*Article*

# Validity process for refactored coupling based non-dispatchable flaws in the design model

**Syed Uzair Ahmad**[1], **Muhammad Naeem**[2]
[1]Department of Information Technology, Hazara University, Mansehra, Pakistan
[2]Department of Information Technology, Abbottabad University of Science and Technology, Abbottabad, Pakistan
E-mail: ssuab.kk@gmail.com, Naeem@hu.edu.pk

*Received 19 November 2015*; *Accepted 22 December 2015*; *Published online 1 March 2016*


**Abstract**
As we know that in the start level, it is better to pick out flaw and refactored it, as soon as possible. It is best way to address flaw through refactoring. All of techniques cover the area of refactoring or validation, no approach has been used to refactor the non dispatchable flaw based on coupling and validate that refactored design. In this paper we are going to address the missing point as for founded in previous works. For example, we will use model based on coupling defined model, second will refactored it, third will check bad fixes and at last we will validate the model, the validation needs because to ensure that the model has changed design or in same design.

**Keywords** validation; flaw; coupling; bad fixes; refactoring; dispatch.

**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

According Jennifer Campbell winter 2007 validation means “Are we building the right system?” the other definition by (Jane Hillston September 19, 2003) Validation is the task of demonstrating that the model is a reasonable representation of the actual system “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 as early as possible to do not disturb the next phases of the system. When a design flaw is found, normally researchers apply some techniques like refactoring to refine the design from that flaw (Yaowarattanaprasert and Muenchaisri, 2013). Later on the flaws are very difficult to handle them all and harder to understand, maintain the system (Mekruksavanich, 2011; Tahvildari and Kontogiannis, 2003; Mens et al., 2004; Bacchelli, 2010; Saxena and Kumar, 2012). So over all it is good to pick out flaw and refactor it as possible, and for checking validate that model to ensure that is model has been disturbed or not.

## 2 Back Ground

Jane Hillston (September 19, 2003) define validation as: "Validation is the task of demonstrating that the model is a reasonable representation of the actual system". Validation concerned with building the right model. It is utilized to determine that a model is an accurate representation of the real system. Validation is usually achieved through the calibration of the model, an iterative process of comparing the model to actual system behavior and using the discrepancies between the two, and the insights gained, to improve the model. This process is repeated until model accuracy is judged to be acceptable. A model is usually developed to analyze a particular problem and may therefore represent different parts of the system at different levels of abstraction.

As a result, the model may have different levels of validity for different parts of the system across the full spectrum of system behaviour For most models there are three separate aspects which should be considered during model validation which are Assumptions, input parameter values and distributions and Output values and conclusions.

There are three validation models or strategies for validating data:

- Rejecting bad data: creating a set of undesirable data and rejecting them. This model is also known as "blacklist" approach.
- Accepting only known good data: data constrained by Five Primary Security Input Validation Attributes which are: type, length, character set, format, reasonableness. Data is rejected unless it matches for known good data. This model is also known as "white list" approach.
- Sanitizing data: sanitizing a defined set of dangerous data so that it does not pose a threat to the software (PedramHayati 2008). It is important to know that how we can apply validation process to our model. Before validation we will use metric to find different class and object number in model. Detail of metric use and types are under:

**Definition1: MetricModel**

*Model metrics are for estimating the size or the amount of information contained in a model.*

We can use metrics according to pointed situation. Each different object and class relation has different metric, as given below.

**Table 1** Software metrics for UML models (Abbreviation UML Metric).

| | |
|---|---|
| CBC | Coupling between classes |
| DIT | Depth of inheritance tree |
| NACM | Number of actors in a model |
| NACU | Number of actors associated with a use case |
| NAGM | Number of the aggregations in a model |
| NASC | Number of the associations linked to a class |
| NASM | Number of the associations in a model |
| NATC1 | Number of the attributes in a class - unweight |
| NATC2 | Number of the attributes in a class - weighted |
| NCM | Number of the classes in a model |
| NDM | Number of the directly dispatched messages of a message |
| NDM* | Number of the elements in the transitive closure of the directly dispatched messages of a message |
| NIM | Number of the inheritance relations in a model |
| NMM | Number of the messages in a model |
| NMRC | Number of messages received by the instantiated objects of a class |
| NMSC | Number of messages sent by the instantiated objects of a class |
| NMU | Number of messages associated with a use case |
| NOM | Number of the objects in a model |
| NOPC1 | Number of the operations in a class - unweight |
| NOPC2 | Number of the operations in a class - weighted |

