On the Automatic Detection and Correction of Software Architectural Defects in Object-Oriented Designs

Naouel Moha and Yann-Gaël Guéhéneuc

GEODES - Group of Open and Distributed Systems, Experimental Software Engineering

Department of Informatics and Operations Research
University of Montreal, Quebec, Canada
E-mail: {mohanaou, guehene}@iro.umontreal.ca

Abstract

Design defects, antipatterns, code smells are software defects at the architectural level that must be detected and corrected to improve software quality. Automatic detection and correction of these software architectural defects, which suffer of a lack of tools, are important to ease the maintenance of object-oriented architectures and thus to reduce the cost of maintenance. A clear understanding of the different types of software architectural defects defects and a classification of these defects is necessary before proposing any techniques related to their detection or correction. We introduce a first classification and summarise existing techniques. Then, we introduce some challenges that our community must meet.

Keywords: Software Defects, Design Patterns, Design Defects, Antipatterns, Detection, Correction, Object-Oriented Architecture.

1. Introduction

We present a first study on automatic detection and correction of software architectural defects in object-oriented architectures in terms of techniques and tools. We englobe in Software Architectural Defects (SAD) design defects and antipatterns. By architecture, we mean the design level as defined in [8]. Automatic detection and correction of software architectural defects are important to ease the maintenance of object-oriented architectures and thus to reduce the cost of maintenance.

Indeed, maintenance of existing software represents over 60 percent of all effort expended by a development team [17]. Only about 20 percent of all maintenance work is spent in fixing faults whereas the remaining 80 percent is spent in adapting existing systems to changes in their external environment, making enhancements requested by users, and reengineering an application for future use [20].

However, software maintenance is a tedious task and is sometimes quite impossible because of the huge amount of code lines. Reengineering object-oriented architectures to understand, to correct, and to maintain programs is one of the solutions to facilitate and to improve software maintenance. Thus, we propose to detect and to correct SAD defects to improve the quality of the code in terms of maintainability[1] and usability[2].

The contribution of this paper is to present issues related to the automated detection and correction of defects at design level and to discuss new perspectives of research.

In the first section, we define what we englobe in SAD defects and propose a first classification of these defects. The two following sections detail the detection and the correction of SAD defects in terms of techniques and tools. We conclude with some challenges and perspectives related to this research.

2. Terminology

A clear understanding of the different types of SAD defects and a classification of those defects is neces-

[1]ISO/IEC 9126-1/2001: The capability of the software product to be modified. Modifications may include corrections, improvements, or adaptation of the software to changes in environment, in requirements, and functional specifications.

[2]ISO/IEC 9126-1/2001: The capability of the software product to be understood, learned, used, and attractive to the user, when used under specified conditions.

sary before proposing any techniques related to their detection or correction.

2.1. Taxonomy

There still is some confusion between the terms design defects, antipatterns, and code smells. To our best knowledge, the taxonomy defined in this paper is the first attempt to clarify and to classify these concepts.

By defects, we refer to the fault or errors in the "fault/error/failure chain". A fault is an adjudged or hypothesized cause of an error. An error is a system state that is liable to lead to a failure if not corrected. As for a failure, it is the behavior of a system differing from that which was intended. Thus, a fault is a cause which can activate an error as an internal manifestation of this cause, and the failure is the potential propagation of this error as an external manifestation in the system. We are situated at the design level and we consider only programs without compilation errors.

We englobe in SAD defects design defects and antipatterns mainly.

Antipatterns. An antipattern is a literary form that describes a commonly occurring solution to a design problem, solution which generates decidedly negative consequences [5]. Antipatterns are bad solutions to recurring design problems. The idea of antipatterns is to show what not to do: Isn't it through errors that people learn? The Blob, the Spaghetti, the Poltergeist, the Lava Flow are among well-known antipatterns. For example, the Blob corresponds to one single complex controller class that monopolizes the processing and is surrounded by simple data classes [5]. The Spaghetti code, which is one of the most famous antipattern, describes a program or system with a software structure that lacks clarity [5]. A Spaghetti code is hard to maintain or extend because of its complexity.

Design Defects. Design defects are similar to design patterns which are today used and studied in the industry and academia. Design patterns propose "good" solutions to recurring design problems in object-oriented architectures, whereas design defects are occurring errors in the design of the software that come from the absence or the bad use of design patterns. Thus, Guéhéneuc *et al.* define design defects as distorted forms of design patterns, i.e., micro-architectures similar but not equal to those proposed by design patterns [12].

