A Systematic Review of the Approaches for Anti-Pattern Detection and Correction

SYEDA GHANA MUMTAZ, HUMA TAUSEEF*, MUHAMMAD ABUZAR FAHIEM, SAIMA FARHAN

Department of Computer Science, Lahore College for Women University, Lahore Pakistan

**Corresponding author's email: humaiftikhar@hotmail.com*

Abstract

Pattern is a prominent strategy amongst the most basic and capable strategies to enhance the outline, and subsequently upgrade the practicality and reusability of code. Anti-pattern identification is a helpful procedure for picking up information on the outline issues of existing systems and enhances the system's perceptions, which therefore upgrade the product viability and development. Various reviews have been directed and many tools have been produced to recognize anti-patterns, while just few reviews have considered the anti-pattern correction, which has not been researched with a similar degree of anti-pattern correction. Anti-pattern detection and correction approach combined together will be an effective approach to handle issues that arises during software modification. This paper reviews the existing approaches for anti-pattern detection and correction.

Keywords: Anti-patterns, Code smell, Software refactoring

INTRODUCTION

Nowadays, we are living in the age of technology, software and gadgets. With the passage of time, technology and gadgets become outdated and their new and updated versions become available in the market. Similarly, software also becomes obsolete to cope up with the changing demand of market. Software needs to be updated with the passage of time by adding new features and functionality. Software program enrichment, alteration and variation to accommodate new demands are complicated tasks [1]. These updates are most of the time done in a hurry by the developers to meet the demanding needs of market and users. Functionality enhancement of existing software systems always fetches parallel decrease in the quality of service (QoS) and an increase in difficulty of code [2], [3].

Software must be updated in such a way that there should be no change in overall functionality. Due to shortage of time and hurry, defects are induced in the design of software that subsequently degrades the quality of the software systems. There is a need to remove these defects to maintain the quality of the software systems. The defects that are introduced during the designing of code can be removed by refactoring process. Refactoring is the process in which written code is improved in a way that the internal structure of the source code is altered without varying its external behavior [4]. Basic concept of refactoring is to change different types of software artifacts.

Refactoring helps to enhance the quality of code by improving reusability, maintainability and modifiability. It is essential to accumulate the proper expertise, abilities, strategies, and equipment to completely benefit from refactoring.

Code smells are structural indicators in the code that propose the presence of a problem in the software code [4]. Code smells make the software maintainability and understandability hard for developers [5]. Presence of Code smells indicate that refactoring of code is needed. Code smells are issues that occur when basic qualities of programming are affected; no error occurs at execution time indicating that there are code sections that need removal or improvement by refactoring [6]. Code smells do not necessarily affect the functionality of the software but it is very important to look deeper to dig out if there is an underlying problem. Code smells don’t give the solution of the problem but they list down the possible refactoring techniques which can be used to solve them. Code smell detection methods are used to identify the parts that require refactoring. There are many kinds of code smells and numerous software refactoring techniques that are available for use. Some design solutions appear to be valuable for the reconstruction of design in updated procedure of code but their long-term consequences are harmful for design and quality of code. These design solutions are called anti-patterns.

Anti-patterns are common reoccurring problems of design introduced by developers in the software projects due to lack of adequate knowledge and experience to resolve a particular problem of design patterns. Just like code smells occurrence of anti-patterns in system decreases the quality of systems. Anti-patterns are used to categorize the common problems and their harmful effects, consequences and preventive measures [7]. Anti-pattern is an industry terminology for commonly occurring mistakes in software projects [8]. Presence of anti-patterns obstructs the maintenance process and escalates the risk of faults in the systems. Anti-patterns have negative impact on the system as they are poor programming practices. Anti-pattern examination is a fundamental need of any software development procedure. Prior knowledge of anti-patterns assists proficiency to prevent them or to recuperate from them. The structures having anti-patterns are much significantly testable. Presence of anti-patterns in any system is a strong indicator of presence of bugs in the system design. These bugs can bring out dubiousness of the system. It becomes incredibly critical to check such framework or system. Anti-patterns hinder the comprehension and maintainability of software [9]. Presence of Anti-patterns in code also increases the chances of its fault proneness [10], [11].

