International Journal of Information Technology and Knowledge Management
January-June 2011, Volume 4, No. 1, pp. 63-67

AN APPROACH FOR DETECTION AND CORRECTION OF DESIGN DEFECTS IN OBJECT ORIENTED SOFTWARE

Nirmal Kr. Gupta[1] & Mukesh Kr. Rohil[2]

The presence of design defects in object oriented software can have a severe impact on the quality of software. The detection and correction of design defects is important for cost effective maintenance. In this work we propose an automatic detection technique which uses the design patterns as reference to detect the design defects in existing software design. We also propose a correction technique which can refactor the code to meet the design specifications using the concept of class slicing. We can use this technique for any code in which classes are excessively coupled together, thereby not meeting with the good design specifications, for an object oriented software.

1. Introduction

Software systems have become an indispensable part of business and commerce in modern world. Therefore the software quality has become vital both to ensure the proper functioning of the systems and to reduce development and maintenance cost. The quality of software is assured during the whole life cycle of software development, which aims to detect errors earlier during development. The errors caused during software design are called design defects [1]. If these errors and defects are detected earlier in design phase they can not move to the next stage of development or maintenance or in worst case to the deployment phase.

Software is said to be evolved. For example, in software maintenance when the software is enhanced, modified or changed for a new requirement, the software code becomes more complex and the software design changes from the original design. One important step towards maintaining object oriented software is to detect its design defects for ease of future maintenance. According to Fowler [2] a "design defect" is "bad smell of design" mainly due to violation of one or more design principles. These design principles are called as heuristics [3] and these rules should be thought of as a series of warning bells that will ring when violated.

According to Moha [4] the "design defects are bad solution" to recurring design problems in object oriented systems. Design defects are problems resulting from bad design ranging from high level design problems such as antipatterns (anti-pattern is defined as a design pattern that may be commonly used but is ineffective and/or counterproductive in practice [10]) to low level or local problems such as code smells. Since the design defects are because of bad solutions to recurring problems design, whose origins are from poor design practices [5] and also different defects which are "deviations from specifications or requirements". These may result in failures in the operations [6] [7]. They include problems at different levels of granularity: Architectural problems such as anti-patterns [8] and problems such as low bad smells [2] which are usually signs of anti-patterns.

The defects have a negative impact on the quality of object-oriented systems and make it difficult to address debugging and development. Therefore, their detection and correction early in the development process can significantly reduce development costs and maintenance [12]. But their detection requires significant resources in time and personnel and is subject to many errors because the defects usually involve several classes and methods. During maintenance of the software, one needs to analyze the code to understand its structure and behavior. After that the identification of correct location is needed so that new requirements could be added or existing defects could be removed. Also, another challenge is to see that the modifications should not break the functionality of rest of the software. Around 75% of software life cycle efforts and around 80% of development time is devoted to the maintenance of the software [9]. The correction of extension of software can be done by experts who are capable with both the forward and reverse engineering skills. After identification of defects, developers can use refactoring techniques to eliminate defects. Design defects can be classified in three categories namely: Intraclass, Interclass and behavioral design defects [13]. To address intraclass design defect class slicing [14] can be used as a refactoring technique to redesign the classes and maintain their relationship and behavior.

[1,2]CSIS Group, BITS-Pilani, Pilani, Rajasthan, India
Email: [1]nirmalgupta@bits-pilani.ac.in, [2]rohil@bits-pilani.ac.in

2. Background Study

Several approaches have been proposed for detection and correction of software design defects. According to Moha

[4] the detection and correction of design defects due to poor design choices, are difficult because of the lack of precise specifications of defects and tools for their detection and correction. He provides a technique for automatic generation and detection of algorithms from the design specifications. Other techniques described by Radu [15] provide metric based rules to detect ten important flaws of object-oriented design found in the literature. The internal structure of the source code helps the programmers to understand a system to improve the overall design. If something goes wrong with the source code, it gives rise to code smell [2]. Munro [16] addresses the bad smells by identifying the characteristics of a bad smell through the use of a set of software metrics. Moha et al. [17] define design defects as occurring errors in the design of software that come from the absence or the bad use of design patterns. According to Moha the design defects are software defects at the architectural level and their detection and correction is important to improve software quality. They presented several techniques and tools to detect design pattern defects and also described a case study to illustrate the use of these techniques and tools.

Majority of the effort is spent on identifying those parts of the system that are affected by particular design defects, and which need to be redesigned in order to achieve the reengineering goal. Radu [18] presented a metrics-based approach for detecting design problems, and describes two concrete techniques for the detection of two well-known design problems: god-classes and data-classes. The detection and correction of design defects are two different approaches and there exists a conceptual gap between these two. There is a lack of appropriate support for the automated mapping of design defects to possible solutions. Adrian Trifu et. al. [19] describe an approach based on the concept of "correction strategies" which serve as reference descriptions that enable a human assisted tool to plan and perform all necessary steps for the safe removal of detected design defects, with special concern towards the targeted quality goals of the restructuring process.

