Software Engineering 2012, 2(2): 29-35
DOI: 10.5923/j.se.20120202.03

GA-driven Automatic Refactoring based on Design Patterns

Takao Shimomura

Dept. of Information Science and Intelligent Systems, University of Tokushima, Tokushima, 770-8506, Japan

Abstract Refactoring is a process of applying behavior-preserving transformations to improve the design, readability, structure, performance, abstraction, and maintainability of existing code. This paper presents an approach to GA-driven refactoring for Java programs to automatically judge the qualities of programs based on design patterns. If a program is judged to be bad, this GA-driven refactoring method will further recommend that the program should be transformed using an appropriate design pattern.

Keywords Design Patterns, Discriminant Analysis, Genetic Algorithm, Refactoring, Syntax Analysis

1. Introduction

Refactoring is a process of applying behavior-preserving transformations to improve the design, readability, structure, performance, abstraction, and maintainability of existing code[1-3]. It applies some transformations to programs such as extracting interface for re-routing the access to a class via a newly created interface and pulling up members for moving members into a superclass. Griffith, et al.[4] developed an automated system utilizing Evolutionary Algorithms to manipulate refactorings correctly without requiring an underlying understanding of the software.

Briand et al.[5] empirically explore the relationships between existing object-oriented coupling, cohesion, and inheritance measures and the probability of fault detection in system classes during testing. The frequency of method invocations and the depth of inheritance hierarchies seem to be the main driving factors of fault-proneness.

Daikon[6] demonstrates the feasibility of automatically finding places in the program that are candidates for specific refactorings. The approach uses program invariants: when a particular pattern of invariant relationships appears at a program point, a specific refactoring is applicable. Hanenberg et al.[7] introduce a number of new aspect-oriented refactorings which help to migrate from object-oriented to aspect-oriented software and to restructure existing aspect-oriented code. Hannemann et al.[8] introduce a role-based refactoring approach to aid developers in re-structuring the implementation of crosscutting concerns using aspect-oriented programming. Tip et al.[9] present an approach in which type constraints are used to verify preconditions and to determine allowable source code modifications for a number of generalization-related refactorings. As object-oriented class libraries evolve, classes are occasionally deprecated in favor of others with roughly the same functionality. In Java's standard libraries, for example, class Hashtable has been superseded by HashMap, and Iterator is now preferred over Enumeration. Balaban et al.[10] present an approach in which mappings between legacy classes and their replacements are specified by the programmer. Then, an analysis based on type constraints determines where declarations and allocation sites can be updated.

A variety of metrics have been proposed to estimate the qualities of programs[11,12]. For example, a class situated deeper in the inheritance hierarchy is more likely to be fault-prone than a class situated higher up (i.e., closer to the root) in the inheritance hierarchy. However, the cost of detecting candidates for refactoring and of choosing an appropriate refactoring transformation could be high.

Genetic algorithm (GA) has been used to obtain the optimal solution in a variety of fields such as aesthetic design of bridge structures, logistics networks, and grid computing[13-15]. This paper presents an approach to GA-driven refactoring for Java programs to automatically judge the qualities of the programs based on design patterns[16]. If a program is judged to be bad, refactoring will be further recommended so that the program can be transformed using an appropriate design pattern.

2. GA-driven Refactoring

2.1. Design Patterns

A design pattern is a general reusable solution to a commonly occurring problem in software design. If we appro-

* Corresponding author:
simomura@is.tokushima-u.ac.jp (Takao Shimomura)
Published online at http://journal.sapub.org/se

priately apply design patterns to software development, we can efficiently make programs of high quality that can be easily enhanced. This GA-driven refactoring uses design patterns to judge a program to be bad if a design pattern can be applicable to the program. There are a number of design patterns, some of which are shown in Table 1.

