A systematic literature mapping on the relationship between design patterns and bad smells: Papers Summary

Bruno L. Sousa
Computer Science Department
Federal University of Minas Gerais (UFMG)
Belo Horizonte – Minas Gerais, Brazil
bruno.luan.sousa@dcc.ufmg.br

Mariza A. S. Bigonha
Computer Science Department
Federal University of Minas Gerais (UFMG)
Belo Horizonte – Minas Gerais, Brazil
mariza@dcc.ufmg.br

Kecia A. M. Ferreira
Department of Computing
Federal Center of Technological Education of Minas Gerais (CEFET-MG)
Belo Horizonte – Minas Gerais, Brazil
kecia@decom.cefetmg.br

1. INTRODUCTION

This document presents a summary of the studies selected for conducting a Systematic Literature Review on the relationship between design patterns and bad smells. In this summary the main idea of each selected studies is discussed. In addition, during the preparation of these abstracts, the papers were analyzed and the main and most relevant information was extracted based on the research questions proposed in the Systematic Literature Review.

The Systematic Literature Review was conducted with 16 studies that address the relationship between design patterns and bad smells. Table 1 lists all 16 selected primary studies. The studies are ordered by year of publication. For each study, we extracted title, authors, publisher and the relation between design patterns and bad smells established with them. During the data summarization, three types of relationships were established: co-occurrences, refactoring and impact on software quality.

The remainder of this document is organized as follows. Section 2 describes the studies that explore the relationships of co-occurrence between design patterns and bad smells. Section 3 reports the studies that apply the design patterns as refactoring solutions for certain structures with the presence of bad smells. Section 4 is intended for studies that analyze the impact of design patterns on software quality.

2. CO-OCCURRENCE BETWEEN DESIGN PATTERNS AND BAD SMELLS RELATIONSHIP

SAC 2018,April 09-13, 2018, Pau, France

DOI: DOI:http://dx.doi.org/10.1145/2851613.2851735

Four of 16 studies establish a relationship of co-occurrence between design patterns and bad smells. This section presents a description of these studies, following the same order as they appear in Table 1.

2.1 Analysing Anti-Patterns Static Relationships with Design Patterns

Jaafar et al. [7] stress that software systems are constantly changing. In the midst of this, inappropriate developer knowledge may be a key factor in introducing anti-patterns. Large, long-lasting systems can present both anti-pattern and design patterns in the source code. In addition, there may be cases where entities of these two structures end up having some relationship. From these facts, Jaafar et al. [7] analyzed the existence, evolution and impact of static relationship between anti-patterns and design patterns in software systems. During the investigation, a case study was conducted, from snapshots of three open source Java systems: ArgoUML, JFreeChart and XercesJ. Information was extracted from design patterns, bad smells, and static relationship between both. Using a contingency table, Fisher's exact test and Odds ratio ([14]), the data were analyzed and the statistical significance of the relationship investigated were verified.

Jaafar et al. [7] concluded that (i) design patterns may have different proportions of static relationship with anti-patterns, (ii) the Command design pattern was identified as having the greatest relationship with the investigated anti-patterns, (iii) existing relationships between these two structures are not random, since they are constantly growing during the evolution of the project, and finally, (iv) classes that participate in static relations between anti-patterns and design patterns are more prone to change and less prone to failures in relation to classes that have the presence of anti-pattern, but do not participate in this relation.

2.2 Co-Occurrence of Design Patterns and Bad Smells in Software Systems: An Exploratory Study

Cardoso and Figueiredo [1] report the importance of design patterns in the literature, and emphasize that these

