

A Taxonomy and a First Study of Design Pattern Defects

Naouel Moha, Duc-loc Huynh, and Yann-Gaël Guéhéneuc

PTIDEJ Team

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

Department of Informatics and Operations Research

University of Montreal, Quebec, Canada

E-mail: {mohanaou, huynhduc, guehene}@iro.umontreal.ca

Abstract

Design patterns propose “good” solutions to recurring design problems in object-oriented architectures. Design patterns have been quickly adopted by the Software Engineering community and are now widely spread. We define design pattern defects as occurring errors in the design of a software that come from the absence or the bad use of design patterns. Design pattern defects 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 improve object-oriented architectures and, thus, to ease maintenance. We propose a first taxonomy of design pattern defects and presents techniques and tools to detect these defects in source code.

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

1. Introduction

Validation and verification as well as maintenance are key activities in the software lifecycle. During these activities, it is important to check the correctness of the design and implementation of a software product against some predefined criteria to detect and to correct software architectural defects early in the development process and, thus, to reduce costs.

We define design pattern defects as a sub-category of software architectural defects (SAD) [8]. Design pattern defects are distorted forms of design patterns, i.e.,

micro-architectures similar but not equal to those proposed by the solutions of design patterns, also called design motifs [4].

Long-term Research Objective. By detecting design pattern defects, it is possible to pinpoint which design patterns have not been well implemented by the developers and, thus, to understand why design patterns are not well applied. We are convinced that the identification of micro-architectures similar, but not equivalent, to design patterns not only highlight poor design solutions needing improvements, but also help in improving the application of design patterns. Through our research, we want to show which design patterns are usually well applied and which are less well applied, but also how they are applied and for which applications (domains, size, etc.). We believe that the detection and study of design pattern defects help to answer the questions: When, how, why and which design patterns are not well applied?

Short-term Research Objective. This paper proposes a first classification of the design pattern defects and presents tools and methodologies to detect them in source code. In section 2, we define what we mean by design pattern defects and propose a first taxonomy of these defects. In section 3, we present the current techniques and tools for detecting these defects. In section 4, we detail the case study we conducted to identify design pattern defects.

2. Terminology

A clear understanding of the different types of design pattern defects and a classification of those defects is

necessary before proposing any techniques related to their detection.

2.1. Taxonomy

This section clarifies what we mean by design pattern defects and proposes a first classification of this category of defects.

Defects. We use the term “defect” to define a flaw or an imperfection, not an erroneous behaviour. It is any unintentional, intentional, or undesirable irregularity in the design or the architecture that could affect performance and maintenance. A defect can lead to a fault that may cause a failure.

Design Pattern Defects. Design pattern 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 pattern 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, referring to design pattern defects, as distorted forms of design patterns, i.e., micro-architectures similar but not equal to those proposed by solutions of design patterns [4].

We propose to distinguish between four kinds of design pattern defects:

- *An approximative or deformed design pattern* is a design pattern that has not been well implemented according to the Gang of Four but that is not erroneous. Sometimes it can be an improvement of the implementation of the design pattern.
- *A distorted or degraded design pattern* is an incorrect occurrence or a distorted form of a design motif which is harmful for the quality of the code.
- *A missing design pattern* is one of the sub-categories of design flaws defined by Marinescu [7]. According to the Gang of Four, missing patterns generate poor design.
- *A excess design pattern* is related to the excessive use of design patterns [12].

Thus, we define design pattern defects as design pattern motifs that are deformed, distorted, missing, or in excess, and that conflict with their motivations as in the Gang of Four. They are different from antipatterns because they were intended to improve design contrary to antipatterns that are bad solutions (we adopt the

philosophical stance that too much of a good thing can be bad).

2.2. Classifications

It is important to have a consistent classification to avoid redundancies in definitions and to ease comprehension. We classify design pattern defects based on the two following classifications.

Gamma Classification. We propose the following classification, which describes the nature of design pattern defects, to maintain consistency between the classification of design patterns as defined by Gamma *et al.* [3] and design pattern defects:

- *Creational Design Patterns Defects* are defects related to creational patterns including Abstract Factory, Builder, Singleton, etc.
- *Structural Design Patterns Defects* are defects related to structural patterns including Composite, Decorator, Facade, etc.
- *Behavioral Design Patterns Defects* are defects related to behavioral patterns including Command, Iterator, Visitor, etc.

Classification of the type of defects. This classification aims to define not the nature but the kind of defects. Based on the different kinds of design pattern defects, we define four kinds of defects: missing, in excess, deformed, and distorted.

