Detecting Bad Smells in Object Oriented Design Using Design Change Propagation Probability Matrix

[1]A. Ananda Rao, [2]K Narendar Reddy

***Abstract*--- Object oriented software systems are subject to frequent modifications either during development (iterative, agile software development) or software evolution. For such systems which have large number of classes, detection of design defects is a complex task. Bad smells are used to identify design defects in object oriented software design. Identification of bad smells allows us to apply appropriate refactorings to improve the quality of design. In existing bad smell detection systems, bad smells are generally detected using human intuition, and recently, people started developing quantitative methods. As human intuition is subjective, the quantitative methods to detect bad smells are effective as they do not include subjectivity (bias) and allows for automation. This paper proposes a quantitative method. The proposed quantitative method makes use of the concept design change propagation probability matrix (DCPP matrix) to detect two important bad smells. The first one is shotgun surgery bad smell and the other one is divergent change bad smell. Two of the advantages of the proposed quantitative method are: Detecting shotgun surgery and divergent change bad smells require that the design change propagation between artifacts that are connected directly and indirectly should be considered quantitatively. The proposed method considers this aspect quantitatively. The second advantage is, the method is amicable for automation.**

Using this proposed method, with typical example designs, the bad smells shotgun surgery and divergent change are detected. Appropriate refactorings are suggested for the detected bad smells. Different advantages of the proposed quantitative method are presented. A broader framework in which this quantitative method is applied is given.

Key words: Bad smells, Design change propagation probability matrix, Framework, Refactoring.

Manuscript submitted for review on December 6, 2007 and accepted on December 29, 2007.

[1]*A. Ananda Rao is with Department of Computer Science and Engineering, JNTU College of Engineering, Jawaharlal Nehru Technological University, Anantapur, AP, India, (*akepogu@yahoo.co.in).

[2]*K Narendar Reddy is with Department of Computer Science and Engineering, CVR College of Engineering, Jawaharlal Nehru Technological University, Mangalpally, Ibrahimpatnam, Hyderabad, AP, India, (*knreddy_mist@yahoo.com).

I. Introduction

Detection of design defects using bad smells during software development (iterative, agile) and software maintenance is an important and complex task. In this paper the software maintenance is in the context of preventive maintenance. Design defects cause the system to exhibit low maintainability, low reuse, high complexity and faulty behavior [1]. One of the ways to detect design defects is by detecting bad smells. The bad smells which affect maintenance mostly are shotgun surgery, divergent change, and parallel inheritance hierarchies. These smells can be characterized as "maintenance smells" because they manifest themselves during maintenance of the software [2]. Hence, accurate detection of these bad smells provides a significant challenge in evolving software. Detection of bad smells allows us to apply appropriate refactorings to improve the quality of design.

In existing bad smell detection systems, bad smells are detected using human intuition which leads to subjectivity and is not amicable for automation. Recently, people started developing quantitative methods. Quantitative methods are effective as they do not include subjectivity (bias) and allows for automation. This paper proposes a quantitative method/approach. The proposed quantitative method makes use of the concept design change propagation probability matrix (DCPP matrix) to detect two important bad smells which are categorised under "maintenance smells". The first one is shotgun surgery and the other one is divergent change. Both of these bad smells are important because the first one creates a rippling effect and the other one indicates an artifact which is sensitive to changes in other artifacts. In both the cases, the change propagation is involved. Change propagation is based on the strength of coupling between the artifacts. Therefore, it is always an advantage to have design change propagation prediction that conveys the number of related artifacts that are going to be affected if a particular artifact is changed. This issue is not addressed in the literature. The paper is aimed at tackling the above issue. Detection of the bad smell "parallel inheritance hierarchies" under the category "maintenance smells" will be considered in the future work.

This paper explores the use of the concept (DCPP matrix) developed for version management [3], to detect the proposed bad smells. The DCPP matrix represents the design change propagation probabilities between artifacts. The

DCPP matrix for a design of N artifacts, is of size N x N. In this matrix, the entry at row A, column B represents the probability that a design change in artifact A requires change in B so as to preserve the overall function of the system. To construct DCPP matrix, first the design should be represented as a unified representation of artifacts graph (URA graph) [4]. The artifacts in URA graph, map to physical entities in different ways like classes, sets of classes, subsystems etc. Making use of URA graph, other concept cdegree (explained in section III-A) and proposed equations (given in section III-A), the DCPP matrix is constructed. For three example designs DCPP matrices are constructed. By making use of these DCPP matrices and formulated conditions (given in section III-A), proposed bad smells are detected.