Title	Author	Publisher	Relationship
1. Assessment of Design Patterns During Software Reengineering: Lessons Learned from a Large Commercial Project	Wendorff [19]	European Conference on Software Maintenance and Reengineering	Impact on software quality
2. Coupling of Design Patterns: Common Practices and Their Benefits	McNatt and Bieman [11]	International Computer Software and Applications Conference	Impact on software quality
3. Defect frequency and design patterns: An empirical study of industrial code	Vokac [16]	IEEE Transactions on Software Engineering	Impact on software quality
4. Do Design Patterns Impact Software Quality Positively?	Khomh and Gueheneuce [9]	European Conference on Software Maintenance and Reengineering	Impact on software quality
5. Automated refactoring to the Strategy design pattern	Christopoulou et al. [2]	Information and Software Technology	Refactoring
6. Analysing Anti-patterns Static Relationships with Design Patterns	Jaafar et al. [7]	Electronic Communications of the EASST	Co-occurrence
7. A multiple case study of design pattern decay, grime, and rot in evolving software systems	Izurieta and Bieman [6]	Software Quality Journal	Impact on software quality
8. Code Quality Cultivation	Speicher [15]	Communications in Computer and Information Science	Impact on software quality
9. Automated pattern-directed refactoring for complex conditional statements	Liu et al. [10]	Journal of Central South University	Refactoring
10. Automatic recommendation of software design patterns using anti-patterns in the design phase: A case study on abstract factory	Nahar and Sakib [12]	Central Europe CEUR Workshop Proceedings	Refactoring
11. A proposal of software maintainability model using code smell measurement	Wagey et al. [17]	International Conference on Data and Software Engineering	Impact on software quality
12. Co-Occurrence of Design Patterns and Bad Smells in Software Systems: An Exploratory Study	Cardoso and Figueiredo [1]	Brazilian Symposium on Information Systems	Co-occurrence
13. ACDPR: A Recommendation System for the Creational Design Patterns Using Anti-patterns	Nahar and Sakib [13]	IEEE International Conference on Software Analysis, Evolution and Reengineering	Refactoring
14. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults	Jaafar et al. [8]	Empirical Software Engineering	Co-occurrence
15. The relationship between design patterns and code smells: An exploratory study	Walter and Alkhaeir [18]	Information and Software Technology	Co-occurrence
16. Automated refactoring of superclass method invocations to the Template Method design pattern	Zafeiris et al. [20]	Information and Software Technology	Refactoring

Table 1: Final result of the implementation phase of the Systematic Review.

solutions are considered good programming practices and encourage the construction of flexible and reusable structures. However, the authors have identified previous studies that point out relations of these solutions with bad smells, originating mainly from the inappropriate application of design patterns. Based on these clues, the authors carried out an exploratory study in order to identify co-occurrences between design patterns and bad smells. This study focused on two types of analysis. The first of these is based on the identification of possible design patterns that can co-occur with bad smells. To conduct this first analysis, Cardoso and Figueiredo [1] extracted data on design patterns and bad smells from five software projects, and applied association rules to detect possible relationships between these structures. The second analysis is based on the discovery of facts that explain the emergence of these relations.

At the end of the study, Cardoso and Figueiredo [1] identified co-occurrences between (i) Command with God Class and (ii) Template Method with Duplicate Code. When analyzing the entities that presented these two structures, the authors concluded that the excessive use of a simple receiver class in the application of the Command design pattern to different interests caused the appearance of God Class bad smell. In the case of the Template Method design pattern, the multiple duplications of implementations were responsible for its co-occurrence with the Duplicate Code bad smell.

2.3 Evaluating the Impact of Design Pattern and Anti-Pattern Dependencies on Changes and Faults

Jaafar et al. [8] argue that previous studies have pointed out that design patterns can correlate with complex struc-

tures and result in failure occurrences. In addition, there are reports of problem propagation in classes with design patterns and bad smells that have static or commonality dependencies with other classes. Dependency of commutation is defined by the authors as change in one class that directly impacts on the change of another. In view of this finding, the authors decided to investigate the impact of such dependencies on object-oriented systems and to analyze their relationship with (i) propensity for failure, (2) types of exchanges, and (iii) types of failures that classes exhibit.

In order to conduct this investigation, they extracted data referring to design patterns, anti-patterns, static relationship of design patterns and anti-patterns, design patterns and anti-pattern commonality relations, and class failures. Such information was obtained from snapshots of three open source Java systems: ArgoUML, JFreeChart and XercesJ by means of the Fischer statistical test and Odds ratio [14]. Jaafar et al. [8] examined the significance of the relationships between the occurrence of a static or commutative dependency on classes with design patterns or anti-patterns and the risk of failure.

As the main results, the authors concluded that (i) classes that have a static or commonality relationship with anti-pattern classes have significantly more failures than other classes, (ii) classes that have commutation dependencies with classes of design patterns have significantly more (iii) structural changes are more likely to occur in classes with anti-pattern dependencies than in other classes, (iv) code additions are more prone to (v) specific types of failures are more prevalent in certain anti-patterns, such as Blob Class and Complex Class, which mainly propagate logical failures.

