (as each class must comprise at least one attribute and one method). Second, attributes and methods are assigned at random to classes within the individual design.

4.4 Performing Global Multi-Objective Search

Multi-objective evolutionary search explores the global design solution space, trading off conflicting objective fitness functions selected by the user. This evolutionary search has been outlined in [51] and is detailed as follows: In essence, global search is a balance between exploration (achieved by diversity promotion) and exploitation (achieved by fitness-based selection).

The multi-objective evolutionary search algorithm used in this paper was inspired by the elitist Nondominated Sorting Genetic Algorithm (NSGA-II) proposed by Deb in 2001 [52]. Since then, NSGA-II has been applied to a variety of multi-objective problem domains and in a recent overview of multi-objective evolutionary algorithms (MOEAs), Coello Coello et al. [53] reported that NSGA-II has been used frequently in comparison with newly designed MOEAs. Following the approach of NSGA-II, the algorithm used in this paper produces an offspring population of the same size as the parent population by mutation (mutation having been found to be more explorative than crossover and computationally more efficient). Using fitness functions, the combined population is nondomination sorted into ordered "fronts" of equivalent optimality. The new population is filled by solutions of different nondominated fronts, one at a time; the filling starts with the best front, and continues with the next best, and so on, until the population size is met or exceeded. Solutions that do not make the new population are discarded. Typically, the last front contains more solutions than there is space available. In selecting which solutions go forward, a "crowding distance sorting" operator is employed to ensure that the most diverse range of solutions is preserved, thus helping the algorithm to explore the fitness landscape. In the early stages of evolution, many fronts are evident in the population. However, as evolution proceeds, the number of fronts decreases until eventually a small number of fronts, including the Pareto-optimal front, remain. At this point, the crowding distance sorting operator is crucial in preserving diversity of solutions. The processes of the nondominated sorting algorithm are illustrated by a flowchart in Fig. 6.

The approach taken to design tasks in the experimental episode in this study is to exploit computational concurrency for performance gains wherever possible. Hence, a software agent performs multiobjective evolutionary search as a task within its own autonomous thread. The approach also takes account of the need, as far as is possible, to prevent potential user fatigue associated with the visual presentation of a population of class designs. Thus, a mechanism is necessary to narrow the global multiobjective search toward the most useful and interesting candidate designs; this is achieved by exploiting the characteristics of the design search space to isolate discrete zones with specific numbers of classes within the global population and taking the zones forward for local search. The results of the subsequent local search(s) are presented to the designer (although the designer may visualize candidate designs prior to local search if they wish).

Nevertheless, there exists a limitation with the Pareto-optimal fronts obtained which relates to the scattered nature of the landscape of discrete class designs in the design search space. Multi-objective search and optimization using evolutionary algorithms have been successfully applied to a variety of problem situations where the goal is to find multiple trade-off optimal solutions with a wide range of real values for objectives [53]. Typically, fronts of trade-off optimal solutions for two conflicting objective functions may be visualized in 2D graphs where x and y-axes use continuous scales, thus enabling a diverse range of values of equivalent optimality to exist within the Pareto-optimal front. However, in this study, although the x-axis uses a continuous scale to reveal, for example, coupling values, the y-axis uses a discrete scale to reveal the number

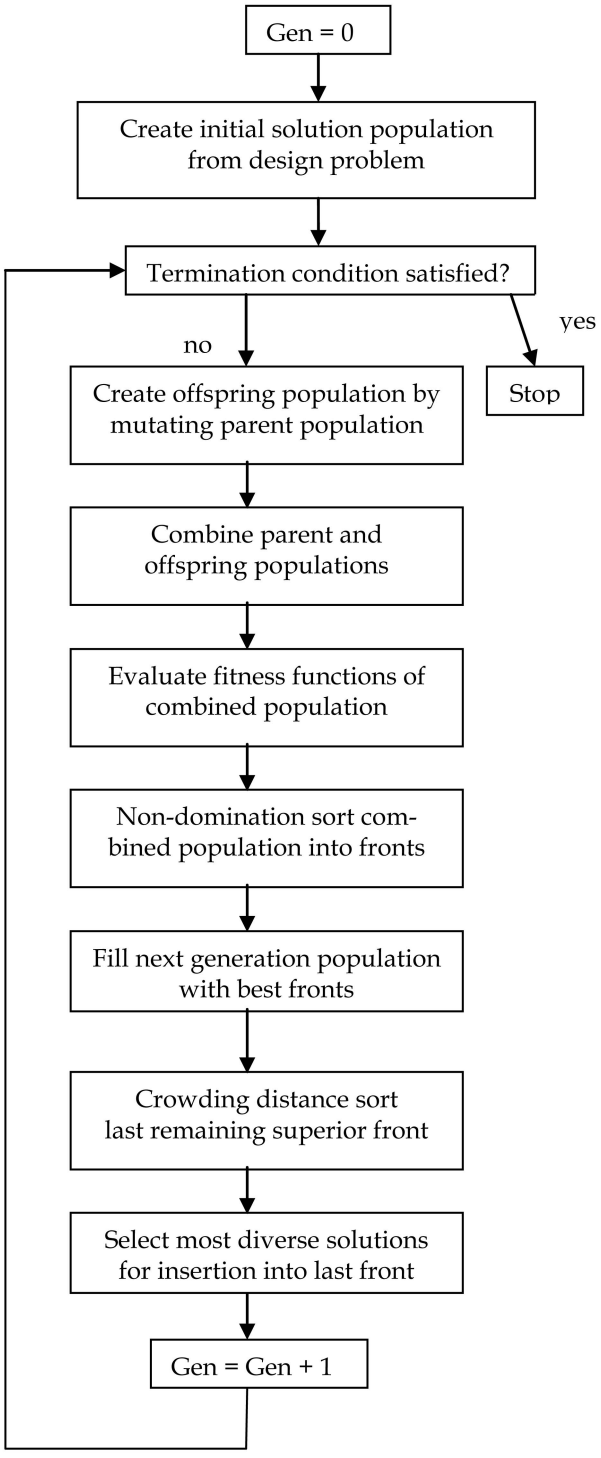

Fig. 6. Nondominated sorting evolutionary algorithm.

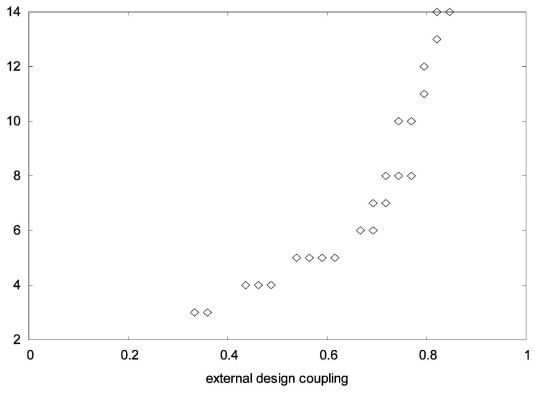

Fig. 7. Population after 50 generations.

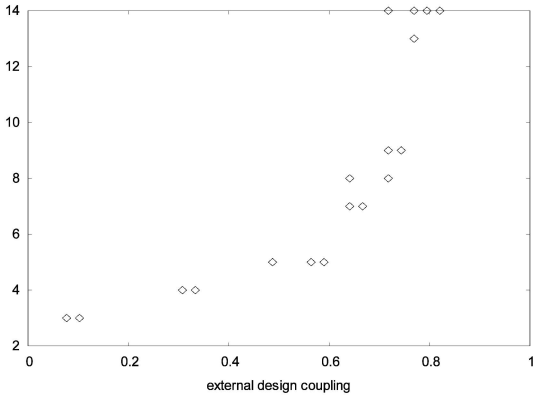

Fig. 8. Population after 100 generations.

