DETECTION AND REFACTORING OF BAD SMELL CAUSED BY LARGE SCALE

Jiang Dexun[1], Ma Peijun[2], Su Xiaohong[3], Wang Tiantian[4]

School Of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
[1]negrocanfly@163.com, [2]silverghost192@163.cn
[3]Suxh@hit.edu.cn, [4]Wangtt@hit.edu.cn

ABSTRACT

Bad smells are signs of potential problems in code. Detecting bad smells, however, remains time consuming for software engineers despite proposals on bad smell detection and refactoring tools. Large Class is a kind of bad smells caused by large scale, and the detection is hard to achieve automatically. In this paper, a Large Class bad smell detection approach based on class length distribution model and cohesion metrics is proposed. In programs, the lengths of classes are confirmed according to the certain distributions. The class length distribution model is generalized to detect programs after grouping. Meanwhile, cohesion metrics are analyzed for bad smell detection. The bad smell detection experiments of open source programs show that Large Class bad smell can be detected effectively and accurately with this approach, and refactoring scheme can be proposed for design quality improvements of programs.

KEYWORDS

Distribution rule; Class length distribution model; Cohesion metrics; Bad smell detection; refactoring scheme

1. INTRODUCTION

Nowadays, with the development of software programming, the number of software analysis tools available for detecting bad smells significantly increase. Although these tools are gaining acceptance in practice, a lack of detection towards some bad smells may decrease the effectiveness, such as Long Method, Large Class and Long Parameter List [1]. The key of these bad smells is about the structure and components.

Software programs are composed of components from every level. The component from higher level is composed of ones from lower level. Characters compose keywords, while keywords, variables and operators compose statements. The composition level of object-oriented programs is shown in Figure 1.

DOI : 10.5121/ijsea.2013.4501

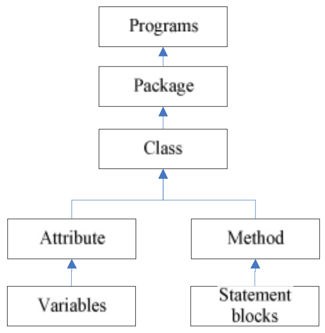


Figure 1. Composition level of object-oriented programs.

Large Class [1] bad smell is one classical bad smells, meaning a class is too large. The cause of large classes may be the large number of instance variables or methods. Large Class has long history, but the detection is always vague. From the definition [1], this bad smell detection should be achieved by the class length statistics. Usually the class length is measured by the lines of code. In practice it is difficult to confirm a threshold value for detecting one particular class is too large or not. So it is also difficult to detect Large Class bad smell particularly in business open source programs. The fixed threshold value is not fastidious for Large Class bad smell detection.

In this paper a detecting method of Large Class bad smell is proposed based on scale distribution. The length of all the classes in one program is extracted, and then distribution model of class scale is built using the length of these classes. In distribution model the groups which are farthest from the distribution curve is considered to be candidate groups of Large Class bad smell. Furthermore, the cohesion metrics of the classes in these groups are measured to confirm Large Class.

The rest of the paper is organized as follows. Section 2 presents a short overview of related work. In Section 3, the class length distribution model is built to present the distribution rules of class length. With this model and cohesion metrics presented, the detection method of Large Class bad smell is proposed in Section 4. Section 5 discusses how to give proper refactoring scheme. And Section 6 shows the experiment results. The conclusion is provided in Section 7.

2. Related Work

In the past decades, a number of studies were conducted for bad smells of programming codes. Webster [2] introduced smells in the context of object-oriented programming codes, and the smells sorted as conceptual, political, coding, and quality assurance pitfalls. Riel [3] defined 61 heuristics characterizing good object-oriented programming that enable engineers to assess the quality of their systems manually and provide a basis for improving design and implementation. Beck Fowler [1] compiled 22 code smells that are design problems in source code, and it is the basis of suggesting for refactorings.

