A GENETIC ALGORITHM APPROACH TO DESIGN EVOLUTION USING DESIGN PATTERN TRANSFORMATION

Mehdi AMOUI [1], **Siavash MIRARAB** [2], **Sepand ANSARI** [2], **Caro LUCAS** [1]

[1]*Control and Intelligent Processing Center of Excellence, Department of Electrical and Computer Engineering, University of Tehran, Tehran, Iran*
E-mail: mehdi.amoui@ece.ut.ac.ir, lucas@ipm.ir

[2]*Department of Electrical and Computer Engineering, University of Tehran, Tehran, Iran*
E-mail: {s.mirarab, sepans}@ece.ut.ac.ir

Abstract

Improving software quality is a major concern in software development process. Despite all previous attempts to evolve software for quality improvement, these methods are neither scalable nor fully automatable. In this research we approach software evolution problem by reformulating it as a search problem. For this purpose, we applied software transformations in a form of GOF patterns to UML design model and evaluated the quality of the transformed design according to Object-Oriented metrics, particularly 'Distance from the Main Sequence'. This search based formulation of the problem enables us to use Genetic Algorithm for optimizing the metrics and find the best sequence of transformations. The implementation results show that Genetic Algorithm is able to find the optimal solution efficiently, especially when different genetic operators, adapted to characteristics of transformations, are used. Overall, we conclude that software transformations can successfully be approached automatically using evolutionary algorithms.

Keywords: Genetic Algorithms, Software Evolution, Software Quality, Transformation

1 Introduction

High quality software needs to meet both its functional and non-functional requirements, such as reusability, performance, and robustness. Design and implementation defects that cause systems to exhibit low maintainability, low reuse, high complexity, and faulty behavior are reduced in high quality software. Because of this, major proportion of total cost of the software development process is devoted to software maintenance [1], [2], [3]. Therefore, the need of techniques that reduce software complexity by incrementally improving the internal software quality becomes more obvious [4]. The first step toward achieving this goal is to quantize non-functional properties. For this reason, metrics have long been studied as a way to assess the quality of software systems [5] and have been applied to object-oriented systems as well [6], [7], [8], [9].

In the reengineering phase, there is a need of systematic way for software quality improvement at different stages and abstraction levels. The research domain that addresses this problem is referred to as restructuring or, in the specific case of object-oriented software development, refactoring [4]. The term refactoring was originally introduced by William Opdyke [10]. The main idea is to redistribute classes, variables and methods across the class hierarchy in order to facilitate future adaptations and extensions [11]. Based on object-oriented refactorings, Tokuda and Batory presented a pragmatic approach for high level object-oriented transformations to Design Patterns while preserving the software behavior [12]. In another approach for higher level transformations, Tahvildari et al. used a catalogue of object-oriented metrics as an indicator to automatically detect where a particular meta-pattern can be applied to improve the software quality via refactoring [13]. However, former attempts on design evolution are hardly automated. In case of software refactoring, the numerous combinations of valid transformations, makes the decision of applying appropriate transformations to achieve the highest software quality difficult. Therefore, search-based and evolutionary methods can be extensively used to automatically find merely the best possible sequence of valid transformations. This type of software evolution by using Genetic Algorithms and other search-based methods has been developed recently for simple *line of codes (LOC)* metric [14].

Throughout this contribution, we will propose a search-based evolutionary method, particularly Genetic Algorithm, to find the best sequence of valid high level design pattern transformations to improve software reusability. Improvement evaluation is based on measurement of some high level object-oriented design metrics.

The rest of this paper is organized as follow: First, we define the characteristics of chosen object-oriented design metrics, and then we discuss the details of our Genetic Algorithm functions, assumptions, and encodings. Next, our developed framework for automating the whole process is introduced. Afterwards we demonstrate our approach through a case study. Finally, conclusion and further works are given.

2 Metrics for Transformation Evaluation

Design patterns, which are described as a general solution to a common problem in software design [15], can be expressed as a series of parameterized program transformations applied to a plausible initial software state [12]. In this research we tend to use high level design transformations, mainly GOF design patterns. Most of these patterns can improve the design quality and reusability by decreasing generic coupling metrics while increasing the cohesion.