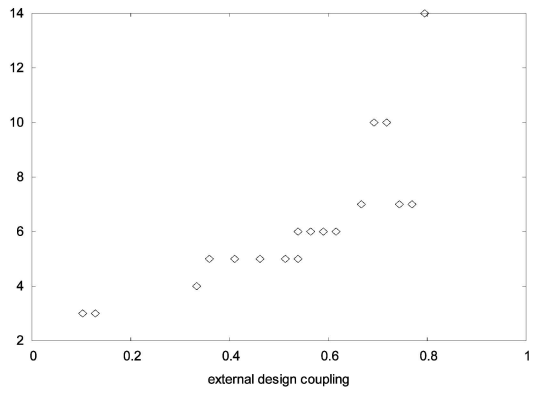

Fig. 9. Population after 200 generations.

of classes in a class design. As the evolutionary search progresses, a direct consequence of the use of a discrete scale is the possibility of reduced diversity among class designs in fronts of equivalent optimality. To illustrate this reduction of diversity, see Figs. 7, 8, 9, and 10, which show trade-off fronts for a multi-objective search using the Cinema Booking System example design problem domain (detailed in Section 3) for a population size of 100 after 50, 100, 200, and 300 generations, respectively. In Figs. 7, 8, 9, and 10, the trade-off curve between design coupling and the numbers of classes in a design can clearly be seen. In addition, an increase in the population fitness with respect to the minimization of coupling is evident as the search proceeds. However, it is also apparent that banding of designs occurs due to the discrete scale of the y-axis, resulting in discrete "zones" of designs with the same number of classes. Effectively, this banding phenomenon partitions the global search space into local zones of designs that all share the same number of classes; so all designs in a zone might be comprised of, for example, 6 classes or 10 classes, etc. While this provides a useful mechanism to narrow the search of the global design space, a further consequence, however, is that diversity in the front of nondominated designs (the Pareto-optimal front) is constrained to the range of integer values on the y-axis. In

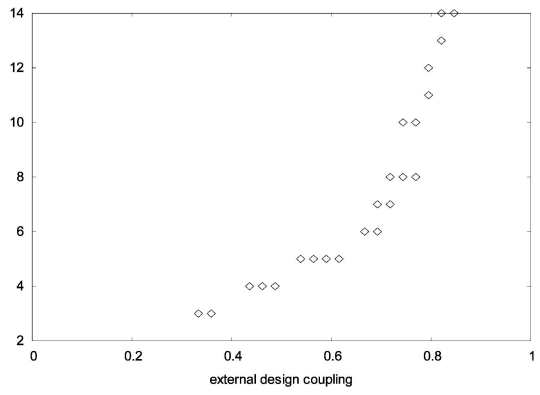

Fig. 10. Population after 300 generations.

the search illustrated in Figs. 7, 8, 9, and 10, the maximum number of classes, as defined by the bounds on the search space, is 14. Thus, within a population of, say, 100 designs, the upper limit of diversity achievable in the Pareto-optimal front is typically many fewer unique designs. Furthermore, the mechanism for selection of individual designs for the next generation as used in a multi-objective nondominated sorting algorithm, such as NSGA-II, may result in individuals from particular zones being eliminated from the population, producing gaps in the fronts of nondominated designs. Thus, a trade-off is apparent: As the search progresses through generations, overall population fitness with respect to coupling improves as discrete zones appear, but this comes at the expense of emerging sparsity in the fronts of nondominated designs. It is clear that allowing the multi-objective evolution to progress to full convergence is undesirable within an interactive search, as useful and interesting designs may be lost. Hence, the approach used in this study is to halt the evolutionary search at a judicious number of generations to enable the trade-off between minimized coupling and diversity preservation to be made. This is achieved by one of two ways: either the designer may manually specify various diversity threshold values for the population of designs which halt multi-objective evolution or an isolation software agent may be employed to monitor the utility of the population with respect to the coupling versus diversity trade-off as evolution proceeds.

4.5 Isolation of Local Search Zones

In order to manually halt the multi-objective search to preserve population diversity, the designer may specify a number of threshold values prior to evolution:

- *Sparsity:* Halt evolution when a specified number of discrete values on the y-axis are eliminated from the population.
- *Duplicates:* Halt evolution when there are two or more identical solutions are evident in the population.
- *Fittest individual class:* Halt evolution when any class in any class design in the population achieves a specified cohesion fitness value.
- *Average design cohesion:* Halt evolution when the average population cohesion achieves a specified cohesion fitness value.
- *Lowest design coupling:* Halt evolution when any class design in the population achieves a specified coupling fitness value.

It is also possible for the designer to combine threshold values, e.g., halt evolution when a specified sparsity value and an average design cohesion are achieved or when a specified duplicates value or a lowest design coupling is achieved.

Alternatively, an isolation software agent may be deployed by the designer to halt evolution at a point when the utility of the population, as evaluated by the agent, becomes unacceptable. The utility of a generation is assessed as a composite measure of the search state at a generation (with respect to fitness, sparsity, and duplicates) and a longer-term reflection of performance of the evolutionary algorithm over time. The utility of a generation is computed using four properties of the population:

- *Fitness*, i.e., average population coupling,
- *Sparsity*, i.e., the sparsity count of the population divided by the maximum number of classes in the design solution search space,
- *Take over*, i.e., the number of individual designs with one or more duplicates divided by the population size,
- *Elapsed time*, i.e., the current generation number divided by the number of generations specified as a preference for evolutionary search by the designer.

Utility (u) is computed as

$$u = fitness - sparsity - take\ over - elapsed\ time.$$

The designer can specify the isolation agent to halt evolution when either:

- utility becomes negative or
- a single fall in utility is observed or
- two sequential falls in utility are observed.

4.6 Local Search

After the local zones have been isolated from the multi-objective search, local search of the individual zones may be initiated in one of two ways: either the designer manually selects particular concept zones for local search or a local search software agent performs the selection based on diversity of the zones. For manual selection, a list of all zones is provided to the designer, together with an indication of the diversity present in each zone in terms of the number of unique designs in the zone. The designer may manually select one or more zones to go forward to local search. Alternatively, a zone isolator agent may also assess the zones for diversity, and select zones where the number of classes lies in the middle two quartiles of range.

Prior to each local search commencing, it is necessary to initialize the design population of each zone. However, at the point when the multi-objective search is halted, the zones may be comprised of fewer individual designs than the population size local search preference expressed by the designer. Should this be the case, individual designs in a zone are first placed in the local search population, and then the individual designs are repeatedly mutated into the local search population until the local population size preference is reached. This mechanism promotes diversity in the local search. Each local search is conducted using a single-objective genetic algorithm using coupling as the objective fitness

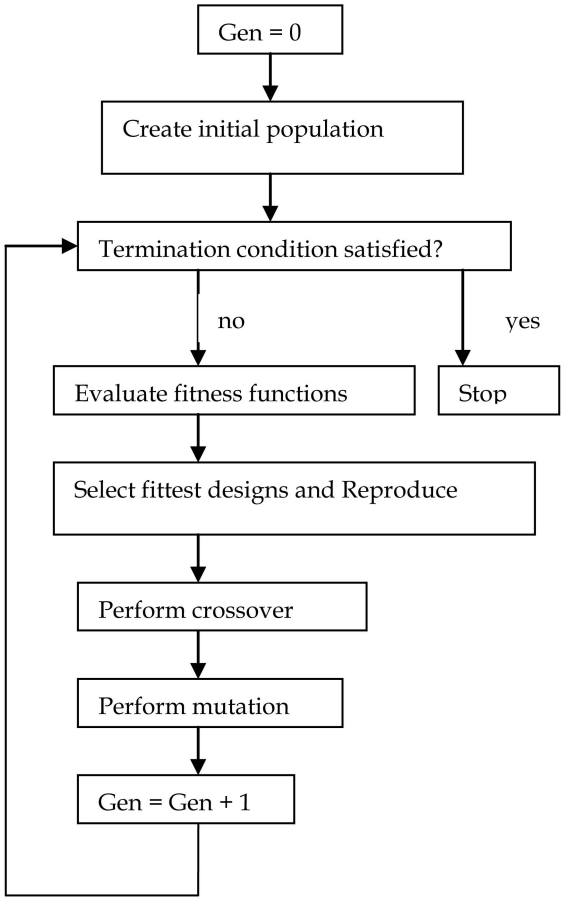

Fig. 11. Local search evolutionary algorithm.

function. A summary flowchart of a genetic algorithm is shown in Fig. 11. The genetic algorithm operates on the same discrete design representation as the multi-objective search, and uses both crossover as its recombination/selection operator and mutation. Within each local search, the number of classes of each design remains unaltered. Local search proceeds until the preferred number of generations of the designer is reached. To achieve the best execution time, computational concurrency is exploited by executing each local search agent in parallel, each agent executing its behaviors as an autonomous task. This enables up to approximately 20 local searches to be conducted in parallel. After local search, candidate designs may be inspected by the designer, who may add useful and interesting designs to the Design Portfolio as required.

The behaviors of all software agents employed in this approach (episode logger agent, multi-objective search agent, discrete concept zone isolator agent, and local search agent) are coordinated and controlled by an Agent Controller (itself a software agent) that regulates both the life cycles and autonomous behaviors of the agents.

5 Results

5.1 Design Episode Event Log Trace

To provide an illustration of an event log trace, an extract of the event log of the design episode for the Graduate Development Program is provided below. A full trace of the event log can be found in [54].

