Article

Refactoring of non-dispatchable flaws in the design model based on coupling

Syed Uzair Ahmad[1], **Muhammad Naeem**[2], **Fawad Qayum**[3], **Faqir Gul**[1], **Faisal Bahadur**[1], **Hafiz Abdul Wahab**[1]

[1]Department of Information Technology, Hazara University, Mansehra, Pakistan

[2]Department of Information Technology, Abbottabad University of Science and Technology, Abbottabad, Pakistan

[3]Department of Mathematics, Hazara University, Mansehra, Pakistan

[4]Department of IT and CS, University of Malakand, Pakistan

E-mail: ssuab.kk@gmail.com, Naeem@hu.edu.pk, Findfawad@yahoo.com, Gul@hu.edu.pk, Msosfaisal@gmail.com, Wahabmaths@yahoo.com

Received 23 June 2015; *Accepted 5 July 2015*; *Published online 1 September 2015*

Abstract

It is always better to detect and dispatch flaws at design level before the start of development for better and economic results. Refactoring is considered as a better way to address the design flaws. To the best of our knowledge, none of the available techniques targets non-dispatchable flaws of the design model in their approaches. In this paper, we are focused on multiple aspects that have been missed by the existing researchers of refactoring. For example, use of coupling to define flaws in the design model; secondly, use of refactoring to address the non-dispatchable flaws in the design models; thirdly, confirmation that whether addressing of a design flaw caused other flaws or not. Furthermore, we have used real life example of a telephonic call system to elaborate our approach.

Keywords dispatch; design flaw; refactoring; coupling.

Selforganizology
ISSN 2410-0080
URL: http://www.iaees.org/publications/journals/selforganizology/online-version.asp
RSS: http://www.iaees.org/publications/journals/ selforganizology/rss.xml
E-mail: selforganizology@iaees.org
Editor-in-Chief: WenJun Zhang
Publisher: International Academy of Ecology and Environmental Sciences

1 Introduction

A design flaw is defined as the classes or objects having an irregular or no association with each other (Objecteeing, 2015; Seidewitz, 2003). In other words, any design pattern that may affect the quality of software is called a design flaw (Mekruksavanich, 2011). This is advised to detect and remove the flaws that may exist in the design of a system. Hence, when a design flaw is found, normally researchers apply some techniques like refactoring to refine the design from that flaw (Yaowarattanaprasert and Muenchaisri, 2013). The later the flaws are handled the harder to understand, maintain the system (Mekruksavanich, 2011; Tahvildari and Kontogiannis, 2003; Mens et al., 2004; Bacchelli, 2010; Saxena and Kumar, 2012).

2 Background

Fowler and Beck in (1999) defines the process of Refactoring as: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor, you are improving the design of the code after it has been written". The refactoring can be summarized by five basic operations as addition, removal, move, generalization and specialization of modeling elements. These operations are discussed as:

- Addition of features and relations to a class can be done when the new feature does not have the same signature as any other features owned by the class or by its parents.
- Removal of associations and features can only be done when these elements are not referenced in the whole model.
- Move is used to transfer a method from a class to another, and create a forwarder method in the former.
- Generalization refactoring can be applied to elements owned by classes, such as attributes, methods, operations, association ends and state charts. It consists in the integration of two or more elements into a single one which is transferred to a common super class. Since private features are not accessible within the subclasses, they cannot be moved.
- Specialization refactoring is the exact opposite of Generalization (Sunyé, 2001).

Now it is important to know that how we can apply refactoring on different models, and how to find that which model needs to be refactored. Here we use coupling technique for finding that model shall be refactored or not. Further the description on coupling as under:

Definition1: Coupling

Coupling is defined as the degree of dependence between subsystems.