To measure the design quality improvement, many object-oriented metrics have been introduced since the birth of object-oriented programming. These metrics are capable of measuring wide range of software properties at different abstraction levels [16]. Throughout this research we narrow our metrics to those related to measuring the reusability of design which is one of the major factors of software quality.

According to object oriented design principles, a high quality design consists of stable packages that are surrounded by other loosely coupled subsystems which could possibly minimize the propagation of software changes. Thus, software stability is highly dependent on the coupling between components. Because none of usual metrics can represent the overall system reusability, we will use *Distance from the Main Sequence (D)* metric introduced by Robert C. Martin [17] as the primary parameter that we tend to optimize in our approach. The *D* metric ranges between [0, 0.707] and can be calculated by:

$$D = \frac{|A + I - 1|}{\sqrt{2}} \tag{1}$$

Where:

$$I = \frac{\textit{Efferent Coupling}}{\textit{Efferent Coupling} + \textit{Afferent Coupling}} \tag{2}$$

$$A = \frac{\textit{Abstract Classes}}{\textit{Total Classes}} \tag{3}$$

In our framework we will use the normalized version of *D* metric which is more convenient and ranges between [0,1]. The visual representation of this metric is available in Fig. 1. According to *D* metric, any package that stands far from the origin is unbalanced and should be reengineered in order to define it more reusable and less sensitive to changes. As a result, the value of *D* metric should be minimized to improve the reusability.

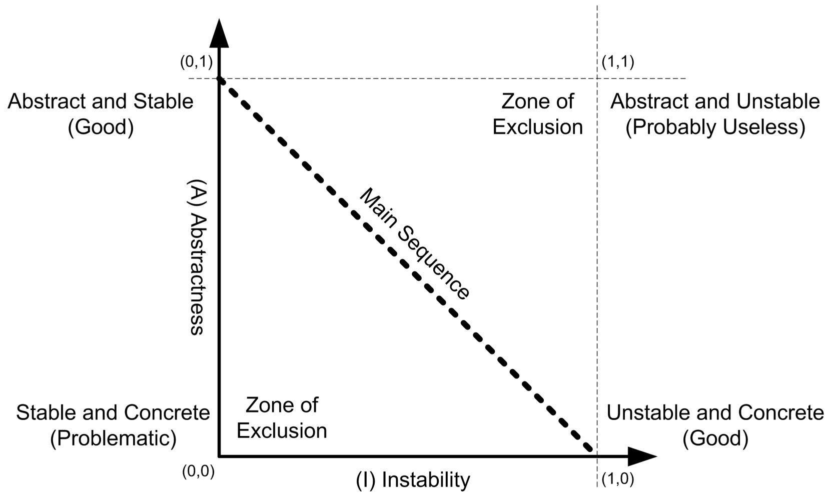

(0,1)
(1,1)
Abstract and Stable (Good)
Zone of Exclusion
Abstract and Unstable (Probably Useless)
(A) Abstractness
Main Sequence
Stable and Concrete (Problematic)
Zone of Exclusion
Unstable and Concrete (Good)
(0,0)
(I) Instability
(1,0)

Figure 1. Distance from the Main Sequence [17]

3 Genetic Algorithm

In the context of software transformation, patterns can be applied to different classes with different parameters, making the enumeration of candidate solutions substantial. On the other hand, there is no efficient or complete algorithm to find the appropriate and optimum sequence of transformations. These characteristics of software transformation problem led us to use genetic algorithms as a good search-based method for handling the problem [27]. Another technique for searching large search spaces is Simulated Annealing, which has reasonable performance on many search problems. In this specific problem because of the independent instinct of neighborhoods and discrete search space, this method is not supposed to perform well. In this section, we describe the formulation of our problem as a genetic algorithm problem:

3.1 Chromosomes

The chromosomes are basically an encoding of a sequence of transformations and their parameters. Each individual consists of several supergenes, each of which represents a single transformation. A supergene is a group of neighboring genes on a chromosome, which are closely dependent, often functionally related, and invalid under certain circumstances. These internal genes are the pattern number, the package number, and the classes that the pattern is applied to, as illustrated in Fig. 2. Since each pattern has different number of parameters and the parameters are dependent on each other, after applying genetic operators some invalid supergenes may be produced. For example the package that a certain pattern should be applied to and the classes that are transformed by the pattern are encoded in the chromosome. By applying mutation operator, the class number could become an invalid number that is not present in the package encoded in the chromosome. Thus after applying the genetic operators, any inconsistency in the parameters of supergenes will be found and those supergenes will be discarded. The population size is 30N. where N is the number of packages. The population is initialized randomly and after random generation of parameters inside the encoding of a chromosome, the invalid supergenes are detected and initialized again until no inconsistency occurs.