Travassos et al. [4] introduced a process based on manual inspections and reading techniques to identify smells. But manual detection of bad smells is one time-consuming process, and is easy to mistake. So researchers pay more attention in automatic detection. Marinescu [5] presented a metric-based approach to detect code smells with detection strategies, implemented in the IPLASMA tool. Tahvildari and Kontogiannis [6] 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. A limitation of their approach is that it indicates the kind of the required transformation but does not specify on which specific methods, attributes, or classes this transformation should apply (this process requires human interpretation). O’Keeffe and O’Cinneide [7] treated object-oriented design as a search problem in the space of alternative designs. This is application of search-based approaches to solving optimization problems in software engineering. Bad smell detecting by metric needs to be selected proper metrics and the judging threshold should be predetermined.

Visualization techniques are used in some approaches for complex software analysis. 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 [8, 9]. 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 [10, 11]. But visual detecting results need manual intervention.

Some bad smells relevant to cohesion can be detected using distance theory. Simon et al. [12] defined a distance-based metric to measure the cohesion between attributes and methods. The inspiration about the approach in this paper is drawn from the work [12] in the sense that it also employs the Jaccard distance. However, the approach has proposed several new definitions and processes to get improvements. The conception of distance metrics is defined not only among entities (attributes and methods) but also between classes. In [13], the distances between entities and classes are defined to measure the cohesion among them. The bad smell detection with distance theory needs more calculation. In this paper the equation of distance between one entity and one class has been used for computing the cohesion degree of one class.

There is less research about bad smell detection of Large Class. Liu et al [14] proposed a detection and resolution sequence for different kinds of bad smells to simplify their detection and resolution, including Large Class bad smell. But Liu paid more attention to the schedule of detection rather than Large Class detection itself, and the specific detecting process was not provided in the paper.

In Large Class bad smell detection, class size measures have been introduced. When class size is large, it is seen as Large Class. In bad smell detection tools, the main way [15] of measuring class size is to measure the number of lines of code, i.e. NLOC, or the number of attributes and methods. PMD[16] and Checkstyle[17] both use NLOC as detection strategy. The former uses a threshold of 1000 and the second a threshold of 2000. The fixed threshold value is not fastidious for Large Class bad smell detection, and easy to cause false detection. And in these tools, there is no function about refactoring of Large Class bad smell.

These researches above show that, the detection of Large Class bad smell is based on fixed threshold comparison. Since the fixed threshold is selected manually, the objectivity is low. Moreover, the refactoring method is decided manually, and there is no suggestion or scheme about that.

3. THE DISTRIBUTION OF CLASS LENGTH

3.1. Class length distribution appearance

In object-oriented programs there are a large number of classes. The length of these classes is not the same. In this paper, it is declared that if the length of one class is larger than the average length of the program, it is called larger class, or smaller class.

There are some programs with more classes are the larger classes, while others are the opposite. This depends on the different function programs should be achieved. And this is also relevant to different coding habits and programming styles of developers. From Table1 it is seen that class length statistics of some open source programs is listed.

Table 1. Class length statistics of open source programs.

Program	**Number of Class**	**Average Length**	**Larger Class**	**Smaller Class**
HSQLDB-2.2.7	111	500	29.73%	70.27%
Tyrant-0.96	117	101	27.35%	72.65%
Tyrant-0.334	262	169	27.10%	72.90%
swingWT-0.60	44	41	22.73%	77.27%
Trama	16	249	25%	75%
ArgoUML	1874	91	19.80%	80.20%
Spring Frk	1531	57	27.69%	72.31%
Azureus_Vuze4812	1597	129	28.18%	71.82%

In Large Class bad smell detecting, the usage of fixed value threshold may cause mistakes: the detection results of some programs (such as HSQLDB-2.2.7 in Table 1) are that most of classes are “too large”, and from the results of other programs (such as Spring Frk) there is no Large Class bad smell at all. Besides that, actually the value of fixed threshold is set manually, with the lower objectivity.