This paper presents an analysis of techniques and their evaluation metrics from literature for anti-pattern detection. Open research problems in the area have also been discussed to implicate new research horizons.

LITERATURE REVIEW

Anti-patterns need to be identified, monitored and removed to improve the maintainability and comprehension. It is not an easy task and becomes a challenge for complex and large systems. Literature survey is conducted to review different types of anti-patterns, their characteristics, detection techniques and detailed comparison. All these are discussed in detail in this section.

ANTI-PATTERNS

There are more than thirty kinds of anti-patterns identified by different researchers [4], [8]. Only those anti-patterns have been discussed here that have tools and techniques proposed for their detection.

Blob/God Class

Blob / God class is a class having too much responsibilities i.e. it has a lot of attributes and methods.

Feature ENVY

Feature envy is a class that takes more interest in some other class. Such a class is largely reliant on the 'envied' class. Cohesion of envied class is reduced due to the feature envy.

Duplicate Code/Clone Code

These are classes with same code structure at multiple places in a code. Duplicate code effects comprehensibility and maintainability of code. Duplicate code anti-pattern is difficult to detect because different copies of same feature suffers from different changes during evolution. If a change is desired in a duplicated feature then it has to be made with great care at all the places in code, thus increasing the effort to change the code. These anti-patterns effect stability and also increase fault proneness of code.

Refused Bequest

This anti-pattern occurs when the interface of superclass is not supported by the subclass [4]. It usually is the case when the inherited class overrides too many methods of the inherited class.

Divergent Change

This is class which is most changed out of all the classes present in the code. Each time it is changed in multiple ways for different reason. Such classes have low cohesion.

Shotgun Surgery

This anti-pattern is present whenever a little change is desired but one has to do lot of changes in many different classes [4]. Such anti-patterns are difficult to detect.

Parallel Inheritance Hierarchies

Parallel inheritance hierarchies occur in the cases when a subclass formation of one class leads to subclass formation of another class.

Functional Decomposition

This anti-pattern exists when inheritance and polymorphism are poorly used in a class. Such classes use private fields and create very few methods [12].

Spaghetti Code

This anti-pattern is present when code contains complex methods without any parameters. Classes interact using instance variables [12]. Spaghetti code is usually encountered when procedural programming approach is adapted.

Swiss Army Knife

It is a highly complex class with lot of responsibilities. This is the case when a class has multiple complex methods with lot of interfaces. This anti-pattern is usually confused with blob but both are different in nature. Blobs are self-centered and work for themselves but the Swiss army knife works for others.

Type Checking

This anti-pattern refers to the class that implements complicated conditional statements. They affect the maintainability and understanding of the code. The resulting problems tend to multiply over time.

CHARACTERISTICS OF ANTI-PATTERNS, THEIR IDENTIFICATION AND EXTRACTION

Three types of characteristics are commonly used for identification of anti-patterns; structural, lexical and historical characteristics. Structural characteristics are usually extracted by analyzing the code. Lexical characteristics are usually extracted using Natural Language Processing (NLP) and Information Retrieval (IR) methods. Extraction of historical characteristics requires data mining of software repositories. The methods to extract these characteristics are discussed in detail in this section.

Structural Characteristics Extraction

Structural characteristics can be extracted by analyzing method calls, shared variable instances and inheritance variables. Method call analysis can be done using Call-based Dependence between Methods (CDM) [13], Information flow-based Coupling (ICP) [14], Message Passing Coupling (MCP) [15] and Coupling Between Object classes (CBO) [15]. CDM is the most widely used technique to extract structural characteristics from code. It represents the coupling of methods. It has been used in different researches to identify feature envy [16], [17], blob [18]– [20], refused bequest [21], type checking [22], divergent change [23] and shotgun surgery [23].

Shared instance-based analysis can be implemented by finding out the structural similarity between methods (SSM) [24] or by using lack of cohesion of methods (LCOM) [25]. Researchers have used shared instance-based analysis to detect blob [13], [26] and spaghetti [26] anti-patterns from code. Analysis of relationships between two classes makes use of a Boolean value to indicate the presence of inheritance between two classes. This method has been utilized by researchers to detect parallel inheritance [27] and refused bequest [21].

