2014 IEEE International Conference on Systems, Man, and Cybernetics
October 5-8, 2014, San Diego, CA, USA

Using Evolutionary Computation to Shed Light on the Effect of Scale and Complexity on Object-Oriented Software Design

Christopher L. Simons
Department of Computer Science & Creative Technology
University of the West of England
Bristol BS16 1QY United Kingdom
chris.simons@uwe.ac.uk

Jim Smith
Department of Computer Science & Creative Technology
University of the West of England
Bristol BS16 1QY United Kingdom
james.smith@uwe.ac.uk

***Abstract*—Early lifecycle software design is an intensely human activity in which design scale and complexity can place a high cognitive load on the software designer. Recently, the use of evolutionary search has been suggested to yield insights in the nature of software engineering problems generally, and so we have applied dynamic evolutionary computation using self-adaptive mutation to the object-oriented software design search space. Using three design problem instances of varying scale and complexity, initial investigations of the discrete search landscape reveal a redundancy in genotype-to-phenotype mapping enabling flexible and effective exploration. In further experiments, mutation probabilities and population diversity are observed to significantly increase in the face of increasing problem scale, but not for increasing complexity (in problems of the same scale). Based on these findings, we conclude that design problem scale rather than complexity has an effect on the software design process, emphasizing the role of decomposition as a design technique.**

Keywords—software design; evolutionary computation.

I. INTRODUCTION

Early lifecycle software design is an intensely human activity in which relevant concepts and information relating to a design problem are identified. In the widely used object-oriented design paradigm of software development, such concepts and information are often represented as classes comprised of attributes and methods. However, there is evidence to suggest that early lifecycle class modelling is non-trivial and difficult to perform, not least due to the scale and complexity of design problem domains. For many class models, numbers of attributes and methods can run to hundreds with a corresponding multiplicity of classes. Petre [1] has suggested that software design problems are often wicked: too big, too ill-defined and too complex for easy comprehension and solution. Glass [2] goes further to suggest that the scale and complexities of some software designs may be beyond human comprehension. Such evidence is consistent with the notion, expressed by Brooks in his seminal paper "No Silver Bullet" [3], that software design and engineering problems exhibit an 'essential complexity' that cannot be reduced. Indeed, Brooks advocates that the way forward for the field is to address this essential complexity in software design and development. Fiadeiro [4] goes further to suggest that *"software is haunted by the beast of complexity and doomed to live in permanent crisis"*. However, Fiadeiro also notes a general trend to *"shift complexity to a place where it can be managed more effectively"*. Consistent with previous authors (e.g. [5]), Fiadeiro suggests two design techniques to help manage complexity: abstraction and decomposition. Use of appropriate design abstractions, e.g. objects and classes, enables effective modelling of the design solution. Decomposition, or 'divide and conquer', enables the breakdown of a complex design into smaller, more manageable component parts.

Interestingly, there is also evidence from other fields e.g. building architecture design that the scale and complexity of design problems impacts the quality of design solutions. Flager et al. [6] report the results of a recent experiment to measure human abilities to solve building design problems parameterized by scale (number of design variables) and design coupling (interactions between variables). They report an exponential decrease in design solution quality as scale increases, although the adverse effect of coupling becomes less significant as the scale of the problem increases.

One way in which insight may be gained into the effect of scale and complexity in software design problems is to apply evolutionary search to various software design search spaces and examine the parameters that control search algorithms e.g. mutation probability. In the course of making the application of evolutionary algorithms effective and efficient, it is often the case that control parameters must be 'tuned' by empirical investigation. However, there are drawbacks to parameter tuning. De Jong [7] observes that with each parameter having a wide range of possible values, there is a corresponding explosion of parameter value combinations due to all possible interactions. Eiben et al. [8] go further and suggest that evolutionary algorithms are inherently dynamic, adaptive processes, and that *"static parameters go against this spirit and different values of parameters may be optimal for different stages of the evolutionary process"*.