Because of the programs with different coding habits and programming styles, the detection result of Large Class bad smell with the fixed value threshold is inaccurate and less persuasive. In Table 1, the percentage of large classes in programs is lower, and the ratio of larger classes and smaller ones is between 1:4 and 3:7.

Functionally, common classes usually are designed to be small and easy to use, particularly for the frequently used ones. Oppositely, large scale classes are designed for complex functionality and computing algorithm. But in programs there are more classes which are simple and common, and complex classes are less. So for the class length statistics, smaller classes are majority, and larger ones are minority.

Additionally, in the step of functional design some classes have been designed to achieve certain functions but these classes are just created but not coded completely. This situation is obvious particularly in multiple versions comparison of program design. Maybe these classes only contain some member variables, comments, or even just class names themselves. This kind of unfinished classes may cause minority smaller classes. From the analysis of statistics and program design, the numerical comparison relationship of larger classes and smaller classes would be clear.

Above all, one conjecture is proposed in this paper. Conjecture: the class length statistics of programs confirm to certain distribution rule. And this distribution rule should be verified in programs statistics.

3.2. The verification of certain distribution conjecture

The process of curve fitting about the statistics data of class length is shown in Figure 2.

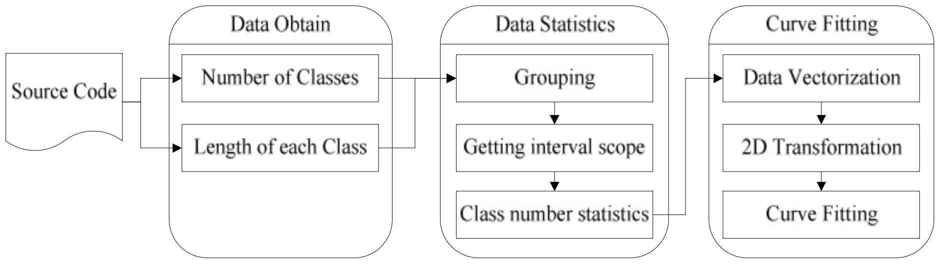


Figure 2. Process of class length statistics curve fitting.

3.2.1 Obtain the data

Get the data about the number of classes, the length of each class. The class length is measured by lines of code. n is the number of classes in the program. The length of class C_i is defined as A_i, and $i = 1,2,\ldots, n$.

3.2.2 Data statistics

Grouping

According to Sturges Equation, the classes need to be grouped. The Sturges Equation is

$$N = 1 + 3.32 \cdot lg\, n \qquad (1)$$

n is the number of classes. With Equation (1), the classes are divided into N groups, named as G_i, $i = 1,2,\ldots, N$

Getting interval scope

Get the maximum value A_{max} and minimum value A_{min} of each class's length, and the span X. The interval $[A_{\min}, A_{\max}]$ is divided into N parts, and the length of sub interval is $m = X / N$. So the interval of group G_i is $[A_{\min} + (i-1) \cdot m, A_{\min} + i \cdot m]$, $i = 1,2,\ldots, N$.

Class number statistics

The number of classes in group G_i is defined as P_i, and the statistics algorithm is shown as Figure 3:

```
Algorithm：Class number statistics
Input：$G_i$
Output：$P_i$,
Begin
 Foreach（ $G_i$ ）
   Foreach（ $j = 1,2,\ldots N$ ）
   If（ $A_i \in [A_{\min} + (j-1)\cdot m, A_{\min} + j \cdot m]$ ）
    $P_i$++;
    EndIf
   EndFor
 EndFor
End
```

Figure 3. Class number statistics algorithm.

After the algorithm the vector P_i is valued.

3.2.3 Curve fitting

Graphical vector

The number i of group interval is defined as the data of x axis, and P_i is defined as the data of y axis. So a series of points is created in the rectangular coordinates to represent the class length statistics.

Curve Fitting

According to the point set of class length statistics in the rectangular coordinates, get one curve with the least value of Mean Squared Error (MSE). The process of curve fitting is executed with all types of statistical curves.