The organization of the paper is as follows. Section II presents the related work on detection of bad smells. Section III addresses the bad smell detection method which is based on DCPP matrix. A generic framework for object oriented software design quality improvement is given in section IV. The conclusions and future directions have been placed in section V.

II. Related Work

In bad smells shotgun surgery and divergent change, the change propagation is involved. The change propagation depends on strength of dependency (coupling) between artifacts. Since the artifacts are related one another directly (adjacently) or indirectly (through intermediate artifacts) the strength of dependency (coupling) should be calculated for the above two cases. This section presents the related works covering the above aspects.

Considering one widely accepted suit of metrics [5], the CBO (Coupling between object classes) is defined as a count of the number of other classes to which it (a class under consideration) is coupled. This definition of coupling counts the classes to which a particular class has some sort of interaction. It does not measure the amount (strength) of coupling between any two classes. Considering the number of discrete messages exchanged between classes, the god classes are identified using link analysis method [6]. The god classes in the system imply a poorly designed model.

The paper [7] describes the ripple effect metric. It considers its applicability as a software complexity measure for object oriented software. It is mentioned that this approach has potential to improve the stability and efficiency of object oriented software and cut the cost of software maintenance. A list of metric based detection strategies for capturing flaws of object oriented design are defined in paper [8]. Papers [7][8] have not included how the strength of dependency (coupling) between artifacts (which are connected through intermediate artifacts in more than one path) is calculated.

JRipples, a tool [9] supports impact analysis and change propagation. This tool does not employ coupling metrics to suggest which classes are most likely to be involved in change. A novel metric based heuristic framework to detect and locate object oriented design flaws from the source code is proposed in paper [10]. In the future work section, it is mentioned that it will be useful to include a degree of possibility or a kind of certainty factor for the heuristics and the detected design flaws as we can not specify strict threshold values for “high” or “low” terms in classifiers rules.

Paper [11] investigated the construction of probabilistic decision models based on coupling measurement to support impact analysis. It provides an ordering of classes where ripple effects are more likely. A metric for measuring the class weakness for object oriented software is proposed in paper [12]. Inter-class weakness is affected by the interconnection of the class over other classes, and increases if the dependency of the class is more. The ripple effect also contributes to the dependency and this effect has also been considered in this paper.

Even though the paper [12] considers the ripple effect in contribution to the dependency between classes, the paper has not correlated the results with the detection of bad smells (shotgun surgery, divergent change). To the best of our knowledge, the present bad smells/design flaws detection methods have not considered the design change propagation probabilities and how the DCPP matrix values are used in detecting these bad smells. Therefore, this paper proposes a quantitative method in detecting two important bad smells using design change propagation probability matrix. Knowing the presence of these bad smells allows us to apply appropriate refactorings so as to improve the quality of software design.

III. A DCPP Matrix Based Bad Smells Detection Method

The proposed bad smells detection method which is based on DCPP matrix is a quantitative method. The proposed method is carried out in three steps:

1. Construction of DCCP matrix for a given design.
2. Representing different possible values of DCPP matrix as different conditions.
3. Checking for the conditions satisfied by a given DCPP matrix and correlating these conditions with bad smells.

Sections A and B covers above three steps of the proposed method. Construction procedure for DCPP matrix and formulated conditions are given in section A. Detection of bad smells in three example designs is given in section B.

A. Construction of DCPP Matrix

To construct DCPP matrix, first the design is represented as a URA graph and then strength of dependency between artifacts is estimated. In this paper, the strength (amount) of dependency between artifacts is represented by the term cdegree [3]. Since the artifacts are related one another

directly (adjacently) or indirectly (through intermediate artifacts) the strength of dependency is calculated for the above two cases.

First the cdegree between adjacent artifacts is estimated and then using proposed equations, combined cdegree between artifacts by considering intermediate artifacts is estimated. Using these cdegree values, N x N DCPP matrix is constructed, where N is number of artifacts in the design.

Definition of cdegree: The degree of coupling (cdegree) of link is the indicator of the amount of dependency that exists between two related artifacts represented by the URA. The value of cdegree has the range [0, 1]. Therefore, a change is propagated to related artifacts based on the cdegree value. Since the artifacts are related one another directly (adjacently) or indirectly (through intermediate artifacts) the change propagation should be calculated for the above two cases.