2.4 The Relationship Between Design Patterns and Code Smells: An Exploratory Study

Walter and Alkhaeir [18] carried out an exploratory study with the goal of (i) determining if and how the presence of design patterns are related to the presence of bad smells, (ii) investigating if and how the presence of these relations changes along the evolution of the code, and (iii) identify relationships between design patterns and bad smells. To achieve these objectives, the authors performed an analysis of the evolution of two systems, Apache Maven and JFreeChart. A total of 87 versions were analyzed, where 32 were related to the first system and 55 were related to the second. In addition, the study was conducted with 7 different types of bad smells and nine design patterns from the GOF catalog [5]. The information concerning design patterns and bad smells instances in these projects were extracted and, through the hypothesis test, non-parametric trend test (Mann-Kendall) and association rules, the authors performed the analysis of the information extracted, the order to reach a conclusion for i, ii and iii goal, respectively.

The main results obtained in this study showed that the presence of design patterns is related to the absence of bad smells in the same class. In other words, a class that makes use of the design pattern tends not to display bad smells. In addition, some design patterns have been pointed out as more likely to lack bad smells. It is the case of the design patterns: State-Strategy, Adapter-Command, Factory Method and Singleton. However, the Composite design pattern showed a stronger relation with the presence of bad smells, being an exception to this trend. According to the authors, the results obtained in relation to the Singleton design pattern were surprising, due to the fact that other research points the contrary.

Finally, on ii goal, the authors conclude that during the evolution of a project, the presence of bad smells in classes with design patterns is not greater than in the creation phase of the software. Therefore, this result implies in the fact that software systems are already designed with bad smells.

3. REFACTORING RELATIONSHIP

Five of the 16 studies establish a refactoring relationship between design patterns and bad smells. In these studies, design patterns are proposed as possible solutions to eliminate occurrences of bad smells or complex structures that impair the quality of software. This section presents a description of these studies following the order in which they appear in Table 1.

3.1 Automated Refactoring to the Strategy Design Pattern

Christopoulou et al. [2] proposed a code refactoring approach with the Complex Conditional Statements bad smell, from the application of the Strategy design pattern. The Complex Conditional Statements bad smell has been reported by Fowler and Beck [4] as complex conditional structures that propagate along a software source code, reducing its readability and making it more complex. These conditional structures may be replaced by a inheritance hierarchy, in which the polymorphism is made, leaving the code more modularized and flexible to modifications. The Strategy design pattern, proposed by Gamma et al. [5], uses this hierarchy in its structure and, therefore, giving to Christopoulou et al. [2] a choice to combat this bad smell.

In the proposed implementation, Christopoulou et al. [2] have created two types of algorithms. The first consists of identifying the anomalous structures in a Java source code. The second applies the Strategy design pattern to transform complex conditional structures into an inheritance hierarchy, replacing the use of conditionals with polymorphisms. This approach was implemented in the JDeodorant plugin of Eclipse IDE as support for automatic refactoring in Java projects.

After the implementation, the authors conducted an experiment involving 8 Java software projects, where six are open source and two are proprietary code. To evaluate this approach, they used the metrics: precision and recall. The results of the evaluation showed good efficacy in relation to quality, since almost half of the suggested refactorings were significant. In addition, efficiency over time was another relevant and quite satisfactory factor. The processing time of the algorithm did not exceed 30 seconds for medium-sized projects and 2.5 minutes for large projects.

3.2 Automated Pattern-Directed Refactoring for Complex Conditional Statements

Liu et al. [10] discuss the importance of avoiding the use of complex conditional statements in the source code of a project. If a program uses these claims on a large scale, there is a high probability that the Switch Statements bad smell [4] be present in the structure of that program. One way to eliminate it is via the application of refactoring techniques, in which a inheritance hierarchy is created and polymorphisms are introduced in place of conditional structures.

Although many developers are aware of the importance of refactoring for quality assurance, there is still some sub-use of this technique. Thus, Liu et al. [10] proposed a refactoring approach using the Factory Method and Strategy design patterns, which consists of (i) identifying refactoring opportunities in the code, i.e. points where there are conditional structures, and (ii) applying refactoring in those points using the Factory Method or Strategy design pattern. In this approach, four algorithms were constructed, two of which were used to identify refactoring opportunities and two for refactoring.

