2012 Second International Conference on Instrumentation & Measurement, Computer, Communication and Control

Detecting Bad Smells with Weight Based Distance Metrics Theory

Jiang Dexun, Ma Peijun, Su Xiaohong, Wang Tiantian
School Of Computer Science and Technology
Harbin Institute of Technology
Harbin, China
silverghost192@163.com

***Abstract*—*Detecting bad smells in program design and implementation is a challenging task. Manual detection is proved to be time-consuming and inaccurate under complex situation. Weight based distance metrics and relevant conceptions are introduced in this paper, and the automatic approach for bad smells detection is proposed based on Jaccard distance. The conception of distance between entities and classes is defined and relevant computing formulas are applied in detecting. New weight based distance metrics theory is proposed to detect feature envy bad smell. This improved approach can express more detailed design quality and invoking relationship than the original distance metrics theory. With these improvements the automation of bad smells detection can be achieved with high accuracy. And then the approach is applied to detect bad smells in JFreeChart open source code. The experimental results show that the weight based distance metrics theory can detect the bad smell more accurately with low time complexity.*

Keywords-refactoring oppotunity; feature envy bad smell; distance metrics theory; weight based distance metrics theory

I. INTRODUCTION

Bad smells are some indications which is trouble and can be solved by a refactoring, and the principle of refactoring is to make it easier to understand and cheaper to modify. So the bad smells are some particular places in codes which make it difficult to understand or modify. Beck's book presented 22 bad smells in the source code that suggest the possibility of refactoring, which are all based on the principle above.

There are two problems should be paid attention. The first is the member to be grouped, or what are the subsystems. In programming languages (particularly object oriented languages) the subsystems could be packages, classes, statements or even variables. The second is the criteria which might lead the grouping process. Grouping criteria can be divided in three categories: functional criteria, data-oriented criteria and time-oriented criteria. The grouping process among different criteria can lead to different results. But no standard grouping result, and criteria can be seen as "correct" or "incorrect". The correctness depends on the usability of the criteria for a specific goal. A functional based grouping is preferred for easily modifying any workflow within a system, while a data-oriented grouping is preferred for modifying some particular data structures.

Grouping should be used as an important factor for the analysis of a given system. A large system with a very coarse grained grouping is much more difficult to be analyzed than that with a fine granularly grouped one. Unfortunately, in most cases the chosen grouping is not made explicit in some documentation but implicitly built into the source code. Especially for reverse engineering of large systems tool support, it is necessary to extract and to comprehend grouping information.

In the level of object oriented classes, this situation is described as the bad smell of "feature envy". In object oriented programs it is a key point that data and the processes used on that data should be packaged together. But the feature envy bad smell is just the opposite. It is described as "a method seems more interested in a class other than the one it actually is in". The action may be that one method invokes much more fields or methods from other class (or classes) than from its own class.

To express and compare methods different invoking times towards same data, the weight based distance metrics theory is introduced. In this theory, for short, if there are more invoking relationships between nodes, the distance is smaller. With this theory the design information (particularly invoking information) of programs can be calculated and analyzed more accurately.

This paper is organized as follows: Section 2 surveys related work. Section 3 introduces the distance metrics theory, and describes the detection goal of distance metrics theory. Section 4 improves the traditional distance metrics theory, and defines the distance formula between entities and classes. With this, the feature envy bad smell in object oriented programs can be detected automatically. Then this section proposes the weight based distance metrics theory aiming at conquering the limitation of traditional distance, and gives the analysis and proof why the weight based distance is more accurate. Section 5 gives the experiment on the JFreeChart open source code with the theories above, and analyzes the results. Section 6 concludes the paper.