Degree of dependence shows the strength of connection of different elements of a system. For example,

$$CBO = \frac{Number\ of\ Links}{Number\ of\ Classes}$$

finding the coupling between objects (CBO) we can use the formula:

Normal range of coupling is from 1 to 4. More than 4 mean that tightly coupled and would complicate in testing and modification (Objeteering, 2015). Less than 1 means no coupling and is out of the system. Thus it shall not be considered as a part of the system.

Types of coupling

1) Message: Component communicates via message passing.
2) Content: a module depends on the internal working of another module
3) Common: two modules share the same global data.
4) External: modules share an externally imposed data format, or communication protocol.
5) Control: one module controls the flow of another, by passing it information on what to do.
6) Stamp: modules share a composite data structure and use only part of it.

The coupling shows all the associations, but to know the description of these associations and details we are going to use Meta model of the models.

Meta model

A Meta model is an arrangement model for a class where each class is itself a valid model articulated in a definite modeling language. A Meta model makes statements about what can be articulated in the valid models

of a definite modeling language. The Meta model conception for software modeling is also mainly important because it forms the foundation for the UML definition (Yaowarattanaprasert and Muenchaisri, 2013).

The rest of the paper is arranged as follows: Sections 2 and 3 contain discussion about the related approaches and our methodology, respectively. The Section 4 elaborates on the case study for the implementation of our methodology, while the Section 5 concludes the paper.

3 Related Work

Moha (2007) provided a systematic method to specify design defects accurately. Their approach is based on detection and correction algorithms by using refactoring semi-automatically. To apply and validate these algorithms on open-source object-oriented programs was used to show that method allows the systematic description, detection, and correction of design defects with a reasonable precision.

Mekruksavanich (2011) proposed a methodology for detection of design flaws. Symbolic logic representation and analytical learning technique are used to diagnose design flaws in simple way and to extrapolate patterned rules for complex flaws. The methodology is validated by detecting design flaws in an open-source system.

Saxena and Kuma (2012) helped to find the flaw in the design model and to remove it as early as possible. They used the flaw pattern for finding the flaw. When design flaw is detected based in the design pattern, the process exits after dispatching that flaw, the proposed approach was composed of model representation of design model and flaws detection using flaw patterns. The design models of UML class and sequence diagrams were used as an input. It would be transformed to the proposed representation model. In detecting flaws, flaw patterns are used in checking against the representation model. This study covered flaw patterns for detecting Large Class, Refused Bequest, and Middle Man.

In Mohamed et al. (2011), authors used the approach of automatic flaw detection in design model. To find the number of flaw number of classes and detection. Which was based on model qualities metrics and design flaws, author suggest a new demarche allowing the mechanized finding of model refactoring opportunities and the assisted model restructuration. Which focused on class and sequence diagrams. That developed a software call's M-Refactor for those works.

According to Trifu et al. (2004) authors used the flaw detection and correction. The process as problem detected, developers obtain a list of design flaws together with their location in the system. The necessary transformations that removed them were left to their own judgment and experience. The mapping between specific design flaw and code transformations is removed.

In Kessentini (2011) authors used an approach to detect the flaw in design and correct the flaw in the source code. Their approach support automatic generation of rules to detect defects by the help of genetic programming. Using a genetic algorithm, adjustment solutions are found by combining refactoring operations in such a way to reduce the number of detected defects. The detection system is physically specified. Projected corrections fix, in standard, more than 74% of detected defects.

Alikacem and Sahraoui (2010) provide support for source code analysis. They proposed a rule-based approach that allowed the specification and detection of flaws. The approach provided a new language to describe flaws as sets of rules. The latter are translated into Jess's rule format, and given as input to Jess inference engine. The current work is an extension of our source code analysis platform and PatOIS, a metric description language. A main advantage of his approach was its extensibility since the tool is not limited to a set of predefined flaws. Existing flaws could be modified to a specific context and new ones could be added.

Budi et al. (2011) provided a framework that automatically labels classes as Boundary, Control, or Entity, and detects design flaws of the rules associated with each stereotype. Their evaluation with programs

developed by both novice and expert developers show that his technique is able to detect many design flaws accurately.