After the curve fitting of the class length statistics from a large number of open source programs, the Exponential curve is found to be the optimal fitting curve defined as

$$y = y_0 + A \cdot e^{R_0 \cdot x} \qquad (2)$$

Through the statistics data obtainment of large amount of programs, the residuals threshold T is calculated. The value of residuals threshold T is the average of each group MSE in open source programs curve fitting. This residuals threshold is used for bad smell detection. With the class length data statistics of programs to be checked, if the residual R_i of group i is larger than the residuals threshold, there is Large Class bad smell in this group, and the bad small classes in this group is $R_i - T$.

4. BAD SMELL DETECTION

Usually the quarantine programs are open source programs which contain a large number of classes. In the detection method, the inputs are the codes, and the outputs are the bad smell classes.

4.1. Bad smell location in group

Classes are divided with their length by Sturges Equation, and the result is created in a dimensional vector $P = \{ P_1, P_2, \ldots P_N' \}$. And this vector P is fitted with Exponential curve in the rectangular coordinates. The optimal fitting curve with least value of MSE is

$$y = y_0' + A \cdot e^{R_0' \cdot x} \qquad (3)$$

After curve fitting, the positive residual R_i^+ is:

$$R_i^+ = P_i - y_i \qquad (4)$$

Where P_i is the number of classes in group G_i, and y_i is the value of Equation (3) curve in i place. If $R_i^+ > T$, there are bad smell classes in group G_i, and the number N_i of bad smell classes is computed in Equation (5).

$$N_i = R_i^+ - T \qquad (5)$$

4.2. Bad smell location in class

As the bad smell group location above, the bad smell groups may not be the largest groups. Similarly, the identifying method is not to simply select the x largest classes. So it is the key of Large Class bad smell detection: the detecting basis is not from the metrics of destination class itself (length or others), but from metrics of all the classes.

In this paper, the bad smell location in class is identified with the inner cohesion of classes. The cohesion metric is defined with the entity distance theory. In entity distance theory, these concepts should be defined.

Definition 1 (Entity): the entity is the attribute a or the method m in one class, which is signed as E.

Definition 2 (Property Set): the property set is the set of entities which have invoking-relations with the given entity E, and it is signed as $P(E)$. If one method uses (accesses/calls) one attribute or another method, they two have invoking-relations with each other.

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.

Definition 3(Distance): the distance value $Dist(E_1, E_2)$ of entity E_1 and E_2 is

$$Dist(E_1, E_2) = 1 - \frac{|P(E_1) \cap P(E_2)|}{|P(E_1) \cup P(E_2)|} \qquad (6)$$

Where $|P(x)|$ is the member count of $P(x)$, and the distance between entity e and class C is the average of the distances between e and every entity in C :

$$\overline{D}(e,C) = \begin{cases} \frac{1}{|E_C|-1} \times \sum_{y_i \in C} Dist(e, y_i) & e, y_i \in E_C \\ \frac{1}{|E_C|} \times \sum_{y_i \in C} Dist(e, y_i) & e \notin E_C, y_i \in E_C \end{cases} \qquad (7)$$

Where E_C is the set of entities C contains.

Definition 4(Cohesion Metric): the Cohesion Metric value is the rate of the average of the distance of entities out of the class and those in the class.

$$Conesion_C = \frac{\frac{\sum_{e_i \notin C} Distance(e_i, C)}{|e_i \notin C|}}{\frac{\sum_{e_i \in C} Distance(e_i, C)}{|e_i \in C|}} \qquad (8)$$

If the cohesion metric value is smaller, the degree of cohesion is lower. So with the x smallest cohesion metric value, these classes are identified to Large Class bad smell.

5. REFACTORING SCHEME

In this section the classes which are sure to have Large Class bad smell is refactored. And the refactoring process is Extract Class, which means the destination class should be divided into two or more new classes. In practice, the destination class would be divided into two parts, and the bad smell detection would be executed again.