In the evaluation of this approach, the following metrics were used: precision, recall and accuracy. By the results, the refactoring approach, both by the Factory Method and Strategy design pattern, were efficient in relation to the reduction of cyclomatic complexity and number of methods code lines. However, some cases of false positives and false negatives were found, originating from some deficiencies in the approach. The authors intend to correct these cases in future works.

3.3 Automatic Recommendation of Software Design Patterns Using Anti-Patterns in the Design Phase: A Case Study on Abstract Factory

Nahar and Sakib [12] believe that choosing the design pattern could be made easier if done in conjunction with a recommendation tool. As a result, the authors created a tool, based on UML class diagrams, which incorporates both the detection of anti-patterns and the recommendation of design patterns in the design phase of the software. Anti-pattern based Design Pattern Recommender (ADPR) is composed of two phases. The first phase consists in analyzing the possible existence of anti-patterns in the software. The second phase detects the existing anti-patterns and recommends solutions composed of design patterns that eliminate the detected anti-patterns.

The authors carried out a case study to evaluate the recommendation of solutions with the Abstract Factory design pattern. The case study used an open source software developed in the Java language, called Painter. This software was chosen because the poor application of the Abstract Factory design pattern led to the occurrence of anti-patterns. In this evaluation, the effectiveness of the proposed approach was proven, which allowed the execution of an experiment with a larger number of software and the comparison of ADPR with other types of tools.

In a second evaluation, the authors compared the proposed tool with a code-based tool. Five Java open source projects were used, which are hosted on Github. In the comparison of the results, the authors concluded that the ADPR achieved a high precision, identifying all the points in which the application of the Abstract Factory design pattern was degraded and culminated in the presence of anti-patterns. The code-based tool did not perform well and had false-negative occurrences.

3.4 ACDPR: A Recommendation System for the Creational Design Patterns Using Anti-Patterns

Nahar and Sakib [13] have proposed a tool for recommending creative design patterns in the design phase of software. This tool uses anti-pattern characteristics to make possible recommendations. Anti-patterns in this study are considered by the authors as source code structures in which there was loss of effectiveness of the design patterns. For example, a structure with a class group instantiated directly without using a factory for its instantiation is considered a loss of the Abstract Factory project pattern. Each breeding design pattern has characteristics that drive its indication as a possible refactoring candidate. A punctuation system was included in the approach to classify the structures that can be refactored and indicate a type of design pattern more likely to be applied in this activity.

The tool was implemented in Java. The authors used a data set consisting of 21 open source projects also developed in the Java programming language for tool evaluation. For this evaluation, the metrics: precision, recall and f-measure were used to analyze the effectiveness of the recommendations. The results of this experiment were considered satisfactory, since the tool did not present cases of false positives and only a single case of false negative was returned.

3.5 Automated Refactoring of Super-Class Method Invocations to the Template Method Design Pattern

Zafeiris et al. [20] point out that inheritance is the mechanism used in object-oriented programming and supports the implementation of polymorphism in the development of a software project. These features enable the creation of modular and flexible software. However, the application of this mechanism must be specified and implemented with care so as not to lead to the occurrence of a bad smell called by Fowler and Beck [3] as Call Super. This bad smell consists of an extension of behavior of a concrete method, through the use of the keyword `super`. Gamma et al. [5] recommend using the Template Method design pattern for controlled extension of the behavior of a method.

Thus, the authors suggest a refactoring technique that aims to use the Template Method design pattern to eliminate structures with the Super Call symptom existing in the source code. Initially, the authors proposed an algorithm that performs static analysis in the source code of Java projects and identifies refactoring opportunities. This algorithm converts all classes of a project into an abstract syntax tree (AST) and parses the instances of methods that include `super` invocations. After this initial phase of identifications, another algorithm is applied in order to perform the transformations

in the source code. This second algorithm consists of seven steps that specify the introduction of the Template Method design pattern into the identified opportunities. The proposed approach was implemented as an extension to the JDeodorant plugin of Eclipse IDE.

For the evaluation, Zafeiris et al. [20] used a set composed of 12 open source projects, developed in Java. The proposed approach was tested in this benchmark, and through an experimental analysis, the authors verified that the identification algorithm suggested 20.5% of refactoring candidate instances. According to the authors, most of the rejected refactorings were related to trivial or semantically irrelevant cases, thus making them their satisfactory behavior. In addition, the application of refactorings in the identified opportunities impacted on the reduction of the Specialization Index (SIX) in the affected subclasses. Finally, the approach proved to be scalable, consuming a runtime of 7 to 25 seconds for small to medium-sized projects, not exceeding 1 minute for large projects.