The main theme of authors in the paper is to find flaw through metric base and convert it into code may in Java or C++. They defined such detection strategies for capturing around ten important flaws of object-oriented design found in the literature and validated the approach experimentally on multiple large-scale case-studies (Marinescu, 2004).

Marinescu (2003) focused on flaw detection through metric base and converted into object-oriented system. This paper presented a metrics-based approach for detecting design problems, which describes two concrete techniques for the detection of two well-known design flaws found in the literature. By an experiment it was showed that the proposed technique found indeed real flaws in the system and it suggests that, based on the same approach.

Moha et al. (2008) used an approach propose a novel approach for defect removal in object-oriented programs that combines the efficiency of metrics with the theoretical strength of formal concept analysis Algorithm. They suggested a novel approach for defect deduction in object-oriented programs that combines the usefulness of metrics with the hypothetical power of formal concept analysis, and case study of an exact fault.

Simon et al. (2006) have worked for finding bad smells. With four typical refactorings and present both a tool supporting the identification and case studies of its application. They showed that special kind of metrics can support these skewed perceptions and thus can be used as effective and efficient way to get support for the decision where to apply which refactoring. They demonstrate this loom for four typical refactorings and present both a tool supporting the classification and case studies of its function.

In the above mentioned techniques there is no single technique that targets dispatchable and non-dispatchable flaws. Similarly There is no confirmation check in the available approaches that whether a flaw has been removed or not. In the existing techniques, there isn't any flaw detection technique that uses coupling. In our work, we are going to address all the weakness discussed above and used a process which will be further discussed in our approach.

4 Our Approach

We specifically address the design flaws because if flaws flow down to further development phases, they become more costly to be addressed. Our approach consists of the following steps:

Step 1: Domain analysis and metrics identification
Step 2: Modeling and meta modeling
Step 3: Flaw detection and flaw pattern
Step 4: Option if flaw >1 or no flaw found
Step 5: Condition check for dispatch and non-dispatchable flaw

The systematic view of our approach:

Step 1 (Domain Analysis): We will do the analysis of the domain area, in this step we associate manually with each design defect to detect them and set refactoring by using metric based identification to find class and their association through coupling.

Step 2 (Modeling): In modeling, the Meta model for software modeling is important, because it forms the basis for the UML definition. The UML specification document is indeed a Meta model for UML. That is, it includes a set of statements that must not be false for any valid UML model. (This Meta model, in its entirety,

includes all the concrete graphical notation, abstract syntax, and semantics for UML.)