Building on this, we hypothesize that employing a dynamic approach to parameter control might yield useful insights into the nature of the object-oriented software design search space with respect to scale and complexity. Specifically, we hypothesize that observing the behaviours of self-adaptive mutation (as used for example by Evolutionary Strategies) will

978-1-4799-3840-7/14/$31.00 ©2014 IEEE

yield significant insights into the effect of scale and complexity the software design search space, which in turn will to help us to better understand a process that software engineers find non-trivial and demanding. Therefore this paper addresses the following research questions:

RQ1. What insights can be gained by the use of dynamic parameter control in the evolutionary search of object-oriented software design search spaces?

RQ2. What are the implications of any insights gained with respect to the difficulty of software design?

To answer these questions, the remainder of the paper is organized as follows. Section II provides a brief context of relevant work in this arena. Section III describes the representation, fitness measures and evolutionary algorithm used. Section IV details the experimental methodology, while Section V presents and analyses the results obtained. To conclude the paper, Section VI summarizes the findings and assesses their implications for object-oriented software design.

II. BACKGROUND

Within the evolutionary computing community, there is an increasing recognition that a dynamic and adaptive approach to search is useful for robust and effective application e.g. [7]. A comprehensive review of dynamic parameter control in evolutionary algorithms generally is provided by Meyer-Neiberg and Beyer [9]. However, one example of an investigation into dynamic parameter control specifically in local search of object-oriented software design can be found at [10]. Building on its findings, self-adaptive mutation probability is used as a starting point for investigations in this paper.

Software designers have long recognized the impact of scale and complexity in software design [3]. For example, Glass [2] suggests that for every 25% increase in problem complexity, there is a 100% increase in solution complexity. Within the object-oriented paradigm, design scale relates to the number of atomic elements i.e. the number of attributes and methods. Software design complexity is measured in different ways across various modelling paradigms, but generally relates to the number of dependencies and interconnections that couple design elements [4]. The greater the number of interconnections, the greater the complexity. As the number of couples between elements is driven by the number of classes in an object-oriented design, we take the number of classes to be an indicator of design complexity. Thus, for a design of a given scale, we consider the higher the number of classes, the greater the design complexity.

III. REPRESENTATION AND ALGORITHM

A. Representation

The representation of object-oriented software designs is discrete and comprises classes, methods and attributes, consistent with the Unified Modeling Language (UML) [11]. Classes are represented as groupings (or placeholders) of methods and attributes, although, of course, there are many ways in which methods and attributes may be grouped into classes. Thus a design solution individual is encoded by a specific assignment of attributes and methods to classes.

The design problem is described by use cases (e.g. [11]), which capture scenarios of interaction between user and the software system-to-be. Within use cases, the steps of the scenarios, and in particular the actions (verbs) and data (nouns) contained in each step, are recorded. If an action and datum are co-located in the same step of the narrative text, the action is said to 'use' the datum. The sets of actions, data and 'uses' thus define the design problem. A set of solution attributes $a_1 .. a_n$ is derived directly from members of the set of data specified in the design problem, while a set of methods $m_1 .. m_n$ are derived from the set of actions, explicitly providing traceability from the design problem to the solution. Each method and each attribute is assigned to exactly one class. We also impose the constraint that each class comprises at least one attribute and one method, to ensure that each class maps to meaningful concepts and information in the design problem domain. Further details of the representation can be found at [12].

B. Measures of Software Design Quality

Three fitness measures are used in search. The first relates to coupling between classes, and the second to the 'elegance' of the software design. The third is an equal combination of them.