4. IMPACT ON SOFTWARE QUALITY RELATIONSHIP

Seven of the 16 studies establish an impact on software quality relationship. These studies carry out empirical analyzes in which aspects and issues related to the application of design patterns are assessed, which can improve the quality of software or generate impacts that degrade the structure of the design pattern. This section presents a description of these studies, following the same order as they appear in Table 1.

4.1 Assessment of Design Patterns During Software Reengineering: Lessons Learned from a Large Commercial Project

Wendorff [19] evaluated the application of design patterns in commercial software in order to discover possible negative impacts generated by this technique. This analysis was conducted by a software engineer with eight years of professional experience. A proprietary software developed in the C++ language was used. In the study, Wendorff [19] realized that this technique can generate some negative impacts. According to him, the first results extracted from this analysis refers to the misuse of these solutions by the developers, due to the lack of understanding of the logic involved in its implementation. Other interesting results are that the application of design patterns does not fit the requirements of the project. In other words, the non-framing of these techniques can be represented by situations such as: overestimation and change of requirements as well as application of the solution without necessity.

At the end of the study, the author developed a simple procedure to mitigate the negative impacts caused and guide the developer in removing inappropriate source code patterns. This guide consists of seven steps: (i) identification of quality attributes relevant to the software, (ii) identification of the patterns used in the code, (iii) attempt to reconstruct the existing reasoning behind the patterns, (iv) evaluation of the concrete benefit, (v) evaluation of the concrete extra cost, (vi) evaluation of the effort required for removal and (vii) application of a decision to remove the pattern. This script was evaluated during the process of reengineering the analyzed software. According to the author, the decisions became more objective, well documented and more solid. However, the removal decision still remained subjective in some places.

4.2 Coupling of Design Patterns: Common Practices and Their Benefits

McNatt and Bieman [11] point out that in the software development process, design patterns can relate to one another by generating pattern couplings. This type of coupling is defined by the authors as two different design patterns that have at least one class in common.

In this study, the authors decided to evaluate the impact of these structures in the context of software quality through works that report occurrences of pattern matching. The set of papers selected for analysis included 16 documents, distributed among industrial applications and analytical studies. These documents were divided into four categories: pure analytical study, analytical study with synthetic examples, industry case studies considering existing code and case study of the new project industry. The coupling types were also classified into loosely, referring to patterns with few connections and simpler to be modified in the future, and tightly, referring to strongly connected patterns with many dependencies, where small changes in one pattern can impact others.

Analyzing the impact of these relationships, the authors concluded that: (i) the tightly coupling type results in a difficult design to be modularized, making it difficult to modify; (ii) the loosely coupling type is more flexible and a possible modification in this type of structure does not generate a large impact compared to the coupling tightly; (iii) the Singleton design pattern presented a detrimental tendency to the attribute of modularity quality, due to the fact that tightly coupling instances were found involving this pattern.

4.3 Defect Frequency and Design Patterns: An Empirical Study of Industrial Code

Vokac [16] argues that, while there is great acceptance of design patterns by researchers and practitioners, some studies show that such solutions can lead to defects in software. As a result, the author proposed to investigate them. Thus, a case study with proprietary software developed in the C++ language, called SuperOffice CRM5, was carried out. The owner of this product provided the author full access to the source code and version history of the software. During the investigation, snapshots for the three-year period were analyzed. For the extraction of instances of design patterns, Vokac [16] has built its own tool that supports the identification of five design patterns proposed by Gamma et al. [5]: Singleton, Template Method, Decorator, Observer, and Factory Method. The defects in the project were extracted through an integration of the version control system (CSV) with the defect detection system, and a textual analysis of the comments in the CSV to recover defects that had no direct connection between both systems.

Data analysis was performed using a logistic regression model

[**?**]. The author found some significant correlations for the patterns explored. The Factory design pattern had a lower defect rate. On the other hand, the Observer design pattern was correlated with higher occurrence rates of defects. The Template Method design pattern presented an inconclusive characteristic in the study, since it occurred in several different contexts. The Decorator design pattern did not present statistically significant results due to its low frequency of occurrence. Another interesting result of this study was that the combination of the Singleton design pattern with Observer tends to be used in complex areas with more code and higher frequency of defects. Therefore, the author concluded that both Singleton and Observer tend to associate complexity within a software.