Cdegree estimation: Let A and B are two artifacts that are related adjacently and attributes of an artifact B access attributes of an artifact A. In this paper an attribute is considered as a feature of an artifact. Since various number of links are possible between two artifacts, each link can be given a weightage. Based on these weightages the total strength between these two artifacts can be calculated.

For example, consider Fig. 1, where a class A has three attributes (a1, a2, and a3) and B has five (b1, b2, b3, b4, and b5) attributes and attribute a2 is called four times by attributes of artifact B. Therefore, the weightage of attribute a2 is 4/7 where total number of calls exist between A and B are 7. Similarly, weightage for other attributes can also be calculated. After calculating the weightages for all the attributes, cdegree between A and B is defined as follows. Cdegree = sum of weightages of each link with respect to attributes of A from B/Total number of possible links from B to A.

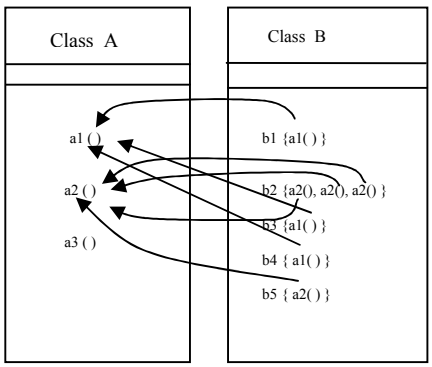

Fig.1. Example class diagram and their interactions

The denominator can be taken as the total number of attributes exist in A, since this many maximum links can be made. In the above equation, a link is defined as a call made by class B with respect to a method. It is irrespective of number of calls made to each method. That is, all calls of a method constitute a link even though if it is called more than once by a method of class B. The number of call references is taken into consideration while calculating weightage of each attribute.

The cdegree considers method invocation outside the class and variable reference outside the class. Higher the value of cdegree indicates higher the method invocation outside the class or higher the variable reference outside the class. It may be the combination of the both. Hence, it indicates the strength of dependency (coupling). A design change is propagated to related artifacts based on the cdegree value. For example, a design change is propagated to related artifacts whose cdegree value is more than the threshold (say 0.5) value.

The above procedure is used to estimate cdegree value between adjacent artifacts. A more general approach would be to estimate the ramifications due to a single change. Therefore, the following method is used to compute combined cdegree value for the artifacts that are connected through intermediate artifacts in more than one path. The combined cdegree value is the probability that the end effect will arise, regardless of the path. This can be calculated using probability lemmas. While calculating the combined cdegree value it is to be noted that the events are not mutually exclusive. Therefore, the following formulas are used to estimate the combined cdegree values [3].

$$c\deg ree_{p,q} \cap c\deg ree_{p,r} = c\deg ree_{p,q} \text{ X } c\deg ree_{p,r} \quad (1)$$

$$c\deg ree_{p,q} \cup c\deg ree_{p,r} = c\deg ree_{p,q} + c\deg ree_{p,r} - \left(c\deg ree_{p,q} \text{ X } c\deg ree_{p,r}\right) = 1 - \left[\left(1 - c\deg ree_{p,q}\right) \text{X } \left(1 - c\deg ree_{p,r}\right)\right] \quad (2)$$

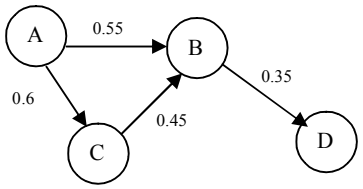

Fig.2. Example URA graph of artifacts

For example, in Fig. 2, values on the top of links represent cdegree values between various artifacts. The combined cdegree value at artifact B in the figure is calculated as follows:

$$c\deg ree_{a,b} = 1 - \left[\left(1 - c\deg ree_{a,b}\right) \text{ X } \left(1 - \left(c\deg ree_{a,c} \text{ X } c\deg ree_{c,b}\right)\right)\right] = 0.67$$

To construct DCPP matrix, first URA graph is constructed to represent the design of the system. Then cdegree values between adjacent artifacts are calculated (using the procedure explained in section A under the heading cdegree estimation). Example design diagram shown in Fig. 2 is an URA graph. In this graph, values on the top of links represent cdegree values between adjacent artifacts. By making use of cdegree values between adjacent artifacts and above proposed equations, the combined cdegree values between artifacts that are connected through intermediate artifacts are calculated. Making use of the cdegree values, N x N matrix is constructed, where N is the number of artifacts in the design. This matrix is termed as DCPP matrix. The entry at row A1, column A3 represents the probability that a design change in artifact A1 requires change in A3 so as to preserve the overall function of the system. The 4 x 4 DCPP matrix constructed using above described procedure for the design in Fig. 2, is given in Table I.