The first fitness measure is *coupling between objects* (CBO), a measure inspired by previous work of Harrison et al. [13]. The CBO measure is based on the notion that if a method uses an attribute of the same class, this promotes the internal cohesion of the class. Conversely, if a method uses an attribute located in a different class, a couple exists between the two classes. As described above, the design problem is specified by a number of use cases, from which solution attributes and methods are derived. Drawing on the use case documentation defining the software problem instance, a matrix U is constructed based on methods using attributes such that:

$$U_{ij} = \begin{cases} 1, & i \le a,\ a < j \le a+m,\ \text{method } j \text{ uses attribute } i \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where the elements are numbered from 1 to a (attributes) and $a+1$ to $a+m$ (methods). The CBO related fitness (to be minimized) is then defined as the proportion of all uses that are external or "out of class" within a given design:

$$f_{CBO} = 100 * \frac{\sum_i \sum_{j,class(j) \neq class(i)} U_{ij}}{\sum_i \sum_j U_{ij}} \quad (2)$$

The second fitness measure, which has been shown to correlate well with designers' recorded feelings of design "elegance", exploits design symmetry, is *Numbers Among Classes* (NAC). NAC is calculated as the arithmetic mean of the standard deviation of the numbers of attributes and methods among the classes of the design [12]. Values of NAC are truncated to the range [0,6] and the fitness to be minimized is calculated as a percentage:

$$f_{NAC} = \frac{100}{6} * \left(\frac{\sigma_a}{2} + \frac{\sigma_m}{2}\right) \quad (3)$$

where σ_a and σ_m denote the standard deviation of attributes and methods among classes of the design respectively. Lastly,

to reflect both measures equally in search, we combine the two as follows:

$$f_{comb} = 0.5 * (f_{CBO} + f_{NAC}) \quad (4)$$

C. Search Algorithm

The EA chosen is an elitist evolutionary algorithm inspired by a local search approach reported earlier [10]. Parents are chosen by deterministic binary tournaments, and the offspring replaces the least fit member of the population. Because crossover has been found previously to be complex and computationally expensive [10], mutation alone is used for diversity creation. To enable self-adaptive mutation, each design solution encodes its own mutation probability which is modified thus:

$$p_m' = p_m \,.\, \mathrm{N}(1, \sigma) \quad (5)$$

where σ, the standard deviation, is set at an empirically determined value of 0.1. The mutation probability determines whether mutation takes place rather than any notion of mutation strength, and is also bounded to lie in the range [0,1].

```
PROCEDURE: EvolutionaryAlgorithm
BEGIN
   initialize POP_SIZE solutions at random
   evaluation = 0
   generation = 0
   WHILE( evaluation < MAX_EVALUATIONS )
        FOR EACH( solution )
            evaluate f_CBO, f_NAC, f_comb
            evaluation = evaluation + 1
        END FOR
        select POP_SIZE parents using tournament
        FOR EACH( parent )
            offspring = AttemptMutation( parent )
            IF( result is success )
                replace one individual in population
                with offspring using 'Delete-Worst'
            END IF
        END IF
        generation = generation + 1
    END WHILE
END PROCEDURE
```

Mutation involves relocating attributes and/or methods among classes. In rotation, attributes and methods are moved and swapped from one to another, as follows:

```
PROCEDURE: AttemptMutation( parent )
BEGIN
   result = failure
   update parent probability [see eq. 5]
   IF( parent probability < random( 0.0, 1.0 ) )
       END PROCEDURE
   END IF
   attemptCount = 0
   WHILE( attemptCount < MAX_ATTEMPTS )
        clone an offspring individual from parent
        mutate offspring by selected mechanism
        evaluate f_CBO, f_NAC, f_comb of mutated offspring
        IF( offspring is no worse than parent )
            result = success
            END PROCEDURE
        ELSE
             attemptCount = attemptCount + 1
        END IF
   END WHILE
END PROCEDURE
```

IV. METHODOLOGY

A. Design Problem Instances

Three instances of software design problem domains have been used spanning a range of scale and complexity. Details of the three problem instances are available at 0. The first is a generalized abstraction of a Cinema Booking System (CBS), while the second is a university student information system for recording graduate development programs (GDP) and was implemented at the University of the West of England, UK, in 2008. The third is based on an industrial case study for booking cruise holidays on tall sailing ships, Select Cruises (SC). Information relating to the numbers of attributes, methods and uses for each problem instance is shown in Table 1. To facilitate easy comparison with the results produced by search, we also show the numbers of classes and values for f_{CBO} and f_{NAC} for manually produced software designs.