Lexical Characteristics Extraction

Lexical characteristics are extracted using latent semantic indexing (LSI) [28], [29], that looks for the textual similarity. Feature envy [16] and blob [13] have been detected using lexical characteristics analysis.

Historical Characteristics Extraction

Different mining techniques are used to analyze software repositories for extraction of historical characteristics. Co-changes either at the file level or method level are analyzed to extract historical features of a code. Mining of log files is done using configuration management tools (CMTs) [30] to analyze co-changes at file level. For analyzing co-changes at method level a tool named Markos project [31] is used.

ANTI-PATTERN DETECTION APPROACHES

There exist many approaches for anti-pattern detection. These approaches have been grouped into classes based on the searching techniques and characteristics used. Some of the commonly used methods are discussed in this section.

Heuristic Based Approaches

These approaches use software metrics to detect anti-patterns. Cohesion and coupling are used to determine blobs in code [25]. Example of such a detection tool is DÉCOR [26] that uses rule cards to detect blobs. Another system named Marinescu [32] uses cohesion, coupling, absolute and relative threshold to discriminate blob and clean classes.

Structural Characteristics Based Approaches

Researchers often use structural characteristics of the code to detect anti-patterns. Move method refactoring (MMR) [16], [17] uses structural information to transfer a method to a particular class if the method uses more features of that class. JDeodorant [22] is a tool that uses MMR to remove feature envy from code. JDeodorant is also used to identify type checking and blob anti-pattern [18], [20], [22]. Syntactic or structural similarity can also be found using abstract syntax tree (AST) [33]. AST is traversed to calculate the referencing classes for each field, so that feature envy can be identified. Many approaches have been proposed to identify duplicate / clone code by finding out syntactic or structural similarity using AST [12], [34], [35]. Static analysis of the source code and dynamic evaluation of unit test are used to detect refused bequest [4], [21]. Structural properties can also be used to find out probabilities cdegree [23] to calculate change propagation probability (DCPP). DCPP helps to identify divergent change and shotgun surgery from code.

Historical Characteristics Based Approaches

Historical characteristics along with the structural characteristics of code are used to identify blobs, feature envy, divergent change, shotgun surgery and parallel inheritance hierarchies [27]. For blobs, regardless the type of change the date of change as indicated by code history log can give useful information. For feature envy detection, researchers lookout for a method that undergo alterations more often with an envied class on its own. Divergent classes are detected as multiple sets of methods, each set containing all the methods that change together but independently from methods in other classes. If a method changes with several other methods in other classes then it is considered to be shotgun surgery anti-pattern.

Lexical Characteristics Based Approaches

These approaches use software metrics to detect anti-patterns. Cohesion and coupling are used to determine blobs in code [25]. Example of such a detection tool is DÉCOR [26] that uses rule cards to detect blobs. Another system named Marinescu [32] uses cohesion, coupling, absolute and relative threshold to discriminate blob and clean classes. CCFinder [36] is a tool that uses lexical characteristics to identify duplicate clone code. Parallel inheritance hierarchies can be identified using lexical characteristics [4].

Mixed Approaches

These approaches use software metrics to detect anti-patterns. Cohesion and coupling are used to determine blobs in code [25]. Example of such a detection tool is DÉCOR [26] that uses rule cards to detect blobs, spaghetti code, swiss army knife and functional decomposition from code. Trees and syntactic analysis are combined to detect duplicate code. DECKARD is a tool that uses the same approach [37]. For borderline classes, while detecting blobs, rule cards are converted to Bayesian network of metric values [19]. Researchers applied the same approach but used signatures to determine similarity instead of rule cards for blob detection [20]. Most of the time rules are combination of different properties like quantitative, structural and lexical manifestations [24]. An automated anti-pattern detection approach based on detection rules is proposed by Kessentini *et al*. [38]. Another anti-pattern detection approach is proposed based on Prolog rules [39].