3.2 Fitness Function

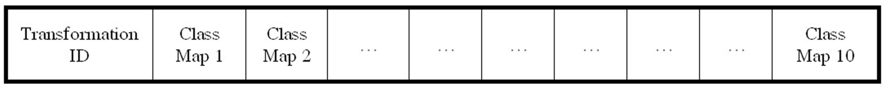
To evaluate a chromosome, its represented transformations are applied to the original design using the *Transformer Engine*. The *Transformer Engine* transforms the design according to the data encoded in the chromosome, then the D metric is evaluated using the *Design Metrics Evaluator Engine* and is set as the fitness value of the given individual. Furthermore, not all the patterns can be applied to all the classes of the project. If applying a design pattern to some specific classes leads to inconsistency with basic principals of software engineering, such as generating a cyclic inheritance, a zero fitness value is assigned to the chromosome. More specifically, we guarantee that a java code can be automatically generated from the transformed code.

3.3 Crossover

We designed two different crossover operators, which are applied either simultaneously or separately. The first crossover is a single-point crossover with a randomly selected point applied at supergene level. This crossover swaps the supergenes beyond the crossover point, but the internal genes of supergenes remain unchanged. This means the offspring produced from the parent chromosomes is the combination of the sequence of the patterns of each parent. The second crossover operator randomly selects two supergenes from two parent chromosomes, and similarly applies single point crossover to the genes inside the supergenes. This crossover operator combines two different patterns and generates a new pattern. Each of these crossovers has different effects. The first one combines the promising patterns of two different transformation sequences, while the second one combines the parameters of two successfully applied patterns. The crossover rate is 0.8 when applied separately and 0.45 each when both of them are applied in each generation.

3.4 Mutation

The mutation operator randomly selects a supergene and mutates a random number of genes, inside that supergene. The mutation operator randomly changes the parameters of a pattern inside a supergene. After applying the mutation, inconsistency may occur in the supergenes, thus the validity of the supergenes are checked. The mutation rate is 0.06.



Transformation ID	Class Map 1	Class Map 2	...	...	...	...	...	...	Class Map 10

Figure 2. Supergene Representation

4 Framework

Our developed framework, *Automatic Design Enhancer (ADE)*, designed to automate the whole process of evolving software design via genetic algorithm, consists of three major subsystems. The block diagram of ADE is illustrated in Fig. 3.

The *Design Transformer engine* is based on an open-source project DPA toolkit [19]. It can reverse engineer the software and extract its design. DPA can also get the required design model directly from an XML. The main feature of DPA is the facility to apply design patterns to the object-oriented design of the project. To do so, the developer can use the existing design pattern plug-in or create a custom plug-in.

To evaluate generated, transformed design of *Transformer engine*, according to object-oriented design metrics, we have developed a Design Metrics Evaluator engine. Finally we put them all together using JGAP [1] GA toolkit as our *Genetic Algorithm engine*.