TABLE I. PROBLEM INSTANCE INFORMATION

	Att^s	Met^s	Uses	Classes	f_{CBO}	f_{NAC}
CBS	16	15	39	5	15.4	8.21
GDP	43	12	121	5	29.7	25.92
SC	62	30	126	16	45.2	15.2

B. Algorithm Parameters

Experiments were run with 100 individuals in the EA population. Initial mutation probabilities are set at random in the range 0.01 to 0.1 in line with previous findings [10]. The EA was run 50 times for each specific problem instance, with each run allowed to make 100,000 calls to the evaluation function.

C. Experimental Design

Three Experiments were conducted as follows to investigate:

1. Baseline EA performance;
2. RQ 1: effect on behaviour of increasing scale;
3. RQ 2: effect on behaviour of increasing complexity.

Firstly, to enable comparison with f_{CBO} and f_{NAC} values obtained for manual designs, the number of classes for software designs for CBS, GDP and SC were fixed at 5, 5 and 16 respectively. As well as average population values for fitness (f_{CBO},f_{NAC} and f_{comb}), average population values for self-adaptation (mutation probabilities and number of mutation attempts) were recorded. Initial empirical investigations quickly revealed that optimization for f_{NAC} was highly effective – optimizing symmetrical elegance did not appear difficult for the EA to achieve – and so focus shifted to f_{CBO} and f_{comb} as key indicators of EA fitness performance. As described later in the Results section, it emerged from initial studies that measures of population diversity are also necessary to evaluate the EA landscape, and so the number of phenotypically unique solutions and groupings have also been recorded.

Secondly, EA behaviour in the face of increasing scale was investigated by comparing performance for problem instances of increasing scale i.e. CBS, GDP and SC and fixing complexity with the number of classes at 5 for all three. Table II shows the measurements recorded.

TABLE II. EXPERIMENTAL MEASUREMENTS RECORDED

Fitness	• Mean Best Fitness for f_{comb} (Bestf_{comb}) • When Best F_{comb} found (WhenBestf_{comb})
Self-adaption	• Mean Maximum Probability (MaxProb) • When Mean Maximum Probability found (WhenMaxProb) • Mean Maximum Number of Mutation Attempts (MaxAttempts) • When Mean Maximum Number of Mutation Attempts Found (WhenMaxAttempts)
Diversity	• Mean Maximum Number of Phenotypical Uniques (MaxUniques) • When Mean Maximum Number of Phenotypical Uniques Found (WhenMaxUniques) • Mean Maximum Number of Phenotypical Groupings (MaxGroupings) • When Mean Maximum Number of Phenotypical Groupings Found (WhenMaxGroupings)

Thirdly, EA behaviour in the face of increasing complexity was investigated by using the largest scale problem instance, SC, but varying the number of classes. Consistent with the previous experiments, a lower bound on the number of classes was fixed at 5 while the upper bound remained at 16. The problem instances are denoted SC05, SC06, ..., SC16. Data recorded are the same as the second experiment. For the second and third experiments, IBM SPSS Statistics Package v20 was used to carry out a one-way Analysis of Variance with fitness, self-adaption and diversity measures as dependent variables and fixed factors problem instance. This was followed by post-hoc testing for significant differences using Tukey's Honestly Significant Difference (HSD) test. Where significant differences are reported, they are at the $p < .01$ level.