The basic idea of refactoring scheme is to divide the entities in the destination class based on the cohesion degree among them. So the key ideas are how to represent cohesion degree between entities in classes and how to cluster entities in classes.

5.1. Cohesion degree representation of entities in class

The cohesion degree is represented as the distance between two entities. The distance value of entity E_1 and E_2 is shown in Equation (5). Before clustering, all the distances between each two entities in the destination class should be computed accurately. The lower distance value is, the higher the cohesion degree is.

5.2. Entities clustering algorithm

The agglomerative clustering algorithm [18] (which is a hierarchical clustering algorithm) is used in this paper. The process is given below:

1) Assign each entity to a single cluster, and the distance value of each two cluster is the distance of the two entities;
2) Repeat merging until the total cluster number reduces to 2. And the considered merging criterion is to merge two clusters with the lowest distance value. After merging once, the distance to the new merging cluster is the average of those to last clusters.
3) Output the two clusters (each of them contains several entities).

The agglomerative clustering algorithm is given in Figure 4:

```
Algorithm：Agglomerative Clustering Algorithm
Input：ench entities and their distance
Output：two new clusters
Begin
 each entity is assigned to be a single cluster;
 While(clustering number is more than 2)
   merge two clusters A, B with the lowest distance value as cluster C;
   Foreach（any other cluster X in the class）
     Dist（C，X）=Avg（Dist（A，X），Dist（B，X））;
   EndFor
 EndWhile
```

Figure 4. Agglomerative clustering algorithm of refactoring.

After the algorithm, according to the two new clusters, Extract Class operation would be executed as refactoring.

6. EXPERIMENTAL RESULTS

In this paper several Java open source programs are used to detect Large Class bad smells. The names of these programs are shown is Table 2:

Table 2. Open source programs in Large Class bad smell detection.

Program name	Number of classes
HSQLDB 2.2.4	111
Tyrant 0.96	116
Swing WT 0.60	44
Trama	16
ArgoUML	1874
JFreeChart 1.0.13	504

6.1. Large Class bad smell location in group

If all the groups of statistics data have high fitting degree (through threshold comparison) after detection, there is no Large Class bad smell at all. And sometimes the positive residual is less than 0, so it is detected to be no bad smell.

The results of Large Class bad smell group location towards the programs in Table 2 is shown in Table 3:

Table 3. Results of Large Class bad smell group location.

Group	**HSQLDB2.2.4**	**Tyrant0.96**	**SwingWT0.60**	**Trama**	**ArgoUML**	**Soul3.0**
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0		0	0
7	0	0	3		0	4
8	0	2			0	0
9					1	0
10					1	0
11					0	
12					2	

In Table 3, the nonzero digit N_i means the existence of Large Class bad smell in its group, which is computed in Equation (5). And the value means the number of Large Class bad smell in the group. If N_i is equal to zero, there is no Large Class bad smell at all in G_i group.

Besides, different programs have different numbers of class, so the group number of each program is different with each other. So maybe there is no digit in N_i position. HSQLDB2.2.4 has only 8 groups, so they are blank spaces in group 8 to group 12. The ArgoUML program has 12 groups, which is more than any others.

6.2. Large Class bad smell class location in class

The cohesion metrics of classes in group location are computed with the Equation (8) to detect which class/classes have bad smell.

As the location method proposed in Section 4.2, N_i classes were detected as Large Class bad smell with smallest cohesion metrics. Table 4 shows the cohesion metrics of group 8 class members of Tyrant0.96 program. With this, the classes Creature and GameScreen are both located to be Large Class.

