Asian Journal of Information Technology 15 (17): 3205-3216, 2016
ISSN: 1682-3915

Detecting Shotgun Surgery Bad Smell Using Similarity Measure Distribution Model

[1]G. Saranya, [1]H. Khanna Nehemiah, [2]A. Kannan and [3]S. Vimala
[1]Ramanujan Computing Centre
[2]Department of Information Science and Technology, College of Engineering Guindy,
Anna University, 600025 Chennai, India
[3]Department of Computer Science and Engineering, Tamil Nadu, India

Abstract: Bad smells are the symptoms of code decay which leads to the severe maintenance problem. Shotgun surgery is a smell where a change in a class may cause many small changes to other different classes. There are several approaches that identify bad smells based upon the definition of rules and change history information. These rules are the combination of software metrics and threshold values which sometimes not able to detect the code decay such as shotgun surgery. Since, it is difficult to find the best threshold value for rule based detection and also finding the best combination of metrics from the historical information seems to be difficult. To detect the shotgun surgery, the co-change should be feasible. In that case, the need for having sufficient history of observable co-changes, without which the approach of change history is not possible. Therefore, these techniques cannot report the accurate instance of smells to detect the shotgun surgery bad smell. So as an alternate, in this study, a framework similarity measure distribution modelfor detecting shotgun surgery bad smell for object oriented program without the need for change history information is proposed. The framework is experimented on HSQLDB, TYRANT, XERCES-J and JFREE CHART open source software. To enable the detection of Shotgun surgery certain class files of the software under experimentation are modified. The results obtained through this framework are compared with the results obtained from the bad smell detection tools namely, inFusion and iPlasma in terms of precision and recall.From the results it is inferred that the shotgun surgery can be detected more accurately using this proposed approach.The proposed framework improves the maintainability by detecting the bad smell shotgun surgery.

Key words: Software maintenance, bad smell detection, similarity measure, distribution model, sturge's rule, frequency histogram

INTRODUCTION

Software maintenance is one of the important phase in the software life cycle. Maintenance of software has become a major challenge due to changing requirements. To adapt to changing requirements, programmers modify or add new functionalities to the existing software system. This results in poor design and violation of the object oriented design principles such as data abstraction, modularity and encapsulation (Riel, 1996) which serves as a cause for bad smell occurrence. The presence of bad smells indicates that there are issues with code quality, such as understandability and changeability (Yamashita and Moonen, 2012). The name bad smell (code smell) was first used by Kent Beck (Fowler 1999). Bad smells makes program modification and reuse more complex which affects the software quality (Dexun *et al*., 2013). There are different types of bad smells in software programs. Fowler (1999) presented twenty two types of code smells namely; comments, speculative generality, long method, lazy class, large class, switch statements, alternative classes with different interfaces, long parameter, temporary field, inappropriate intimacy, data clumps, divergent change, data class, refused bequest, feature envy, shotgun surgery, duplicate code, message chains, primitive obsession, parallel inheritance hierarchies, dead code and middle man. Traditionally, inspection of bad smell was done manually for large systems and it is a time consuming process for programmers to detect the bad smell. According to Fowler *et al*. bad smells can be removed using refactoring. Studies show that bad smells hinder clarity (Abbes *et al*. 2011) and possibly increase change and fault proneness (Khomh *et al*. 2012; Palomba *et al*., 2013). Also, the interaction between bad smells in different classes/ methods can negatively affect maintain ability (Yamashita *et al*., 2013). Hence, the bad smells have a negative impact on software evolution; it should be

Corresponding Author: G. Saranya, Ramanujan Computing Centre, College of Engineering Guindy, Anna University, 600025 Chennai, India

carefully monitored and removed through refactoring operations (Palomba *et al*., 2013). Although, many tools are practically used to detect the bad smell, there is still a lack in the detection of bad smells in code.

Shotgun Surgery is a bad smell where, a change in a class may cause many small changes to other different classes (Fowler, 1999). Shotgun Surgery bad smell can also take the form of a piece of code which is replicated repeatedly in various methods, belonging to various classes which might otherwise not give the impression of being coupled to each other. This happens when the programmers use duplicate copies of code. The methods in a class affected by the shotgun surgery have many design entities dependent on it. If a change is implied on such inter dependent methods, the programmer is required to make all cascading changes. Due to system-wide dispersion of changes and high amount of coupled entities, there is a risk of missing required changes which may cause maintenance problems (Olbrich *et al*., 2009). In the existing work, the majority of the work is focused on bad smell detection some which is based on the rule based and historical change information (Palomba *et al*., 2013, 2015). For each bad smell, the rules are based on the combination of metrics and threshold. To find the best threshold value for the rule based detection and finding the best combination of metrics from the historical information is difficult.

To overcome the above mentioned limitations, in this study, shotgun surgery bad smell is detected using similarity based distribution model which does not rely on the rule and change history information. Using similarity measure the methods replicated throughout the program is identified and a similarity matrix is constructed. From the similarity matrix the frequency histogram is generated and from the histogram residual threshold is computed to detect the shotgun surgery bad smell. The framework similarity measure distribution model was evaluated on four open source software's namely, HSQLDB, TYRANT, XERCES-J and JFREE CHART, respectively. When comparing the framework performances in terms of precision and recall against the tools inFusion and iPlasma, it is observed that the similarity measure distribution model tends to provide good performances both in precision and recall, i.e., the framework is able to detect more code smell that the tools omit.

Literaure review: This section analyzes the introduction and various detection approaches for detecting bad smell in source code is discussed. Detecting bad smells and refactoring of bad smells in programs is a challenging task faced during the maintenance phase. Fowler (1999) introduced code smells that are design defects in source code. Fowler (1999) presented twenty types of bad smells without classifying it. Since, there are more number of bad smells and they are closely related to each other Mika and Mantyla (2003) presented a classification of bad smells. Their classification was more understandable and can identify the relationship between the smells. Travassos *et al*. (1999) introduced reading technique for detecting bad smell which is a manual detection process and no attempt was made to automate this. The limitation of this technique is manual detection of bad smell is time consuming and the results are inaccurate. Instead of detecting bad smell manually, Marinescu (2004) proposed a metric-based approach to detect bad smells with detection strategies. The strategies capture deviations from good design principles and heuristics. However, there are some limitations in their detection strategies which have no justification for choosing the metrics, thresholds and the combination of the metrics and threshold.