V. RESULTS

A. Empirical Evaluation

Initial empirical evaluation of the EA landscape using fCBO revealed that values achieved for the minimization of f_{CBO} compared favourably with those of the manual designs i.e. 15.0% with 15.4% for CBS, 32.2% with 29.7% for GDP, and 44.8% with 45.2% for SC. Superior fitness values are achieved in approximately 100 generations. Values achieved for f_{comb} shown in figures 1 reveal a similar pattern of fitness curves for the three problem instances, although the combination of F_{NAC} (for design elegance) with F_{CBO} for f_{comb} gives the lowest fitness for GDP after 10 generations. Figures 2 show that self-adaptation of mutation probabilities appears effective. Average population mutation probability values climb for the first 40 or so generations to promote exploration of the search space but thereafter decline to focus search on promising regions of the discrete search space. Consistent with this, figures 3 shows that the population average number of mutation attempts also climb in the first 40 or so generations but plateau thereafter. Taken together, the results shown in figures 1 to 3 suggest that the evolutionary algorithm performance is effective (i.e. achieves fitness values comparable to manual designs for f_{CBO}) and efficient (i.e. superior fitness values are achieved in a reasonable number of generations). However, to examine the characteristics of the search landscape further, a plot of F_{CBO} against F_{NAC} after convergence is shown in figure 4.

Figure 4 shows that the discrete nature of the object-based genotype encoding has a strong influence on the phenotypical landscape, especially with respect to f_{CBO}. This is explained by the redundancy in the genotype-to-phenotype mapping in the object-based discrete representation. For example, two (or more) classes in different design may contain the same attributes and methods but encoded in a different order. However, the ordering of methods and attributes in classes has no effect on the calculation of F_{CBO}. To investigate this effect on population diversity, the number of phenotypically unique individuals and number of groupings of phenotypically equivalent individuals were recorded and the results shown in figures 5 and 6. Figures 5 and 6 would initially appear to indicate that values for these phenotypically-based measures do indeed vary with problem instance, and so these measures of population diversity were recorded in subsequent experiments on the effect of increasing scale and diversity. Results are given in the following section.

B. Increasing Scale

TABLE III. RESULTS OF ANALYSIS OF VARIANCE AND TUKEY'S HSD TEST ON MEAN VALUES USING F_{COMB}. BOLD FONT INDICATES SIGNIFICANT DIFFERENCES.

	CBS	GDP	SC05
Bestf_{comb}	**0.4297**	**0.4651**	**0.2478**
WhenBestf_{comb}	44.22	126.32	**221.92**
MaxProb	**0.7018**	**0.8771**	**0.8960**
WhenMaxProb	**166.94**	58.76	51.68
MaxAttempts	**47.208**	**43.73**	**49.97**
WhenMaxAttempts	404.72	**621.68**	346.90
MaxUniques	**48.26**	**78.96**	**85.78**
WhenMaxUniques	33.88	47.02	39.32
MaxGroupings	**22.54**	27.03	27.08
WhenMaxGroupings	**40.66**	57.76	67.08

With increasing scale of the three problem instances, the results of analysis shown in table III reveal significantly different mean values for Bestf_{comb}, MaxProb, MaxAttempts and MaxUniques. Interestingly, an increase in MaxProb would seem to reflect effective self-adaptation with increasing scale, and consistent with this in an increase in MaxUniques. It is important to recall that the mutation procedure repeats until an offspring individual is of better or equal fitness compared to the parent. Because of this, an offspring individual of different genotype but similar (phenotypical) fitness to the parent may be arrived at. The redundancy of representation appears to promote diversity as the values of MaxUniques increase in the face of increasing scale. The results appear to indicate that the evolutionary search adapts effectively with increasing scale.

A. Increasing Complexity

With increasing complexity (i.e. numbers of classes) for a fixed scale of problem instance, all measurements show no statistically significant difference between problem instances.