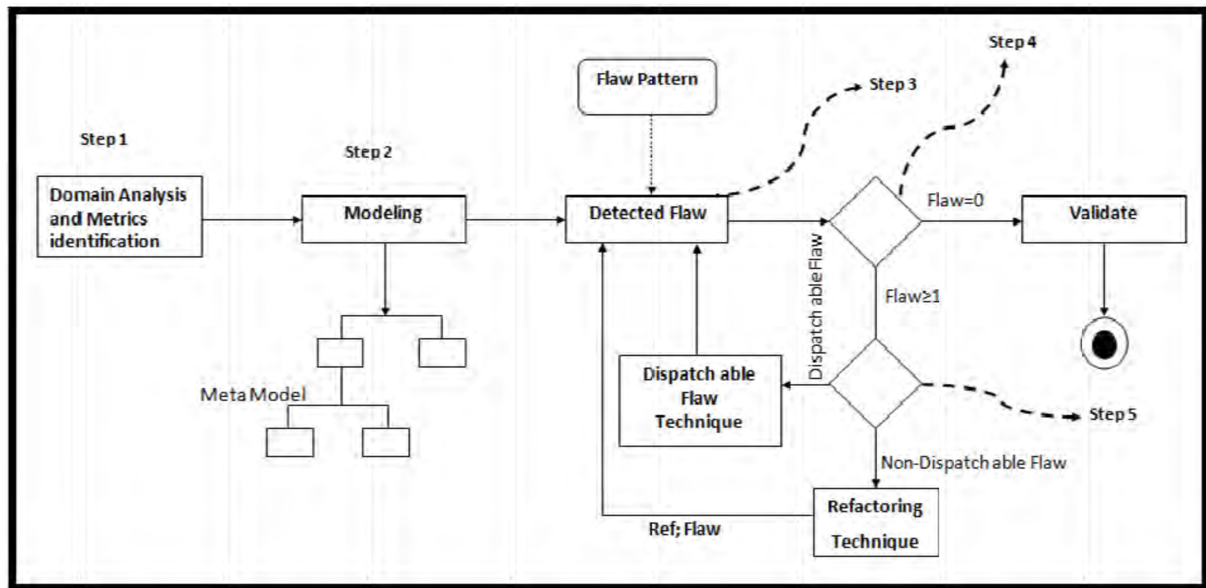


Fig. 1 Our Approach.

Step 3 (Flaw Detection): In this step, we find the flaws by using flaw detection patterns. Basically, a pattern is a format which will identify a flaw in the model. We use coupling for the detection of flaws. Coupling measures the strength of all relationships between functional units. For the calculation of coupling between elements, we use the formula that is already discussed in Section 1.

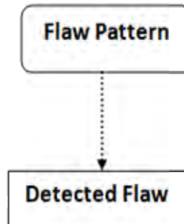


Fig. 2 Flaw Detection.

Step 4 (Flaw does exist or not): Using the above formula, if flaw exists then to be refactored or dispatched, if not then exit. Using the pattern as depicted in Fig. 3

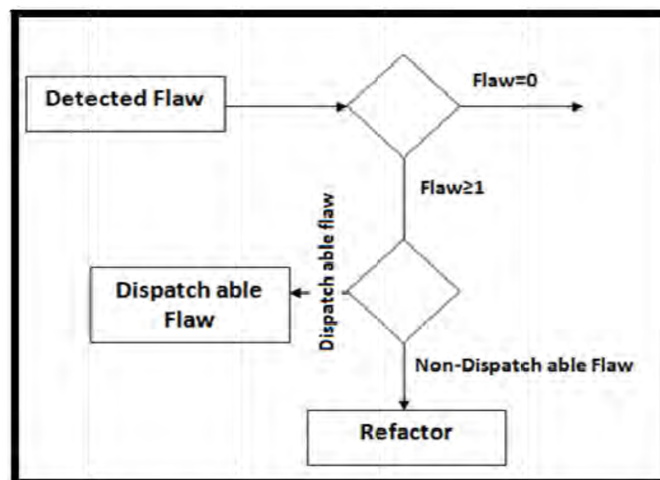


Fig. 3 Finding dispatchable and non-dispatchable flaws

When the flaw =0, it will exit else if flaw≥1then condition shall be checked. That either flaw is dispatchable or non-dispatchable, if dispatchable go to dispatchable module otherwise non-dispatchable and go for refactoring to refactor module. As shown in the Fig. 3 above.

Step 5: (Flaw checking of dispatchable or non-dispatchable): Condition checking whether the flaw is dispatch able or non-dispatchable. The dispatchable flaw goes to dispatchable flaw module and the non-dispatchable flaw goes to the non-dispatchable flaw module as shown in Fig. 4.

First condition is to check that if flaw is dispatchable, the flaw goes to the dispatchable module and removed there. The second condition: If the flaw is non dispatchable and been removed through refactoring a tag Ref; attached to the refactored flaw as a comment for the detected flaw module to understand that this flaw has refactored and didn't need to catch it again. Both from dispatch and non-dispatchable flaw modules the model goes again for rechecking flaws to the detection flaw module. The cycle continues until all flaws are dispatched or refactored.