Abstract Factory design pattern can create a set of instances easily. Figure 1 shows two sample programs for the pattern. Figure 1 (a) illustrates a bad program to which the pattern can be applicable, and Fig. 1 (b) illustrates a good program that is obtained after the pattern is applied to the bad program. If the number of products is p, and the number of factories is f, the bad program requires the invocations of p*f new operations. On the other hand, the good program only requires the invocations of f create() methods.

Template Method design pattern can customize a series of common processes partly. Figure 2 shows two sample programs for the pattern. Figure 2 (a) illustrates a bad program to which the pattern can be applicable, and Fig. 2 (b) illustrates a good program that is obtained after the pattern is applied to the bad program. If the number of classes that use a template method is c, and the number of sub-operations of the template method is s, the bad program requires c*s method invocations. On the other hand, the good program only requires c*t method invocations, where t is much less than s.

Decorator design pattern can enhance a class without modifying the class. Figure 3 shows two sample programs for the pattern. Figure 3 (a) illustrates a bad program to which the pattern can be applicable, and Fig. 3 (b) illustrates a good program that is obtained after the pattern is applied to the bad program. If the number of components is c, and the number of decorators, which will enhance each component is d, the bad program requires c*d more classes. On the other hand, the good program only requires d more classes.

```
public class Main {
    public static void createSet(String type) {
        Burger burger = null;
        Dish dish = null;
        Drink drink = null;
        if ("Japanese".equals(type)) {
            burger = new JapaneseBurger();
            dish = new JapaneseDish();
            drink = new JapaneseDrink();
        } else if .........
        burger.display();
        dish.display();
        drink.display();
    }
    public static void main(String[] args) {
        String[] types = {
            "Japanese",
            "American",
            "French",
        };
        for (int i = 0; i < types.length; i++) {
            String type = types[i];
            createSet(type);
        }
    }
}
```

(a) a bad program

```
public class Main {
    public static void createSet(Factory factory) {
        Burger burger = factory.createBurger();
        Dish dish = factory.createDish();
        Drink drink = factory.createDrink();
        burger.display();
        dish.display();
        drink.display();
    }
    public static void main(String[] args) {
        Factory[] factories = {
            new Japanese(),
            new American(),
            new French(),
        };
        for (int i = 0; i < factories.length; i++) {
            Factory factory = factories[i];
            createSet(factory);
        }
    }
}
```

(b) a good program

Figure 1. Sample programs for Abstract Factory

```
public class Japanese {
    public void introduce() {
        nationality();
        breakfast();
        lunch();
        dinner();
        supper();
    }
    public void nationality() {
        System.out.println("I am Japanese.");
    }
    public void breakfast() {
        System.out.println("I eat fruit at breakfast.");
    }
    public void lunch() {
        System.out.println("I eat rice at lunch.");
    }
    public void dinner() {
        System.out.println("I eat meat at dinner.");
    }
    public void supper() {
        System.out.println("I eat nothing at supper.");
    }
}
```

(a) a bad program

```
public class Japanese extends Person {
    public void nationality() {
        System.out.println("I am Japanese.");
    }
    public void lunch() {
        System.out.println("I eat rice at lunch.");
    }
}
```

```
abstract public class Person {
    public void introduce() {
        nationality();
        breakfast();
        lunch();
        dinner();
        supper();
    }
    abstract public void nationality();
    public void breakfast() {
        System.out.println("I eat fruit at breakfast.");
    }
    public void lunch() {
        System.out.println("I eat toast at lunch.");
    }
    public void dinner() {
        System.out.println("I eat meat at dinner.");
    }
    public void supper() {
        System.out.println("I eat nothing at supper.");
    }
}
```

(b) a good program

Figure 2. Sample programs for Template Method

```
public class Main {
    public static void main(String[] args) {
        new Japanese().introduce();
        new JapaneseSinger().introduce();
        new JapaneseSinger().sing();
        new American().introduce();
        new AmericanSinger().introduce();
        new AmericanSinger().sing();
        new French().introduce();
        new FrenchSinger().introduce();
        new FrenchSinger().sing();
        new German().introduce();
        new GermanSinger().introduce();
        new GermanSinger().sing();
    }
}
```

```
public class AmericanSinger extends American {
    public void sing() {
        System.out.print("I am singing ... ");
        super.introduce();
    }
}
```