II. RELATED WORK

In the past decades, a number of studies were conducted for bad smells of programming codes. Webster [3] introduced smells in the context of object-oriented programming codes, and the smells sorted as conceptual, political, coding, and quality assurance pitfalls. Riel [4] defined 61 heuristics characterizing good object-oriented programming that enable

978-0-7695-4935-4/12 $26.00 © 2012 IEEE
DOI 10.1109/IMCCC.2012.74

engineers to assess the quality of their systems manually and provide a basis for improving design and implementation. Beck Fowler [2] compiled 22 code smells that are design problems in source code, and it is the basis of suggesting for refactorings. Code smells are described in an informal style and associated with a manual detection process. Ma¨ntyla¨ [5] and Wake [6] proposed classifications for code smells. Brown [7] focused on the design and implementation of object-oriented systems and described 40 anti-patterns textually. However in the specification above the detection of these bad smells depends on manual inspection to a great extent, which is a time-consuming and error-prone activity.

Travassos et al. [8] introduced a process based on manual inspections and reading techniques to identify smells. No attempt was made to automate this process, and thus, it does not scale to large systems easily. Also, the process only covers the manual detection of smells, not their specification.

Tsantalis and Chatzigeorgiou [9] proposed one methodology to identify Extract Method refactoring opportunities and present them as suggestions to the designer of an object-oriented system. Their approach is based on the union of static slices that result from the application of a block-based slicing technique. What's more, Limei Yang, Hui Liu and Zhendong Niu [10] gave an approach to recommend fragments within long methods for extraction, and it is integrated as a prototype called AutoMeD. But the methodologies concentrate obsessively on dividing codes into fragments relevance rather than design intention, and their approaches cannot deal with fragments which has interactive variables transmitting.

Marinescu [11] presented a metric-based approach to detect code smells with detection strategies, implemented in the IPLASMA tool. The strategies capture deviations from good design principles and consist of combining metrics with set operators and comparing their values against absolute and relative thresholds.

Munro [12] noticed the limitations of text-based descriptions and proposed a template to describe code smells systematically. This template is similar to the one used for design patterns [13]. It consists of three main parts: a code smell name, a text-based description of its characteristics, and heuristics for its detection. It is a step toward more precise specifications of code smells. Munro also proposed metric-based heuristics to detect code smells. He also performed an empirical study to justify the choice of metrics and thresholds for detecting smells. Alikacem and Sahraoui [14] proposed a language to detect violations of quality principles and smells in object-oriented systems. This language allows the specification of rules using metrics, inheritance, or association relationships among classes, according to the engineers' expectations.

Some approaches for complex software analysis use visualization techniques. These semi-automatic approaches are interesting compromises between fully automatic detection techniques that can be efficient but loose in track of context and manual inspection that is slow and inaccurate [15] [16]. However, they require human expertise and are thus still time-consuming. Other approaches perform fully automatic detection of smells and use visualization techniques to present the detection results [17], [18].

Naouel Moha and Yann-Gae¨l Gue´he´neuc [19] proposed three contributions to the research field related to code and design smells. Tahvildari and Kontogiannis [20] used an object-oriented metrics suite consisting of complexity, coupling, and cohesion metrics to detect classes for which quality has deteriorated and re-engineer detected design flaws. O'Keeffe and O'Cinneide [21] treated object-oriented design as a search problem in the space of alternative designs. For this purpose, they employ search algorithms, such as Hill Climbing and Simulated Annealing, using metrics from the QMOOD hierarchical design quality model [22] as a quality evaluation function that ranks the alternative designs.

Du Bois, Demeyer and Berelst [23] theoretically analyzed the best and worst-case impact of refactorings on coupling and cohesion dimensions. According to the authors, extracting a group of statements from more than one method can decrease import coupling as it removes dependencies which are duplicated across methods.

Seng and Stammel [24] used a special model that examines a set of pre- and post-conditions in order to simulate the application of refactoring operations and a genetic algorithm. But the result may be worse or even bringing new refactoring opportunities which don't present in original program. Kataoka, Andou and Rukaya [25] proposed a quantitative evaluation methodology to measure the maintainability enhancement effect of refactoring. However, the authors do not provide a systematic approach for estimating the coefficient values, and they did not include cohesion as a metric.