30 July 2008 15:02:49 BST New design episode started. Episode name is: 'lee nick 30 07 2008'
30 July 2008 15:02:59 BST Design problem selected. Graduate Development Programme (GDP)
30 July 2008 15:03:11 BST Multi-objective search parameter selected. MOGA population size set to 100
30 July 2008 15:03:11 BST Multi-objective search parameter selected. MOGA number of generations set to 500
30 July 2008 15:03:11 BST Multi-objective search parameter selected. MOGA search objective function set to external coupling
30 July 2008 15:03:11 BST Multi-objective search parameter selected. MOGA search objective function set to number of classes
30 July 2008 15:03:11 BST Multi-objective search parameter selected. MOGA search priority set to variety
30 July 2008 15:03:17 BST Local search parameter selected. Population size set to 100
30 July 2008 15:03:17 BST Local search parameter selected. Number of generations set to 100
30 July 2008 15:03:17 BST Local search parameter selected. Crossover rate set to 70 percent
30 July 2008 15:03:17 BST Local search parameter selected. Mutation rate set to 3 percent
30 July 2008 15:03:17 BST Local search parameter selected. Reproduction/selection operator set to tournament
30 July 2008 15:03:22 BST Isolation of discrete zones selected.
30 July 2008 15:03:22 BST Isolation factor specified. preference of two consecutive falls in utility selected for global search halting
30 July 2008 15:03:40 BST Multi-objective search parameter selected. MOGA population size set to 100
30 July 2008 15:03:40 BST Multi-objective search parameter selected. MOGA number of generations set to 500
30 July 2008 15:03:40 BST Search strategy selected. Search strategy set to multi-objective
30 July 2008 15:03:40 BST Search population initialised. Multi-objective search
30 July 2008 15:03:42 BST Multi-objective search parameter selected. MOGA search priority set to variety
30 July 2008 15:03:42 BST Multi-objective search parameter selected. MOGA search objective function set to number of classes
30 July 2008 15:03:42 BST Multi-objective search parameter selected. MOGA search objective function set to external coupling
30 July 2008 15:03:42 BST Isolation agent specified. two falls
30 July 2008 15:03:42 BST MOGA Search started.
30 July 2008 15:03:46 BST Isolation agent stopped MOGA search. average population coupling is 0.697, sample generation is 57, sparsity count is 0, duplicate count is 3, utility is 0.462
30 July 2008 15:04:45 BST Local search agent starting local search. Agent is not selected to choose discrete zones. Designer has manually selected the following discrete zones: 4, 5, 6, 7,

30 July 2008 15:04:46 BST Local search initatiated. Discrete zone 4
30 July 2008 15:04:46 BST Local search initatiated. Discrete zone 5
30 July 2008 15:04:46 BST Local search initatiated. Discrete zone 6
30 July 2008 15:04:46 BST Local search initatiated. Discrete zone 7
30 July 2008 15:04:56 BST Local search completed. Discrete zone 7
30 July 2008 15:04:57 BST Local search completed. Discrete zone 6
30 July 2008 15:04:58 BST Local search completed. Discrete zone 5
30 July 2008 15:04:59 BST Local search completed. Discrete zone 4
30 July 2008 15:08:09 BST Design added to Portfolio. design 'award class' added.star rating is: 2
30 July 2008 15:10:46 BST Design added to Portfolio. design 'possible student class with high cohesion' added.star rating is: 2
30 July 2008 15:16:12 BST Design added to Portfolio. design 'zone 7 too dispersed' added.star rating is: 1
30 July 2008 15:20:16 BST Design added to Portfolio. design 'z6 d1 number of possible classes' added.star rating is: 3
30 July 2008 15:22:39 BST Design added to Portfolio. design 'z6 d2 possible mix of classes' added.star rating is: 3
30 July 2008 15:27:01 BST Design added to Portfolio. design 'z6 d3 very similar to z6 d1, d2' added.star rating is: 3

5.2 Global Multi-Objective Search

Default search parameters have been chosen empirically for the NSGA-II inspired genetic operators based on previous studies [45]. All results of global search given below have been achieved using preset default global search parameters as follows:

- *selection:* no selection performed;
- *crossover and mutation probabilities:* (0.0, 1.0) (i.e., no crossover performed);
- *offspring creation and replacement strategy:* (100, 100) (i.e., 100 parents generate 100 offspring by mutation) and (100, 100) replacement (i.e., the least dominated 100 individuals out of the combined population of 200 go forward to the next generation).

Fig. 12 shows results of experiments into the average utility of a population of class designs (plus or minus one standard deviation) for the Cinema Booking System problem domain averaged over 20 global multi-objective searches. The lower curve on Fig. 12 reveals utility values as calculated by the previously stated formula:

$$u = (1 - coupling) - sparsity - take\ over - elapsed\ time.$$

(Fitness is expressed as $1 - \text{coupling}$, as the Coupling between Objects metric is a minimization function.) The lower curve reveals that the utility of the global search population climbs to a peak of 0.2 at approximately 20 generations and becomes negative at approximately 90 generations. Thereafter, as global search proceeds,

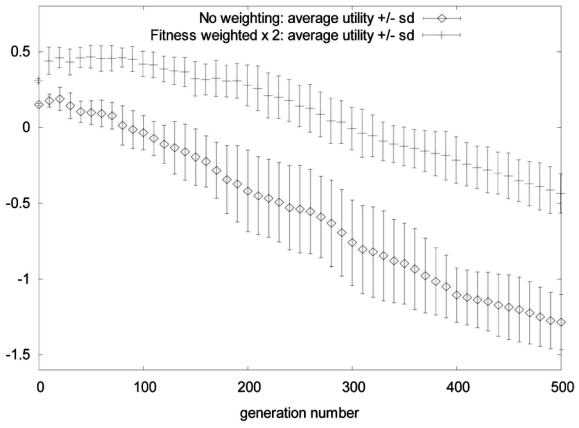

Fig. 12. Average population utility.

population utility falls steadily to -1.2 at 500 generations, revealing the combined impact of the three negative factors (sparsity, take-over, and elapsed time) despite improving fitness of the population. In terms of halting the search, it seems likely that a peak utility of 20 generations occurs a little early in the search. To address this, the fitness factor in the utility calculation has also been weighted by 2 to balance against the three contrary factors. The resulting average population utility is shown as the upper curve in Fig. 12, where average population utility peaks at approximately 0.5 after 100 generations and, although declining, remains positive until approximately 300 generations. The weighted calculation of utility appears more useful as an expression of utility; its slower decline in utility allows the search to progress and thus improve population fitness overall. The weighted calculation of utility has, therefore, been used in this study as a whole.

The number of generations for evolutionary global search is thus not specified a priori. Indeed, stopping global search, in general, prior to population convergence can be advantageous in preventing user interaction fatigue, and need not prevent useful results. For example, Tagaki [55] surveys a wide range of open system applications of interactive evolutionary computing (IEC), including various industrial fields of design (and many others, such as virtual reality, data mining, image processing, control and robotics, geophysics, social system simulations, etc.). Based on his observations of a wide range of applications of interactive evolutionary computing, Tagaki asserts that it generally does not require a large number of generations to achieve satisfactory results as it is sufficient for a subjectively evaluated task to reach an optimum area of the solution search space rather than a single point. In software design refactoring, Harman and Tratt [32] observe similar findings. While refactoring software designs through multi-objective search, Harman and Tratt find that, by generation 20, the Pareto front of designs evolves to a reasonable approximation to the front achieved after 200 generations. Harman and Tratt go on to suggest that achieving good results after relatively few runs is useful, for example, in large-scale designs, as it allows developers to get reasonable quality answers quickly, and helps prevent user fatigue. This is consistent with the global search approach presented in this

TABLE 4
Results of Global Search Halting Tactics for CBS

	Zero Utility		Single Fall		Two Falls	
	Value	SD	Value	SD	Value	SD
Generation	291.9	56.2	18.55	7.11	58.40	29.54
Coupling	0.298	0.194	0.700	0.025	0.626	0.064
Sparsity	5.00	0.85	1.35	0.81	1.95	1.14
Duplicates	50.35	34.94	1.85	2.11	4.25	3.74
Utility	0.00	0.00	0.44	0.05	0.45	0.09

TABLE 5
Results of Global Search Halting Tactics for GDP

	Zero Utility		Single Fall		Two Falls	
	Value	SD	Value	SD	Value	SD
Generation	356.9	82.3	17.50	6.50	46.30	20.30
Coupling	0.480	0.127	0.743	0.016	0.710	0.029
Sparsity	1.9	1.07	0.05	0.22	0.10	0.31
Duplicates	11.2	10.4	1.40	1.80	2.75	2.60
Utility	0.00	0.00	0.46	0.03	0.45	0.03

paper, as designers are free to manually restart a halted global search should they consider, after inspecting design visualizations, that population design coupling is of insufficient coupling fitness.