Goal-Question-Metric (GQM) based approach deliberately builds Bayesian Belief Networks (BBNs) to distinguish detection of anti-patterns in projects [19]. Oliveto *et al.* proposed a technique based on numerical analysis for anti-patterns detection using B-Splines (ABS) [20]. Logic based approach using prolog predicates is also used to detect structural and behavioral aspects of anti-patterns by researchers [39]. Palladio architectural models are used by Trubiani *et al.* for detecting and solving performance anti-patterns [40]. A metrics-based approach is also used for the identification of the area for code improvement [41]. Detection algorithms are also proposed from conditions using domain-specific language [26].

Automatic refactoring tools can be used to increase the quality of code [42]. Graphical user interface (GUI) based testing technique is proposed for computerization testing [43]. Genetic programming is used to detect web service anti-patterns [44]. Support vector machines (SVM) are also used for anti-pattern detection [45]. Process models for control-flow anti-patterns are devised by Han *et al.* [46].

COMPARISON

A detailed comparison of techniques for anti-pattern detection and correction as well as of the research work conducted by different researchers is conducted. Summarized tables of both comparisons for anti-pattern detection and correction are given.

COMPARISON OF DIFFERENT ANTI-PATTERN DETECTION APPROACHES

A comparison of anti-pattern detection approaches to improve system code, their advantages and limitations are given in Table 1. Different factors are considered for comparison like refactoring approach, types of anti-patterns detected, tools used in detection process, methodology adopted, system on which the technique is evaluated. It is obvious from the comparison that most of the work is inclined towards automated detection and in combinational field the major advantage of automated refactoring in anti-pattern detection is time saving mode as compared to semi-automated or manual detection.

COMPARISON OF DIFFERENT ANTI-PATTERN CORRECTION APPROACHES

As mentioned earlier that there is very little work found in literature for correction purpose of anti-pattern. The comparison of anti-pattern correction approaches to improve system code, their advantages and limitations are given in Table 2. Different factors are considered for comparison like refactoring approach, types of anti-patterns corrected, tools used in correction process, methodology adopted, system on which the technique is evaluated.

It is evident from the comparison that new approaches are inclined towards automated correction and in combinational field the major advantage of automated refactoring in anti-pattern correction is time saving mode as compared to semi-automated or manual detection.

Table 1: Comparison of Different Anti-pattern Detection Approaches

REFERENCE	OBJECTIVE	AUTOMATIC APPROACH	ANTI-PATTERNS DETECTED	TOOLS	METHODOLOGY	NO. OF SYSTEMS EVALUATED	ADVANTAGES	LIMITATIONS	CONCLUSION
Szoke, 2016 [42]	Development of automated tool to improve the source code quality	Yes	-	-	• Analysis, • Design & Development refactoring framework, • Application	05	The project provided the businesses a chance to refractor their code and improve code maintainability	The framework supporting refactoring of specific group of coding issues.	Recommendations can serve as a guideline in future for design and development of automatic refactoring tools.
Ujhelyi, 2014 [47]	Provide certain evaluation of memory utilization in extraordinary ASG representations and run time carryout performance of different Program query techniques.	Yes	-	Columbus framework, dedicated Linux-based server	• Java programs representation as an ASG or EMF • Graph pattern formalism • Implementation of queries using: ✓ ASG visitors ✓ Local search techniques ✓ Rete networks	05	Select the query analysis strategy supporting the specified usage profile	Results were consistent.	Accentuate on articulacy and brief formalism of pattern matching solutions over hand-coded approaches. Recommend quick execution and an easier way to experiment with queries.
Kaur & Kaur 2014 [43]	Suggest a GUI primarily based testing method for identity of anti-patterns through automation testing.	Yes	Blob	Eclipse	• Create a UML diagram • Generated UML conversion to XML • Read the XML file • Eclipse testing module • Code analysis • Finding Anti-pattern	-	Reduction in time and cost of anti-pattern detection	-	Computerization in testing for anti-pattern identification and of refactoring. can be proficient by digging out the key causes accountable for the occurrence of anti-patterns.