Simon et al. [26] defined a distance-based metric, which measures the cohesion between attributes and methods. The calculated distances are visualized in a three-dimensional perspective supporting the developer to manually identify bad smells of codes. However, visual interpretation of distance in large systems can be a difficult and subjective task.

The conception of distance metrics is defined not only among entities (attributes and methods) but also between classes, thus after the computing the existence of bad smell can be detected by data automatically rather than visualization manually. Weight based distance metrics theory has been proposed to express the design quality of programs more accurately, so the code smells detected are more accurate and credible. And it has been achieved and evaluated on large-scale open-source codes.

III. DISTANCE METRICS THEORY

A. Distance metric

What makes two things degree similar or dissimilar? It is helpful to loot at Bung's ontology [27]. Bunge's consideration indicates that the similarity between two things is the collection of their shared properties. One quantitative concept of degree of similarity [28] for a special set of aspects can be defined:

Definition 1 (Degree of Similarity): the *degree of similarity* between two things x, y relative to a finite subset B of all properties P_i is

$$s(x,y)=\frac{|p(x)\cap p(y)\cap B|}{|p(x)\cup p(y)\cap B|} \quad (1)$$

In this, $p(x)$ is the set of properties which have relationship with x, and $|p(x)|$ means the element number of $p(x)$.

So the distance should be defined. The distance between two things x, y is defined as:

$$d(x,y)=1-s(x,y) \quad (2)$$

B. The application of distance metrics in object oriented programs

Before the definition of formula there are several definitions should be shows.

Definition 2 (Entity): the *entity* is the attribute a or the method m in one class, which is signed as E.

Definition 3 (Property Set): the *property set* is the set of entities which have invoking-relations with the given entity E, and it is signed as $P(E)$.

In more detail, $P(a)$ contains a itself and all the methods use a, and $P(m)$ contains itself and all the attributes and methods m uses.

So the distance formula is:

$$Dist(x,y)=1-\frac{|P(x)\cap P(y)|}{|P(x)\cup P(y)|} \quad (3)$$

In this formula, $Dist(x,y)$ is the distance value of entity x and y, and $|P(x)|$ means the member count of $P(x)$.

IV. IMPROVED DISTANCE METRIC TO DETECT BAD SMELLS

A. Average distance between entities and classes

The distance just records the relativity between each two entities, but not the information of location, belonging and so on. So the one-to-one entity distance needs to be improved.

The conception of distance between one entity and one class is defined to show and store the invoking-relations. The distance of entity e and class C is the average of the distances between e and every entity in C. The formula is show as below:

$$\overline{D}(x,C)=\begin{cases}\frac{1}{|E_C|-1}\times\sum_{y_i\in C}Dist(x,y_i) & x,y_i\in E_C\\ \frac{1}{|E_C|}\times\sum_{y_i\in C}Dist(x,y_i) & x\notin E_C, y_i\in E_C\end{cases} \quad (4)$$

B. Distance ratio to detect the bad smell

It is that the distance from entity e to its own class C should be less than to any other classes in any case. If this distance principle is forbidden in the programs, it is sure that there exists feature envy bad smell. The discrimination formula is as follow:

$$r(x,B)=\frac{\overline{D}(x,A)}{\overline{D}(x,B)} \qquad x\in A, x\notin B \quad (5)$$

C. Weight based distance metric

For the purpose of disability that the multiple times invoking of entities cannot be measured in traditional distance metrics theory, the concept of weight value in the calculation of entities distance is introduced.

Definition 4 (Weight Value): in the property set $P=\{E_1,E_2,...,E_n\}$, if the invoking times of entity E_i in the program is x_i, then the *weight value* w_i of E_i is the percentage of invoking times of E_i in the times of all the entities in the set P. The calculating formula of w_i is:

$$w_i=\frac{x_i}{\sum_{i=1}^{n}x_i} \quad (6)$$

In actual calculating w_i is usually instead by x_i, for $\sum_{i=1}^{n}x_i$ is one constant value in certain calculating fields.