(a) a bad program

```
public class Main {
    public static void main(String[] args) {
        Person[] people = {
            new Japanese(),
            new American(),
            new French(),
            new German(),
        };
        for (int i = 0; i < people.length; i++) {
            Person person = people[i];
            person.introduce();
            Singer singer = new Singer(person);
            singer.introduce();
            singer.sing();
        }
    }
}
```

```
public class Singer extends AbstractPerson {
    public Singer(Person person) {
      super(person);
    }
    public void sing() {
        System.out.print("I am singing ... ");
        person.introduce();
    }
}
```

```
public abstract class AbstractPerson implements Person {
    protected Person person;
    protected AbstractPerson(Person person) {
        this.person = person;
    }
    public void introduce() {
        person.introduce();
    }
}
```

(b) a good program

Figure 3. Sample programs for Decorator

Table 1. Example of design patterns

Classification	Design patterns	Features
Creation	Factory Method	Create an specified instance
	Builder	Create an instance with complicated parameters
	Abstract Factory	Create a set of instances
Behavior	Template Method	Customize a series of common processes partly
	Interpreter	Interpret a hierarchical structure
	Mediator	Leave message passing to a mediator
	Observer	Notify listeners of events
	State	Invoke a process according to a state
	Strategy	Leave a process to another class
	Visitor	Add an operation to a class without modification
	Command	Capsulate operations
Structure	Adapter	Provide a standard interface to a class
	Composite	Form a hierarchical structure
	Decorator	Enhance a class without modification

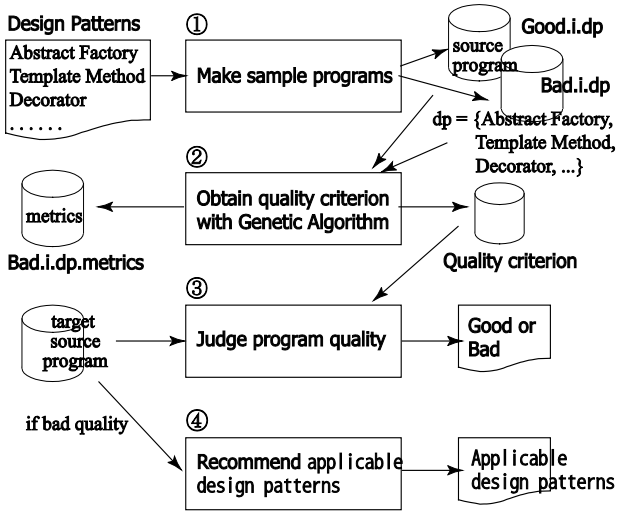

Figure 4. Automatic refactoring flow

2.2. Automatic Refactoring Flow

Figure 4 shows a flow of GA-driven automatic refactoring. (1) First, for each design pattern, we make bad sample programs to which the design pattern should be applied. Then, we make good programs by applying the design pattern to

each of them. (2) Then, the system read these bad and good programs to obtain a quality criterion with which a target program will be judged to be bad or good. The quality criterion is obtained based on the metrics of the bad and good programs. A genetic algorithm is used to determine the weight of each metric, which indicates how much impact the metric has to distinguish bad programs from good programs. (3) A target program is judged to be good or bad based on the obtained quality criterion. (4) If the target program is judged to be bad, the system will recommend design patterns applicable to the target program.