However, a further issue arises: What tactic should the isolation agent apply to halt the search? To experiment, three tactics have been trialed and include halting the global search at

- zero utility,
- a single fall in utility,
- two contiguous falls in utility.

Tables 4 and 5 reveal the average values of 20 multi-objective searches for the CBS domain and the GDP domain, respectively. The generation number at which the Isolation Agent halts the search is given, and then at that generation, average population coupling, sparsity, number of duplicates, and utility. Tables 4 and 5 are analyzed, respectively, as follows: For the zero utility tactic, search is halted at generation 291/356 with a coupling fitness of 0.298/0.480. While this is a promising coupling value, population sparsity and the duplicate count are high, indicating a lack of diversity in the population. For the single fall tactic, search is halted at generation 18/17 with a poor coupling value of 0.700/0.743. Although population diversity is good, the search has not progressed sufficiently to yield a good coupling balance. However, for the two falls tactic, search is halted at generation 58/46, giving a better trade-off between coupling and population diversity. Thus, in this study, the Isolation Agent adopts the tactic of halting search when two contiguous falls in utility are detected.

Having said that, the use of weights in the calculation of evolving population utility could be treated as a parameter to the search approach, which potentially raises a number of issues. First, while the use of weighted objectives in the utility calculation is computationally straightforward, the choice of value for the weights is not, and empirical investigation is typically necessary. It is also necessary to ensure that the scales of the weighted objective values are comparable, and this may involve normalization of the objective values prior to weighting. Furthermore, difficulties may arise either where positive and negative fitness values or maximization and minimization fitness function are to be traded off. In the above approach, for example, the coupling fitness value is expressed as $(1 - \text{coupling})$ in the utility calculation. Nevertheless, the relatively straightforward computation of weights has led to the application of weights in a number of multiobjective search-based approaches, for example, automated design improvement [56], software effort estimation [57], test data generation [58], and pairwise interaction testing [59].

On the other hand, the above weighted approach to utility calculation might be adapted to avoid weights by using Pareto-optimality multi-objective approaches. It seems likely that a nondominated sorting genetic algorithm, similar to the one employed in the global search described in this paper, could derive Pareto-optimal fronts that trade off the coupling of designs against sparsity in the evolving population. Progression of Pareto-optimal fronts might be monitored as search progresses, and global search halted at a judicious generation when the Pareto front is an acceptable approximation of the converged population. Because of the issues associated with weights in multi-objective search, approaches using Pareto optimal have been increasingly applied to search-based software engineering and include, for example, the software quality classification problem [60], the next release problem [61], and test data generation [62]. Interestingly, Zhang et al. [63] investigated both weights and Pareto-optimal approaches in the next release problem and observed that the NSGA-II algorithm [52] is well suited.

5.3 Local Search

Optimum settings for local search parameters for both example design domains have been chosen empirically based on previous studies [45] and are as follows:

- *selection:* tournament;
- *crossover and mutation probabilities:* (0.7, 0.03);
- *offspring creation and replacement strategy:* (100, 100) (i.e., 100 parents generate 100 offspring and only those offspring become parents of the next generation).

After local search has been performed, the designer may inspect individual class designs and add them to the portfolio. Individual classes are visualized in colors indicative of their cohesion: highly cohesive classes are red

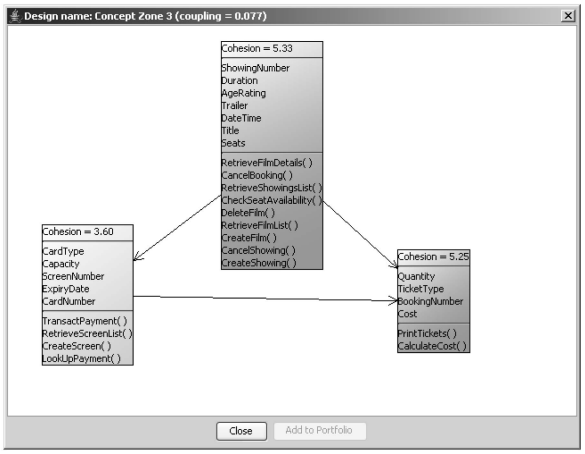

Fig. 13. Sample CBS class design from zone 3.

(hot) while poorly cohesive classes are blue (cool), with intermediate cohesion shown in gray and yellow. The designer might typically explore the fittest zones first, moving through zones of decreasing fitness. Given that local search is population-based, many design solutions of equivalent optimality may be inspected by the designer in each zone. Space constraints unfortunately preclude a complete listing of the populations of designs arrived at by local search; rather, typical example class designs are presented and analyzed as follows:

5.3.1 CBS Local Search Results

An example class design from zone 3 is shown in Fig. 13; all designs in zone 3 have three classes. Immediately apparent is the low coupling in the design. However, although the class design in Fig. 13 has the same coupling value as the manual design presented previously, the cohesion of the classes is adversely affected by their size: The classes are too large to be comprehensible to the designer. This finding is not unexpected given the number of classes in this zone, i.e., three. This design being not hugely interesting to the designer, he/she thus moves on to examine class designs from zone 4.

An example class design from zone 4 is shown in Fig. 14. All designs in this zone have four classes. Fig. 14 also exhibits the same coupling value as the manual design, but as the modularity is more fine-grained with four classes in the design, the beginnings of meaningful classes emerge to interest the designer. With cohesion of 5.85, a class similar to the manual concept of "Film" is present, together with classes similar to "Booking" (cohesion 4.50) and "Payment" (Cohesion 4.00). However, there exists a class with a cohesion value of 4.58 whose design concept remains confused; this class is also strongly coupled to the class resembling the "Film" concept. With the suggestions of the design concepts of "Film" and "Booking" in mind, the designer now moves to zone 5 to investigate if further suggestions for useful classes exist. The class design in Fig. 15 reveals that modularity is a little more fine-grained than the previous zone, with smaller classes evident. Once again, design coupling is the same as found in the manual class design (see Fig. 2). A class with a cohesion value of 6.50 is presented to the designer as the highest cohesion value observed in any class in the manual design and also the highest cohesion value generated by

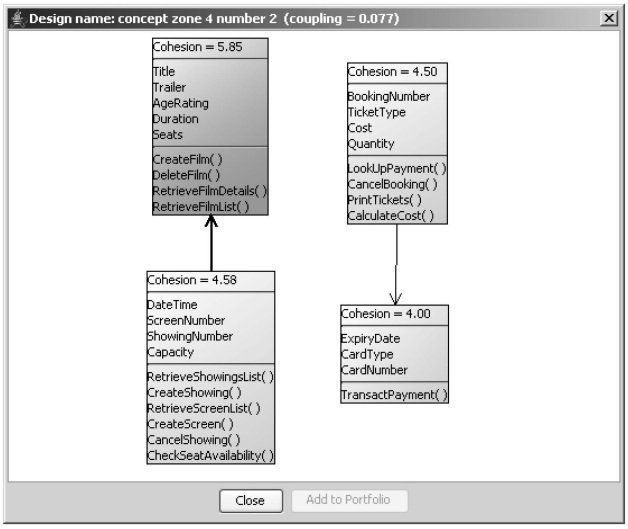

Fig. 14. Sample CBS class design from zone 4.

search. This high cohesion class exactly resembles the manual class mapping to the concept of "Film." A further class with a cohesion value of 4.17 bears a resemblance to the manual design class "Screen." However, two other classes with cohesion values of 4.00 and 2.00 might appear to trace to the manual concept of "Showing" but are cleaved into two classes. As zone 5 shows promise, the designer examines a further class design from the concept zone, shown in Fig. 16.