2.3. Origins of Design Pattern Defects

Software aging, as defined by Parnas [10], is one of the first explanation of the origins of design pattern defects. Indeed, the apparition of design pattern defects can be explained by the evolution of a program has been subject to a lot of changes: restructuring, additions, or removals of functionalities, of methods, of classes, etc. These changes may degrade the design of a program and, thus, the solutions of design patterns initially implemented.

The second explanation is the lack of experience of developers. Novice developers may not have an advanced knowledge of design patterns. They make an effort to structure well their programs by implementing design patterns but fail to implement some concepts related to design patterns or to restrain the use of design patterns. Others are simply not aware of these good practices implementations and use alternatives to solve well-known problems (for example, see the many possible solutions to the design pattern “Iterator”).

3. Detection Techniques and Tools

The problem of automating the detection and correction of design pattern defects have not yet been largely studied.

Actually, the detection of design pattern defects is left to the intuition of the developers or architects based on their experience. But it is a tedious task, especially for large systems. The automatic detection of the design patterns requires to define specific and structured automatic techniques.

However, no technique or measure is as precise as the faculty of the developer to evaluate the code quality and most of the techniques generate some false positive detections. Thus, we suggest that the detection of design pattern defects must be semi-automatic.

The following techniques are currently used for the detection of design pattern defects. These techniques are split into three distinct categories: manual, semi-automatic, and automatic techniques.

- *Manual.* Manual techniques include the naïve approach that consists of reviewing comments and the name of classes, methods to get some keywords related to design patterns. The basic tool that can be used for applying the naïve approach is the textual search provided by most of the code editors to assert the presence of a design pattern defects based on the results of the textual search. This technique requires to have a strong knowledge on design patterns.
- *Semi-automatic.* Reengineering tools such as Describe [2], EclipseUML [9], Code Logic [6] prove to be helpful in the detection of design pattern defects. These tools do not detect automatically design patterns but reverse-engineer class diagrams, which facilitates the visual detection of patterns. When the code has been reverse-engineered, a methodology consists of identifying the abstract classes and interfaces and their inheritance relationships. Another methodology is based on the “Pattern Map” of the Gang of Four which gives the design pattern relationships. The existence of a pattern can suggest the existence of another pattern.
- *Automatic.* Automatic techniques include using Constraint Satisfaction Problem (CSP) as in the Ptidej tool suite [1], recently extended with numerical signatures [5] and techniques based on metrics. They define the problem of detecting a design pattern in terms of its variables, the constraints among the variables, and their domains.

The set of constraints corresponds to the relationships among the entities defined by the design pattern. Marinescu detected design flaws by applying measurement-based rules on a system via his fully automatized ProDeOOS tool [7].

4. Case Study

We perform a first case study to prove the presence of design pattern defects. We ask bachelor students to look for design patterns, but not explicitly design defects, in an application. The object of the study is DrJava [11], a lightweight development environment for writing Java programs designed primarily for students. The subjects are two groups of 3 bachelor students with a good knowledge on design patterns.

Hypotheses. Our hypothesis is that DrJava, as a small program (version 2004: 413 classes for 28,522 LOC), is believed to contain design patterns because it is relatively well written.

We define several attributes to well identify the design pattern defects found:

- Defect Type can have several values: Missing, Deformed, Distorted, in Excess.
- Defect Fix: This attribute gives the refactoring solution to solve this defect.
- Defect Impact: This attribute gives the measure of the impact of the defect in terms of the effect on the system (performance, security, maintainability). The advantage is to present to the maintainers the impact of not correcting this defect. The impact can also related to the developers or the users in terms of usability.
- Defect Visibility : This attribute specifies if the defect can be revealed at a static or dynamic state i.e. if the defect is structural or behavioral.

Research Questions. The questions that we try to answer are: When, how why and which design patterns are not well applied?

Procedure. Three different versions of DrJava were analyzed to point out the changes between the different versions. The reverse engineering of DrJava was performed by the following tools: Describe, Code Logic, and Eclipse. The analysis was based on the semi-automatic methodology as defined previously and lasted around 2 weeks (i.e., 30 hours) for each group. The second step of this case study consisted of evaluating the correctness of the design patterns found.

Results. 38 design patterns were found by the two teams. As generally accepted, the use of design patterns increases the quality of an architecture. So, considering the relative important number of patterns, we could deduce that DrJava is well structured, and thus, confirm our hypothesis. Moreover, the comparison of three different versions of DrJava shows that the developers use the benefits provided by the design patterns (specially the State and Strategy design patterns) to extend the program. Table 1 gives the results of the design patterns found in DrJava. We notice the high use of the Singleton and the Template Method.