4.4 Do Design Patterns Impact Software Quality Positively?

Khomh and Gueheneuce [9] raise a hypothesis that design patterns may not improve quality attributes as expected and, in addition, may generate negative impacts. In order to confirm the hypothesis raised, the authors prepared a survey to provide evidence on the impact of design patterns on software quality. In this survey, the following attributes were considered: expandability, simplicity, reusability, learning, comprehensibility, modularity, generalization, real-time modularity, scalability and robustness. The design patterns used in this study were the 23 standards proposed by Gamma et al. [5]. The questionnaire used to apply the survey was constructed based on these data. Thus, for each design pattern, one should classify the impact that design pattern has on each of the quality attributes. The scale used in the classification is composed of 6 different types of responses: very positive, positive, not significant, negative, very negative and not applicable. This questionnaire was applied during the period from January to April 2007 and, after this period, the authors started the analysis of the answers and applied a hypothesis test and a statistical test called Bernoulli distribution.

Khomh and Gueheneuce [9] have concluded that the use of design patterns does not always improve system quality. Some design patterns, such as Flyweight, for example, decrease some attributes, negatively impacting software quality. For this reason, some care is required in the use of these solutions during development, as they may hinder the maintenance and evolution of the software.

4.5 A Multiple Case Study of Design Pattern Decay, Grime, and Rot in Evolving Software Systems

Izurieta and Bieman [6] report that design patterns can degrade over time. The demand for requirements and maintenance performed on the project may imply the inclusion of class responsibilities and modifications to the project structure that increase the complexity of the source code and cause possible design patterns in the source code to lose their effectiveness. Due to these possibilities, the authors of this study proposed an investigation to understand to what extent, in the evolution of a software, the design patterns keep its structure flexible and easy to maintain. Furthermore, it was investigated whether the system maintain the initial levels of quality.

A multiple case study was conducted with three open source software projects developed in Java. In this case study, some time versions of these projects were obtained so that the effectiveness of the design patterns could be evaluated in the course of the evolution of each project. To extract the design patterns, they used the tools: Design Pattern Finder and PatternSeeker Tool. The evaluated design patterns were: Factory Method, Adapter, Singleton, State, Iterator, Proxy and Visitor. While conducting this study, the authors (i) identified the instances of the design patterns in each version of the software projects, (ii) reverse-engineered the UML diagram of the design patterns to verify entity relationships, (iii) mined software versions to capture information about the loss of effectiveness of design patterns, (iv) analyzed the results obtained and (v) evaluated twelve prepositions elaborated at the beginning of this evaluation.

Izurieta and Bieman [6] did not identify evidence of structural integrity decomposition of design patterns in these systems. However, considerable evidence of pattern decay has been found due to the accumulation of non-class artifacts that play roles in the patterns. Dependencies among components have increased, thus reducing modularity, testability, and adaptability of systems. Such occurrences are directly related to the increase of the coupling in the evolution of these projects.

4.6 Code Quality Cultivation

Speicher [15] argues that design patterns are considered as good programming practices and encourage the production of flexible and extensible software. However, these structures can still lead to bad smells in the source code of the project. One motivation regarding the Visitor design pattern is discussed by the author. This pattern tends to exhibit the bad smell Feature Envy. Speicher [15] presents an example in which this pattern separates the functionalities of the data, in order to aid the construction of complex objects and extend them to the use of new functionalities. However, how the Visitor design pattern accesses the elements data contributes to a false Feature Envy identification. Other design patterns such as Flyweight, Interpreter, Mediator, Memento, State, and Strategy are discussed and the author suggests that, in addition to separating objects and data, some properties of the patterns may be responsible for the occurrences of bad smells in the code.

Based on this discussion, an approach to automated identification of bad smells is proposed, which takes into account some decisions of the developers that may be considered false indications of bad smells. This approach was implemented based on a logical meta-programming, where Java code was represented as Prolog facts, and detection strategies and design patterns were defined as predicates. Amid the modeling of this approach, possible decisions of the developers that could be considered false indications of bad smells were modeled as verifications in their respective predicates. For example, in the Visitor design pattern, Visitor class access to data from other objects can be considered an envious feature, even though this is a responsibility of this class. Such occurrence represents a kind of natural odor and should be disregarded in the identification of bad smells.