To overcome these limitations, Munro (2005) defined a combination of conventional metrics to identify the bad smell. In their work, a framework to extract the main characteristics of bad smell design problems and interpretation rules that use existing and extended metrics to identify candidates in the results are defined. They presented the evaluation results of an automatic detection of two bad smells namely lazy class and temporary field. The technique used in this study is applicable to small software system and cannot perform well for large scale software systems. Marinescu (2003) and Munro (2005) used metrics based approaches which are insufficient to detect bad smell because metrics cannot expose important structural and semantic properties.

Sahraoui have presented an approach for large scale software systems based on visualization. They claim that the automatic analysis is poorly understood and the manual analysis is slow and incorrect; they have defined a semi automatic approach based on visualization for large scale software. Eva presented an approach that uses full automatic analysis to detect the bad smell and visualization technique to display the result which is done manually. However, human interpretation in both the approaches is a time consuming process.

Moha *et al*. (2010) proposed three contributions related to code and design smell. First, Moha *et al*. (2010) proposed a method named DECOR which describes all necessary steps to detect code and design smell. Second, a detection technique DETEX was instantiated to detect

the bad smell at a high level of abstraction and finally the DETEX technique was validated with the help of precision and recall. Their proposed work lacks in handling uncertainty in deciding whether the class is antipatterns or not.

Liu *et al.* (2012) presented a detection and resolution sequence for different kinds of bad smell including large class bad smell. This evaluation was performed on two open source software, namely, Java Source Metric and Thout Reader and the results were validated. In their work, they focused more on the schedule of detection rather than the large class bad smell detection and the detecting process was not clear in their work. Dexun *et al.* (2013) proposed a detection and refactoring method for large class bad smell which is based on scale distribution. All classes were extracted in one program and its length is measured using average length of the program, i.e., if the length of one class is greater than the average length of the class then it is the large class. Then the distribution model of class scale is built based on the length of these classes. Moreover, the cohesion metrics are measured to confirm large class. Then, using agglomerative clustering algorithm the extract class refactoring operation is performed. For clustering algorithm, the distances between the entities are calculated using the cohesion degree. This method is applied to open source software JFreeChart and the results are analyzed. In their work, the distribution rule used to confirm the length of the class is not defined properly.

Jiang *et al.* (2014) proposed distance metrics and K-nearest neighbor algorithm to detect divergent change bad smell. In their research, the dependency between the entities is defined using the distance metrics theory. The distance metrics values give the relationship between the entities. Then K-nearest neighboring algorithm is used to detect the bad smell. The value of K is dynamic in this approach. This approach has been applied in HSQLDB and Tyrant open source software's and the result is analyzed. Rao proposed a design change propogation probability matrix approach for detecting shotgun surgery and divergent change bad smell. The matrix is constructed using the unified representation of artifacts graph. Using the DCPP matrix the bad smells shotgun surgery and divergent change is detected. This matrix method has lower computation complexity.

Tsantalis and Chatzigeorgiou (2009) proposed a methodology for the identification of move method refactoring opportunities that provide a way for solving Feature Envy bad smell. In this study, the notion of distance between an entity and a class is employed to support the automated identification of feature envy bad smell and algorithm has been developed to extract the move method refactoring suggestions. This study clearly indicates which method and to which class they should be moved. It is evaluated on large scale open source projects namely, JFree Chart, JEdit, JMol and Diagram.

Oliveto *et al.* (2010) has introduced a new approach based on numerical analysis technique using B-splines for the identification of antipatterns. This approach has been illustrated on the blob and compared with DECOR and the approach based on Bayesian Beliefs Network. They classify classes strictly as being or not antipatterns and thus cannot report accurate information for borderline classes.

Bavota *et al.* (2013) proposed a new technique for automatic re-modularization of packages using structural and semantic measures to decompose a package into smaller or more cohesive package. The result of this approach indicates that the decomposed packages have low coupling and high cohesion. The structural and semantic measure is used in this study for the feature extraction of the bad smell shotgun surgery.

Tufano *et al.* (2015) conducted a study in 200 open source projects from different software ecosystems and investigated when bad smells are introduced by developers and the circumstances and reasons behind their introduction. The results of the approach contradict common wisdom stating that smells are being introduced during evolutionary tasks.

Kessentini *et al.* (2010) proposed an approach for the automatic detection of potential design defects in code. The detection is based on the notion that the more code deviates from good practices, the more likely it is bad. Instead of characterizing each symptom of each possible defect type, here, the principle of biological immune systems has been applied to identify what is abnormal. The more something is different, the more it is considered risky. In this work, they looked only at the first step of immune systems: the discovery of risk. But the identification and correction of detected design defects (refactoring) were not implemented.

Sahin *et al.* (2014) proposed a code smell detection rules as a bi-level optimization problem. In this the upper level problem generates a set of detection rules and combination of quality metrics. The lower level maximizes the number of generated artificial code smells that cannot be detected by the rules produced by the upper level. The main advantage of our bi-level formulation is that the generation of detection rules is not limited to some code smell examples identified manually by developers which are difficult to collect but it allows the prediction of new code smells behavior that are different from those in the base of examples.

Palomba *et al.* (2015, 2013) proposed an approach HIST to detect five different code smells by exploiting change history information mined from versioning systems. The main disadvantage of HIST is there is a need for having sufficient history of observable co-changes, without which the approach falls short. The results indicate that HIST is able to identify code smells that cannot be identified by competitive approaches.