The number of property set $P(E)$ is defined as $||P(E)||$ in the weight based distance metrics theory, which is different from the $|P(E)|$ in the simple distance metrics theory. The formula of $||P(E)||$ is as follow:

$$||P(E)||=\sum x_i \quad (7)$$

In this, $E_i\in P(E)$ and x_i is the using time of entity E_i.

So the weight distance calculating formula would be changed.

$$WeightDist(E_1,E_2)=1-\frac{||P(E_1)\cap P(E_2)||}{||P(E_1)\cup P(E_2)||} \quad (8)$$

And the immediate entity should be defined.

Definition 5 (Immediate Entity): entity E is the *immediate entity* of $P(E)$, and others are non-immediate entities.

The immediate and non-immediate entity weight maximum and minimum values can be defined as $M_{\min}(i,j)$ and $M_{\max}(i,j)$.

The non-immediate entity weight maximum and minimum values are as follows:

$$M_{min}(i,j)=\sum_{t=i}^{j}Min(w_{t1},w_{t2}) = Min(w_{i1},w_{i2})+...+Min(w_{j1},w_{j2}) \quad (9)$$

$$M_{max}(i,j)=\sum_{t=i}^{j}Max(w_{t1},w_{t2}) = Max(w_{i1},w_{i2})+...+Max(w_{j1},w_{j2}) \quad (10)$$

The immediate entity weight maximum value $N_{\max}(m,n)$ is defined as:

$$N_{max}(m,n)=\sum_{t=m}^{n}Max(w_{t1},w_{t2}) = Max(w_{m1},w_{n2})+...+Max(w_{m1},w_{n2}) \quad (11)$$

After the immediate and non-immediate entities are dealt with separately, the final weight distance formula will be formed as:

$$WeightDist(E_1,E_2)=1-\frac{|P(E_1)\cap P(E_2)-(E_i+...+E_j)-(E_m+...+E_n)|+M_{\min}(i,j)+N_{\max}(m,n)}{|P(E_1)\cup P(E_2)-(E_i+...+E_j)-(E_m+...+E_n)|+M_{\max}(i,j)+N_{\max}(m,n)} \quad (12)$$

1) The analysis and proof of weight based distance formula

a) When E is an immediate entity

There is one entity E in both the property sets of E_1 and E_2. E may be invoked by E_1 or E_2,or oppositely.

We consume that E is the immediate entity of E_1, and that means $E=E_1$. The weight value of E in the property set of E_2 is defined as η, and $\eta>1$. So

$$WD(E_1,E_2)=1-\frac{|P(E_1)\cap P(E_2)-E|+\eta}{|P(E_1)\cup P(E_2)-E|+\eta}$$

①Monotony analysis: When η increases, $\eta'=\eta+\Delta\eta$ （$\Delta\eta>0$）, and

$$WD'(E_1,E_2)-WD(E_1,E_2) = \frac{|P(E_1)\cap P(E_2)-E|+\eta}{|P(E_1)\cup P(E_2)-E|+\eta}-\frac{|P(E_1)\cap P(E_2)-E|+\eta+\Delta\eta}{|P(E_1)\cup P(E_2)-E|+\eta+\Delta\eta} = \frac{(|P(E_1)\cap P(E_2)-E|-|P(E_1)\cup P(E_2)-E|)\cdot\Delta\eta}{(|P(E_1)\cup P(E_2)-E|+\eta)\cdot(|P(E_1)\cup P(E_2)-E|+\eta+\Delta\eta)}<0$$

So the distance between E_1 and E_2 decreases monotonically.

②Extreme value analysis: Theoretically the scope of η is $[1,+\infty)$, and η is one integer. When $\eta=1$, the weighted distance and non-weighted distance are the same. When $\eta\to+\infty, WD\to 0$.

b) When E is non-immediate entity

The weight value of E in property set E_1 is defined as η, and that in property set E_2 is λ. We consume that only η changes in value (monotonically increasing).