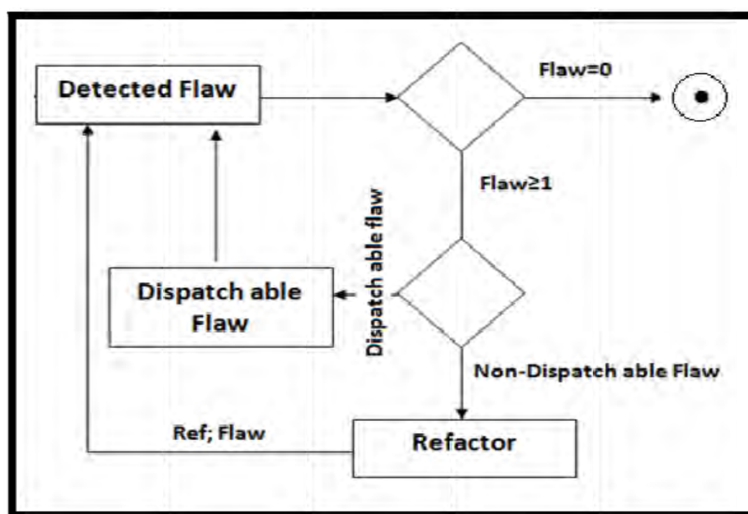


Fig. 4 Condition check and non-dispatchable flaw.

5 An Illustrative Example

Let us consider an example of a telephone with different states (inspired by (Sunyé et al, 2001)).We apply the process to find the flaws and remove the dispatchable and non-dispatchable flaws through our approach. We will check that how we can improve our design model through our approach the case is under in Fig. 5:

In order to improve understandability.We group the states modeling thebehavior of the phone. When it is in use into a compound state, thus segregatingtheidle state and allowing the use of high-level transitions.

To obtain the result shown in Fig. 6, four refactoring steps are needed:

1) Create a composite super state, named Active, surrounding the whole current diagram.
2) Move Idle and the initial pseudo state out of Active.
3) Merge the "hang up" transitions into a transition leaving the boundary of Active.

4) Finally, split the “lift" transition into a transition from Idle to the boundary of Active and a default pseudo state/transition targeting Dial Tone.