Table 2. Example of metrics that determine program quality

Categories	Metrics	Values	GA Weights
Method	mIf	number of if statements	0.815
	mNew	number of new operators	0.828
Class	cSim	number of methods that contain the same method call as the others	0.919
	cLayer	number of super classes	0.715
	cComp	number of fields that refer to other source classes	0.762
	cMethod	number of methods	0.708
Program	pSim	number of methods that contain the same method call as the other classes	0.999
	pSub	number of subclasses	0.782

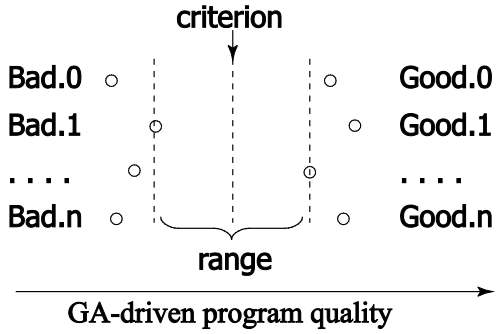

Figure 5. Determine the criterion of program quality

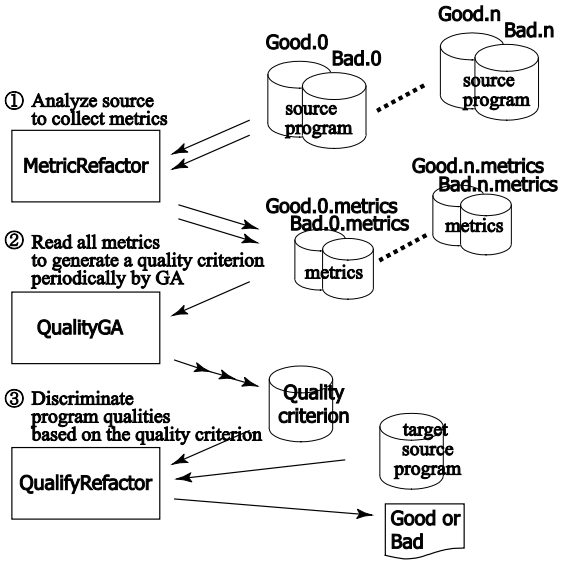

Figure 6. GA-driven discriminant analysis of program quality

2.3. Metrics and Their Weights

We introduce some metrics of a program to estimate its quality. These metrics are expected to become important factors that determine the quality of the program. However, it is not easy to judge whether each metric will have a good or bad influence on the program quality. Moreover, it is not easy to estimate to what extent each metric will give an impact to the program quality. Therefore, we introduce a weight for each metric, and define a factor as the value of a metric multiplied by some weight. Table 2 shows an example of some metrics of a program. For example, metric mIf is a factor that is derived from the number of if statements. The program quality of a program that consists of multiple source files will be calculated from the class qualities of the classes defined in the program. The class quality of a class will be calculated from the method qualities of the methods defined in the class.

The value of a metric (m) is normalized to range from 0.0 to 1.0. For example, the normalized value of metric mIf is defined as

$$1 - (1- \mathrm{mIf}) / (1 + \mathrm{steps})$$

where steps indicates the number of lines of program code in a method. A weight (w) also ranges from 0.0 to 1.0. A weight indicates to what extent each metric will give an impact to the program quality. When a metric (m) is introduced, we also introduce its dual metric (m^d) because we cannot know in advance whether the metric will have a good or bad influence on the program quality. The dual metric m^d of a metric m is defined as (1 - m). A factor (f) is defined as

$$f = (m * w) + m^d * (1 - w)$$

which ranges from 0.0 to 1.0. Each weight will be determined by a genetic algorithm as described in Section 2.4. If weight w becomes close to 1.0, a metric m will become an important factor. If weight w becomes close to 1/2, neither of metric m nor dual metric m^d will have an impact on program qualities. If weight w becomes close to 0.0, a dual metric m^d will become an important factor.