TABLE I

DCPP MATRIX FOR THE DESIGN IN FIG. 2

	A	B	C	D
A	1	0.67	0.6	0.24
B		1		0.35
C		0.45	1	0.16
D				1

Different possible values of DCPP matrix can be represented as different conditions.

Condition 1: Majority of the elements in a row containing larger values (greater than threshold value, say 0.5)
Condition 2: Majority of the elements in a column containing larger values (greater than threshold value, say 0.5)
Condition 3: Both conditions 1 & 2 exist
Condition 4: All the diagonal element values are one
Condition 5: All the matrix element values are one.
Condition 6: With respect to a particular artifact, row or column elements contain zero values, except diagonal element value (which is one for all the artifacts).

Satisfying different conditions indicate different things in the design. Satisfying the condition 4 indicates the probability that any change in an artifact, affects its design maximum. The unit DCPP matrix (satisfying condition 5) indicates the violation of basic heuristic of developing low coupled system. Condition 5 can be used in validating best practices that are used for software development. Condition 6 indicates that the artifact is not collaborating with any of the artifacts and that this artifact is an isolated one. Isolated artifact is created may be because of accidental omission of a link, or it may be a redundant artifact. In the first case, it may create/indicate problem in the overall functioning of software. In large software systems identifying such type of artifacts is very important. Conditions 1, 2, and 3 can be used in detecting the presence of bad smells (shotgun surgery, divergent change) in the design. The detection procedure for the shotgun surgery and divergent change bad smells, and suggested refactorings are given in the following section.

B. Detecting Bad Smells in Example Design Diagrams

Three different hypothetical object oriented designs are taken and corresponding DCPP matrices are constructed. Using the conditions formulated in section A, bad smells are detected. The detection of shotgun surgery bad smell in example design 1 is given in section B-1, whereas the detection of divergent change bad smell in example design 2 is presented in section B-2. The example design which contains both the bad smells is given in section B-3.

B.1. Detecting "shotgun surgery" bad smell

When every time you make a kind of change, you have to make a lot of little changes to a lot of different classes [13]. This indicates that the change is propagated to many other artifacts. Change propagation depends on cdegree value between the artifacts. The cdegree values for artifacts which are not adjacent but connected through intermediate artifacts represent combined cdegree which is calculated considering rippling effect.

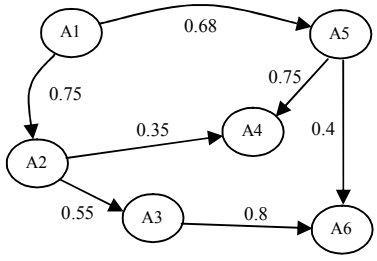

Fig.3. Diagram of example design 1

TABLE II

DCPP MATRIX FOR THE EXAMPLE DESIGN 1

	A1	A2	A3	A4	A5	A6
A1	1	0.75	0.41	0.64	0.68	0.51
A2		1	0.55	0.35		0.44
A3			1			0.8
A4				1		
A5				0.75	1	0.4
A6						1

If a row contains high values with respect to a particular artifact (satisfying condition 1), this situation indicates that, any change to this artifact will require changes in more number of artifacts (artifacts that are indicated in the columns corresponding to high values). The DCPP matrix for the example design 1 (Fig. 3) is given in Table II. The row 1 satisfies the condition 1. This condition indicates the presence of shotgun surgery bad smell in the design. When the artifact A1 is changed, the other artifacts which need to be changed are A2, A4, A5, and A6. To rectify the design defect indicated by shotgun surgery bad smell, appropriate refactorings have to be applied.

Proposed refactorings: Refactoring is basically changing an object oriented software system in such a way that it does not alter the external behavior of the code, yet improves internal structure [14]. The key idea here is to redistribute classes, variables, and methods across the class hierarchy in order to facilitate future adaptations and extensions [15]. To rectify the bad smell shotgun surgery in the design, use "move method" and "move field" refactorings to pull all the changes into a single class. If no current class looks like a good candidate, create one [13].