At first glance, the second class design from zone 5 might appear less promising than the previous example as its coupling value is higher at 0.103. However, like the previous design, this further example also suggests a class with a cohesion value of 6.50, exactly mapping to the manual class "Film," and a class of cohesion 4.00 that maps closely to the manual class "Screen." Furthermore, this design stimulates the designer to consider a number of interesting suggestions. For example, a class of cohesion 4.167 is arrived at which maps closely to the manual class of "Showing." The large confused class of previous designs has now been cleaved to suggest two new classes mapping closely to the manual

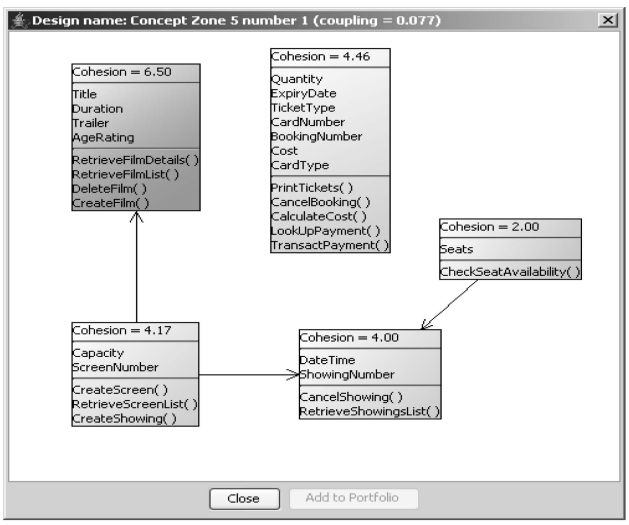

Fig. 15. Sample (1) CBS class design from zone 5.

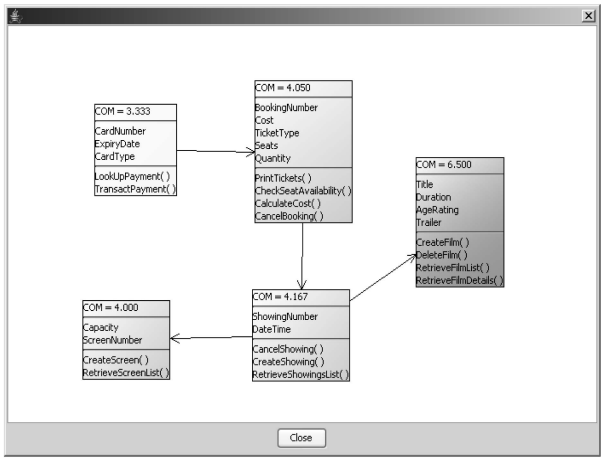

Fig. 16. Sample (2) CBS class design from zone 5.

concepts of "Booking" and "Payment." Perhaps most interesting is the suggestion of a couple from "Booking" to "Showing" interesting and useful as it addresses a possible deficiency in the manual design, namely the need for a structural relationship between "Booking" and "Showing" to relate all of the classes in the design; without this couple, the concepts of "Booking" and "Payment" may seem divorced from the concept of "Showing" to which a "Booking" is related. Having been stimulated by promising suggestions of class designs, the designer might place the promising design in the portfolio.

5.3.2 GDP Local Search Results

A sample class design from zone 3 is shown in Fig. 17. This class design has a coupling value of 0.019, which is markedly

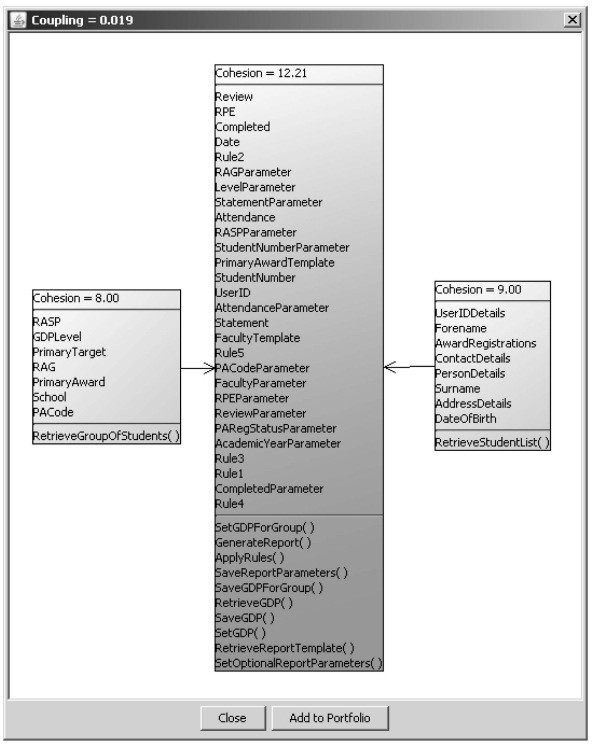

Fig. 17. Sample GDP class design from zone 3.

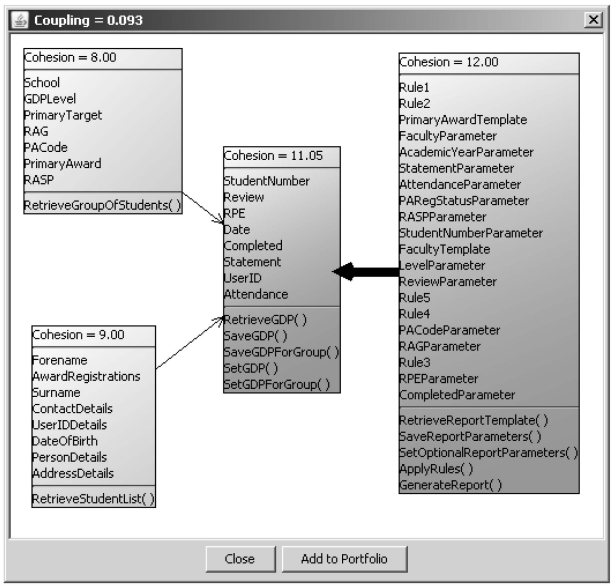

Fig. 18. Sample GDP class design from zone 4.

superior to the coupling value of the manual design (0.099). A large class with a cohesion value of 12.21 is immediately apparent, comprising 28 attributes and 10 methods. This large class is typical of the classes found in this zone of three class designs, and despite the superior coupling value of the design overall, the relation of the large class to the design problem is unclear. Nevertheless, the local search arrives at a class with a cohesion value of 8.00 that exactly matches with the manual design class "Group," and also a class with a cohesion value of 9.00 that closely matches the manual design class "Student."

A sample class from zone 4 is shown in Fig. 18. With an external coupling value of 0.093, the class design has a similar coupling value as the manual design. The local search arrives at the design classes "Group" and "Student," but interestingly suggests a new class with a cohesion value of 11.05 that bears a resemblance to the manual design class "Development." At 11.05, the cohesion value of this new class is superior to the cohesion value of the manual "Development" class at 10.628. However, a large class is also evident in the design, holding 20 attributes and 5 methods, and despite a cohesion value of 12.00, the relation of this class to the design problem is not entirely clear. It is possible that elements of the "Report" class and the "Rule" class are combined, which may be a consequence of the number of classes in this local search zone, i.e., 4. A sample class design from zone 5 is shown in Fig. 19. With an external coupling value of 0.084, the class design has a superior coupling when compared with the manual design. Once again, classes with a strong resemblance to the manual classes "Group" and "Student" are evident. Interestingly, the zone 5 local search suggests a class of cohesion 13.62 with a strong resemblance to the manual design class "Report." Furthermore, the search suggests a design class with a cohesion of 11.33 which seems to encapsulate the attributes and methods of the "Development" and "Rule" manual classes. However, the search has struggled to resolve the strong coupling between the "Development" and "Rule" classes, resulting in a class of zero cohesion with one attribute and one method. We

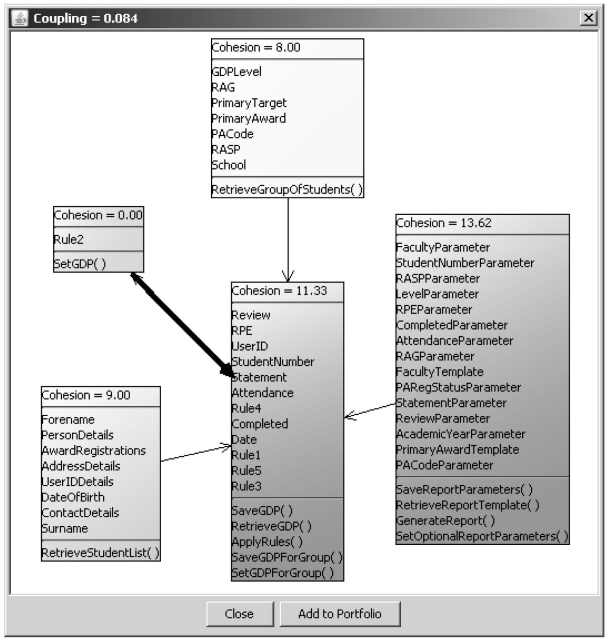

Fig. 19. Sample (1) GDP class design from zone 5.