| | |
|---|---|
| NPM | Number of the packages in a model |
| NSCU | Number of system classes associated with a use case |
| NSUBC | Number of the subclasses of a class |
| NSUBC* | Number of the elements in the transitive closure of the subclasses of a class |
| NSUPC | Number of the super classes of a class |
| NSUPC* | Number of the elements in the transitive closure of the super classes of a class |
| NUM | Number of the use cases in a model |

## 3 Model Metrics

1. Number of the packages in a model (NPM): This metric counts the number of packages in a model. Package is a way of managing closely related modeling elements together. Also by using packages, naming conflicts can be avoided.
2. Number of the classes in a model (NCM): A class in a model is an instance of the meta class "class". This metric counts the number of classes in a model. This metric is comparable to the traditional LOC (lines of code) or a more advances McCabe's cyclomatic complexity (MVG) metric for estimating the size of a system [7]. Thus, in OOP this metric can be used to compare sizes of systems.
3. Number of actors in a model (NAM): According to the UML specification [10], an actor is a special class whose stereotype is "Actor". This metric computes the number of actors in a model.
4. Number of the use cases in a model (NUM): The rationale behind the inclusion of this metric is that a use case represents a coherent unit of functionality provided by a system, a subsystem, or a class.
5. Number of the objects in a model (NOM): In a similar manner that a class is an instance of the metaclass "Class", an object is an instance of a class.
6. Number of the messages in a model (NMM): A message is an instance of the metaclass "Message". Messages are exchanged between objects manifesting various interactions.
7. Number of the associations in a model (NASM): An association is a connection, or a link, between classes. This metric is useful for estimating the scale of relationships between classes.
8. Number of the aggregations in a model (NAGM): An aggregation is a special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part.
9. Number of the inheritance relations in a model (NIM): This metric counts the number of generalization relationships between classes existing in a model.

## 4 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 refactoring's 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 refactoring's and present both a tool supporting the classification and case studies of its function.

Syed et al. (2015) worked on Refactoring of non-dispatchable flaws in the design model based on coupling. In this work he cover all the aspects which was blank in the above all works. But one thing which is validation still remained in his work.

The entire above techniques draw backs have been covered by Syed el al. but, one thing still remains that is: When the process has been done of dispatch and non-dispatchable flaw, how we will find that our model has been in required position. For this we are going to use an approach to validate our model after passing through the process as under.

## 5 Our Approach

The previous work was divided into six steps, now we are going to fix the previous error in approach. For this we will add another one step in the existing approach. So the new step will be validation of model.

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

Step 6: validation check

The systematic view of our approach is

**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. In the metric forms that shows different sort of associations.

**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.

**Step 4: (Flaw does exist or not)**: Using the above formula, if flaw exists then to be refactored or dispatched, if not then exit. 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 dispatch-able module otherwise non-dispatchable and go for refactoring to refactor module.

**Step 5: (Flaw checking of dispatch-able or non-dispatch-able):** Condition checking whether the flaw is dispatch able or non-dispatch able. The dispatch able flaw goes to dispatch able flaw module and the non-dispatch able flaw goes to the non-dispatch able flaw module. First condition is to check that if flaw is dispatch able, the flaw goes to the dispatch able module and removed there. The second condition, If flaw is non-dispatch able 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-dispatch able flaw modules the model goes again for rechecking flaws to the detection flaw module. The cycle continues until all flaws are dispatched or refactored.