Serban (2013) presented a case study based on the Shotgun Surgery design flaw detection. The detection approach is based on software metrics and Fuzzy Divisive Hierarchical Clustering (FDHC). Camelia applied the framework on an open source software namely log4net. Camelia compared the framework with the similar approach by Marinescu (2003). Camelia achieved all the suspected entities affected by the shotgun surgery bad smell like Marinescu except one entity due to the change of class metric values.

In addition to the detection techniques, there are many tools to detect the bad smell. Fontana *et al.* (2012) analyzed and compared the bad smell detection tools namely, Checkstyle, inFusion, iPlasma, PMD and JDeodorant. In their work they focused on six bad smells namely duplicated code, feature envy, god class, large class, long method and long parameter list. They automatically identified the bad smells using the detection tools and applied to six different versions of Gantt project which is open source software written in java. Francesca concluded that only Jdeodorant bad smell detection tool can detect the bad smell and do refactoring to remove the smell.

The researches discussed in the literature deals with various detection of bad smell and refactoring techniques used for object oriented software. In the literature there is less work on shotgun surgery bad smell detection and no methods or tools locate the bad smell shotgun surgery in the source code. Hence, the work proposed in this study suggests a framework that detects the bad smell shotgun surgery based on similarity measure distribution model which is focused only on object oriented software. The frame work has the potential to detect the bad smell shotgun surgery and there by improve the software maintenance of object oriented software.

MATERIALS AND METHODS

Similarity measure distribution model: Let S be the object oriented software for which the bad smells have to be detected and C be the set of classes such that C = {c_1, c_2, c_3, ..., c_n}, C⊂S, where n is the number of classes. From the set of classes C each class $c_k(1 \le k \le n)$ is a set of methods and attributes such that c_k = {mk_1, mk_2, mk_3, ..., mk_{pk}, ak_1, ak_2, ak_3, ..., $ak_{r,k}$} ∀ $1 \le pk \le n$, $1 \le rk \le n$ where mk_j are methods ∀j, $1 \le j \le pk$ and ak_j are the attributes ∀j, $1 \le j \le rk$ from c_k. For the software S the computation of the similarity measure is presented.

Similarity matrix: Shotgun Surgery is a smell, where change(s) made to a class creates ripple effect(s) in one or more related classes. This may be caused when programmers copy blocks of code and paste them in many different modules. To spot this type of code blocks the similarity measure is used. Computing similarity measure plays a key role since it captures the relationship between the methods (Bavota *et al.*, 2010). To compute the similarity measure, the three different measures, Structural Similarity between Methods (SSM) (Gui and Scott, 2006), Call-based Interaction between Methods (CIM) (Bavota *et al.*, 2010) and Conceptual Similarity between the Methods (CSM) (Marcus *et al.*, 2008, Poshyvanyk *et al.*, 2009) has to be computed first.

The first measure is SSM which was introduced by Gui and Scott (2006). SSM calculates the similarity between the methods, i.e., if the instance variable shared between the two methods is higher and then the similarity between the methods is also higher. It is calculated by using the following mathematical model:

$$SSM_{(mk_j, ml_j)} = \begin{cases} \dfrac{\left|ak_j \cap al_j\right|}{\left|ak_j \cup al_j\right|} & \text{if } \left|ak_j \cup al_j\right| \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where, mk_j, ml_1 are methods ∀j, 1,≤j≤pk from the class c_k and c_1, respectively. Where ak_j, al_j (∀j, $1 \le j \le rk$) be the two instance variables referenced by the method, mk_j, ml_j, respectively. The second measure CIM is to be calculated. The CIM was introduced by Bavota *et al.* (2010) which is necessary to know the set of methods called by any method where it is used to reduce the coupling between the classes. If the interactions between the methods are higher, then the value of CIM is higher. It is calculated by using the following mathematical model:

$$CIM_{mk_j \to ml_j} = \begin{cases} \dfrac{calls(mk_j, ml_j)}{calls\ in(ml_j)} & \text{if calls in}(ml_j) \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

where, calls (mk_j, ml_j) be the number of calls performed by method mk_j to ml_j and calls in (ml_j) , the total number of incoming calls to ml_j.

In addition to the structural information, there is also a need for the conceptual information. So, CSM is considered and it is the third measure. It was first introduced by Marcus *et al.* (2008). CSM is based on the conceptual information. Two methods are said to be conceptually related if they perform similar actions. CSM computes the relationship between the methods by measuring the textual similarity of the code components. CSM is calculated by using the following mathematical model:

$$CSM_{(mk_j, ml_j)} = \frac{\overrightarrow{mk_j} \cdot \overrightarrow{ml_j}}{\left\|\overrightarrow{mk_j}\right\| \cdot \left\|\overrightarrow{ml_j}\right\|} \quad (3)$$

where $\overrightarrow{mk_j}$ and $\overrightarrow{ml_j}$ are the vectors corresponding to the methods mk_j,ml_j, respectively and $\left\|\overrightarrow{mk_j}\right\| \cdot \left\|\overrightarrow{ml_j}\right\|$ represents the product of the Euclidean length. If the similarities between the methods are higher, then the value of CSM is higher.

All the measures mentioned above have the value set [0 or 1]. Finally, the similarity measure is calculated by combining the SSM, CIM and CSM (Marcus *et al.*, 2008; Poshyvanyk *et al.*, 2009). Similarity, measure is computed using the following mathematical model:

$$sim(c_k, c_l) = w_{SSM} \times SSM(mk_j, ml_j) + w_{CIM} \times CIM_{mk_j \rightarrow ml_j} + w_{CSM} \times CSM(mk_j, ml_j) \quad (4)$$

where, c_K and c_l is the two different classes in the set C; w_{SSM}, w_{CIM} and $w_{CSM} \epsilon (0, 1)$ and $w_{SSM}+w_{CIM}+w_{CSM} = 1$. The value of w_{SSM}, w_{CIM} and w_{CSM} ensures the weight in each measure (Bavota *et al.*, 2010).