speculate that the use of external coupling as a fitness function in the local search drives down the coupling value of the design as a whole but is less effective in steering the local search when high levels of coupling exist between design classes. This is also illustrated in a further example class design from zone 5 shown in Fig. 20. Again, the external coupling value of the design is marginally superior to the manual class design. Once again, design classes with a strong resemblance to manual design classes "Group," "Student," and "Report" are evident. However, it is also evident that the class with a cohesion value of 11.25 encapsulates many of the

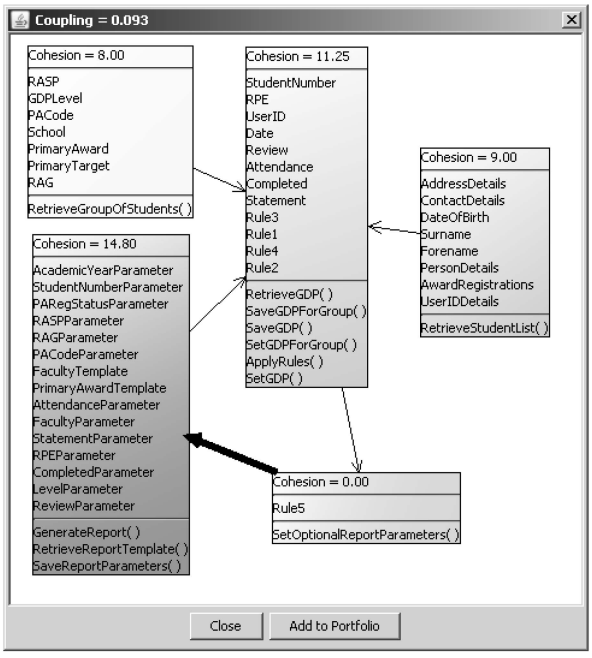

Fig. 20. Sample (2) GDP class design from zone 5.

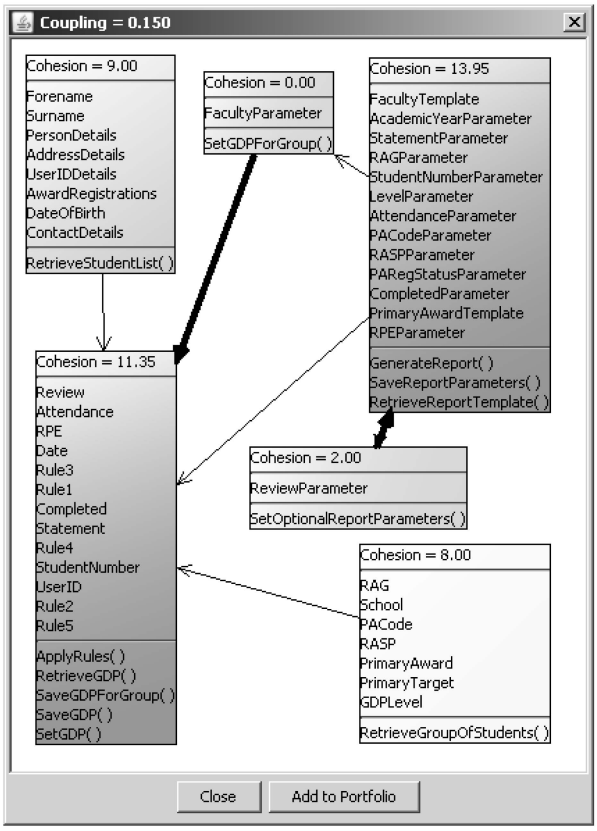

Fig. 21. Sample GDP class design from zone 6.

attributes and methods of the manual design classes "Development" and "Rule." In the manual class design, there exists a high level of coupling between "Development" and "Rule"; that the local search arrives at a class that appears to coalesce these two manual design classes suggests to the designer that a subtle design trade-off is happening at this point. On the one hand, the quantitative measure of coupling suggests that "Development" and "Rule" are coalesced; on the other hand, the qualitative judgment of the human designer is to tolerate inferior coupling in favor of a more fine-grained design solution whereby comprehensibility is preferred.

A sample class design from zone 6 is shown in Fig. 21. With an external coupling value of 0.150, the class design has an inferior coupling when compared with the manual class design. As in zone 5 example designs, classes bearing a strong resemblance to "Group," "Student," and "Report" are evident and, with a cohesion value of 11.35, the local search has arrived at a class that appears to attempt to encapsulate the attributes and methods of the manual design classes "Development" and "Rule." However, when compared with zone 5, this attempt at encapsulation has not been so successful as two small, highly coupled classes with low cohesion values of 0.00 and 2.00 are evident. It would appear that the presence of six classes in the local search rather than five does not help the designer to resolve the strong coupling between "Development" and "Rule." This finding, when taking with an inferior coupling value, suggests that zone 6 might not be so fruitful in discovering useful and interesting class designs as zone 5.

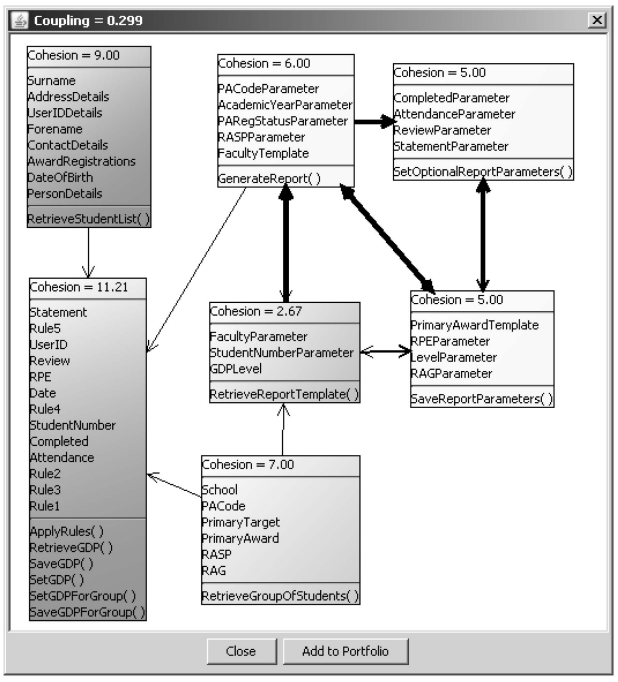

Fig. 22. Sample GDP class design from zone 7.

A sample class design from zone 7 is shown in Fig. 22. The external coupling value of this sample class design is 0.299, which is inferior to the coupling evident in the zone 6 example and much inferior to the coupling value of the manual class design. As in many zone 7 class designs, classes bearing a resemblance to the manual design classes "Group" and "Student" have been arrived at by the local search, although the cohesion value of the "Group" class has diminished from 8.00 in zone 6 to 7.00 in zone 7. However, unlike zone 6, a number of classes with moderate cohesion values (2.67, 5.00, 5.00, and 6.00) are evident. Given that class designs from zone 7 must contain seven classes, it would appear that the elements of the "Report" class have been scattered over these four classes of moderate cohesion. Indeed, the high levels of coupling seen in the wide coupling arrows in Fig. 22 is further evidence to confirm this suspicion. Because of this, the designer may conclude that increasing the number of classes in the design has not been productive; zone 7 classes are too fine-grained with inferior design coupling overall. Local searches of zones 4 and 5 have been more fruitful, yielding more useful and interesting class designs.

5.4 Computational Performance Times

All searches have been performed on a standard desktop PC running Microsoft Windows XP. The time taken for an interactive design scenario is largely dependent on the time taken for designer interaction rather than computational search. For example, in the Graduate Design Program (DGP) design example, examination of the event log trace in Section 5.1 reveals that the multi-objective global search is of four seconds duration, while four local searches of zones 4, 5, 6, and 7 execute concurrently for 10 seconds. Indeed, the event log trace reveals that greater time is taken up by designer interaction, for example, inspecting the visualizations of class design solutions and adding interesting and useful designs to the design portfolio.

6 Limitations and Further Work

In the course of experimentation during an interactive design session, a number of limitations of the approach became apparent. First, the fidelity of the class designs is only as good as the fidelity of the design problem domain description. Thus, great care (as ever) must be taken to faithfully record the design problem. Second, at present, the interactive search approach does not cater to sudden flashes of human designer recognition, of abstract thought, or of analogical transfer of potential design patterns from other design problem domains.