Code smells. A code smell is a term that refers to a symptom or a problem at the code level. Beck and Fowler describe code smells as "certain structures in code that suggest the possibility of refactoring" [10]. Duplicated code, long method, large class, long parameter list, data class are some examples of code smells. The presence of code smells suggests the possible presence of an antipattern as well as of a design defect.

Code smells are generally related to the inner workings of classes while design defects include the relationships among classes and are more situated on a micro-architectural level. Thus, we qualify code smells as intra-class defects while design defects as inter-class defects. Antipatterns are in both classes of defects.

2.2. Classifications

Software defects are usually classified by priority and this classification is composed of the following criteria: Minor, major, severe, and critical. Such a simple classification is used for assigning priorities in repairing defects. Other more complex classifications have been defined [3] but all these classifications are related to software defects at the code level. There exist only few classifications of defects at the design level.

Brown classifies antipatterns in three main categories: Development, architecture, and project management antipatterns. In our case, we study only development and architectural antipatterns because they represent respectively coding practices and system-level structure, and thus could be detected and corrected in programs automatically. Moreover, each antipattern defined in [5] proposes formal or informal refactored corrective guidelines.

Among SAD defects, we distinguish three important categories [12]:

- *Intra-classes*: SAD defects related to the internal structure of a class.
- *Inter-classes*: SAD defects related to the external structure of the classes (public interface) and their relations (inheritance, association, etc.)
- *Behavioral*: the SAD defects related to the semantics of the program.

Also, SAD defects can be classified according to their nature:

1. *Antipatterns*, as described in [5].
2. *Distorted design patterns* or *design defects*: Distorted forms of design patterns.

3. *Distorted form of antipatterns*: A distorted form of an antipattern could be a good form.

Figure 1 describes the different categories of software defects that we are considering for automated detection and correction in object-oriented architectures. Antipatterns and design defects are the main sub-categories of SAD defects. Code smells are small defects that surround antipatterns and design defects and that suggest the possible presence of one or the other.

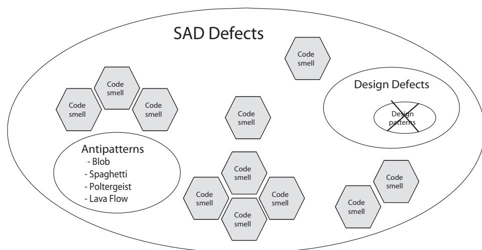

Figure 1. Categories of Software Architectural Defects

An exhaustive study on the definitions and classifications of the software defects found in the literature will enable us to refine and to verify the completeness and correctness of this classification. Such classification is important to target concretely which defects to address.

2.3. Formalization

In parallel to this classification, there is a need to formalize SAD defects. So far, there is only textual descriptions of software defects but no formal description of SAD defects. For example, the descriptions of antipatterns are purely textual and provide no specific detection methods: Detection of antipatterns is left to the intuition of programmers. The specification of design patterns described in [11] is more formal: It gives not only the solutions of the different design patterns but also implementation guidelines of these solutions called design motifs. Design defects, which are degraded forms of design patterns, are so more "formal" than antipatterns. Thus, it is easier to detect approximative forms of design patterns. Code smells are also "formal" indications that enable to detect that there is trouble in the code and that can be solved by a refactoring [10].

Metamodel. An abstract model, named PADL, has been defined to formalize design motifs [2]. This model shall be extended to take into account design defects. With this metamodel, we shall be able to identify approximative motifs of design defects.

Another interesting metamodel to describe design defects is FAMIX [7]. This metamodel provides a language-independent representation of object-oriented source code and supports refactorings.

Mens *et al.* proposes to use a declarative meta-language for expressing and reasoning about programming patterns in object-oriented programs [18]. Besides checking and searching for patterns, this language allows to write queries that check the source code for *violations* or defects of patterns. This language could be efficient for the detection of the design defects.

We are investigating on the choice of the right metamodel that will enable us to detect the SAD defects in object-oriented architectures.

3. Detection of Software Defects

This section presents some techniques and tools related to the detection of SAD defects in object-oriented architectures.

3.1. Techniques

Several techniques have been studied and validated for detecting defects in object-oriented designs, but most of them are "reading techniques" such as software inspections [22] and–or at the code level such as functional or structural testing [23]. We survey different techniques and algorithms to detect automatically the software defects at the design level:

- Information extraction of the code comments [21];
- Behavioral analysis based on the sequence diagrams: The analysis of the dynamic communication among objects of the system by reengineering the sequence diagrams from the code source;
- Dynamic detection during the execution of the program by expressing assertions, pre- and post-conditions [16].