The similarity measure has the values in the range of [0…1]. Using Eq. 4 a similarity matrix is computed by calculating the similarity measure for each class in the set C. And the similarity matrix is given as $sim c_1 c_n$.

Frequency histogram: From Eq. 4, the similarity matrix is obtained. From the similarity matrix, the order of the matrix is computed. Then, the frequency histogram is computed. The algorithm COMP_FREQUENCY_HISTOGRAM is as follows:

Input:

Similarity Matrix $^{sim}c_1c_n$

Process logic

Step 1: Compute the number of groups K. From the similarity matrix, the data is grouped using Sturge's formula (Sturges, 1926). Sturge's formula is used to find the total number of class intervals and to set the number of intervals as close as possible. The Sturge's formula is calculated using the mathematical model (Sturges, 1926):

$$K = 1+3.322*\log N \quad (5)$$

Where, N is the order of the matrix and K is the number of groups.

Step 2: Compute the width of the interval I. To have the same width for all the class intervals, the width of the interval is calculated (Pearson, 1895) using the mathematical model:

$$I = R/K \quad (6)$$

where, I is the width of the class interval, R is the range which is used to find out the class width and K is the number of groups. The range is calculated by using the following mathematical model (Pearson, 1895):

$$R = maxSim-minSim \quad (7)$$

where, maxSim = maximum value of the similarity matrix and minSim = minimum value of the similarity matrix.

Step 3: Compute the class interval CI_i: Using the width computed in step 2 the class intervals are calculated by using the following mathematical model (Pearson, 1895):

$$CI_i = \min sim+(i-1)*I, \min sim+i*I \quad (8)$$

where, i = 1, 2, ..., K and CI = Class Interval

Step 4: Compute the frequency distribution: Using Eq. 9, the midpoint is computed:

$$X = ½ (Lower class limit+Upper class limit) \quad (9)$$

where, X is the mid point. Using Eq. 10, the frequency is calculated:

$$f = Total number of occurrence of values in each class interval \quad (10)$$

where f = frequency. Using the Eq. 8 and 10 a frequency histogram is constructed. Where X-axis is the CI_i (Class Interval) and Y-axis is the f (frequency) of the graph.

Output:

Frequency Histogram

Classes are divided with the similarity measure by Sturge's equation (Sturges, 1926) and the frequency for each interval is calculated. A series of points is created in rectangle co-ordinates to represent the similarity measure statistics. From the histogram, the residual threshold is computed. The value of residual threshold RT is the average of each group of Standard Deviation (SD) in curve fitting. The value of SD is calculated by using the following mathematical model:

$$SD = \sqrt{\frac{1}{N}\sum_{i=1}^{k}(X_i - \bar{X})^2 \times f} \quad (11)$$

where, $N = \Sigma Xf$ and $\bar{x}$ is the mean which is calculated by using the following mathematical model:

$$\bar{X} = \frac{\Sigma Xf}{\Sigma X} \quad (12)$$

Where:

ΣXf = The sum of the product of midpoint and frequency

ΣX = The sum of midpoints

If the value of RT_i of each class interval is larger than the residual threshold RT then there is shotgun surgery bad smell in the class interval. Where, RT_i is given as the SD of each group in the curve fitting. After calculating the residual threshold, the value of class interval CI_i corresponding to the highest residual threshold RT_i is obtained and repeated occurrence in each row of the similarity matrix is computed by Eq. 13:

$$CI_i = sim_{c_1,c_n} \quad (13)$$

The row with the maximum number of occurrence of the value CI_i is obtained and that corresponding class is detected to have the bad smell shotgun surgery.

Empirical study definition and design: The goal of the study is to examine the framework similarity measure distribution model, in detecting the bad smell shotgun surgery in software systems. The quality focus is on the detection accuracy on the bad smell shotgun surgery and compared to the bad smell detection tools while the perspective is of researches who want to evaluate the effectiveness of the framework in identifying code smells to build better recommenders for developers.

The context of the study consists of four open source projects, namely HSQLDB, TYRANT, XERCES-J and JFREE CHART. HSQLDB (http://hsqldb.org) (Hyper SQL DataBase) is SQL relational database open source software written in Java. Tyrant (http://tyrant.sourceforge.net/index.php) is a game written in java and Xerces-J (http://xerces.apache.org) is a library for parsing, validating and manipulating XML written in java. JFree chart (http://www.jfree.org/jfreechart) is a free chart library for java where the developers display the professional quality chart easily. The four open source projects have different size and are of different domains. Table 1 reports the characteristics of the analyzed systems, namely the software versions, number of classes and number of methods.

Research question, data analysis and metrics

The study aims at addressing the following two research questions

RQ1: How does the framework similarity measure distribution model perform in detecting bad smell shotgun surgery?

RQ2: How does the framework similarity measure distribution model compare to the techniques based on static code analysis?

To answer RQ1, the framework similarity measure distribution model was simulated in a real usage scenario.

Table 1: No. of classes and methods in open source programs

Programm name	Version	No. of classes	No. of methods
HSQLDB	2.2.9	525	6258
TYRANT	0.334	262	2106
XERCES	7.0	991	14994
FREE CHART	1.0.13	583	7964

Table 2: No. of group and width of the class interval in open source programs

Programm name	Order of the matrix	No. of groups	Width of the class intervel
HSQLDB	275625	19	0.05
TYRANT	68644	17	0.06
XERCES-J	982081	20	0.05
JFREE CHART	339889	21	0.04

At first, the different similarity measure is applied to the open source software's where the relationship between the classes and methods is identified. After applying the three different measures the similarity matrix is computed. And the best result is obtained by setting the weights to $w_{SSM} = 0.1$, $w_{CIM} = 0.2$, $w_{CSM} = 0.7$ on HSQLDB, $w_{SSM} = 0.3$, $w_{CIM} = 0.2$, $w_{CSM} = 0.5$ on TYRANT,$w_{SSM} = 0.2$, $w_{CIM} = 0.1$, $w_{CSM} = 0.7$ on XERCES-J and $w_{SSM} = 0.2$, $w_{CIM} = 0.2$, $w_{CSM} = 0.6$ on JFREE CHART. From the similarity matrix, the order of the matrix, number of group and the width of the class interval is computed and it is listed in Table 2.