①Monotony analysis:

A. When $\eta<\lambda$ and the scope of η is not more than λ ($\eta+\Delta\eta<\lambda$),

$$WD'(E_1,E_2)-WD(E_1,E_2) = \frac{|P(E_1)\cap P(E_2)-E|+\eta}{|P(E_1)\cup P(E_2)-E|+\lambda}-\frac{|P(E_1)\cap P(E_2)-E|+\eta+\Delta\eta}{|P(E_1)\cup P(E_2)-E|+\lambda} = \frac{-\Delta\eta}{|P(E_1)\cup P(E_2)-E|+\lambda}<0$$

So the distance of E_1 and E_2 decreases monotonically.

B. When $\eta<\lambda$ but $\eta'=\eta+\Delta\eta>\lambda$,

$$WD'(E_1,E_2)-WD(E_1,E_2) = \frac{|P(E_1)\cap P(E_2)-E|+\eta}{|P(E_1)\cup P(E_2)-E|+\lambda}-\frac{|P(E_1)\cap P(E_2)-E|+\lambda}{|P(E_1)\cup P(E_2)-E|+\eta+\Delta\eta} = \frac{[|P(E_1)\cup P(E_2)-E|+|P(E_1)\cap P(E_2)-E|+\eta+\lambda]\cdot(\eta-\lambda)+(|P(E_1)\cap P(E_2)-E|+\eta)\cdot\Delta\eta}{(|P(E_1)\cup P(E_2)-E|+\lambda)\cdot(|P(E_1)\cup P(E_2)-E|+\eta+\Delta\eta)}$$

When this formula is equal to 0,

$$\Delta\eta=\frac{[|P(E_1)\cup P(E_2)-E|+|P(E_1)\cap P(E_2)-E|+\eta+\lambda]\cdot(\lambda-\eta)}{|P(E_1)\cap P(E_2)-E|+\eta}=\eta_0$$

Then analyze this formula:

When $\Delta\eta<\eta_0$, $WD'(E_1,E_2)-WD(E_1,E_2)<0$;

When $\Delta\eta>\eta_0$, $WD'(E_1,E_2)-WD(E_1,E_2)>0$.

It means that in the process where η increases across λ, the distance of E_1 and E_2 decreases first then increases till the end, and the separation point is η_0.

C. When $\eta > \lambda$

$$WD'(E_1,E_2) - WD(E_1,E_2)$$
$$= \frac{|P(E_1)\cap P(E_2) - E| + \lambda}{|P(E_1)\cup P(E_2) - E| + \eta} - \frac{|P(E_1)\cap P(E_2) - E| + \lambda}{|P(E_1)\cup P(E_2) - E| + \eta + \Delta\eta}$$
$$= \frac{(|P(E_1)\cap P(E_2) - E| + \lambda)\cdot\Delta\eta}{(|P(E_1)\cup P(E_2) - E| + \eta)\cdot(|P(E_1)\cup P(E_2) - E| + \eta + \Delta\eta)} > 0$$

②Extreme value analysis: η and λ are both the integer in the scope of $[1,+\infty)$, and when $\eta \to +\infty$, $WD \to 1$. And when η and λ are infinite of the same order, $WD \to 0$

After the analysis from the aspects of monotony and extreme value, there are two properties can be inferred from the analysis results.

V. EXPERIMENTS AND ANALYSIS

In this section we detect feature envy bad smell with the technique of improved distance metrics theory, and get the comparison between the traditional and weight based distance metrics theory. Java open source JFreeChart 1.0.13 is used to detect bad smells.

In JFreeChart open source there are 583 classes, including 3202 attributes and 7964 methods, the code lines are more then 90,000. Totally there are 289 distances which violate the distance principle in Section 4 from the simple distance metrics theory, with 299 from weight based distance metrics theory.