We do not focus on model checking or refinement techniques because it is quite tedious to apply these techniques in millions lines of code. Model checking is an automatic technique for verifying finite state concurrent systems, which deals with the state space explosion problem [9]. Refinement is a systematic top-down methodology that consists in decomposing a software

component into subcomponents and specializing components under certain compatibility conditions [6].

Metrics. We believe that the detection of SAD defects could use metrics. For example, in the case of the Blob antipattern, it would be interesting to identify classes of sizes superior to L LOC, classes with more than M methods and N attributes. It is necessary to define for each antipattern the methods and metrics that enable their identification and also the thresholds related to these metrics.

3.2. Tools

The detection of SAD defects must be fully automated via tools. Several tools for the assessment of the code quality have been developed to identify and visualize the structure of object-oriented programs:

- OptimalAdvisor is a static code analysis and refactoring tool, which shows packages and classes dependencies with UML class diagrams and which provides detailed analysis related to the classes, methods, and fields [15].
- IBM Structural Analysis for Java (SA4J) claims to detect antipatterns but in fact detects 7 predefined bad design examples based on structural dependencies among objects [13].
- SmallLint, a tool for Smalltalk programs, automatically checks for over 60 common types of bugs in the code level such as identification of long methods or inutile code[4].

These tools have some benefits : They help in understanding and in visualizing the structure of the code, provide some basic measures (size, inheritance), and highlight some problems. However, these tools are limited because they do not support the detection of software defects at the design level. For example, OptimalAdvisor detects only dependency structures and suggests to fix them.

4. Correction of Software Defects

This section presents techniques and tools related to the correction of SAD defects in object-oriented architectures.

4.1. Techniques

We investigate techniques that help in correcting SAD defects. Refactoring is a key technique [19]. Refactoring is defined as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [10]. We distinguish between behavior preserving and non-behavior preserving refactoring techniques.

We must precise, develop, or extend refactoring techniques to correct any kind of defects easily. These techniques must be generic to be uniformly applicable on any SAD defects. Thus, we shall be able to add any other SAD defect and detect it without modifying the algorithms of these techniques.

Unlike detection, the correction shall be semi-automatic. We shall detect SAD defects but it is up to the maintainers to refactor or not the code. In the long term, we plan to develop fully automatic corrections. The detection and the correction of SAD defect can generate new SAD defects, which must then be detected and corrected.

4.2. Tools

The tools mentioned previously perform some basic refactorings such as renaming classes, extracting methods, introducing constants, etc.

- OptimalAdvisor: This tool enables developers to refactor automatically their code based on the code analysis. OptimalAdvisor supports class/package rename/move, remove unused import, and the dependency inversion refactoring.
- IBM Structural Analysis for Java (SA4J) does not refactor the code but gives a detailed map of dependencies for assistance in refactoring.
- The Refactory Browser supports also like OptimalAdvisor class insert/move/remove, variable insert/remove/rename, method rename/move/remove refactoring [14].

5. Challenges

The aim of our research is to formalize SAD defects including antipatterns and design defects for their detection and correction in object-oriented architectures and to correct them. The techniques of detection and correction shall be generic and thus shall work for any SAD defect.

Our aim is to develop a system similar to Ptidej [1] to detect automatically SAD defects based on their formalization and to propose corrections with explanations to maintainers. There shall be no fully automatic correction. We first plan to detect the SAD defects by hand and then try to verify them through our system

of detection. A usability evaluation will be required to validate the comprehensiveness and the ease of use of the detection and correction system.

6. Conclusion

This paper clarifies the confusion in the terminology related to design defects and antipatterns. It also introduces a new term "SAD defect" for software architectural defects. Some techniques and tools necessary for the detection and correction of code defects have been surveyed.

Future work includes:

- Classifying SAD defects.
- Formalizing SAD defects.
- Defining techniques, tools, and metrics for the automatic detection and the semi-automatic correction of SAD defects.
- Implementing and validating these techniques and tools.

We would be glad to discuss the challenges of SAD defects detection and correction with the participants of the workshop.

References

[1] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In Debra Richardson, Martin Feather, and Michael Goedicke, editors, *proceedings of the* 16th *conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, November 2001.

Available at: `www.yann-gael.gueheneuc.net/Work/Publications/`.