Table 2 the classes of open source software's are grouped and the width is computed. There are 19 groups for HSQLDB, 17 groups for TYRANT, 20 groups for XERCES-J and 21 groups for JFREE CHART. From this the frequency histogram is computed and the histogram graph is given in Fig.1-4.

From the output of frequency histogram, it is clear that the similarity codes are grouped and arranged in each class interval. After computing the frequency histogram, the residual threshold is computed. The residual threshold RT and RT_i is calculated and compared and the bad smell in the class interval is detected. Besides, different open source software have different number of classes, the class interval of each program may differ with each other. For HSQLDB 9 class intervals are detected, for TYRANT 5 class intervals are detected, for XERCES-J 7 class intervals are detected and for JFREE CHART 3 class intervals are detected. Now, the bad smell in the class/classes is computed using Eq. 13. Table 3 the number of shotgun surgery bad smell detected for open source software's is given.

Once the bad smell shotgun surgery has been detected in open source software's using the framework, then the performance of the framework is evaluated using the widely two adopted Information Retrivel (IR) metrics, namely precision and recall (Baeza-Yates and Ribeiro-Neto 1999):

$$recall = \frac{|cor \cap det|}{|cor|}\% \quad (14)$$

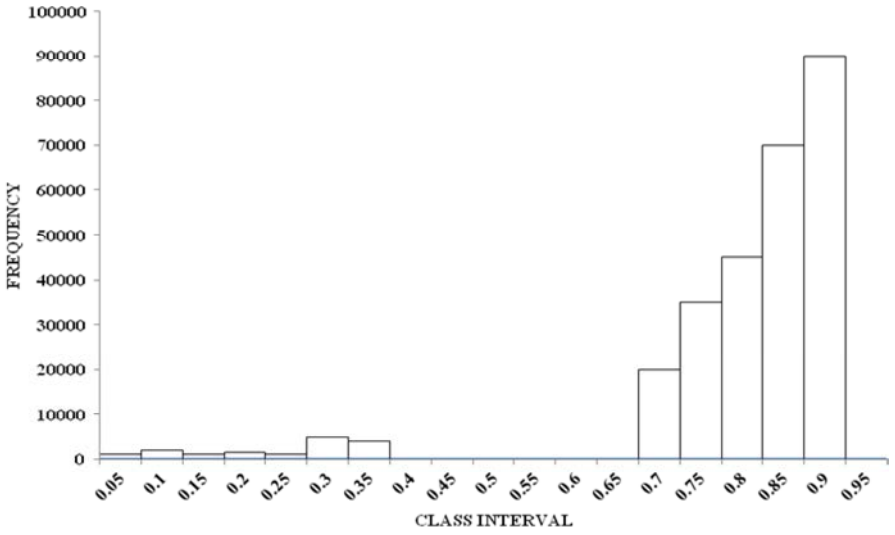

Fig. 1: Histogram of the open source software HSQLDB

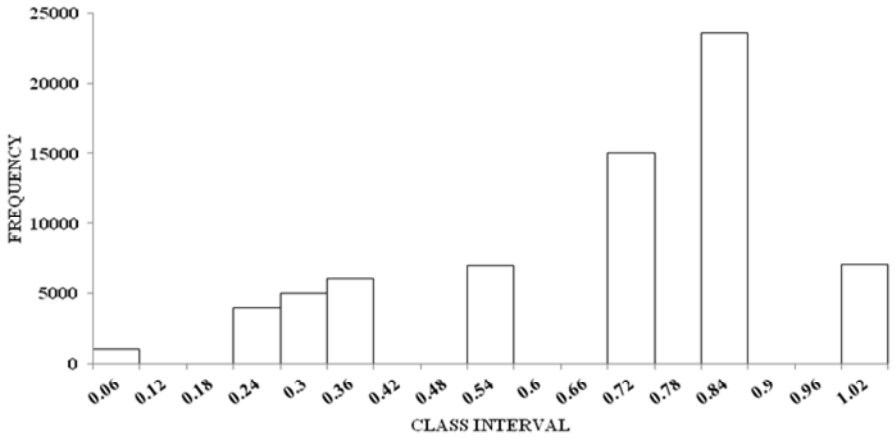

Fig. 2: Histogram of the open source software TYRANT

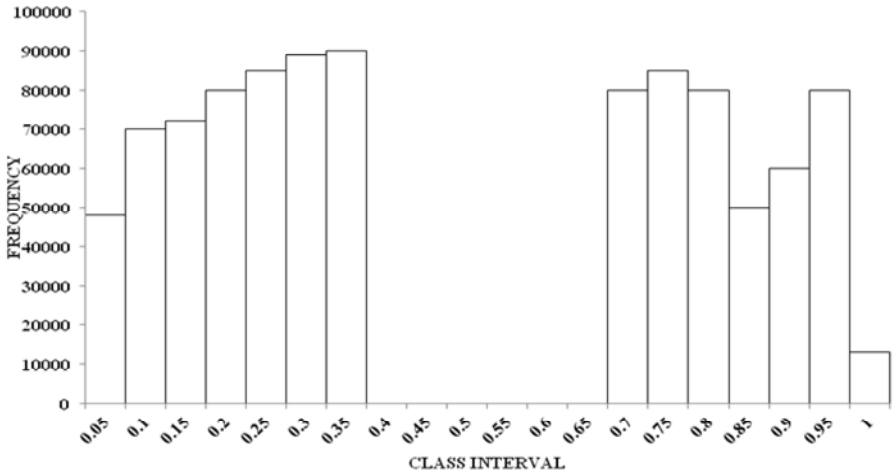

Fig. 3: Histogram of the open source software XERCES-J

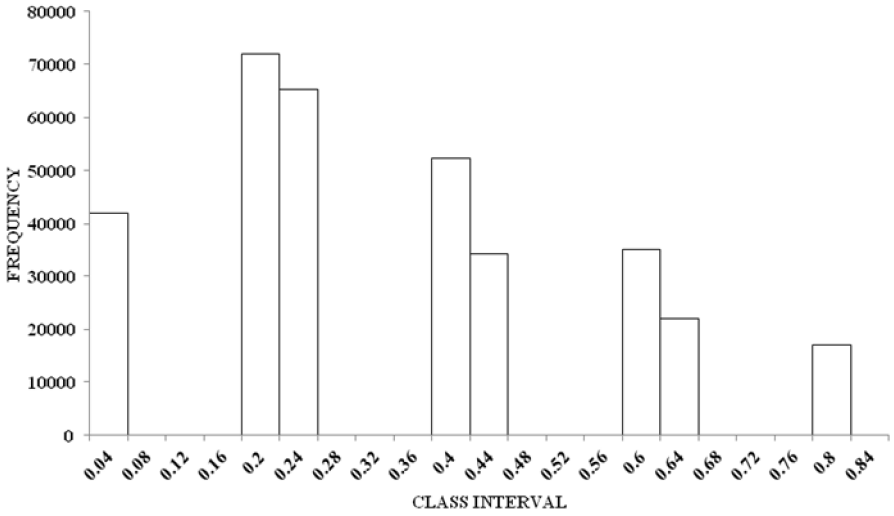

Fig. 4: Histogram of the open source software JFREE CHART

Table 3: No. of shotgun surgery bad smell detected in the four open source software

No. of shotgun surgery	HSQLDB	TYRANT	XERCES-J	JFREE CHART
Bad smell	7	3	5	4

$$\text{Precision} = \frac{|\text{cor} \cap \text{det}|}{|\text{det}|}\% \quad (15)$$

where, cor and det represent the set of true positive smells (those manually identified) and the bad smell shotgun surgery detected by the framework similarity measure distribution model respectively. The F-measure is reported based on the indicator precision and recall. And the F-measure is defined as follows:

$$\text{F}-\text{measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}\% \quad (16)$$

To answer RQ2, the framework is compared with the bad smell detection tool namely inFusion (inFusion-Design Flawdetection tool. Availableat://www.intooitus.com/products/infusion) and iPlasma (iPlasma: http://loose.upt.ro/iPlasma/index.html) (Marinescu, 2003). iPlasma is a detection tool implemented using the detection strategies by Marinescu (2004). The detection strategies identify the code deviation from good design principles of object oriented systems. iPlasma allows to detect more than 15 bad smells including shotgun surgery. inFusion has its root in iPlasma and then extended with more functionality. To improve the quality of the system infusion focuses at architectural and at code level. inFusion allows to detect more than 20 bad smells, like code duplication, shotgun surgery, feature envy, methods and so on. These tools have the ability to detect the bad smell shotgun surgery in the source code of object oriented programs. But it does not provide the location of the source code in the bad smell detection. These tools do not perform the refactoring operation. To compare the performances of the framework similarity measure distribution model with the above mentioned tools, recall, precision and F-measure were calculated. Table 4 and 5 reports the results of the comparison. Moreover, to provide a further comparison of the framework with the tools, the following overlap metrics has been computed (Palomba *et al*. 2013):