As a way of evaluating the approach, a case study was carried out, in which an example is presented to identify bad smells taking into account intentions of developers. In this case study, an open software developed in the Java language, called ArgoUML, was used. After the execution of this case study, the author concludes that the proposed approach for identifying bad smell obtained a good precision.

4.7 A Proposal of Software Maintainability Model Using Code Smell Measurement

Wagey et al. [17] highlight the importance that the maintenance phase has on a software. They emphasize that 66% of the software lifecycle and 60% of the costs are spent just at this stage. This information shows that in the planning of a project it is necessary to take into account the maintainability of the same to guarantee a reduction of costs. Some solutions, such as design patterns, for example, tend to have a positive impact on the design of applications, avoiding the emergence of complex structures.

From this, the authors proposed a new model for measuring the external maintainability attribute in a software application, examining bad smells. This model was built based on an attribute dependency graph consisting of three levels: low, medium and high. The low level is responsible for including the metrics used in this model. They used 11 different metrics according to the bad smells characteristics to be identified in this model. The average level is composed of five bad smells, proposed by Fowler and Beck [4]. The high level contains maintainability characteristics: modularity, reusability, analysability, modifiability and testability. During the creation of this quality model, the authors derived metric scales for each of the bad smells. The scales used varied between the following characteristics: Very Good, Good, Medium, Bad, Very Bad.

To evaluate this model, Wagey et al. [17] conducted a case study with six open source Java applications. In these applications, data were collected regarding design patterns, metric values in each software and the density of each application's pattern. When analyzing the data obtained, the authors realized that the density value of the pattern has a positive impact on the maintainability of a software. Therefore, the higher the density of the pattern, the greater the maintainability of a project.

5. REFERENCES

[1] B. Cardoso and E. Figueiredo. Co-occurrence of design patterns and bad smells in software systems: An exploratory study. In *Proc. of the Brazilian Symposium on Information Systems*, pages 347–354, 2015.

[2] A. Christopoulou, E. A. Giakoumakis, V. E. Zafeiris, and V. Soukara. Automated refactoring to the strategy design pattern. *Information and Software Technology*, 54(11):1202–1214, 2012.

[3] M. Fowler. Callsuper. https://martinfowler.com/bliki/CallSuper.html, 2015. Accessed March 2017.

[4] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[6] C. Izurieta and J. Bieman. A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Software Quality Journal*, 21(2):289–323, 2013.

[7] F. Jaafar, Y. Guéhéneuc, S. Hamel, and F. Khomh. Analysing anti-patterns static relationships with design patterns. *Electronic Communications of the EASST*, 59, 2013.

[8] F. Jaafar, Y.-G. Gueheneuc, S. Hamel, F. Khomh, and M. Zulkernine. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empirical Software Engineering*, pages 896–931, 2016.

[9] F. Khomh and Y.-G. Gueheneuce. Do design patterns impact software quality positively? In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 274–278, 2008.

[10] W. Liu, Z.-G. b. Hu, H.-T. Liu, and L. Yang. Automated pattern-directed refactoring for complex conditional statements. *Journal of Central South University*, 21(5):1935–1945, 2014.

[11] W. B. McNatt and J. M. Bieman. Coupling of design patterns: Common practices and their benefits. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, pages 574–579, 2001.

[12] N. Nahar and K. Sakib. Automatic recommendation of software design patterns using anti-patterns in the design phase: A case study on abstract factory. In *CEUR Workshop Proc.*, pages 9–16, 2015.

[13] N. Nahar and K. Sakib. Acdpr: A recommendation system for the creational design patterns using anti-patterns. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 4*, pages 4–7, 2016.

[14] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition, 2007.

[15] D. Speicher. Code quality cultivation. *Communications in Computer and Information Science*, 348:334–349, 2013.

[16] M. Vokac. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering*, 30(12):904–917, 2004.

[17] B. C. Wagey, B. Hendradjaya, and M. S. Mardiyanto. A proposal of software maintainability model using code smell measurement. In *International Conference on Data and Software Engineering*, pages 25–30, 2015.

[18] B. Walter and T. Alkhaeir. The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology*, pages 127–142, 2016.

[19] P. Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In *Proceedings of the Fifth European Conference on Software Maintenance and*

Reengineering, pages 77–84, 2001.

[20] V. E. Zafeiris, S. H. Poulias, N. Diamantidis, and E. Giakoumakis. Automated refactoring of super-class method invocations to the template method design pattern. *Information and Software Technology*, pages 19–35, 2017.