Among the 38 design patterns found, three were design pattern defects. An Iterator, a Memento, and a Command design patterns have not been well applied by the developers. All three of them were categorized as approximative or deformed design patterns which mean that they are not exactly design patterns as defined by Gamma *et. al* but are not erroneous and do not affect the quality of the code (see Table 2). These three design pattern defects have appeared in the first version and were not corrected in the two next versions. We can deduce that they were not harmful for the architectural quality and we categorize them as deformed defects.

Threats to validity. This case study was a first exploratory experiment, and thus, has several limitations. It is difficult to generalize our results because we did not get a significant number of design pattern defects. We planned to re-conduct the same case study but asking the students to explicitly list all design patterns found, including those that are not well implemented. This case study does not enable to give a response to the questions: When and why design pattern defects are not well applied? However, we detected which design patterns were not well applied and how. To be able to response to the questions when and why, a case study on several different applications is required.

5. Conclusion

Design pattern defects are defects that come from the absence or the bad use of design patterns. We define design pattern defects as design patterns that are deformed, distorted, missing, or in excess, and conflict with the motivations of the Gang of Four.

We presented techniques and tools to detect design pattern defects and a case study that illustrates the use of these techniques and tools.

The automatic detection of design pattern defects is an help provided to the developer, the maintainer, and the architect to facilitate their work. Our objective is

to provide indications on the presence, the localization, and the nature of design pattern defects in programs. Moreover, thanks to the localization of defects in the implementation of programs, it is possible to understand and to explain the reasons for which the design patterns are badly implemented.

We hope that this first work will stir the interest of the participants of the workshop and we would appreciate any comments about this ongoing research.

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] Embarcadero. Describe, August 2005. Embarcadero Describe is a UML design solution that provides your software development team with immediate visibility into your source code. The product adds a set of powerful visual tools for manipulating code, all directly within your existing development environment.
Available at: <http://www.embarcadero.com/products/describe/>.
- [3] 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.
- [4] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Quiyun 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/.
- [5] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Jean-Yves Guyomarc’h. Identification of approximate occurrences of design patterns: An experimental study. In John Knight, editor, *Transactions on Software Engineering*. IEEE Computer

Creational Patterns					Structural Patterns						Behavioral Patterns											
Abstract Factory	Builder	Factory Method	Prototype	Singleton	Adapter	Bridge	Composite	Decorator	Facade	Flyweight	Proxy	Chain of Responsibility	Command	Interpreter	Iterator	Mediator	Memento	Observer	State	Strategy	Template Method	Visitor
1	0	1	0	12	2	1	0	0	0	0	1	0	2	0	1	1	1	0	3	3	9	0

Table 1. Results of the Detection of Design Patterns in DrJava application

Design Pattern Defects	Defect Type	Defect Fix	Defect Impact	Defect Visibility	Comments
Semi-automatic Analysis of DrJava					
Iterator	Deformed	No Fix	No Impact	Static	Iterate the real list and not a clone of the list.
Memento	Deformed	No Fix	No Impact	Static	Fusion of the caretaker and the originator in the same class.
Command	Deformed	No Fix	No Impact	Static	Fusion of the client and the invoker in the same class.

Table 2. Results of the Detection of Design Pattern Defects in DrJava application

Society Press, 2006. **To be submitted for publication in June 2005.**

- [6] Inc. Logic Explorers. Code logic, August 2005. CodeLogic is a revolutionary system for discovering and graphically representing the deep, internal logic of any Java code. Developers can simply point CodeLogic at any existing Java or C# project and immediately get an intuitive view of exactly how the code works.

Available at: <http://www.logicexplorers.com/products/codelogic/>.

- [7] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. Ph.D. thesis, Politehnica University of Timisoara, October 2002.

Available at: www.cs.utt.ro/~radum/papers.html.

- [8] Naouel Moha and Yann-Gaël Guéhéneuc. On the automatic detection and correction of software architectural defects in object-oriented designs. In *Proceedings of the 4th ECOOP Workshop on Object-Oriented Reengineering*, July 2005.

- [9] Omondo. Describe, August 2005. EclipseUML Free Edition is a visual modeling tool, natively integrated with Eclipse 3.1 and JDK 5. EclipseUML Studio Edition offers full support for UML

diagrams, team work, data J2ee modeling and dynamic collaboration to any other plugins.

Available at: <http://www.omondo.com/>.

- [10] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN: 0-8186-5855-X.

- [11] the JavaPLT group at Rice University. Drjava, August 2005. DrJava is a lightweight development environment for writing Java programs. It is designed primarily for students, providing an intuitive interface and the ability to interactively evaluate Java code.

Available at: <http://drjava.sourceforge.net/>.

- [12] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Pedro Sousa and Jürgen Ebert, editors, *proceedings of 5th Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.

Available at: www.computer.org/proceedings/csmr/1028/10280077abs.htm.