**Step 6: (validation of model):** here the major work takes palace, as we refactor the dispatch and non-dispatchable flaws. But we don't know that the refactored model has been changed from original shape or not, for this checking we will validate our model. The last step of validation take place with the comparison of step 2, where we made metrics. Simply we check all the associations and compare last step with step 2, if both of step have similar associations so model will go to exit state else model will be rearranged according to step 2 associations.

The whole system over view is indicated in Fig.1.

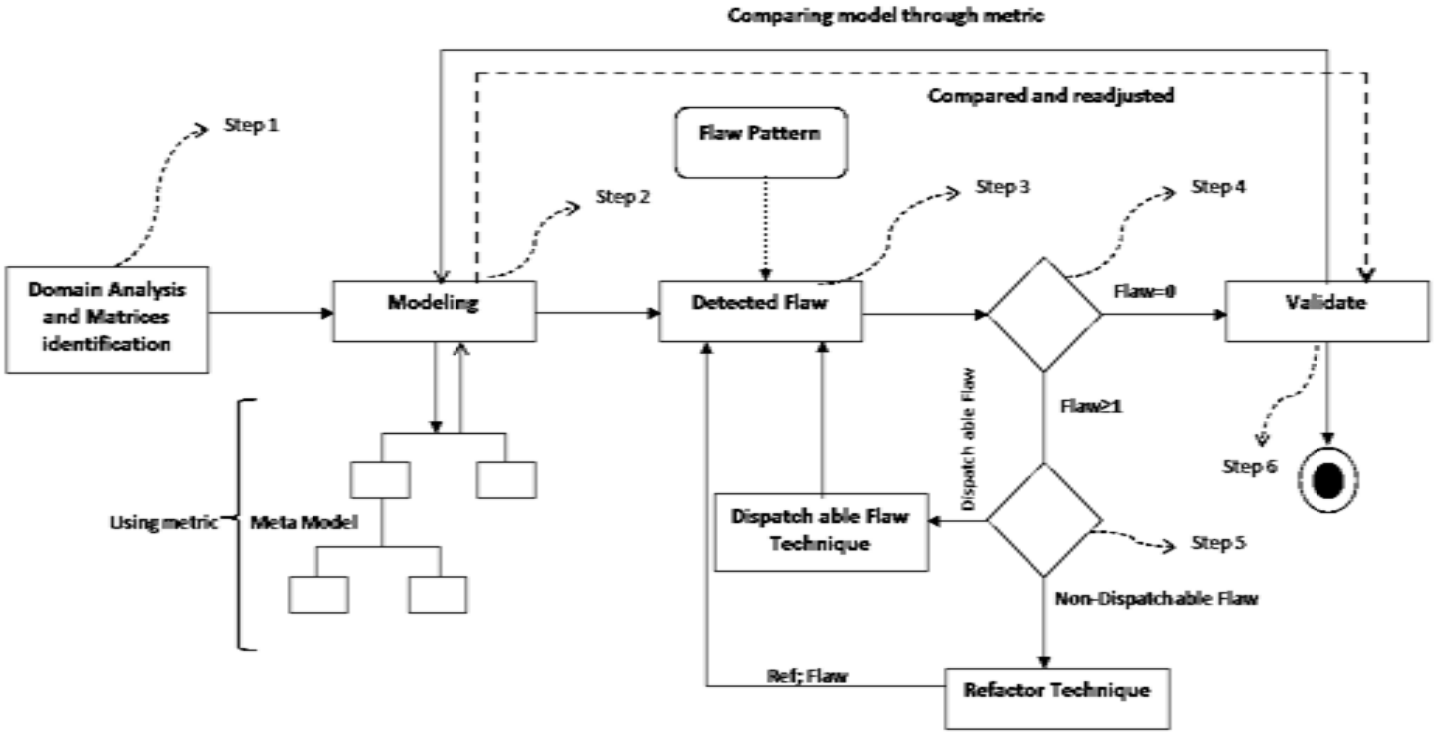


**Fig. 1** Our approach.

## 6 Class Diagram Example

The class diagram given in Fig. 1 is a simple model of a graphical hierarchy for a vector graphics program. Graphics are constituted of geometric Primitives and sub graphs; they have a method to be displayed. Primitives have a matrix attribute representing how they are scaled, rotated or translated in the global coordinate system.