Last, the scale of the example design domains may be small compared to industrial design problem domains. However, the cardinalities of the example search spaces (2.77248×10^{19} for CBS and 1.31668×10^{17} for GDP) are nontrivial and thus of sufficient size and complexity to provide meaningful results. Having said that, experienced software engineers, should they wish to perform class design manually, would likely not evaluate every possible candidate design combination to arrive at a satisfactory design. He/she might use their expertise in analysis and design patterns or perhaps use existing knowledge from a different problem domain by analogical transfer to restrict the initial search to a mental "chunk" of the design problem and arrive at design abstractions and solutions for the chunk before moving on to other parts of the design problem search space. Indeed, we speculate that this manual "divide and conquer" strategy could be exploited by software agents to split the problem space into subsets for parallel evolutionary search. This might be of great benefit in industrial scale design problem domains, and could exploit previously described hierarchical clustering techniques that have been used to divide large-scale software designs into possible software component architectures (e.g., [34]).

Other future work that addresses scaling the approach described in this paper to industrial scale design problem domains includes the use of self-adapting control parameters, enhanced halting of search prior to convergence, and enhancing the interactive design experience. Self-adaptation of control parameters, such as population size, crossover, and mutation probabilities, has been previously shown to make evolutionary search more robust in the face of differing and dynamic search spaces [64], [65]. Even with a consistent representation, different search problems can require considerable a priori empirical effort to "tune" individual control parameter values, and further effort can then be required when the effects of combining individual control parameter values are investigated. Generally, given the wide range of possibilities among the combinations of control values, it may not even be possible to reliably determine optimum values. Self-adaptive control values, on the other hand, typically embed evolutionary control parameters within the individual's genome and evolve them. It seems highly feasible that self-adaptive control parameters could enable the search approach described in this paper to effectively and efficiently address not only a wide variety of design problem domains but also a range of design problem scale, without the need to

expend effort fine tuning the combinations of control parameters empirically for each design problem domain. Enhancing the halting search tactics of isolation agents by Pareto-optimal approaches may also assist the scalability of the approach, particularly if the sampling frequency and number of software agent evaluations could be effectively reduced. The issue of user fatigue is, of course, crucial to the interactive search of large-scale search spaces; inspecting entire populations of design visualizations is not desirable. User interaction could therefore be enhanced by building on the previous work on Parmee and others [26], [28], [29], employing software agents that further reduce the monitoring and filtering load on the designer and also employing machine-learned preferences and heuristics to proactively select interesting design visualizations for presentation to the designer. In addition, the authors have recently conducted an empirical investigation of the approach with collaborators in an industrial software development organization in order to evaluate both the usability and scalability of the approach [66]. It is interesting to note that significant opportunistic moments of sudden design discovery ("aha!" moments) have been observed during the industrial design episode. The authors look forward to using the feedback thus gained from the empirical investigation to further enhance the interactivity and scalability of the approach.

7 Conclusions

Results from the experimental design episode show that interactive, evolutionary search appears to be a highly promising basis for computationally intelligent tool support in upstream software design. The upstream class designs arrived at are inherently traceable to the design problem domain and the population-based evolutionary search presents a mass of useful and interesting class designs without the need for initial design guesses from the designer. When visualized by the designer, colorful UML software designs show quantitative measures of structural integrity, i.e., design coupling and class cohesion. Due to exploitation of concurrency, the computational performance of the software agents underpinning the interactive search appears more than adequate. However, the performance of an interactive episode depends on the human designer too—we envisage that a period of time is necessary for the designer to become familiar with the computational search approach and the interactive support tool. We also conclude that an open systems approach is highly appropriate for search-based computational tool support wherein both human designers and software agents are able to express their preferences to jointly steer the search to useful and interesting upstream software designs.

References

[1] F.P. Brooks Jr., *The Mythical Man Month,* 20th anniversary ed. Addison-Wesley, 1995.

[2] R. Guindon, "Designing the Design Process: Exploiting Opportunistic Thoughts," *Human-Computer Interaction,* vol. 5, nos. 2-3, pp. 305-344, 1990.

[3] C. Zannier, M. Chiasson, and F. Maurer, "A Model of Design Decision Making Based on Empirical Results of Interviews with Software Designers," *Information and Software Technology,* vol. 49, no. 6, pp. 637-653, June 2007.

[4] R.L. Glass, *Facts and Fallacies of Software Engineering,* pp. 81-84. Addison-Wesley, 2003.

[5] D. Svetinovic, D.M. Barry, and M. Godfrey, "Concept Identification in Object-Oriented Domain Analysis: Why Some Students Just Don't Get It," *Proc. 13th Ann. IEEE Int'l Conf. Requirements Eng.,* pp. 189-198, 2005.

[6] R.J. Wirfs-Brock and A. McKean, *Object Design: Roles, Responsibilities, and Collaborations.* Addison-Wesley, 2003.

[7] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2004.

[8] D. Damian, J. Chisan, L. Vaidyanathasamy, and Y. Pal, "Requirements Engineering and Downstream Software Development: Findings from a Case Study," *Empirical Software Eng.,* vol. 10, no. 3, pp. 255-283, 2005.

[9] A. Bhagwat, *Object-Oriented Analysis and Design: Tools,* http://www.cetus-links.org/oo_ooa_ood_tools.html, 2003.

[10] *IBM Rational Software Architect,* http://www-01.ibm.com/software/rational/, 2009.

[11] *Enterprise Architect,* www.sparxsystems.com.au/products/ea/index.html, 2009.

[12] *Select Architect,* http://www.selectbs.com/adt/analysis-and-design/select-architect, 2009.

[13] G.S. Ananda Mala and G.V. Uma, "Automatic Construction of Object-Oriented Design Models [UML Diagrams] from Natural Language Requirements Specification," *Proc. Ninth Pacific Rim Int'l Conf. Artificial Intelligence,* pp. 1155-1159, 2006.

[14] D. Lui, K. Subramaniam, E. Eberlien, and H. Behrouz, "Natural Language Requirements Analysis and Class Model Generation Using UCDA," *Proc. 17th Int'l Conf. Industrial and Eng. Application of Artificial Intelligence and Expert Systems,* pp. 295-304, 2004.

[15] L. Mich and R. Garigliano, "NL-OOPS: A Requirements Analysis Tool Based on Natural Language Processing," *Proc. Third Int'l Conf. Data Mining,* pp. 321-330, 2002.

[16] I.C. Parmee, "Towards Interactive Evolutionary Design Systems," *Proc. Evolutionary and Adaptive Computing in Eng. Design,* pp. 205-232, 2001.

[17] I.C. Parmee, D. Cvetkovic, C. Bonham, and A.H. Watson, "Interactive Evolutionary Conceptual Design Systems," *Proc. Sixth Int'l Conf. Artificial Intelligence in Design,* pp. 249-268, 2002.

[18] I.C. Parmee and J.A. Abraham, "User-Centric Evolutionary Design," *Proc. Eighth Int'l Design Conf.,* pp. 1441-1446, 2004.

[19] J. Rech, E. Ras, and B. Becker, "Intelligent Assistance in German Software Development: A Survey," *IEEE Software,* vol. 24, no. 4, pp. 72-79, Aug. 2007.

[20] R. Milner, "Computing and Communication," *Interactive Computation: The New Paradigm,* D. Goldin, S.A. Smolka, and P. Wegner, eds., pp. 1-8, Springer-Verlag, 2006.

[21] F. Arbab, "Computing and Interaction," *Interactive Computation: The New Paradigm,* D. Goldin, S.A. Smolka, and P. Wegner, eds., pp. 9-23, Springer-Verlag, 2006.

[22] D. Goldin and P. Wegner, "Principles on Interactive Computation," *Interactive Computation: The New Paradigm,* D. Goldin, S.A. Smolka, and P. Wegner, eds., pp. 25-37, Springer-Verlag, 2006.

[23] I.C. Parmee, *Evolutionary and Adaptive Computing in Engineering Design.* Springer-Verlag, 2001.

[24] I.C. Parmee, "Improving Problem Definition through Interactive Evolutionary Computing," *Artificial Intelligence for Eng. Design, Analysis, and Manufacturing,* vol. 16, no. 3, pp. 185-202, June 2002.

[25] D. Cvetkovic and I.C. Parmee, "Agent-Based Support within an Interactive Evolutionary Design System," *Artificial Intelligence for Eng. Design, Analysis, and Manufacturing,* vol. 16, no. 5, pp. 331-342, Nov. 2002.

[26] I.C. Parmee, "Human-Centric Intelligent Systems for Exploration and Knowledge Discovery," *Analyst,* vol. 130, pp. 29-34, 2005.

[27] B. Sharma, I.C. Parmee, M. Whittaker, and A. Sedwell, "Drug Discovery: Exploring the Utility of Cluster Oriented Genetic Algorithms in Virtual Library Design," *Proc. IEEE Congress Evolutionary Computation,* pp. 668-675, 2005.