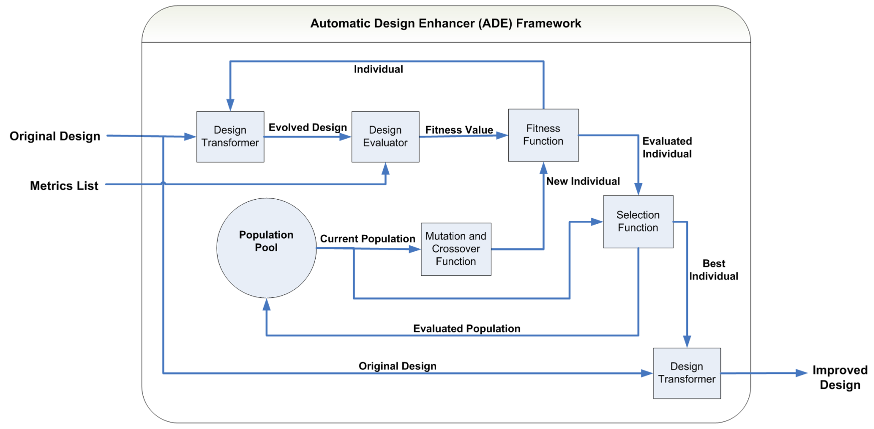


Figure 3. Automatic Design Enhancer (ADE) Framework Schema

5 Case study

In order to test our framework, we extracted the UML design of some free, open source applications. Then we executed our framework with genetic algorithm in two versions. In one version only the first crossover operator, as described in GA section, was applied but in the other both operators was used. Fig. 4 and Fig. 5 illustrate the improvement in fitness value for both runs, tested on an application called Gold Parser [2], which consists of 9 packages, 101 classes, and D metric value of 0.172. Also we applied other search methods such as uninformed search and simulated annealing, but because of the decreet nature of search space, GA finds the optimal solution much more efficiently and accurately. Table 1 compares genetic algorithm and random search.

From software design perspective, the transformed design of the best chromosomes are evolved such that abstract packages become more abstract and concrete packages turn to become more concrete. This is the nature of D metric and reduces the coupling between concrete packages.

[1]http://jgap.sourcefoge.net/
[2]http://www.devincook.com/goldparser/engine/dot-net.htm

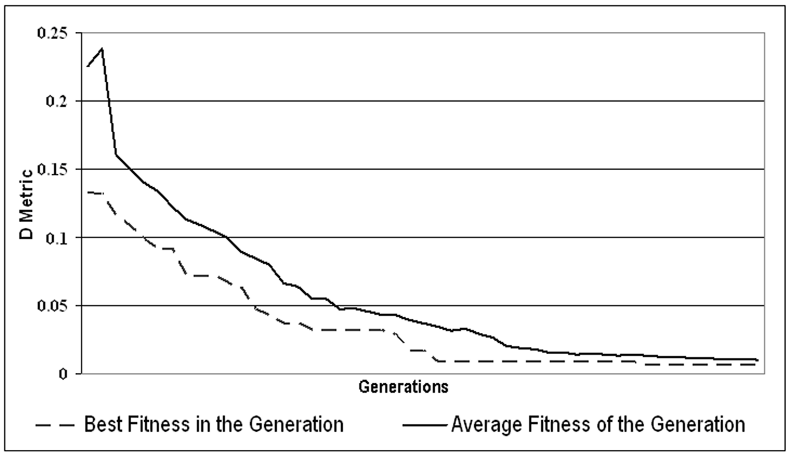


Figure 4. Metric Improvement through 100 generations using 1 crossover

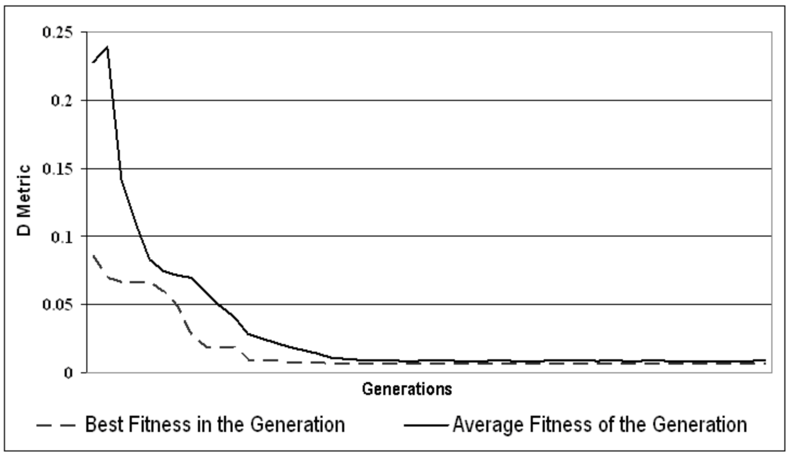


Figure 5. Metric Improvement through 100 generations using 2 crossovers

Table 1. Different initial test conditions for GoldParser case study

	Random Search	**1 Crossover**	**2 Crossovers**
No. of Generations/Searches	10000	100	100
Average Population Size	N/A	82	105
Best Metric Value	0.06	0.02	0.01
Average Metric Value	0.41	0.04	0.03
% of Valid Transformations	44.5	94.0	95.1