Table 4. Cohesion metrics of group 8 class members of Tyrant0.96 program

Class name	**Number of lines**	**Cohesion metric**
Creature	898	5.763
GameScreen	625	3.125
Map	788	12.061

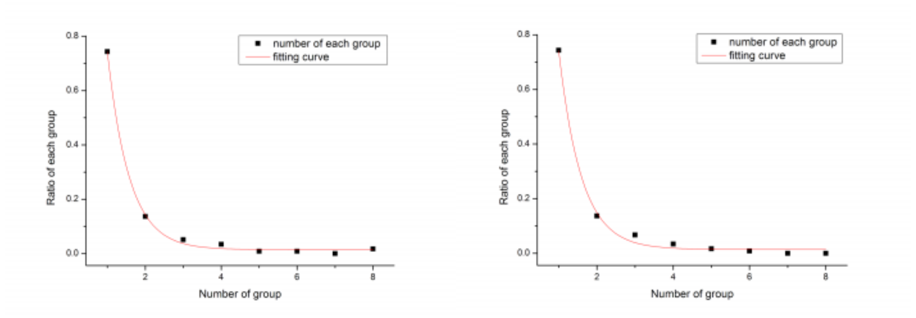
Table 5 shows the cohesion metrics of group 7 class members in JFreeChart1.0.13 program. After the cohesion metrics computing and analysis, the classes AbstractRenderer, PiePlot, CategoryPlot and ChartPanel are identified as Large Class.

Table 5. Cohesion metrics of group 7 class members of JFreeChart1.0.13 program

Class name	**Number of lines**	**Cohesion metric**
AbstractRenderer	1879	4.932
PiePlot	1725	5.375
DatasetUtilities	1808	5.864
ChartPanel	1642	3.509
DateAxis	1752	12.826

6.3. Code refactoring results

The classes with bad smell should be refactored by Extract Class according to the entities distance and agglomerative clustering algorithm. After refactoring the programs should be test again. Figure 5 shows the test results of Tyrant0.96 before and after refactoring.



(a) Fitting curve before refactoring (b) Fitting curve after refactoring

Figure 5. Comparisons on the results of Tyrant0.96 before and after refactoring.

In Figure 5(a), MSE of the data is 0.01810, and that is 0.01037 in Figure 5(b). In Figure 5(b), the curve has better approximation than that in (a). The MSE is less than threshold, so the refactoring is effective and there is no Large Class bad smell at all.

6.4 Comparisons with PMD and Checkstyle tools

In the section of related work, the refactoring tools PMD and Checkstyle are introduced. PMD and Checkstyle have the ability for Large Class bad smell detection, and no refactoring operation suggestion. As mentioned, in these tools, if the line number of one class is higher preset threshold, the class is detected as Large Class. The threshold of PMD for Large Class is 1000, and the threshold of Checkstyle is 2000. But PMD and Checkstyle cannot provide refactoring schemes for existing Large Class bad smells. The detecting results from these two refactoring tools are compared with the approach in this paper.

The results of comparison are shown in Table 6. After manual confirmation, the precision comparisons of the methods in this paper and PMD & Checkstype are displayed in Table 7.

Table 6. Cohesion metrics of group 7 class members of JFreeChart1.0.13 program

Detection tools & methods	**Tyrant0.96**		**JFreeChart1.0.13**	
	Large Class Number	Large Class Name	Large Class Number	Large Class Name
Method in this paper	2	Creature GameScreen	4	AbstractRenderer, PiePlot, DatasetUtilities, ChartPanel
PMD	0	--	12	AbstractRenderer, PiePlot, DatasetUtilities, ChartPanel, DateAxis, ChartFactory, AbstractXYItemRenderer, ContourPlot, ThermometerPlot, AbstractCategoryItemRenderer XYPlot, CategoryPlot
Checkstyle	0	--	2	XYPlot, CategoryPlot

Table 7. Precision comparisons of the methods in this paper and PMD & Checkstype

Program	**Bad smell detection Precision (%)**			**Refactoring scheme Precision (%)**
	PMD	Checkstyle	This paper method	This paper method
Tyrant0.96	--	--	100	100
JFreeChart1.0.13	33.33	0	100	100

"--" means that the precision rate cannot be computed.

From the comparison in Table 7, the method in this paper is much better than the existing Large Class bad smell detection tools.

In small scale programs the classes are general small, the potential probability of Large Class is low, and vice versa. The CLDM is more suitable for larger scale programs Large Class bad smell detection and refactoring schemes. Small scale programs have less Large Class, so the false positive rate of CLDM is higher.