$$\text{Correct}_{(m_i \cap m_j)} = \frac{\left|\text{correct}_{m_i} \cap \text{correct}_{m_j}\right|}{\left|\text{correct}_{m_i} \cup \text{correct}_{m_j}\right|}\% \quad (17)$$

$$\text{Correct}_{(m_i \setminus m_j)} = \frac{\left|\text{correct}_{m_i} \cap \text{correct}_{m_j}\right|}{\left|\text{correct}_{m_i} \cup \text{correct}_{m_j}\right|}\% \quad (18)$$

where, $^{\text{correct}}m_i$ represents the set of correct code smells detected by the method m_i, $^{\text{correct}_{m_i} \cap \text{correct}_{m_j}}$ measures the overlap between the set of true code smells detected by both methods m_i and m_j and $^{\text{correct}_{m_i} \setminus \text{correct}_{m_j}}$ measures the true smells detected by m_i only and missed by m_j.

Table 4: Similarity measure distribution model as compared to the tool inFusion

Code smell	Open source software	Affected components	Similarity based distribution model			inFusion tool		
			Precision (%)	Recall (%)	F-measure (%)	Precision (%)	Recall (%)	F-measure (%)
Shotgun	HSQLDB	7	80	100	88	30	38	34
	TYRANT	3	95	100	97	8	57	14
Surgery	XERCES-J	5	100	100	100	5	33	12
	JFREE CHART	4	100	100	100	28	20	25

Table 5: Similarity measure distribution model as compared to the tool iPlasma

Code smell	Open source software	Affected components	Similarity based distribution model			iPlasma tool		
			Precision (%)	Recall (%)	F-measure (%)	Precision (%)	Recall (%)	F-measure (%)
Shotgun	HSQLDB	7	80	100	88	15	25	19
	TYRANT	3	95	100	97	7	44	12
Surgery	XERCES-J	5	100	100	100	5	25	8
	JFREE CHART	4	100	100	100	29	20	24

RESULTS AND DISCUSSION

This study reports the results of the proposed research with the aim of addressing the research questions formulated. Table 4 and 5 reports the results in terms of recall, precision and F-measure achieved by the framework and the tools on the four open source software. From the result it is clear that the framework attained very good recall at a maximum of 100% for XERCES-J and JFREE CHART, i.e., the framework is able to detect at most all the smells from the source code. Table 6 reports the values concerning overlap and differences between the framework and the tools. Column similarity measure distribution model ⋂ inFusion reports the percentage of correct code smells identified by both similarity measure distribution model and tools. Column similarity measure distribution model\inFusion reports the percentage of correct code smells identified by similarity measure distribution model but not by the tool infusion, column inFusion\similarity measure distribution model reports the percentage of correct code smells identified by the tool infusion but not by similarity measure distribution model. Table 7 reports the values concerning overlap and differences between the framework and the tools. Column similarity measure distribution model ⋂ iPlasma reports the percentage of correct code smells identified by both similarity measure distribution model and tools. Column similarity measure distribution model\iPlasma reports the percentage of correct code smells identified by similarity measure distribution model but not by the tool infusion, column iPlasma\similarity measure distribution model reports the percentage of correct code smells identified by the tool infusion but not by similarity measure distribution model.