REFRENCE	OBJECTIVE	AUTOMATIC APPROCH	ANTI-PATTERNS DETECTED	TOOLS	METHODOLOGY	NO. OF SYSTEMS EVALUATED	ADVANTAGES	LIMITATIONS	CONCLUSION
Fourati *et al.* 2011 [48]	Advise an approach for detection of anti-patterns in UML designs with existing and newly defined quality metrics.	No	Blob, Functional Decomposition Swiss Army Knife	-	• Anti-pattern detection ✓ Structural anti-pattern detection ✓ Behavioral anti-pattern detection ✓ Semantic anti-patterns detection • Correction proposition	-	Improvement in the quality of design and code. Working at design level and reduction in correction cost	-	Approach produced very adequate levels of recall and precision.
Dhambri *et al.* 2008 [41]	Provide a visualization-based approach for detection of design anomalies.	No	Blob, Functional Decomposition Swiss Army Knife, Divergent Change, Shotgun Surgery	VERSO, PMD	• Detection Principle ✓ Mappings metric, graphical attribute ✓ Distribution filter application ✓ Mark class ✓ Inspect class ✓ Apply relationship filters ✓ Save the occurrence	02	Allows the analysis of enormous sets of data during progressing time in an economical manner.	The performance inconsistency of systems still needs enhancement.	Semi-automatic detection has superior recall for some anomaly types.
Ouni *et al.* 2013 [44]	Introduce an automatic approach to stumble on net provider anti-patterns through genetic programming	Yes	Service related anti-patterns	-	• Input web service anti-pattern instances, • Input Web service metrics • Derives a set of detection rules	310 Web services	Gives elevated preciseness and recall values, and acceptable execution time.	Approach works on a limited set of anti-patterns	Quality of rule is based on the maximum number of anti-patterns detect in terms of precision and recall.

REFRENCE	OBJECTIVE	AUTOMATIC APPROCH	ANTI-PATTERNS DETECTED	TOOLS	METHODOLOGY	NO. OF SYSTEMS EVALUATED	ADVANTAGES	LIMITATIONS	CONCLUSION
Stoianov & Sora 2010 [39]	Propose detection methods by using a logic-based approach for a set of patterns and anti-patterns.	Yes	Blob	JTransfor mer	• Data Class ✓ Called predicate get method's name parameters and body ✓ Check name ✓ Check setter method parameter ✓ Calculate code lines verification • Blob ✓ Calculates the complexity of the class, ✓ Use predicate for verification	06	Simplicity of defining Prolog predicates that are able to describe both structural and behavioral aspects of patterns and anti-patters.	-	In manual inspection of the code, no false positives were found among the automatic detected anti-patterns.
Tan *et al.* 2016 [49]	Utilize anti-patterns to mitigate the issue of wrong or inadequate fixes coming about from program repair.	No	Conditional Statement anti-patterns	Gen Prog, SPR	• Anti-patterns filtering algorithm ✓ Program, ✓ Transformations functions ✓ Evaluation • mSPR Repair generation algorithm ✓ Program ✓ Positive and negative test cases ✓ Transformation functions. ✓ Repaired program	-	Simplicity of defining Prolog predicates ready to describe structural and behavioral aspects of patterns and anti-patters.	Proposed set of anti-patterns is not complete.	Obtain repairs that localize the correct lines, involve less deletion of program functionality, and are mostly obtained more efficiently.

REFRENCE	OBJECTIVE	AUTOMATIC APPROCH	ANTI-PATTERNS DETECTED	TOOLS	METHODOLOGY	NO. OF SYSTEMS EVALUATED	ADVANTAGES	LIMITATIONS	CONCLUSION
Maiga *et al.* 2012 [45]	Present SVM Detect, a novel way to deal with detection of anti-patterns , in view of a machine learning strategy SVM.	Yes	Blob, Functional Decomposition, Spaghetti Code, Swiss Army Knife	POM, Weka	• Object Oriented Metric Specification ✓ Training dataset ✓ Calculate object-oriented metrics • SVM Classifier training ✓Define the training dataset ✓Find the optimal hyper-plane • Dataset Construction • Detection of an anti-pattern	03	SVM can be applied on subsetsof systems.	Experimentationc annot be performed on other anti-patterns detection due to lack of manually validated oracle.	SVM-based approach can overcome the limitations of previousapproaches.
Han *et al.* 2013 [46]	Reason an approach which can bolster client characterized control-flow anti-pattern identification with various process demonstrating languages.	Yes	Conditional statements anti-patterns	-	• Merging of Duplicate predicates/rules in CAPDL definition • Conduct node rule query • Conduct block rule query. • Obtain anti-pattern information according to above rules	278 Process models	Detection of multiple anti-patterns requires only one execution of preprocessing and one execution of RPST traversal.	Difficulty to sustain precise control flowstructures with modeling languages.	The proposed approach can detect User-defined control-flow anti-patterns effectively.