These techniques are restricted by the services-platforms for discovering the underlying defects. They mainly use the measurements to detect defects, while ignoring other important characteristics of systems, such as architecture. Refactorings are structural transformations that can be applied to a software system to perform design changes without modifying its behavior. They are actually the change suggestions and are not directly applicable over a system. There are different stages of a refactoring technique. A review and a thorough analysis begin before applying a refactoring technique, which results in a list of findings. These various stages of refactoring are described by Mens and Tourwe [20].

3. Defect Detection Approach

We provide an approach to get a systematic method to detect design defects precisely and correct them through a refactoring technique. The basic concept for design defect detection and correction technique is shown in Figure 1.

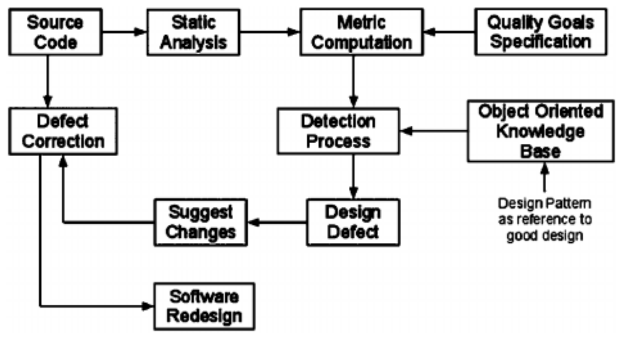

Fig. 1: Design Defect Detection and Correction Process

The detection process is explained below:

3.1. Specify the Quality Goals: For any particular system the quality goals which have more priority than other quality goals must be chosen. The impact of the specific quality factor in relation with that quality goal and the decision of detection of any particular design defect can be determined.

3.2. The Static Program Analysis: It follows the program analysis techniques like control and data flow analysis techniques so that it gives a program dependency graph. This program dependency graph works as an abstract model for the program and can be used effectively by the metric computation block to compute the object oriented metrics for the given program.

3.3. Metric Computation: According to specific quality goals the corresponding quality metrics may be targeted such as computed object oriented metrics like size, complexity, coupling and cohesion metrics. This type of model provides a strategy for detecting design defects using metrics. Metric values are divided into five different levels: "very low", "low", "medium", "high" and "very high". Classes which have metrics values belonging to a specified level are kept as candidate.

3.4. Object Oriented Design Knowledge Base: An object oriented design knowledge base consists of four parts which are called as design principles, design defects, design metric suites and features. The design features relate the primary classifier to this level. A design pattern can be a reference for a good design. The solution given by a design pattern is based on the design concepts of experienced software engineers [21]. The design knowledge base consists of a meta model to represent the design patterns. There exists various meta-models to represent the design patterns but they are not specifically designed towards detection and code transformation. We can use a meta model given in [13].

The design principles consist of rules for good design as described by the meta model for design patterns. These design rules are based on the design principles and their goal is to improve quality factors of the system and to avoid occurrence of design defect.

3.5. Detection Process: The detection process is a two level process. The first level is a primary classifier which aims at indicating a preliminary indication of design defect. Detection of potential defects is done by some primitive rules defined in the object oriented knowledge base. For describing the categories at this level the fuzzy terms are most suitable because they can be configured later for purpose of flexibility and it also reflects the uncertainty of the precise value in detection of design defects.

At the second level of detection, the aim is to detect the design defects accurately. At this level the secondary classifier measures more details about the defects detected by primary classifier. To improve the accuracy of level two measurements we measure each design feature by using more than one metrics.

3.6. Design Defect Identification: Identifying all the possible design defects doesn't make sense because when it is performed automatically it may lead to a large number of probable design defects present in the system. These automatically generated defects must be inspected manually and the user must decide which are relevant according to earlier defined quality goals. It identifies the design defects in terms of design patterns.

3.7. Suggest Changes: In this step the process suggests the necessary changes to be done in the design to achieve the quality goals. These changes are actually UML specifications obtained from design patterns. These change suggestions can be used to do the correction in the defective design.

4. Defect Correction Approach

The correction approach, on receiving the suggested changes, generates the required changes in the code. The technique considers the description of solution and its variation to some given defect. The technique is explained through an example in figure 2.

The technique, based on class slicing [22], targets correction which includes the specific behavior preserving code transformation technique including a redesign proposal. It may contain code refactoring technique, behavior altering transformation, conditional or iterative behavior altering transformations. The code refactoring technique which includes refactoring plan can either be executed automatically or in a semi automatic manner. The overall behavior of the code should not change i.e. the preconditions and post conditions of the method remains the same. In this way the behavior of class object would be same as a whole.