6 Future Works and Conclusion

Further researches may include developing other and more complex crossovers, mutations, and selection algorithms. Also more intelligent crossover functions with memory can be implemented using reinforcement learning approaches. Besides, different pattern and meta-pattern collections may be developed to support transformations, as well further improvements may be considered to ensure the overall quality of software by using data fusion approaches on design metrics.

Finally, the whole process might be applied on design models with different applications. It is clear that different metrics are appropriate for different applications according to the nature of those programs.

This research introduces the possibility of completely automated design evolution using search-based methods especially genetic algorithms. It uses the pool of high level transformations to achieve the required design quality improvement.

Investigation on the results of the whole evolution process make us conclude that search based algorithms, particularly genetic algorithm, are helpful in improving special design metrics.

References

[1] Coleman D. M., Ash D., Lowther B., Oman P. W., 1994, *Using Metrics to Evaluate Software System Maintainability*, IEEE Computer, Vol. 27, No. 8, pp. 44-49.

[2] Guimaraes T., 1983, *Managing Application Program Maintenance Expenditure Comm*, ACM, Vol. 26, No. 10, pp. 739-746.

[3] Lientz B. P.: Swanson E. B., 1980, *Software Maintenance Management. A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, Addison-Wesley.

[4] Mens T., Tourwe T., 2004, *A Survay of Software Refactoring*, IEEE Transaction on Software Engineering, Vol. 30, No. 2, pp. 126-139.

[5] Fenton N., Pfleeger S. L., 1997, *A Rigorous and Practical Approach*, International Thomson Computer Press, London, UK, second edition.

[6] Chidamber S. R., Kemerer. C. F., 1994, *A Metrics Suite for Object-Oriented Design*, IEEE Transaction on Software Engineering, Vol. 20, No. 6, pp. 476-493.

[7] Lorenz M., Kidd J., 1994, *Object-Oriented Software Metrics A Practical Approach*, Prentice-Hall.

[8] Marinescu R., 1998, *Using Object-Oriented Metrics for Automatic Design Flaws in Large Scale Systems*, In Demeyer S. and Bosch J. (Ed.), Object-Oriented Technology (ECOOP'98 Workshop Reader), LNCS 1543, Springer-Verlag, pp. 252-253.

[9] Nesi P., 1998, *Managing OO Projects Better*, IEEE Software.

[10] Opdyke W., 1992, *Refactoring Object-Oriented Frameworks*, PhD thesis, University of Illinois at Urbana-Champaign.

[11] Fowler M., 1999, *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley.

[12] Tokuda L., Batory D. S., 2001, *Evolving Object-Oriented Designs with Refactorings*, Automated Software Engineering, Vol. 8, No. 1, pp. 89-120.

[13] Tahvildari L., Kontogiannis K., 2004, *Improving Design Quality Using Meta-pattern Transformations: A Metric-based Approach*, Journal of Software Maintenance and Evolution, Vol. 16, pp. 331-361.

[14] Fatiregun D., Harman M., Hierons R. M., *Evolving Transformation Sequences using Genetic Algorithms. Source Code Analysis and Manipulation*, Fourth IEEE International Workshop on (SCAM'04), pp. 66-75.

[15] Gamma E, Helm R., Johnson R., Vlissides, J., 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading MA.

[16] Mens T., Demeyer S., 2001, *Evolution metrics*, In Proceedings of International Workshop on Principles of Software Evolution.

[17] Martin R. C., 2000, *Design Principles and Design Patterns*, http://www.objectmentor.com.

[18] Clarke J., Harman M., et al, 2003, *Reformulating Software Engineering as a Search Problem*, IEE Proceedings - Software, pp. 161-175.

[19] Design Pattern Automation Toolkit (DPA), http://sourceforge.net/projects/dpatoolkit/.