REFRENCE	OBJECTIVE	AUTOMATIC APPROCH	ANTI-PATTERNS DETECTED	TOOLS	METHODOLOGY	NO. OF SYSTEMS EVALUATED	ADVANTAGES	LIMITATIONS	CONCLUSION
Maiga 2012 [45]	Introduce SMURF, a singular method to come across anti-patterns, primarily based on support vector machines.	Yes	Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife	POM, PADL, Weka	• Object Oriented Metric Specification ✓ Training dataset TDS ✓ Calculate object-oriented metrics • Train the SVM Classifier ✓ Find the maximum margin hyper-plane ✓ Dataset DDS construction • Detection ✓ The occurrences of anti-pattern ✓ Interactive learning and practitioners' feedback	03	Can be applied in both intra-system and inter-system configurations.	Study is not pretentious in case of potential dependence of the obtainedresults on the chosen anti-patterns and systems.	SMURF precision improves when using practitioners' feedback.
Moha 2010 [26]	Advocate DÉCOR and DETEX	No	Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife	-	• Décor method ✓ Description Analysis ✓ Specification ✓ Processing ✓ Detection ✓ Validation • DETEX detection technique of décor ✓ Domain Analysis ✓ Specification ✓ Algorithm Generation Detection	01	Detection technique can be generalizedto other smells.	-	The results of the validation are repeatable and reliable.

Table 2: Comparison of Different Anti-pattern Correction Approaches

REFRENCE	OBJECTIVE	AUTOMATIC APPROCH	ANTI-PATTERNS DETECTED	TOOLS	METHODOLOGY	NO. OF SYSTEMS EVALUATED	ADVANTAGES	LIMITATIONS	CONCLUSION
Morales 2015 [50]	Propose meta-heuristics based technique for automated refactoring.	Yes	-	Eclipse	• meta-heuristics • purpose search-based approach • level of design-defects • refactoring the code • Implement a search-based approach	-	Comprehension of developers to improve the design quality.	There is no formal approach.	Search-based approach and an Eclipse plug-in can be used to improve the design quality of projects.
Moha *et al.* 2008 [51]	Scrutinize FCA and concept lattices for the improvement of design defect	No	Blob	PADL, Galicia	• Build a model of the source code • Apply well-known algorithms • Contexts are fed into a FCA engine	01	Approach is extensive to other design defectsusing high coupling and low cohesion.	-	FCA represent a better tradeoff between coupling and cohesion.
Llano & Pooley 2009 [52]	Open upthe likelihood to automate the detection and correction of Object-Oriented Anti-patterns	Yes	Blob	-	• Built model for specification of anti patterns • production rules	-	UML specification can be used to overcome problems.	Correction rules may produce side effects.	A transformation is proposed for the correction of the anti-patterns.