[28] A.T. Machwe and I.C. Parmee, "Integrating Aesthetic Criteria with Evolutionary Processes in Complex, Free-Form Design—An Initial Investigation," *Proc. IEEE Congress Evolutionary Computation,* pp. 165-172, 2006.

[29] A.T. Machwe and I.C. Parmee, "Introducing Machine Learning within an Interactive Evolutionary Design Environment," *Proc. Ninth Int'l Design Conf.,* pp. 283-290, 2006.

[30] M. Harman and B. Jones, "Search-Based Software Engineering," *Information and Software Technology,* vol. 43, no. 14, pp. 833-839, Dec. 2001.
[31] Y. Zhang, *Repository of Publications on Search Based Software Engineering,* http://www.sebase.org/sbse/publications/, 2008.
[32] M. Harman and L. Tratt, "Pareto Optimal Search-Based Refactoring at the Design Level," *Proc. Genetic and Evolutionary Computation Conf.,* pp. 1106-1113, 2007.
[33] M. O'Keefe and M.O. Cinnaide, "Search-Based Refactoring for Software Maintenance," *J. Systems and Software,* vol. 81, no. 4, pp. 502-516, Apr. 2008.
[34] O. Maqbool and H.A. Babri, "Hierarchical Clustering for Software Architecture Recovery," *IEEE Trans. Software Eng.,* vol. 33, no. 11, pp. 759-780, Nov. 2007.
[35] K. Beck, *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.
[36] M. Fowler, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
[37] S.-C. Lo and J.-H. Chang, "Application of Clustering Techniques to Software Component Architecture Design," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 14, no. 4, pp. 429-439, Aug. 2004.
[38] A. Egyed and D.S. Wile, "Support for Managing Design Time Decisions," *IEEE Trans. Software Eng.,* vol. 32, no. 5, pp. 299-314, May 2006.
[39] P. Maes, "Agents that Reduce Work and Information Overload," *Comm. ACM,* vol. 37, no. 7, pp. 31-40, Jul. 1994.
[40] N. Negroponte, *Being Digital.* Hodder and Stoughton, 1995.
[41] M. Wooldridge, *An Introduction to Multi-Agent Systems,* Wiley, 2002.
[42] J.M. Bradshaw, "Making Agents Acceptable to People," *Proc. Third Int'l/Central and Eastern European Conf. Multi-Agent Systems,* pp. 1-3, 2003.
[43] G. Klein, D.D. Woods, J.M. Bradshaw, R.R. Hoffman, and P.J. Feltovich, "Ten Challenges for Making Automation a 'Team Player' in Joint Human-Agent Activity," *IEEE Intelligent Systems,* vol. 19, no. 6, pp. 91-95, Nov./Dec. 2004.
[44] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach,* Addison-Wesley, 1992.
[45] C.L. Simons and I.C. Parmee, "A Cross-Disciplinary Technology Transfer for Search-Based Evolutionary Computing: From Engineering Design to Software Engineering Design," *Eng. Optimization,* vol. 39, no. 5, pp. 631-648, 2007.
[46] R. Harrison, S. Councell, and R. Nithi, "An Investigation into the Applicability and Validity of Object-Oriented Design Metrics," *Empirical Software Eng.,* vol. 3, no. 3, pp. 255-273, Sept. 1998.
[47] L.C. Briand, J.W. Daly, and J.K. Wust, "A Unified Framework for Coupling Measurements in Object-Oriented Systems," *IEEE Trans. Software Eng.,* vol. 25, no. 1, pp. 91-121, Jan. 1999.
[48] C.L. Simons, *Use Case Specifications for Cinema Booking System,* http://www.cems.uwe.ac.uk/~clsimons/CaseStudies/CinemaBookingSystem.htm, 2009.
[49] C.L. Simons, *Use Case Specifications for Graduate Development Program,* http://www.cems.uwe.ac.uk/~clsimons/CaseStudies/GraduateDevelopmentProgram.htm, 2009.
[50] B. Lawson, *What Designers Know,* pp. 17-18. Architectural Press (Elsevier), 2004.
[51] C.L. Simons and I.C. Parmee, "User-Centered, Evolutionary Search in Conceptual Software Design," *Proc. IEEE Congress Evolutionary Computation at IEEE World Congress Computational Intelligence,* pp. 869-876, 2008.
[52] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms.* John Wiley and Sons, 2001.
[53] C.A. Coello Coello, G.B. Lamont, and D.A. Van VelderHuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems,* second ed., Springer, 2007.
[54] C.L. Simons, *Example Event Log for Design Episode,* http://www.bit.uwe.ac.uk/~clsimons/CaseStudies/ExampleEventLog.htm, 2008.
[55] H. Tagaki, "Interactive Evolutionary Computation: Fusion of the Capabilities of EC Optimization and Human Evaluation," *Proc. IEEE,* vol. 89, no. 9, pp. 1275-1296, Sept. 2001.
[56] M. O'Keefe and M.O. Cinneide, "Towards Automated Design Improvement through Combinatorial Optimisation," *Proc. 26th Int'l Conf. Software Eng. and Workshop Directions in Software Eng. Environments,* pp. 75-82, 2004.
[57] S.-J. Huang, N.-H. Chiu, and L.-W. Chen, "Integration of the Grey Relation Analysis with Genetic Algorithm for Software Effort Estimation," *European J. Operational Research,* vol. 188, no. 3, pp. 898-909, 2008.
[58] I. Hermadi and M.A. Ahmed, "Genetic Algorithm Based Test Data Generator," *Proc. IEEE Congress on Evolutionary Computation,* pp. 85-91, 2003.
[59] R.C. Bryce and C.J. Colburne, "The Density Algorithm for Pairwise Interaction Testing," *Software Testing, Verification and Reliability,* vol. 17, no. 3, pp. 159-182, 2007.
[60] T.M. Khoshgoftar and Y. Liu, "A Multi-Objective Software Quality Classification Model Using Genetic Programming," *IEEE Trans. Reliability,* vol. 56, no. 2, pp. 237-245, June 2007.
[61] J.J. Durillo, Y. Zhang, E. Alba, A.J. Nebro, "A Study of the Multi-Objective Next Release Problem," *Proc. First Int'l Symp. Search-Based Software Eng.,* pp. 49-58, 2009.
[62] K. Lakhotia, M. Harman, P. McMinn, "A Multi-Objective Approach to Search-Based Test Data Generation," *Proc. Genetic and Evolutionary Computation Conf.,* pp. 1098-1105, 2007.
[63] Y. Zhang, M. Harman, and S.A. Mansouri, "The Multi-Objective Next Release Problem," *Proc. Genetic and Evolutionary Computation Conf.,* pp. 1129-1137, 2007.
[64] A.E. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter Control in Evolutionary Algorithms," *IEEE Trans. Evolutionary Computation,* vol. 3, no. 2, pp. 124-141, July 1999.
[65] *Parameter Setting in Evolutionary Algorithms,* F.G. Lobo, C.F. Lima, and Z. Michalewicz, eds. Springer, 2007.
[66] C.L. Simons and I.C. Parmee, "An Empirical Investigation of Search-Based Computational Support for Conceptual Software Engineering Design," *Proc. IEEE Int'l Conf. Systems, Man, and Cybernetics,* pp. 2577-2582, 2009.

Christopher L. Simons received the BSc degree from the University of Aston in Birmingham in 1978 and the MSc degree from Bristol Polytechnic in 1989. He practiced software engineering over many industrial software design domains before joining the University of the West of England (UWE), United Kingdom. His research interests include intelligent computational support for software engineers during development. He is a member of the British Computer Society.

Ian C. Parmee received the BSc degree in civil engineering in 1985 and the PhD degree in 1990. He practiced civil engineering before becoming a reader at the EPSRC Engineering Design Centre, the University of Plymouth, United Kingdom. He is now a professor in the Department of Computer Science at the University of the West of England (UWE), United Kingdom. He is associate editor of both the *IEEE Transactions on Evolutionary Computation* and *IEEE Transactions on Engineering Management*. He achieved Best Overall Paper at the 2004 IEEE Congress on Evolutionary Computation. He is a member of the United Kingdom EPSRC Peer Review College, the United Kingdom Engineering Council, and the United Kingdom Energy Institute. His research interests include user-centered, computationally-intelligent design, and decision support systems.

Rhys Gwynllyw received the BSc degree from the University of Wales, Bangor, in 1985, the MSc degree in numerical analysis and mathematical modeling fromt the University of Oxford in 1986, and the PhD degree in computational fluid dynamics from the University of Bristol in 1991. He practiced as a software engineer in telecommunication systems before joining the University of the West of England (UWE). He is a member of the Institute of Mathematics and Its Applications.