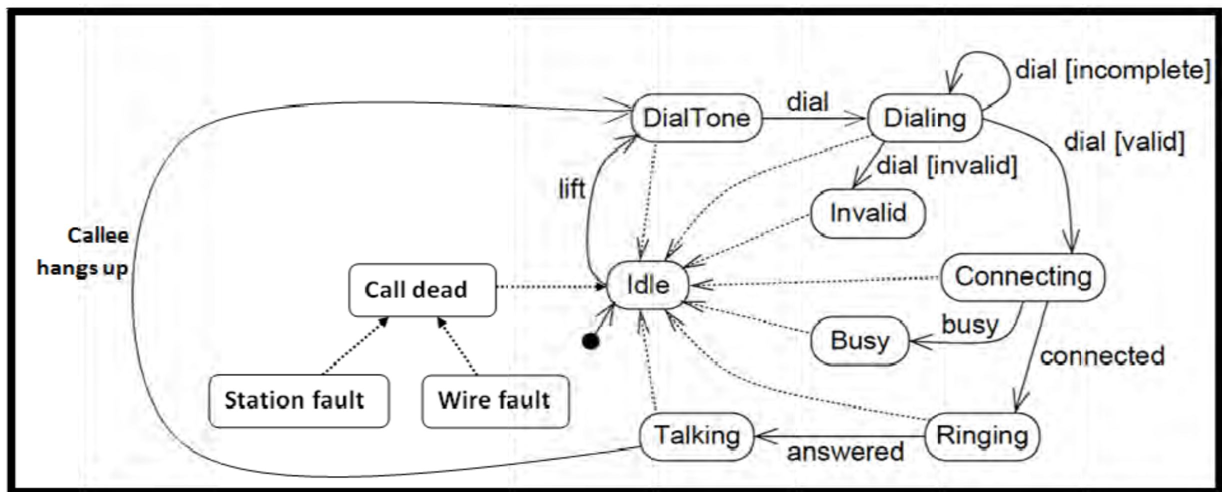
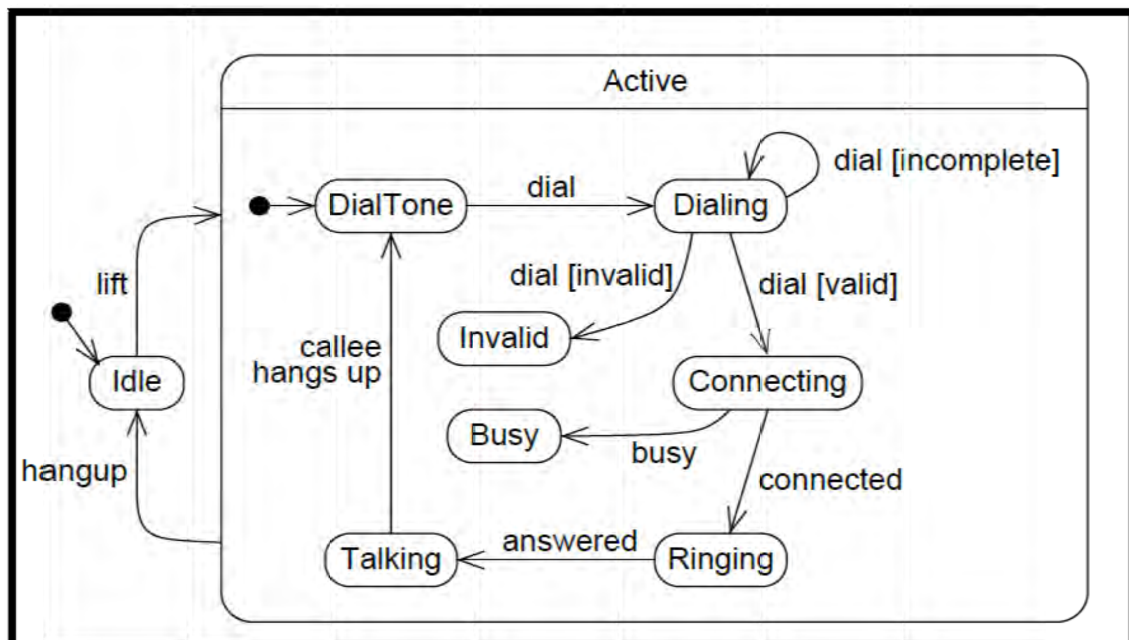


Fig. 5 Initial phone state diagram (dotted transitions are triggered when the caller hangs up.

These are the justifications for the previous transformations:
- Creating a surrounding state is trivially behavior-preserving.
- Moving the Idle state out is legal here: Active has no entry or exit actions, and so the execution order of existing actions is unchanged.
- Transitions exiting Active can be folded to a top-level transition since they are equivalent (they hold the same label and target the same state).
- The replacement of the “lift" transition by a top-level one is possible given that there is no other top-level transition entering Active (Sunyé, 2001).



- **Fig. 6** Passing through process.

According to the formula of coupling as idle state have more transitions. Hence it considers being a flaw but it is a flaw which we cannot dispatch totally. So, we refactored it and refined it as in Fig. 6.

On the other hand the call dead, station fault, wire fault are those states which has only doted arrow means hang up transitions. So when hang up transitions are refactored then those states have no more transitions with Idle. As no transitions according to the definition of coupling its mean it is not a part of this model and shall be dispatch, so we dispatch all these three states as in Fig. 6. The refactoring presented here can be summarized in five basic operations: *addition, removal, move, generalization* and *specialization* of modeling elements.

The two last actions use the generalization relationship to transfer elements up and down a class hierarchy. Most part of the modeling elements composing the class diagram may have a direct connection to the elements of other views. Therefore, some of the refactoring that apply to class diagrams may have an impact on different UML views (Sunyé et al, 2001).