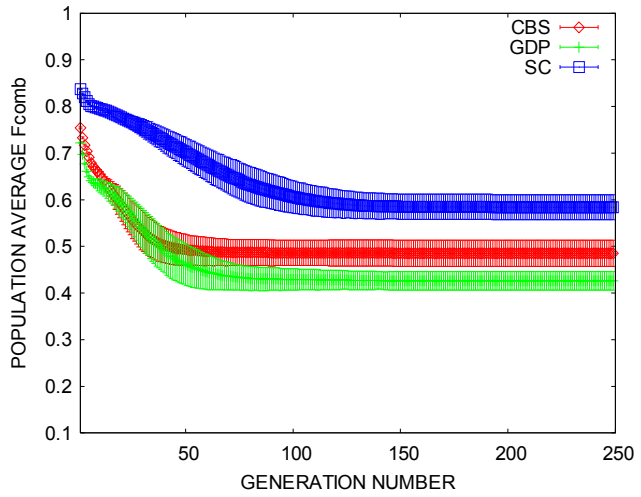

Fig. 1. Average population f_{comb} fitness values

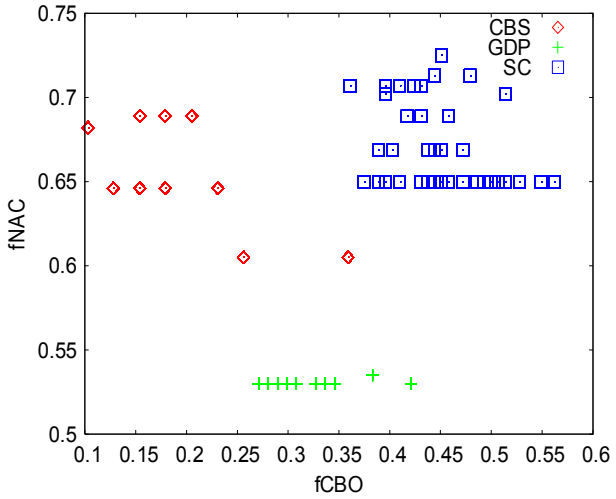

Fig. 4. Plots of f_{CBO} and f_{NAC} after convergence

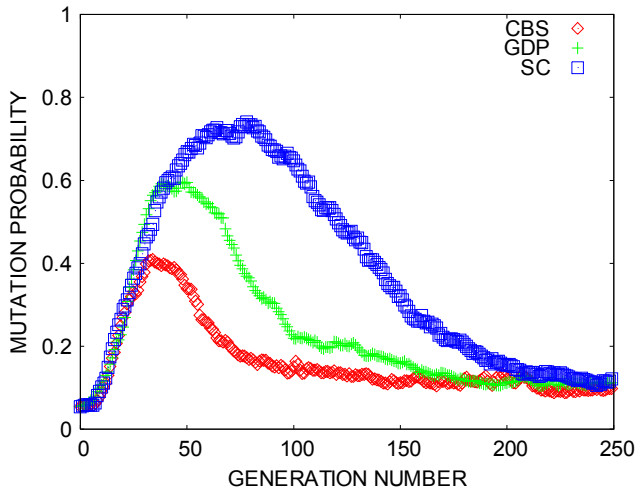

Fig. 2. Average population mutation probability values using f_{comb}

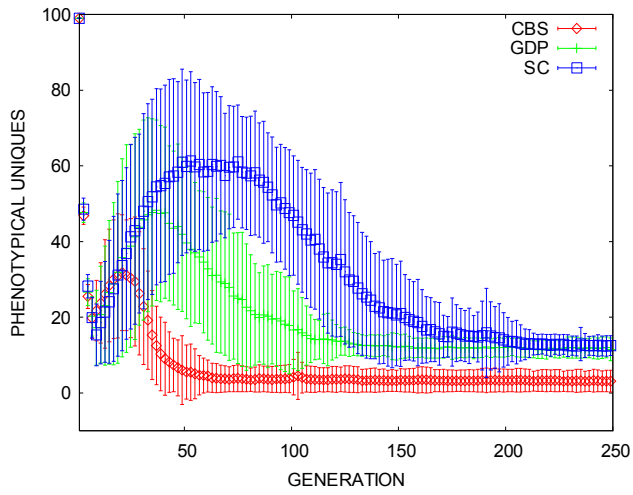

Fig. 5. Mean number of population phenotypical uniques using f_{comb}

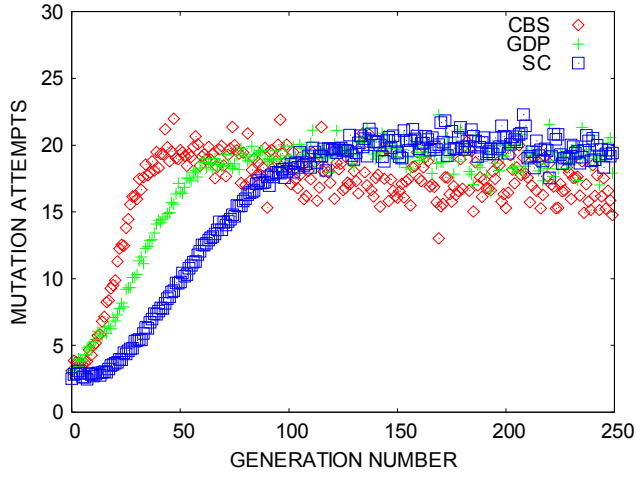

Fig. 3. Average population mutation attempt count values using f_{comb}

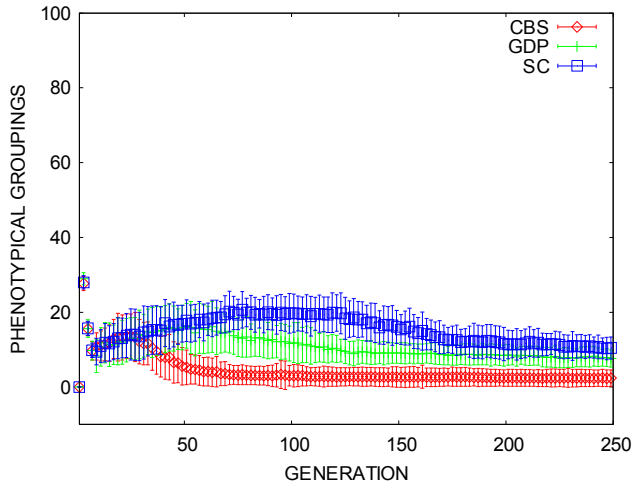

Fig. 6. Mean number of population phenotypical groupings using f_{comb}

However, the results for Mean Bestf_{comb} values do provide some insight into the relationship between the number of classes in a software design and coupling between classes. Given the constraint of the discrete representation that each class must contain at least one attribute and one method, increasing the number of classes leads to an increase in coupling, and this is shown in the search results and illustrated in figure 7. It is interesting, however, to speculate that although the results of evolutionary search show this clearly, this assumes an even spread of couples among the classes. In practice, software designers take great pains to manage the distribution of coupling in a design with design principles and patterns.

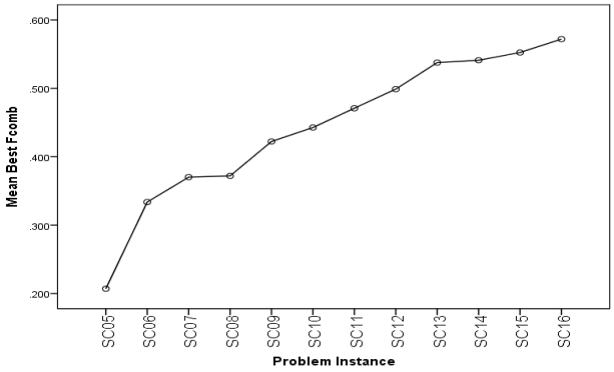

Fig. 7. Mean best f_{comb} with increasing numbers of classes.

B. Analysis of Results

Initial evaluation of the evolutionary algorithm using adaptive self-mutation reveals a robust algorithm that appears effective (e.g. fitness values for f_{CBO} are comparable to those produced manually) and efficient (e.g. superior fitness values are arrived at in a reasonable number of generations). It is clear that results obtained for increasing scale are different to those for increasing complexity. In the face of increasing scale, maximum mutation probabilities and the number of phenotypically unique individuals increase significantly, suggesting that the evolutionary algorithm is adapting to increasing scale by exploiting representational redundancy to becoming more explorative. To put it another way, the search landscapes clearly contain a larger number of more diverse peaks, since there is an evolutionary advantage towards adopting a more explorative strategy. However, the picture for complexity is different. No significant differences in maximum mutation probabilities or numbers of phenotypically unique individuals are observed. Taken overall, the results suggest that the self-adaptive evolutionary algorithm is more explorative in the face of problem instance scale rather than complexity (for a fixed scale), which in turn suggests that scale of the discrete search landscape rather than its complexity has a significant effect on search.