From the result, it is seen clearly that the framework similarity measure distribution model performs good in detecting the code smell shotgun surgery. The framework was able to outperform when compared to the tool inFusion and iPlasma. From the framework it is claimed that there is no need to refer to the past change history. Using our framework the maintainer can able to detect the exact location of the smell when a change occurs. So, there is no need to have the sufficient past histories of the co-changes. The advantage of this framework is it highlights and identifies the exact location of the smell shotgun surgery which is not focused in the tools inFusion/iPlasma, they all just gives the number of smells but can't able to detect the exact location.

Summary of RQ1: The framework similarity measure distribution model performed good in detecting the bad smell shotgun surgery considered in this research. The F-measure was between (88-100%). While the result obtained was quite expected on the bad smell shotgun surgery since it features are extracted using both the structural, semantic and call based interaction between the methods.

Summary of RQ2: The framework was able to outperform the design flaw detection tools namely inFusion and iPlasma in terms of recall, precision and F-measure. The comparison suggests that the design flaw detection tool and the framework could be nicely complemented to obtain a good result.

Threats to validity: In this study, construct validity, internal validity and external validity is addressed. Construct validity concern with the relationships between theory and observation. In general, the threat internal validity focuses on the factors that influence the results. External validity concern the generalization of the result and finally reliability concerns the replication of the study.

Construct validity: The main contribution of this study is to detect the shotgun surgery bad smell. The framework

Table 6: Overlap between the framework similarity measure distribution model and the tool inFusion

Code smells	Similarity measure distribution model ∩ inFusion	Similarity measure distribution model\inFusion	inFusion\similarity measure distribution model
Shotgun surgery	40	100	40

Table 7: Overlap between the framework similarity measure distribution model and the tool inPlasma

Code smells	Similarity measure distribution model ∩ iPlasma	Similarity measure distribution model\iPlasma	iPlasma\similarity measure distribution model
Shotgun surgery	30	100	30

used in the study is the stepwise selection approach. The memory consumption would be substantially large because the similarity matrix and the values of the residual threshold are maintained until the bad smell is identified.

Internal validity: In this study, the relationship between the classes is focused. The dependencies related to the methods that affect the maintainability and other types of dependencies caused by structural relation between methods, call based interaction between methods and conceptual semantic between the methods is considered. The detection of the bad smell shotgun surgery using the frequency distribution model is easy and reduces the time to detect the smell.

External validity: The framework was experimented on four open source software was highly effective and accurate. The framework deals with only one smell shotgun surgery while the other smells are not uncovered. The open source software chosen exhibit the characteristics of the object oriented software that are subjected to the detection of the bad smell shotgun surgery. The degree of maintainability improvement would vary from software to software.

Reliability: The data used for the experiment is available online. The efficient similarity measure is used to calculate the similarity matrix. And from it, the distribution model is computed. So the detection of the bad smell shotgun surgery reduces the computation time.

CONCLUSION

In this study, a framework for detecting shotgun surgery bad smell is modeled using similarity measure distribution model. In this model, a piece of code which is scattered throughout the system is identified using similarity measure. From the result of the similarity measure a frequency table is constructed. In the frequency table the width of the class interval is constructed using sturge's rule and a frequency histogram is generated. From the generated frequency histogram, the residual threshold is computed to detect the bad smell in the class interval CI_i . In each row of the similarity matrix, the maximum number of occurrence of the value of class interval Cl_i for which the residual threshold is largest is computed to detect the bad smell shotgun surgery. The proposed framework maximizes the number of detected defects. The study aims at evaluating the framework performances in terms of precision and recall against the tools inFusion and iPlasma. From the result, it is inferred that the framework attains a good recall when compared to the tools. Thus, this study sheds light on a branch of statistics which has received attention from software engineers. In future, the different kind of distribution model is to be considered and applied to other types of bad smell.

REFERENCES

Abbes, M., F. Khomh, Y.G. Gueheneuc and G. Antoniol, 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), March 1-4, 2011, IEEE, Montreal, Quebec, Canada, ISBN:978-1-61284-259-2, pp: 181-190.

Baeza-Yates, R.A. and B. Ribeiro-Neto, 1999. Modern Information Retrieval. 1st Edn., Addison-Wesley Longman Publishing Co., Boston, MA., USA.

Bavota, G., A.D. Lucia, A. Marcus and R. Oliveto, 2010. A two-step technique for extract class refactoring. Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, September 20-24, 2010, Antwerp, Belgium, pp: 151-154.

Bavota, G., Lucia, D.A., A. Marcus and R. Oliveto, 2013. Using structural and semantic measures to improve software modularization. Empirical Software Eng., 18: 901-932.

Dexun, J., M. Peijun, S. Xiaohong and W. Tiantian, 2013. Detection and refactoring of bad smell caused by large scale. Int. J. Software Eng. Appl., 4: 1-13.

Fontana, F.A., P. Braione and M. Zanoni, 2012. Automatic detection of bad smells in code: An experimental assessment. J. Object Technol., 11: 1-5.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, New York, USA., ISBN-13: 9780201485677, Pages: 431.