From the detecting results there are 289 simple distance results and 299 weight based ones.

In 289 simple distance results, there are 280 distance pairs which make the value of simple distance ratio and weight based distance ratio both more than 1. But the values of these two are not same completely, because of adding the calculation of multi-invoking information.

And 19 ditance pairs changed the simple distance ratio. Take the 281st distance pair line. The method *createStandardTickUnits* has the distance 0.896 towards its local class and 0.901 towards class *NumberAxis*. But actually *createStandardTickUnits* invokes the method *add* in class *TickUnits* for 57 times, so its weight based distance towards *TickUnits* is changed to 0.3, and the weight based distance ratio is more than 1. Therefore it may be feature envy bad smell. It can be seen as that weight based distance detection points out the false negative of simple distance detection.

And 9 distance pairs changed the weight based distance ratio. In the 300th line, *calculateLabelAnchorPoint* invokes the method *itemLabelAnchorOffset* from its own class *AbstractRenderer* for 40 times, and the weight based ratio is less than 1, so there is no bad smells. It is just the false positive of simple distance detection.

From the comparisons, the false positive rate is improved 3.0%, and the false negative rate is improved 6.4%. This indicates that the application of weight based distance metrics theory to detect feature envy bad smell is more accurately and effectively.

The other indicator is to detect time consuming. The computing space is n^2 when there are n entities. If there are 10,000 entities (actually more than 10,000 entities in JFreeChart codes), the number distance values will up to nearly 0.1 billion. The storage and computing of this large number of data is difficult.

VI. CONCLUDING REMARKS

In this paper the approach of computing and comparing the distance metrics between entities and classes is proposed to detect feature envy bad smells. With this approach the problem of low precision and speed in tradition bad smells detection which is visual and manual can be improved. The advantage is that after computing the distance metrics and distance metrics ratio between entities and classes, bad smells can be detected numerically and automatically, so the efficiency is increased.

The weight based distance metric is defined to manifest the multiple invoking relationships between each two entities. Based on this conception, the weight based approach is proposed to improve the problem of high fail positive and fail negative rates. This is caused by fail distinguishing different invoking frequencies among entities. The data of invoking times among entities is used in distance metrics computing, and entities invoking relationships with different invoking frequencies can be reflected by the distance metrics value quantitatively, which makes bad smells detection more accurately. The results of detecting bad smells in large scale source code indicate that the approach of computing weight based distance metrics decrease the fail passive and fail negative rates and increase the total precision compared with the traditional detection methods.

ACKNOWLEDGMENT

This research is supported by the National Natural Science Foundation of China under Grant No.61173021 and the Research Fund for the Doctoral Program of Higher Education of China (Grant No. 20112302120052 and 20092302110040).

REFERENCES

[1] Tom Mens, Tom Tourwe. A Survey of Software Refactoring. IEEE Trans. Software Eng., vol. 30, no. 2, Feb. 2004, pp.126-139.

[2] Martin Fowler. Refactoring: Improving the design of existing code. Addison-Wesley, 1999.

[3] B.F. Webster, Pitfalls of Object Oriented Development, first ed. M&T Books, Feb. 1995.

[4] A.J. Riel, Object-Oriented Design Heuristics. Addison-Wesley, 1996.

[5] M. Ma¨ntyla¨ , "Bad Smells in Software—A Taxonomy and an Empirical Study," PhD dissertation, Helsinki Univ. of Technology, 2003.

[6] W.C. Wake, Refactoring Workbook. Addison-Wesley Longman Publishing Co., Inc., 2003.

[7] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. McCormick III, and T.J. Mowbray, Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, first ed. John Wiley and Sons, Mar. 1998.

[8] G. Travassos, F. Shull, M. Fredericks, and V.R. Basili, "Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality," Proc. 14th Conf. Object-Oriented Programming, Systems, Languages, and Applications, pp. 47-56, 1999.