This model has some design flaws; for instance, as Primitives have no inheritance relation with Graphics, they must be treated differently, thus making the code unnecessarily complex. Fortunately, the Composite design pattern addresses this type of problem, where a structure is composed of basic objects that can be recursively grouped in a part-whole hierarchy. We will therefore introduce this pattern in the model through the following steps, leading to the diagram presented in Fig. 2.

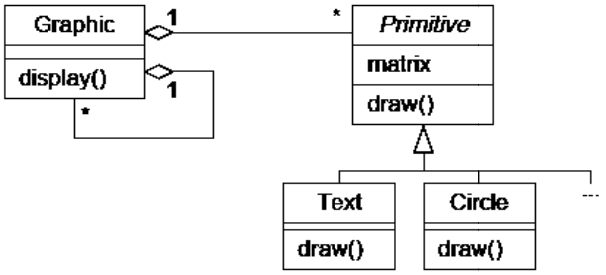


**Fig. 2** Initial class diagram.

1. Renaming the Graphic class to Group;
2. Adding an abstract superclass named Graphic to Group.
3. Making the class Primitive a subclass of Graphic.
4. Merging the Group-Group and Group-Primitive aggregations into Group-Graphic.
5. Finally, we can move relevant methods and attributes up to Graphic.

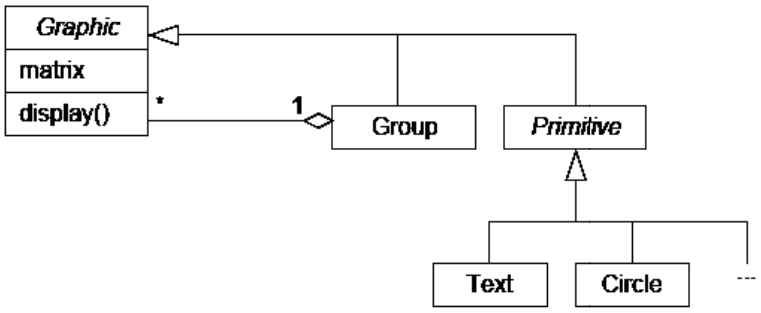


**Fig. 3** Restructured class diagram.

We need to justify why the behavior preservation condition holds for these model transformations: Renaming of a model element does not change anything to the model behavior; provided the new name is legal (i.e. it does not already exist in the model).

The added abstract superclass has no attributes or methods. It is a \empty" model element; its addition has no effect on the model.

Creating a generalization between two classes does not introduce new behavior, provided no conflict (due to multiple inheritance, for instance) is introduced; in our case, Primitive had no superclass and Graphic is empty.

Merging two associations is only allowed when these two associations are disjoint (they do not own the same objects), when the methods invoked through these associations have the same signature, and when the invocation through an association is always followed by an invocation through the other.

Finally, moved methods or attributes to the superclass will simply be inherited afterwards (overriding is not modeled).

While most of these transformations - namely element renaming and the addition of a superclass - do not have an impact on other views, the merging of two associations may require changes on collaborations and object diagrams (GersonSuny, 2001).

## 7 Conclusion

As for performing this sort of process we will be able to refactor all the dispatch able and non-dispatchable flaws, we will also be able to re-structure the model if the model has been misplaced. The model simply makes meta models using metrics and the store that metric code. After performing all the operations for validation the model again comes to modeling state. Here the previous and new model metric compared and find the differences between them. If the model metrics same validate and exit, or else it restructure the model and go to exit state.

## References

Ahmad SU, Naeem M. 2015. Refactoring of non-dispatchable flaws in the design model based on coupling. Selforganizology, 2015, 2(3): 46-54

Alikacem H, 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 IEEE20th 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, 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. 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, Jézéquel J. 2001. Refactoring UML Models. UML2001—TheUnified 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 ThroughMeta-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