Gui, G. and P.D. Scott, 2006. Coupling and cohesion measures for evaluation of component reusability. Proceedings of the 2006 International Workshop on Mining Software Repositories, May 22-23, Shanghai, China, pp: 18-21.

Jiang, D., P. Ma, X. Su and T. Wang, 2014. Distance metric based divergent change bad smell detection and refactoring scheme analysis. Int. J. Innovative Comput. Inf. Control, 10: 1519-1531.

Kessentini, M., S. Vaucher and H. Sahraoui, 2010. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. Proceedings of the IEEE-ACM International Conference on Automated Software Engineering, September 20-24, 2010, ACM, Antwerp, Belgium, ISBN:978-1-4503-0116-9, pp: 113-122.

Khomh, F., M.D. Penta, Y.G. Gueheneuc and G. Antoniol, 2012. An exploratory study of the impact of antipatterns on class change and fault-proneness. Empirical Softw. Eng., 17: 243-275.

Liu, H., Z. Ma, W. Shao and Z. Niu, 2012. Schedule of bad smell detection and resolution: A new way to save effort. IEEE. Trans. Software Eng., 38: 220-235.

Mantyla, M., 2003. Bad smells in software-a taxonomy and an empirical study. Ph.D Thesis, Helsinki University of Technology, Espoo, Finland.

Marcus, A., D. Poshyvanyk and R. Ferenc, 2008. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE. Trans. Software Eng., 34: 287-300.

Marinescu, R., 2003. Measurement and quality in object oriented design. Ph.D Thesis, Faculty of Automatics and Computer Science, Politehnical University of Timisoara, Timisoara, Romania.

Marinescu, R., 2004. Detection strategies: Metrics-based rules for detecting design flaws. Proceedings of the 20th IEEE International Conference on Software Maintenance, September 11-14, 2004, IEEE, Timisoara, Romania, ISBN:0-7695-2213-0, pp: 350-359.

Moha, N., Y.G. Gueheneuc, L. Duchien and L.A.F. Meur, 2010. DECOR: A method for the specification and detection of code and design smells. IEEE. Trans. Software Eng., 36: 20-36.

Munro, M.J., 2005. Product metrics for automatic identification of bad smell design problems in java source-code. Proceedings of the 11th IEEE International Symposium on Software Metrics (METRICS'05), September 19-22, 2005, IEEE, New York, USA., ISBN:0-7695-2371-4, pp: 15-15.

Olbrich, S., D.S. Cruzes, V. Basili and N. Zazworka, 2009. The evolution and impact of code smells: A case study of two open source systems. Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, October 15-16, 2009, IEEE, Washington, USA., ISBN: 978-1-4244-4842-5, pp: 390-400.

Oliveto, R., F. Khomh, G. Antoniol and Y.G. Gueheneuc, 2010. Numerical signatures of antipatterns: An approach based on b-splines. Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CSMR), March 15-18, 2010, IEEE, Salerno, Italy, ISBN: 978-1-61284-369-8, pp: 248-251.

Palomba, F., G. Bavota, D.M. Penta, R. Oliveto and D. Poshyvanyk *et al*., 2015. Mining version histories for detecting code smells. IEEE. Trans. Software Eng., 41: 462-489.

Palomba, F., G. Bavota, D.M. Penta, R. Oliveto and D.A. Lucia *et al*., 2013. Detecting bad smells in source code using change history information. Proceedings of the 2013 IEEE-ACM 28th International Conference on Automated Software Engineering (ASE), November 11-15, 2013, IEEE, Williamsburg, Virginia, USA., ISBN: 978-1-4799-0215-6, pp: 268-278.

Pearson, K., 1895. Contributions to the mathematical theory of evolution. II. Skew variation in homogeneous material. Trans. R. Philos. Soc. Ser. A, 186: 343-414.

Poshyvanyk, D., A. Marcus, R. Ferenc and T. Gyimothy, 2009. Using information retrieval based coupling measures for impact analysis. Empirical Software Eng., 14: 5-32.

Riel, A.J., 1996. Object Oriented Design Heuristics. Addison-Wesley Company, Boston, Massachusetts, ISBN:9780201633856, Pages: 379.

Sahin, D., M. Kessentini, S. Bechikh and K. Deb, 2014. Code-smell detection as a bilevel problem. ACM. Trans. Software Eng. Methodology (TOSEM.), Vol. 24, 10.1145/2675067

Serban, C., 2013. Shotgun surgery design flaw detection: A case-study. Studia Universitatis Babes Bolyai Inf., 58: 65-74.

Sturges, H.A., 1926. The choice of a class interval. J. Am. Stat. Association, 21: 65-66.

Travassos, G., F. Shull, M. Fredericks and V.R. Basili, 1999. Detecting defects in object-oriented designs: using reading techniques to increase software quality. ACM. Sigplan Not., 34: 47-56.

Tsantalis, N. and A. Chatzigeorgiou, 2009. Identification of move method refactoring opportunities. IEEE. Trans. Software Eng., 35: 347-367.

Tufano, M., F. Palomba, G. Bavota, R. Oliveto and D.M. Penta *et al*., 2015. When and why your code starts to smell bad. Proceedings of the 37th International Conference on Software Engineering, Vol. 1, May 16-24, 2015, IEEE, Piscataway, New Jersey, USA., ISBN:978-1-4799-1934-5, pp: 403-414.

Yamashita, A. and L. Moonen, 2012. Do code smells reflect important maintainability aspects?. Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), September 23-28, 2012, IEEE, New York, USA., ISBN:978-1-4673-2313-0, pp: 306-315.

Yamashita, A. and L. Moonen, 2013. Exploring the impact of inter-smell relations on software maintainability: An empirical study. Proceedings of the 2013 International Conference on Software Engineering, May 18-26, 2013, IEEE, Piscataway, New Jersey, ISBN:978-1-4673-3076-3, pp: 682-691.