VI. CONCLUSIONS

To address the first research question posed by this paper, we find that the different strategies emerging from self-adaptive mutation yield useful insights into object-oriented software design search spaces We conclude that that the scale of the search landscape rather than its complexity (for a given scale) has a significant effect on search. To address the second research question, the implication of this insight for the software design is to confirm the usefulness of decomposition as a design technique. Decomposition effectively reduces design scale to enable the breakdown of large, complex software designs into smaller, more manageable component parts.

We had wished to compare our findings with empirical data in the field. Unfortunately, such studies are not readily available in the literature. Nevertheless, it is interesting to note that the findings of this paper appear broadly consistent with an empirical study from a different field i.e. architectural building design results obtained show an exponential decrease in solution quality as scale increases, but complexity became less significant as the scale of the problem increased.

REFERENCES

[1] M. Petre, "Insights from expert software design practice", in Proceedings of the 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '09), ACM Press, 2009, pp. 233-241.

[2] R. L. Glass, Facts and Fallacies of Software Engineering. Boston, MA, USA, Addison-Wesley Pearson Education, 2003.

[3] F. P. Brooks, "No Silver Bullet – Essence and Accidents in Software Engineering", chapter 16 in The Mythical Man Month (Anniversary Edition), Addison-Welsey, 1995.

[4] J.L. Fiadeiro, "The many faces of complexity in software design", chapter 2 in Conquering Complexity, M. Hinchley, L. Coyle, (Eds.), London, Springer, 2012.

[5] G. Booch, Object-oriented analysis and design with applications (2nd Ed.), Califorina, USA, Benjamin Cummings, 1994.

[6] F. Flager, D. J. Gerber, and B. Kallman, "Measuring the impact of scale and coupling on solution quality for building design problems", Design Studies, vol. 36, no. 2, pp. 180-199, 2014.

[7] K. De Jong, "Parameter setting in EAs: a 30 year perspective", chapter 1 in Parameter Setting in Evolutionary Algorithms, F. G. Lobo, C. F. Lima, and Z. Michalewicz, (Eds.), Berlin Heidelberg, Springer, 2007.

[8] A. E. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter control in evolutionary algorithms", IEEE Transactions on Evolutionary Computing, vol. 3, no. 2, pp. 124-141, 1999.

[9] S. Meyer-Neiberg, and H.-G. Beyer, "Self-adaptation in evolutionary algorithms", chapter 3 in Parameter Setting in Evolutionary Algorithms, F. G. Lobo, C. F. Lima, and Z. Michalewicz, (Eds.), Berlin Heidelberg, Springer, 2007.

[10] C. L. Simons, and I. C. Parmee, "Dynamic parameter control of interactive local search in UML Software Design", in Proceedings of the 2010 IEEE International Conference on Systems, Man, and Cybernetics, (SMC '10). IEEE Press, 2010, pp. 3399-3404.

[11] Object Management Group, 2014, http://www.uml.org/

[12] C. L. Simons, and I. C. Parmee, "Elegant object-oriented software design via interactive, evolutionary computation", IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews, vol. 42, no. 6, pp. 1797-1805, 2012.

[13] R. Harrison, S. J. Counsell, and R. V. Nithi, "An investigation in the applicability and validity of object-oriented design metrics", Empirical Software Engineering, vol. 3, no. 3, pp. 255-273, 1998.

[14] C. L. Simons, 2014, http://www.cems.uwe.ac.uk/~clsimons/CaseStudies/index.htm