[9] Nikolaos Tsantalis, Alexander Chatzigeorgiou. Identification of Extract Method Refactoring Opportunities. European Conference on Software Maintenance and Reengineering. March,2009, pp.119-128.

[10] Limei Yang, Hui Liu, Zhendong Niu. Identifying Fragments to be Extracted from Long Methods. 16th Asia-Pacific Software Engineering Conference. December,2009, pp.43-49.

[11] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws," Proc. 20th Int'l Conf. Software Maintenance, pp. 350-359, 2004.

[12] M.J. Munro, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code," Proc. 11th Int'l Software Metrics Symp., F. Lanubile and C. Seaman, eds., Sept. 2005.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns—Elements of Reusable Object-Oriented Software, first ed. Addison-Wesley, 1994.

[14] E.H. Alikacem and H. Sahraoui, "Generic Metric Extraction Framework," Proc. 16th Int'l Workshop Software Measurement and Metrik Kongress, pp. 383-390, 2006.

[15] K. Dhambri, H. Sahraoui, and P. Poulin, "Visual Detection of Design Anomalies," Proc. 12th European Conf. Software Maintenance and Reeng., pp. 279-283, Apr. 2008.

[16] G. Langelier, H.A. Sahraoui, and P. Poulin, "Visualization-Based Analysis of Quality for Large-Scale Software Systems," Proc. 20th Int'l Conf. Automated Software Eng., T. Ellman and A. Zisma, eds., Nov. 2005.

[17] M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice. Springer-Verlag, 2006.

[18] E. van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," Proc. Ninth Working Conf. Reverse Eng., Oct. 2002.

[19] Naouel Moha, Yann-Gae¨l Gue´he´neuc, Laurence Duchien. DECOR: A Method for the Specification and Detection of Code and Design Smells. Software Engineering, IEEE Trancsactions on 2010 Jan, Volume 36 Issue 1, pp.20-36.

[20] Ladan Tahvildari, Kostas Kontogiannis. A Metric-Based Approach to Enhance Design Quality through Meta-Pattern Transformations. 7th European Conference Software Maintenance and Reengineering. March. 2003, pp.183-192.

[21] Mark O'Keeffe, Mel O'Cinneide. Search-based refactoring: an empirical study. JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE. August, 2008, pp.345–364.

[22] Jagdish Bansiya, Carl G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. IEEE Trans. Software Eng., January, 2002, pp.4-17.

[23] Bart Du Bois, Serge Demeyer, Jan Verelst. Refactoring—Improving Coupling and Cohesion of Existing Code. 11th Working Conf. Reverse Eng., Nov. 2004, pp.144-151.

[24] Olaf Seng, Johannes Stammel, David Burkhart. Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems. 8th Ann. Conf. Genetic and Evolutionary Computation. 2006, pp.1909-1916.

[25] Y. Kataoka, T. Imai, H. Andou, T. Fukaya. A Quantitative Evaluation of Maintainability Enhancement by Refactoring. 18th IEEE Int'l Conf. Software Maintenance. Oct. 2002, pp.576-585.

[26] F. Simon, F. Steinbru¨ ckner, and C. Lewerentz, "Metrics Based Refactoring," Proc. Fifth European Conf. Software Maintenance and Reeng., pp. 30-38, Mar. 2001.

[27] Mario Bunge. Treatise on Basic Philosophy. D. Reidel Publishing Company, Dordrecht-Holland, Volume 3, Ontology I, 1977.

[28] Frank Simon, Silvio Löffler, Claus Lewerentz. Distance based cohesion measuring. in proceedings of the 2nd European Software Measurement Conference (FESMA) 99, Technologisch Institute Amsterdam, 1999

[29] James M. Bieman, Byung-Kyoo Kang: "Measuring Design-Level Cohesion", IEEE Transactions on Software Engineering, Vol 24, Nr. 2, February 1998

[30] Ghezzi Carlo, Jazayeri Mehdi, Mandrioli Dino. Fundamentals of Software Engineering. Prentice Hall, 2003