The following are the steps through which the technique will work.

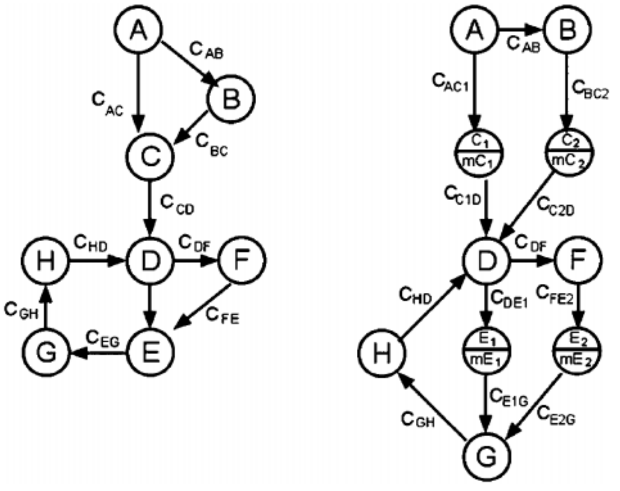

Fig. 2: Defect Correction through Slicing Classes

4.1. Trimming the Classes: This step takes Program Dependency Graph (PDG) as input and forward trim starts with all nodes with no ancestors and removes them and all their edges. After removal the nodes with no ancestors may appear so this trimming will continue until no more nodes can be removed [22]. Similarly the backward trim performs the same operation but starts with classes with no descendants and uses topological sorting to find test orders. The purpose of trimming is to separate classes which have no concerning cycles and to pre-order these classes.

4.2. Refactoring Formalization: The refactoring of the target software architecture must be done step-wise via a number of well-defined, small increments. Each increment might include a top-down refinement activity to detail and complete the software architecture. We can also use bottom-up refactoring activities to refine and clean-up inconsistent or insufficient design decisions. The decision is influenced by the fact that behavior preserving conditions must be checked. Class slicing applies Object Oriented program slicing on PDG to extract class. The strategy to refactoring a class involves various steps. In first step we break Strongly Coupled Classes (SCC).

In Figure 2 for any given two nodes X and Y (represents two classes X and Y) in PDG, C_{XY} denotes the measure of Coupling Between Objects (CBO) between classes X and Y [23]. CBO is measured by counting the number of distinct non- inheritance related class hierarchies on which a class depends. Excessive coupling prevents reuse of classes by other user classes [11].The higher the coupling the more sensitive the system is to changes in other parts of the design, and therefore maintenance is more difficult.

4.3. Slicing Classes: This step searches for a class which can be isolated at first. The main purpose of this step is to search a class that can be isolated. We separate this class to break cycles from the SCC. First of all, we must select one cycle to be considered at a time. But the first priority type is the smallest cycle of all SCCs from cycle detection. If desired class is found, we will use class slicing to separate and order them. After that the adjacency cycle is connected and class slicing is repeated until no cycle remains. We use the class slicing technique discussed by Jaroenpiboonkit and Suwannasart [22] with modification that we have considered CBO between two classes. The value of CBO should be below a threshold value defined by the designer of the classes. If it is not then the corresponding class must be sliced. For example in Figure 2 C_{CD} is above threshold and therefore class C must be sliced. The class C itself is also coupled with classes A and B in such a way that these classes have methods which use the methods of class C. The class C is analyzed to find which of its methods are called by classes A and B. We make two groups of methods in class C which are called by classes A and B respectively. These are designated as mC_1 and mC_2 respectively. The class C now can be sliced into C_1 and C_2 having methods mC_1 and mC_2 respectively. These sliced classes may also have coupling with other classes. If the CBO of sliced classes is also greater than threshold then slicing again to be applied for target class.

4.4. Software Redesign: Software redesign is concerned with identiûcation, application and refinement of new ways to improve and transform software processes. Based on the suggestions to change the design this step modifies the earlier design in defect.

4.5. Result Refinement: The detected design defects must be checked again to remove the false positives. The source code is again analyzed for the detected/undetected design defects. The design defects which look to affect severely must be figured out first.

5. Conclusion and Future Work

Detection of design defects is important for improving the quality of object oriented software systems. By automated correction of these defects at appropriate time, total cost of software development is reduced because the manual detection of defective design is tedious and time consuming. In this work we presented a systematic technique that covers the process of design defects detection using object oriented patterns as knowledge base, and also proposed a correction technique which uses class slicing to refactor the classes to improve the design. This method may facilitate the development of concrete tools for the detection and correction of design defects. The tool developed will be helpful for validating the concepts by applying to various case studies. The limitation of this approach is that it may take more effort in refactoring if most of the classes are highly coupled with other classes.