2.4. Determine the Criterion of Program Quality

To define the quality of a program, we need to determine the weight of each metric. We do not have any preconception about the metrics we have introduced. Therefore, we determine the weight of each metric by using a genetic algorithm. To automatically determine the weights, we need to give some information about which program has a high quality and which program has a poor quality to the genetic algorithm. We have made some programs of high and poor qualities based on several design patterns, such as Abstract Factory, Template Method, and Decorator patterns. For example, in an Abstract Factory pattern, bad programs require more conditional statements than good programs. In a Template Method pattern, bad programs have some classes, where a method contains a sequence of the same method calls as a method another class defines. In a Decorator pattern, bad programs have more pairs of a superclass and a

subclass and more instantiations than good programs.

The genetic algorithm program calculates the program qualities of both of good and bad programs. Its fitness function returns a range

(min{good programs' qualities} - max{bad programs' qualities}),

as shown in Fig. 5. The optimal weights are determined so that this range will be as large as possible. We take the middle point as a criterion to judge whether a program has a good or bad quality. If the quality of a program that is calculated with the optimal weights is less than the criterion, this program will be judged to be a bad program, and refactoring is recommended so that the program will be transformed using an appropriate design pattern.

2.5. GA-Driven Discriminant Analysis

Figure 6 illustrates a sequence of processes from collecting metrics to judging program qualities.

(1) We first prepare several pairs of good and bad programs based on design patterns. Then, we execute MetricRefactor program to compile them and collect metrics.

(2) We execute QualityGA program, which is a genetic algorithm program, to calculate the optimal weights of the metrics. It first reads the metrics that are output by MetricRefactor program. By using the fitness function described before, it determines the optimal weights of the metrics and periodically outputs them into a quality criterion file. This file contains the criterion to judge the quality of a program as well as a set of the optimal weights of the metrics.

(3) If we execute QualifyRefactor program by specifying a target program, it will read the latest quality criterion file to calculate the quality of the target program and finally judge whether it is a good or bad program. Both of MetricRefactor and QualifyRefactor programs collect the metrics of a program. The way to collect metrics from a program will be described in detail in Section 3.

2.6. Recommendation of Applicable Design Patterns

If the target program is judged to be bad, the system will recommend design patterns that are most applicable to the target program. Figure 7 shows how applicable design patterns are determined. Each bad program Bad.i.dp represents a point (m_i*w_i) in a multi-dimensional affine space of metrics, where m_i is a metric and w_i is its weight. If the target program is located most close to Bad.i.dp, its design pattern dp will be recommended. More than one design patterns can also be recommended.

3. Refactoring Analysis

3.1. Collecting Metrics by Using Java Compiler Visitor Pattern

We have revised a Java compiler to collect metrics from a source program. The Java compiler is developed by using Visitor design pattern[16] so that its abstract syntax tree can be traversed without modifying the nodes of the syntax tree. Therefore, we have inserted some code to start traversing the syntax tree after compilation. Figure 8 shows a mechanism to compile a program and get metrics.

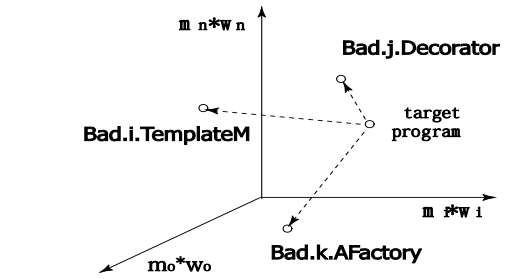

Figure 7. Recommendation of applicable design patterns

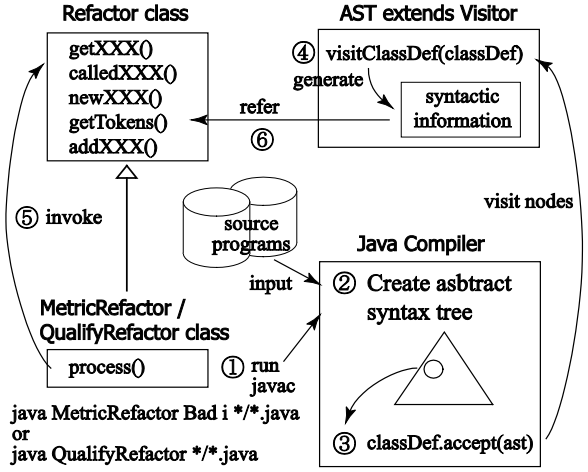

Figure 8. Collecting metrics by using the Visitor pattern of Java compiler

(1) For example, to execute MetricRefactor program for the i-th bad program, we run a command "java MetricRefactor Bad i */*.java".