[2] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In Bedir Tekinerdogan, Pim Van Den Broek, Motoshi Saeki, Pavel Hruby, and Gerson Sunyé, editors, *proceedings of the* 1st *ECOOP workshop on Automating Object-Oriented Software Development Methods*. Centre for Telematics and Information Technology, University of Twente, October 2001. TR-CTIT-01-35.

Available at: `www.yann-gael.gueheneuc.net/Work/Publications/`.

[3] Boris Beizer. *Software Testing Techniques*. van Nostrand Reinhold, 2nd edition, 1990. ISBN: 0442206720.

[4] John Brant. Smalllint, April 1997. The Smalllint tool checks for over 60 common types of bugs.

Available at: `http://st-www.cs.uiuc.edu/users/brant/Refactory/Lint.html`.

[5] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1st edition, March 1998. ISBN: 0-471-19713-0.

Available at: `www.amazon.com/exec/obidos/tg/detail/-/0471197130/ref=ase_theantipatterngr/103-4749445-6141457`.

[6] Carlos Canal, Ernesto Pimentel, and José M. Troya. Specification and refinement of dynamic software architectures. In *proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 107–126, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V. ISBN: 0-7923-8453-9.

[7] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why FAMIX and not UML? Technical report, Software Composition Group, University of Bern, 1999.

Available at: `iamwww.unibe.ch/~famoos/FAMIX/whyFAMIX/whyFAMIX.html`.

[8] Amnon H. Eden and Rick Kazman. Architecture, design, implementation. In Laurie Dillon and Walter Tichy, editors, *proceedings of the* 25th *International Conference on Software Engineering*, pages 149–159. ACM Press, May 2003.

Available at: `www.eden-study.org/publications.html`.

[9] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[10] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1st edition, June 1999. ISBN: 0-201-48567-2.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, 1994. ISBN: 0-201-63361-2.

[12] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Quioyun Li, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, *proceedings of the* 39th *conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, July 2001.

Available at: `www.yann-gael.gueheneuc.net/Work/Publications/`.

[13] IBM. Structural analysis for java, March 2004. Structural Analysis for JavaTM (SA4J) is a technology that analyzes structural dependencies of Java applications in order to measure their stability. It detects structural "anti-patterns" (suspicious design elements) and provides dependency web browsing for detailed exploration of anti-patterns in the dependency web. SA4J also enables "what if" analysis in order to assess the impact of change on the functionality of the application; and it offers guidelines for package re-factoring.

Available at: `http://www.alphaworks.ibm.com/tech/sa4j`.

[14] The Refactory Inc. Refactoring browser, October 1999. The Refactoring Browser is an advanced browser for VisualWorks, VisualWorks/ENVY, and IBM Smalltalk. It includes all the features of the standard browsers plus several enhancements.

Available at: `http://st-www.cs.uiuc.edu/users/brant/Refactory/, http://www.refactory.com/RefactoringBrowser/`.

[15] Compuware JavaCentral. Optimaladvisor, May 2005. OptimalAdvisor is a package and code analysis tool that delivers insight and advice into the code structure and effectiveness of Java applications.

Available at: `http://javacentral.compuware.com/products/optimaladvisor/`.

[16] Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering tobias combinatorial test suites. In Tiziana Margaria-Steffen Michel Wermelinger, editor, *proceedings of ETAPS/FASE04 - Fundamental Approaches to Software Engineering*, volume 2984. LNCS, Springer-Verlag, April 2004.

[17] M. Manna. Maintenance burden begging for a remedy. *Datamation*, pages 53–63, April 1993.

[18] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Elsevier Journal on Expert Systems with Applications*, 23(4). Lecture Notes in Computer Science (LNCS), November 2002.

[19] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.

[20] Roger S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, November 2001. ISBN: 0-07-249668-1.

Available at: `www.accu.org/bookreviews/public/reviews/s/s000005.htm`.

[21] Ellen Riloff and Wendy Lehnert. Information extraction as a basis for high-precision text classification. *ACM Transactions on Information Systems (TOIS)*, 12(3). ACM Press, July 1994.

[22] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *Conference on Object Oriented Programming Systems Languages and Applications, proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–56. ACM Press, New York, NY, USA, 1999.

[23] Murray Wood, Marc Roper, Andrew Brooks, and James Miller. Comparing and combining software defect detection techniques: a replicated empirical study. In *proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 262–277. Springer-Verlag New York, Inc., 1997.