Table 3: Comparison of Combinational Approaches for Detection and Correction of Anti-patterns

REFRENCE	OBJECTIVE	AUTOMATIC APPROCH	ANTI-PATTERNS DETECTED	TOOLS	METHODOLOGY	NO. OF SYSTEMS EVALUATED	ADVANTAGES	LIMITATIONS	CONCLUSION
Kessentini *et al.* 2011 [38]	Propose an automated approach for detection and correction of design defects	Yes	Blob, Spaghetti Code, Functional Decomposition	DECOR	Heuristic method. Find detection rules Correction solutions Combination of refactoring operations. Rule extraction Use genetic algorithm for correction.	04	The technique can be used in different situation and will produce good detection, correction and recall results for the detection of anti-patterns.	Might require more work than identify, specify, and adapt rules.	The best solution has the minimum fitness value.
Ouni et al. 2013 [44]	Propose automated approach to detect and correct maintainability defects.	Yes	Blob, Spaghetti Code, Functional Decomposition	ECLIPSE	Defect detection process Finding code fragments Applying refactoring operations Correction of maintainability defects Software code Refactoring operations Values for each operation Defect detection rules	06	Approach provide better precision results on genetic algorithm.	Show low recall score for the different systems.	Our study shows that technique outperforms DECOR.

REFRENCE	Hadimani et al. 2011 [53]	Moha 2007 [54]
OBJECTIVE	Class slicing based correction technique to meet the design specifications.	Provide a logical method to specify design defects and produce detection and correction algorithms.
AUTOMATIC APPROCH	No	No
ANTI-PATTERNS DETECTED	-	-
TOOLS	-	-
METHODOLOGY	Defect Detection Approach Specify the quality goals Perform static program analysis Do Metric Computation Detection process Primary classifier Design defects accurately Design Defect Identification Suggest changes Defect Correction Approach Trimming the classes Refactoring Formalization Slicing Classes Software Redesign Result Refinement	Textual descriptions of design defects Analysis Taxonomy Specification Meta-modelling Modelling Generation Detection Validation 1 Correction Validation 2
NO. OF SYSTEMS EVALUATED	-	-
ADVANTAGES	Facilitate the development of concrete tools for the detection and correction of design defects	It allows the efficient measurement and detection of design defects, in comparison with previous works.
LIMITATIONS	-	-
CONCLUSION	A systematic technique that covers the process of design defects detection as knowledge base, and given a correction technique by using class slicing.	This method allows maintainers to specify design defects in terms of structural, semantic, and quantifiable property of classes and structural relationships among classes.

FUTURE DIRECTIONS

While taking anti-patterns refactoring into consideration there is a lack of detailed correction approach to correct anti-patterns in source code to improve the quality of code and to minimize testing efforts. An approach that can first detect and then correct the detected anti-patterns for minimizing testing efforts will be very helpful. With the help of automated correction approach developers can modify the project very easily while keeping testing efforts low and service quality of system high. The continuous involvement of developer is compulsory for source code related projects. Therefore, if correction approach is used to manage anti-patterns in source code it will be a remarkable contribution.

Linguistic anti-patterns [55] are attracting many researchers nowadays. Inappropriate or missing comments, ambiguous selection of identifiers and poorly used coding standards increase the risk of presence of anti-patterns in code [56]–[61]. Design patters focus on reoccurring problems of code while linguistic anti-patterns emphasize on symptoms and their consequences.

Most of the identification systems rely on the static analysis of code. There is a need to dynamically analyze the behavior of code and extract the characteristics accordingly. Historical characteristics should also be incorporated with structural characteristics to improve precision of the identification systems.

CONCLUSION

With the rapidly moving technology market, the software also needs to evolve day by day to meet the changing market and user needs. Several approaches are proposed for anti-pattern detection at different levels of software lifecycle. There is a need of an automated anti-pattern correction approach that can help the developers in developing and modifying software.

REFERENCES

[1] D. Coleman, D. Ash, B. Lowther, and P. Oman, “Using metrics to evaluate software system maintainability,” *Computer*, vol. 27, no. 8, pp. 44–49, Aug. 1994.

[2] A. Garrido, G. Rossi, and D. Distante, “Refactoring for usability in web applications,” *IEEE Softw.*, vol. 28, no. 3, pp. 60–67, 2011.

[3] I. Balaban, F. Tip, and R. Fuhrer, “Refactoring support for class library migration,” in *ACM SIGPLAN Notices*, 2005, vol. 40, no. 10, pp. 265–279.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the design of existing code*. Addison-Wesley Professional, 1999.