In addition, because of the different threshold, the detecting Precision and Recall of PMD and Checkstyle are not the same in different scale programs. But it is not sure which threshold is more effective for all the programs.

7. CONCLUSION

In this paper the approach of Large Class bad smell detection and refactoring scheme has been proposed. Fixed-threshold-based detection method is analyzed to be rigid and error-prone. And a new model is developed to describe the statistic distribution of class length. In this model, the class groups that are far away from the distribution curve are treated as containing bad smells potentially. And combining with cohesion metric computing, the bad smell classes are confirmed in the class groups. After using Agglomerative Clustering Technique, the scheme of Extract Class is proposed for refactoring.

The contributions of this paper are as follows. First, the characteristics of Large Class bad smell are quantified with statistical analysis. Second, the length and cohesion metrics based approach is proposed for Large Class bad smell detection.

ACKNOWLEDGEMENTS

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] M. Fowler, (1999) “Refactoring: Improving the design of existing code”, Addison-Wesley, pp89-92.

[2] B.F. Webster, (1995) “Pitfalls of Object Oriented Development”, first M&T Books, Feb.

[3] A.J. Riel, (1996) “Object-Oriented Design Heuristics”, Addison-Wesley.

[4] G. Travassos, F. Shull, M. Fredericks, & V.R. Basili., (1999) “Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality”, Proceeding of 14th Conference in Object-Oriented Programming, Systems, Languages, and Applications, pp47-56.

[5] R. Marinescu, (2004) “Detection Strategies: Metrics-Based Rules for Detecting Design Flaws”, Proceeding of 20th International Conference in Software Maintenance, pp350-359.

[6] Ladan Tahvildari & Kostas Kontogiannis, (2003) “A Metric-Based Approach to Enhance Design Quality through Meta-Pattern Transformations”, 7th European Conference Software Maintenance and Reengineering, pp183-192.

[7] M. O’Keeffe & M. O’Cinneide, (2008) “Search-based refactoring: an empirical study”, Journal of software maintenance and evolution: research and practice,pp345-364.

[8] K. Dhambri, H. Sahraoui & P. Poulin, (2008) “Visual Detection of Design Anomalies”, Proceeding of 12th European Conference in Software Maintenance and Reeng, pp279-283.

[9] G. Langelier, H.A. Sahraoui & P. Poulin, (2005) “Visualization-Based Analysis of Quality for Large-Scale Software Systems”, Proceeding of 20th International Conference in Automated Software Engineering , pp214-223.

[10] M. Lanza & R. Marinescu, (2006) “Object-Oriented Metrics in Practice”, Springer-Verlag. pp125-128.

[11] E. van Emden & L. Moonen, (2002) “Java Quality Assurance by Detecting Code Smells”, Proceeding of 9th Working Conference in Reverse Engineering, pp120-128.

[12] F. Simon, F. Steinbruckner C. Lewerentz, (2001) “Metrics Based Refactoring”, Proceeding of 5th European Conference in Software Maintenance and Reengineering, pp30-38.

[13] D.X. Jiang & P.J. Ma, (2012) “Detecting Bad Smells With Weight Based Distance Metrics Theory”, Proceeding of 2nd International Conference on Instrumentation, Measurement, Computer, Communication and Control, pp299-304.

[14] H. Liu, Z.Y. Ma & W.Z. Shao, (2012) “Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort”, IEEE Transactions on Software Engineering, Vol. 38, No. 1, pp220-235.

[15] D. Fontana, A. Francesca & P.Braione, (2012) “Automatic detection of bad smells in code An experimental assessment”, Journal of Object Technology, Vol. 11, No. 2, pp1-38.

[16] http://pmd.sourceforge.net.

[17] http://checkstyle.sourceforge.net.

[18] J.W. Han & M. Kamber, (2005) “Data Mining Concepts and Techniques”, Morgan Kaufmann Publishers.