Applying refactoring like move method, may lead to creation of a new version for the refactored artifact. But, if the shotgun surgery is not rectified by refactoring, for every change in the artifact (which is causing the bad smell shotgun surgery) as part of corrective and perfective maintenance, a new version may be created for every artifact which is affected by shotgun surgery bad smell. Hence, lot of new versions may be created and in turn increases maintenance cost.

B.2. Detecting "divergent change" bad smell

Divergent change occurs when one class is commonly changed in different ways for different reasons [13].

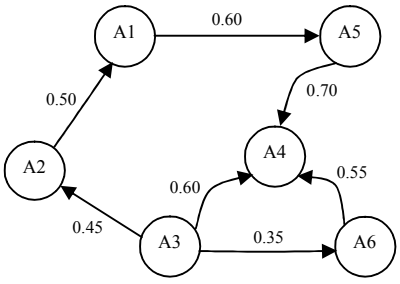

Fig.4. Diagram of example design 2

If a column (in DCPP matrix) contains high values with respect to a particular artifact (satisfying condition 2), then it can be inferred that this artifact is likely to undergo frequent changes during evolution. The DCPP matrix for the example design 2 (Fig. 4) is shown in Table III. Fourth column satisfies the condition 2. The artifact which is likely to undergo frequent changes is A4.

TABLE III

DCPP MATRIX FOR THE EXAMPLE DESIGN 2

	A1	A2	A3	A4	A5	A6
A1	1			0.42	0.6	
A2	0.5	1		0.21	0.3	
A3	0.23	0.45	1	0.71	0.14	0.35
A4				1		
A5				0.7	1	
A6				0.55		1

This condition indicates the presence of divergent change bad smell in the design. Presence of divergent change bad smell in the design may indicate/lead to the following:

1. The class (artifact) is trying to do too much. It may be a "god class". God class could be the result of placing disjoint features into one class. LCOM [5] could be used for detecting god classes when disjoint features are placed into one class.
2. The class (artifact) is depending on many other artifacts
3. Frequent changes may deteriorate the design of class (artifact) undergoing change
4. Lot of versions for the same class (artifact) may be created.

To overcome the above disadvantages, the artifact A4 (Fig. 4) should be refactored. Identify everything that changes for a particular cause and use extract class refactoring to put them all together [13]. Due to extract class refactorings a few new classes will be created.

B.3. Detecting "shotgun surgery" and "divergent change" bad smells in the same design

It is possible that both the bad smells shotgun surgery and divergent change can exist in the same design. The design shown in the Fig. 5 has these two bad smells. These bad smells can be detected from the DCPP matrix values given in Table IV (satisfying condition 3).

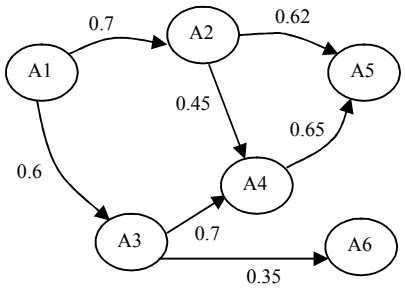

Fig.5. Diagram of example design 3

The row 1 (Table IV) containing larger values with respect to artifact A1, indicates the presence of shotgun surgery bad smell in the design, and any change to artifact A1 will require changes in more number of artifacts (A2, A3, A4, and A5). Column 5 containing larger values with respect to artifact A5, indicates the presence of divergent change bad smell in the design and artifact A5 is likely to undergo frequent changes during evolution.

TABLE IV

DCPP MATRIX FOR THE EXAMPLE DESIGN 3

	A1	A2	A3	A4	A5	A6
A1	1	0.7	0.6	0.6	0.66	0.21
A2		1		0.45	0.73	
A3			1	0.7	0.46	0.35
A4				1	0.65	
A5					1	
A6						1

Both these bad smells indicate design defects. More number of bad smells in the design indicates high complexity. As the software is enhanced, modified, and adapted to new requirements the code becomes more complex and drifts away from its original design, there by lowering the quality of software. To cope with this increased complexity, there is a need for techniques that reduce software complexity by incrementally improving the internal software quality. The research domain that addresses this problem in case of object oriented software development is referred to as refactoring [14]. If the design defects are not corrected using either refactoring or proper redesign they are bound to increase maintenance cost because of high complexity, faulty behavior and low maintainability.