6 Conclusions

Form all our work we are now able to check out a flaw from the design pattern, to find a flaw the statistical formula has been used. After that two sort of flaw may occur the one is that flaw which is dispatchable and the other one which is non-dispatchable. Applying rules for removing both of the flaws if found, we remove the flaws of both sorts and got a flaw free design for high quality of software to produce.

In future, we will apply our approach over the larger case studies. Our work is in process to automate the whole process for automatic checking of flaws using coupling.

References

Alikacem H and Sahraoui HA. 2010. Rule-Based System for Flaw Specification and Detection in Object-Oriented Programs. 13th TOOLS Workshop on Quantitative Approaches in Object-Oriented Software Engineering. 1-11

Bacchelli, Lanza M. 2010. On the Impact of Design Flaws on Software Defects. IEEE 10th International Conference on Quality Software (QSIC), 23-31

Budi A, Lucia, Lo D, Jiang L, Wang S. 2011. Automated Detection of Likely Design Flaws in Layered Architectures, Research Collection School of Information Systems, 1-6

Fowler M, Beck K. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, USA

Kessentini M, Kessentini W, Erradi A. 2011. Example-based Design Defects Detection and Correction. IEEE 19th International Conference Program Comprehension (ICPC). 81-90

Marinescu R. 2001. Detecting Design Flaws via Metrics in Object-Oriented Systems. 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS), 173-182

Marinescu R. 2004. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. International IEEE 20th Conference on Software Maintenance, 350-359

Mekruksavanich S. 2011. Design flaws detection in object-oriented software with analytical learning method. International Journal of e-Education, e-Business, e-Management and e-Learning, 1(3): 210-216

Mens T, Tourw T. 2004. A Survey of Software Refactoring. IEEE Transactions on Software Engineering, 30(2): 126-139

Moha N. 2007. Detection and Correction of Design Defects in Object-Oriented Architectures. Companion to the 22nd ACM Special Interest Group ACM SIGPLAN Conference, 949-950

Moha N, Rezgui J, Gueheneuc Y, Valtchev P, Boussaidi G. 2008. Using FCA to Suggest Refactorings to Correct Design Defects. 4th International Conference on Concept Lattices and Their Applications, LNCS

4923, 269-275

Mohamed M, Romdhani M and Ghedira K. 2011. M-REFACTOR: A new approach and tool for model refactoring. American Resources Policy Network (ARPN) Journal of Systems and Software, 1(4): 1-6

Objecteeing. 2015. http://support.objecteering.com

Saxena V, Kumar S. 2012. Impact of coupling and cohesion in object-oriented technology. Journal of Software Engineering and Applications, 671-676

Seidewitz E. 2003. What Models Mean. IEEE Software. Publisher IEEE Computer Society Press Los Alamitos, CA, USA, 20(5): 26-32

Simon F, Steinbrückner F, Lewerentz C. 2001. Metrics Based Refactoring. IEEE 5th European Conference on Software Maintenance and Reengineering, 30-38

Sunyé, Gerson, Pollet D, Traon Y, and Jézéquel J. 2001. Refactoring UML Models.≪UML≫ 2001--The Unified Modeling Language. Modeling Languages, Concepts and Tools. 134-148, Springer Berlin Heidelberg, Germany

Tahvildari L, Kontogiannis K. 2003. Metric-Based Approach to Enhance Design Quality Through Meta-Pattern Transformations. Proceedings of 7th European Conference on Software Maintenance and Reengineering, 183-192

Trifu A, Seng O, Genssler T. 2004. Automating Design Flaw Correction in Object-Oriented Systems. Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR 2004). 174-183

Yaowarattanaprasert N, Muenchaisri P. 2013. Graphical pattern matching approach for detecting design flaw in design model. International Journal of Advanced Research in Computer Science and Software, 2: 1-5