(2) Java compile starts to create the abstract syntax tree of the program that consists of multiple classes.

(3) After compilation, for each class definition, "classDef.accept(ast)" is invoked.

(4) Then, "visitClassDef(classDef)" in AST class, which extends Visitor, is invoked. AST is defined to traverse child nodes for each node in the abstract syntax tree. AST class provides visit*Node*(*node*) method for each *node* type. This visit*Node*(*node*) method invokes *child*.accept(this) for a *child* node of the *node*. After an instance ast of AST class is created, when *node*.accept(ast) is invoked for a *node* in the abstract syntax tree, ast.visit*Node*(*node*) will be invoked. Therefore, when topLevel.accept(ast) is invoked for the top level node of the abstract syntax tree, all nodes of the tree can be traversed in turn.

topLevel.accept(ast)--->
ast.visitTopLevel(topLevel)--->....--->*child*.accept(ast)
--->ast.visit*Child*(*child*)--->....

(5) The process() method of MetricRefactor class invokes a variety of methods defined in Refactor superclass to calculate the metrics of the program. The methods of Refactor superclass refer the syntactic information that is created by the visit*Node*(*node*) methods of AST class.

Table 3. Example of methods provided by Refactor class

Categories	Methods	Functions
Class	int getClassCount()	returns the number of classes
	String getClassName(int classIndex)	returns the name of a class
Method	int getMethodCount(int classIndex)	returns the number of methods in a class
	String getMethodName(int classIndex, int methodIndex)	returns the name of a method in a class
	int getNumOfIfStms(int classIndex, int methodIndex)	returns the number of if statements in a method of a class
	int getNumOfLoopStms(int classIndex, int methodIndex)	returns the number of loop statements in a method of a class
	int getNumOfAssignStms(int classIndex, int methodIndex)	returns the number of assign statements in a method of a class
	int getNumOfMethodCalls(int classIndex, int methodIndex)	returns the number of call statements in a method of a class
	String [] calledMethodName(int classIndex, int methodIndex, int callIndex)	for a method call in a method of a class, returns the names of the method and the class that defines it
	int [] calledMethodIndex(int classIndex, int methodIndex, int callIndex)	for a method call in a method of a class, returns the indexes of the method and the class that defines it; returns null if it is not defined in the source code.

3.2. Calculation of Program Qualities Using the methods of Refactor Superclass

Table 3 shows an example of methods provided by Refactor class. Because both of MetricRefactor and QualifyRefactor classes are Refactor subclasses, they can directly invoke these methods to collect metrics. For example, if we use calledMethodIndex(int classIndex, int methodIndex, int callIndex), we can know about each method call contained in a method definition of a class, including whether the invoked method is defined in the source code or not.

4. Observation

Table 4. GA parameters

GA type	real value
Number of genes	12 (metrics)
Number of individuals	20
Minimum value of a gene	0.0
Maximum value of a gene	1.0
Selection	roulette wheel selection
Crossover	one-point blend crossover
Mutation	pseudorandom value uniformly distributed between 0.0 and 1.0
Crossover rate	0.6
Mutation rate	0.1
Interval of generation outputs	60 (seconds)
Number of generations	172369766
Number of design patterns	6
Number of pairs of good and bad programs	3

To investigate the effectiveness of the method this paper proposes, we have prepared three pairs of good and bad programs for each of six design patterns, which amount to thirty-six small programs in total. The number of lines of program code ranges from 142 to 226. Table 4 summarizes the parameters of the genetic algorithm this method uses. We have introduced twelve metrics and their corresponding dual metrics. The range that discriminates between good and bad programs has converged to 0.0157, which is not big. However, because this range is positive, we can judge the quality of a target program based on the result of these example programs. A pair of a target good and bad programs were able to be judged to be good and bad respectively by the criterion obtained from a set of the other pairs of the programs. Appropriate design patterns were able to be recommended to fifteen out of eighteen bad programs, whose probability amounts to 0.833.