C. Advantages of DCPP Matrix Based Bad Smells Detection Method

Detecting shotgun surgery and divergent change bad smell requires that the design change propagation between artifacts that are connected directly and indirectly should be considered quantitatively. The proposed method considered this aspect quantitatively. The DCPP matrix values not only helps in finding out the presence of two bad smells (shotgun surgery and divergent change) in the design but also helps in number of ways. Proposed detection method which is based on DCPP matrix will help in:

1. Knowing the complete picture of ripple effects in the software, based on which we can assess the software complexity and maintainability. Comparing different designs of software system with respect to complexity and providing optimal maintenance solutions. One of the solutions could be refactoring. For example, the example design 3 indicates that it is more complex than the example designs 1 and 2. In other words, this comparison will help in predicting the maintainability of software. Such type of comparisons for large software systems will be of great use.
2. Validating best practices that are used for software development. The unit DCPP matrix (satisfying condition 5) indicates the violation of basic heuristic (design guideline/principle) of developing low coupled system.
3. Automating the detection process. Mapping the software into URA graph, estimation of cdegree, construction of DCPP and checking for the conditions, can be automated.
4. This matrix can be used to identify artifacts which are going to be affected by a change during software development or as part of software maintenance (corrective, perfective). For large software systems this identification is an important task. This identification helps in making appropriate changes to the affected artifacts. Knowing the affected artifacts due to ripple effect of a change will help in identifying the bad design locations as part of preventive maintenance. Identifying bad design locations enables us to apply appropriate refactorings to make software maintainable. This matrix can be used during software development and software maintenance. This is similar to spiral model, where it can be applied until software retires.

IV. A FRAMEWORK FOR OBJECT ORIENTED SOFTWARE DESIGN QUALITY IMPROVEMENT

In general, a framework is a real or conceptual structure intended to serve as a support or guide for the building of something that expands the structure into something useful [16]. A framework (D^3ARTI: design defects detection and refactoring to improve) is proposed for object oriented design quality improvement. This framework is being taken as a support or guide in formulating better design defects detection methods/approaches, methods and tools for refactoring. These methods will be based on metrics, bad smells, design heuristics, and other novel techniques like DCPP (design change propagation probability) matrix. There is a need for processes, methods, and tools that address refactoring in more consistent, generic, scalable, and flexible way [15]. As part of this bigger aim a method for detecting design defects (indicated by the presence of bad smells) is proposed in this paper. The proposed method is based on DCPP matrix. As shown in Fig. 6, this matrix should be constructed as part of phase 2 before refactoring and as part of phase 7 after refactorings. The DCPP matrix which is reconstructed as part of phase 7 can be used to ascertain the elimination of design defects and hence the improvement in design quality. The proposed framework has an iterative characteristic similar to spiral model. The detection process and refactoring (improvement) goes on in

iterations until the design is free of defects or required quality is attained.

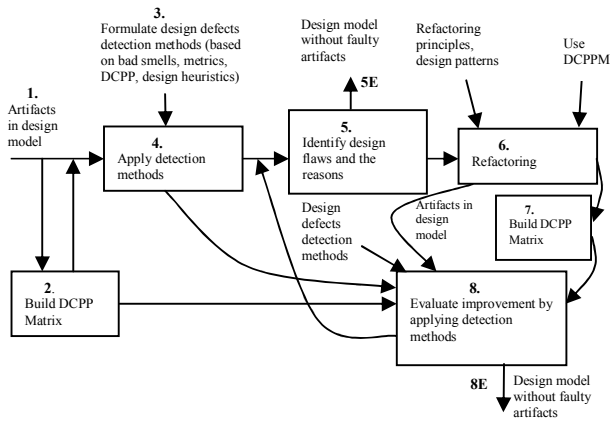

Fig.6. Framework for object oriented software design quality improvement (D^3ARTI)

V. Conclusions and Future Directions

The use of DCPP matrix based bad smells detection method, for detecting two bad smells (shotgun surgery and divergent change) is explored. The proposed method is a quantitative method for detecting bad smells. Presence of bad smells in the design, indicate design defects. Three different example designs are considered for analysis in this paper. These example designs are represented by URA graphs. Cdegree values for adjacent artifacts (artifacts in the URA graph) and combined cdegree values for artifacts that are connected through intermediate artifacts are calculated. Using these cdegree values, DCPP matrices are constructed. In one example design, shotgun surgery bad smell and in the second example design, divergent change bad smell is detected. In the third example design, two bad smells are detected. The required refactorings for the two bad smells are suggested to improve the quality of design. The number of advantages of proposed DCPP matrix based bad smells detection method are discussed. The broader framework in which this detection method is used is given.