References

[1] Subramanyam, R. Krishnan, M. S., “Empirical Analysis of CK Metrics for Object-oriented Design Complexity: implications for Software Defects”, IEEE Transactions on Software Engineering, 2003, 29, Part 4, Pages 297-310.

[2] M. Fowler, “Refactoring - Improving the Design of Existing Code”, Addison-Wesley, 1st Edition, June 1999.

[3] Riel, A. J. 1996 Object-Oriented Design Heuristics, 1st. Addison-Wesley Longman Publishing Co., Inc.

[4] Moha, N. 2007, “Detection and Correction of Design Defects in Object-oriented Designs”, In Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (Montreal, Quebec, Canada, October 21-25, 2007). OOPSLA ’07. ACM, New York, NY, 949-950.

[5] Perry, D. E. and Wolf, A. L. 1992, “Foundations for the Study of Software Architecture”, SIGSOFT Softw. Eng. Notes 17, 4 (Oct. 1992), 40-52]. They are the Opposite of Design Patterns [Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley.

[6] Fenton, N. E. and Neil, “M. 2000. Software Metrics: Roadmap”, In Proceedings of the Conference on the Future of Software Engineering, (Limerick, Ireland, June 04-11, 2000). ICSE ’00. ACM, New York, NY, 357-370.

[7] M. H. Halstead, “Elements of Software Science”, Elsevier New Holland, New York, 1977.

[8] Brown et al. 1998, “AntiPatterns, Refactoring Software, Architectures and Projects in Crisis”, Wiley Computer Publishing.

[9] S. Woods, A. Quilici, and Q. Yang, “Constraint-Based Design Recovery for Software Re-engineering: Theory and Experiments”, Kluwer Academic Publishers, 1998.

[10] Budgen, D. (2003), “Software Design”, Second Edition, Addison-Wesley.

[11] A. A. Zakaria and H. Hosny, “Metrics for Aspect-oriented Software Design”, In AOM: Aspect-Oriented Modeling with UML, AOSD, March 2003.

[12] Pressman, “R. S. 1996 Software Engineering: a Practitioner’s Approach”, 4th. McGraw-Hill Higher Education.

[13] Guéhéneuc, Y. and Albin-Amiot, H. 2001, “Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects”, In Proceedings of the 39th international Conference and Exhibition on Technology of Object-Oriented Languages and Systems (Tools39) (July 29 - August 03, 2001). TOOLS. IEEE Computer Society, Washington, DC, 296.

[14] Larsen, L. and Harrold, M. J. 1996, “Slicing Object-oriented Software”, In Proceedings of the 18th International Conference on Software Engineering (Berlin, Germany,

March 25 - 29, 1996), International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 495-505.

[15] Radu Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws", Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004), IEEE Computer Society Press, pages 350 - 359, 2004.

[16] Munro, M. J. 2005, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code", In Proceedings of the 11th IEEE International Software Metrics Symposium (September 19-22, 2005). METRICS. IEEE Computer Society, Washington, DC, 15.

[17] Naouel Moha, Duc-Loc Huynh, Yann-Gaël Guéhéneuc, "A Taxonomy and a First Study of Design Pattern Defects", Proceedings of the STEP International Workshop on Design Pattern Theory and Practice, September 25-30, 2005, Budapest, Hungary.

[18] Radu Marinescu, "Detecting Design Flaws via Metrics in Object-Oriented Systems", Proceedings of 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39), IEEE Computer Society Press, 2001.

[19] Trifu, A., Seng, O., and Genssler, T. 2004, "Automated Design Flaw Correction in Object-Oriented Systems", In Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (Csmr'04) (March 24 - 26, 2004), CSMR. IEEE Computer Society, Washington, DC, 174.

[20] T. Mens, "T. Tourwe: A Survey of Software Refactoring", IEEE Transactions on Software Engineering, 30, No. 2, February 2004.

[21] Beck, K., Crocker, R., Meszaros, G., Vlissides, J., Coplien, J. O., Dominick, L., and Paulisch, F. 1996, "Industrial Experience with Design Patterns", In Proceedings of the 18th International Conference on Software Engineering (Berlin, Germany, March 25 - 29, 1996), International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 103-114.

[22] Jaroenpiboonkit, J. and Suwannasart, T. 2007, "Finding a Test Order using Object-Oriented Slicing Technique", In Proceedings of the 14th Asia-Pacific Software Engineering Conference (December 04 - 07, 2007), APSEC. IEEE Computer Society, Washington, DC, 49-56.

[23] Briand, L. C., Daly, J. W., and Wüst, J. K. 1999, "A Unified Framework for Coupling Measurement in Object-Oriented Systems", IEEE Trans. Softw. Eng. 25, 1, Jan. 1999, 91-121.