The weights of the metrics that are related to class cohesion have shown their impacts on program quality. Although we need to have more experiments, the advantages of this GA-driven method are as follows: (1) A new metric can be easily introduced to this system. We have only to revise Refactor subclasses to calculate the metric, and add one gene to the genetic algorithm program for the metric. (2) The GA-driven system gives some information about which metrics have significant impacts on program qualities.

5. Conclusions

This paper has presented an approach to discriminate between programs of high and poor qualities by using a genetic algorithm based on design patterns. Like corpuses in the natural language processing, we need to collect more samples for bad and good programs. At present, this system uses one-dimensional quality. We intend to classify some metrics into groups and apply multivariate analysis.

REFERENCES

[1] Fowler, M. and Beck, K., "Refactoring: improving the design of existing code," Addison-Wesley Professional, 1999.

[2] Batory, D., "Program Refactoring, Program Synthesis, and Model-Driven Development," Springer, 2007.

[3] Mens, T. and Tourwe, "A survey of software refactoring,"

IEEE Transactions on Software Engineering, 30(2):126-139, 2004.

[4] Griffith, I., Wahl, S. and Izurieta, C., "Evolution of legacy system comprehensibility through automated refactoring," Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering, ACM, 35-42, 2011.

[5] Briand, L. C., Wust, J., Daly, J. W. and Porter, D. V., "Exploring the relationships between design measures and software quality in object-oriented systems," Journal of Systems and Software, 51(3):245-273, 2000.

[6] Kataoka, Y., Notkin, D., Ernst, M. D. and Griswold, W. G., "Automated support for program refactoring using invariants," IEEE International Conference on Software Maintenance, page 736, 2001.

[7] Hanenberg, S., Oberschulte, C. and Unl, R., "Refactoring of aspect-oriented software," Proceedings of the 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World, pages 1-17, 2003.

[8] Hannemann, J., Murphy, G. C. and Kiczales, G., "Role-based refactoring of crosscutting concerns," Proceedings of the 4th international conference on Aspect-oriented software development, pages 135-146, 2005.

[9] Tip, F., Kiezun, A. and Baumer, D., "Refactoring for generalization using type constraints," ACM SIGPLAN Notices}, 38(11):13-26, 2003.

[10] Balaban, I., Tip, F. and Fuhrer, R., "Refactoring support for class library migration," ACM SIGPLAN Notices, 40(10):265-279, 2005.

[11] Kan, S. H., "Metrics and models in software quality engineering," Addison-Wesley, 1998.

[12] Singh, S. and Kahlon, K, "Effectiveness of refactoring metrics model to identify smelly and error prone classes in open source software," ACM SIGSOFT Software Engineering Notes, 37(2):1-11, 2012.

[13] Furuta, H., Maeda, K. and Watanabe, E., "Application of genetic algorithm to aesthetic design of bridge structures," Computer-Aided Civil and Infrastructure Engineering, 10(6):415-421, 2008.

[14] Min, H., Koa, H. J. and Ko, C. S., "A genetic algorithm approach to developing the multi-echelon reverse logistics network for product returns," Omega, 34(1):56-69, 2006.

[15] Lim, D., Ong, Y.-S., Jin, Y., Sendhoff, B. and Lee, B.-S., "Efficient hierarchical parallel genetic algorithms using grid computing," Future Generation Computer Systems, 23(4):658-670, 2007.

[16] Metsker, S. J. and Wake, W. C., "Design Patterns in Java," Addison-Wesley, 4 2006.