The use of DCPP matrix based bad smells detection method to detect the two bad smells (shotgun surgery and divergent change) has to be evaluated empirically using a case study. Detection of the bad smell "parallel inheritance hierarchies" under the category "maintenance smells", will be considered in our future work. In addition, while applying refactorings, the inclusion of design patterns into the overall design should be considered. Design patterns improve the maintainability of software [17]. The ideas presented in this paper will be taken further by using and studying as part of framework (given in section IV), design change propagation probabilities between artifacts and improvement in design quality due to refactorings with and without design patterns for object oriented software.

Acknowledgment

Authors of this paper would like to acknowledge and express their thanks to JNTU College of Engineering, Anantapur and CVR college of Engineering, Ibrahimpatnam, Hyderabad, of Jawaharlal Technological University, for giving financial support for attending this conference.

References

[1] Ladan Tahvildari, Kostas Kontogiannis, *A metric-Based Approach to Enhance Design Quality Through Meta-Pattern Transformations,* Proceedings of the Seventh IEEE European Conference on Software Maintenance and Reengineering (CSMR'03), 2003, pp. 183-192.

[2] Eva van Emden, Leon Moonen, *Java Quality Assurance by Detecting Code Smells*, Proceedings of the 9th IEEE Working Conference on Reverse Engineering (WCRE'02), 2002, pp. 97-106.

[3] A Ananda Rao, D. Janaki Ram, *Software Design Versioning using Propagation Probability Matrix,* In Proceedings of Third International Conference on Computer Applications (ICCA 2005), Yangon, Myanmar, March 2005

[4] S. Srinath, *URA: A Paradigm for Context Sensitive Reuse*, A Thesis of Master of Science & Engineering, Indian Institute of Technology, Madras, India, April 1998.

[5] Shyam R. Chidamber, Chris F. Kemerer, *A Metrics Suite for Object Oriented Design*, IEEE Trans. Software Eng., vol.20, no.6, June 1994, pp. 476-493.

[6] Alexander Chatzigeorgiou, Spiros Xanthos, George Stephanides, *Evaluating Object-Oriented Designs with Link Analysis,* Proceedings of the 26th IEEE International Conference on Software Engineering (ICSE '04), 2004, pp. 656-665.

[7] Haider Bilal, Sue Black, *Computing Ripple Effect for Object Oriented Software, Quantitative Approaches in Object Oriented Software Engineering* (QAOOSE) Workshop, Nantes, France, July 3rd 2006.

[8] R. Marinescu, *Detection Strategies: Metrics-Based Rules for Detecting Design Flaws,* In Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM), September 2004, pp.350-359.

[9] Jonathan Buckner, Josph Buchta, Maksym Petrenko, Vaclav Rajlich, *JRipples: A Tool for Program Comprehension during Incremental Change,* Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC '05), 2005, pp. 149-152.

[10] Mazeiar Salehie, Shimin Li, Ladan Tahvildari, *A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws,* Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC '06), 2006, pp.159-168.

[11] Lionel C. Brand, Jurgen Wust, Hakim Lounis, *Using Coupling Measurement for Impact Analysis in Object-Oriented Systems,* Proceedings of the IEEE International Conference on System Maintenance (ICSM'99), 1999, pp. 475-482.

[12] Jitender Kumar Chhabra, K.K.Aggarwal, *Measurement of Intra-Class & Inter-Class Weakness for Object-Oriented Software*, Proceedings of the Third IEEE International Conference on Information Technology: New Generations (ITNG'06), 2006, pp. 155-160.

[13] Martin Fowler, K.Beck, J.Brant, W.Opdyke, D.Roberts, *Refactoring: Improving the Design of Existing Code.* Addison- Wesley, New York, 1999

[14] W .F. Opdyke, *Refactoring : A Program Restructuring Aid in Designing Object-Oriented Application Frameworks,* PhD thesis, Univ. of Illinois at Urbana-Champaign, 1992.

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

[16] http://whatis.techtarget.com/definition/0,,sid9_gci1103696,00.html

[17] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns : Elements of Reusable Object- Oriented Software*. Pearson Edn. Inc., 2006