[5] J. Padilha, J. Pereira, E. Figueiredo, J. Almeida, A. Garcia, and C. Sant’Anna, “On the effectiveness of concern metrics to detect code smells: an empirical study,” in *Int. Conf. Advanced Information Systems Engineering*, 2014, pp. 656–671.

[6] J. Fields, S. Harvie, M. Fowler, and K. Beck, *Refactoring: Ruby Edition*. Pearson Education, 2009.

[7] M. H. Dodani, "Patterns of Anti-Patterns" *J. Object Technol.*, vol. 5, no. 6, pp. 29–33, 2006.

[8] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[9] M. M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle," *J. Syst. Softw.*, vol. 1, pp. 213–221, 1979.

[10] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empir. Softw. Eng.*, vol. 17, no. 3, pp. 243–275, 2012.

[11] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *J. Syst. Softw.*, vol. 80, no. 7, pp. 1120–1128, 2007.

[12] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. Int. Conf. Software Maintenance,* 1998, pp. 368–377.

[13] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *J. Syst. Softw.*, vol. 84, no. 3, pp. 397–414, 2011.

[14] Y. Lee, B. S. Liang, S. F. Wu, and F. J. Wang, "Measuring the coupling and cohesion of an object-oriented program based on information flow," in *Proc. Intl. Conf. Software Quality*, 1995, pp. 81–90.

[15] W. Li and S. Henry, "Maintenance metrics for the object oriented paradigm," in *Proc. First Int. Symp. Software Metrics*, 1993, pp. 52–60.

[16] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 671–694, 2014.

[17] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 347–367, 2009.

[18] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *J. Syst. Softw.*, vol. 85, no. 10, pp. 2241–2260, 2012.

[19] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *9th Int. Conf. Quality Software (QSIC'09)*, 2009, pp. 305–314.

[20] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *14th European Conf. Software Maintenance and Reengineering (CSMR 2010)*, 2010, pp. 248–251.

[21] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," in *IEEE Int. Conf. Software Maintenance*, 2013, pp. 392–395.

[22] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells," in *12th European Conf. Software Maintenance and Reengineering (CSMR 2008)*, pp. 329–331.

[23] A. A. Rao and K. N. Reddy, "Detecting bad smells in object-oriented design using design change propagation probability matrix," in *Int. Multi Conf. of Engineers and Computer Scientists*, 2008, vol. I, pp. 1001–1007.

[24] G. Gui and P. D. Scott, "Coupling and cohesion measures for evaluation of component reusability," in *Proc. 2006 Int. Workshop Mining Software Repositories*, 2006, pp. 18–21.

[25] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[26] N. Moha, Y.-G. Gueheneuc, A.-F. Duchien, and Others, "Decor: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[27] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proc. 28th IEEE/ACM Int. Conf. Automated Software Engineering*, 2013, pp. 268–278.

[28] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 287–300, 2008.

[29] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval-based coupling measures for impact analysis," *Empir. Softw. Eng.*, vol. 14, no. 1, pp. 5–32, 2009.

[30] A. Leon, *Software configuration management handbook*. Artech House, 2015.

[31] A. Laskowska, J. G. Cruz, I. Kedziora Pawełand Lener, B. Lewandowski, C. Mazurek, and M. Di Penta, "Best practices for validating research software prototypes-MARKOS case study," in *Conf. eChallenges (e-2014)*, 2014, pp. 1–9.

[32] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. 20th IEEE Int. Conf. Software Maintenance*, 2004, pp. 350–359.

[33] C. De Roover, T. D'Hondt, J. Brichau, C. Noguera, and L. Duchien, "Behavioral similarity matching using concrete source code templates in logic queries," in *Proc. 2007 ACM SIGPLAN Symp. Partial evaluation and semantics-based program manipulation*, 2007, pp. 92–101.

[34] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS®: Program transformations for practical scalable software evolution," in *Proc. 26th Int. Conf. Software Engineering*, 2004, pp. 625–634.

[35] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *Fourth IEEE Int. Workshop on Source Code Analysis and Manipulation*, 2004, pp. 128–135.

[36] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[37] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf Software Engineering*, 2007, pp. 96–105.

[38] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *19th IEEE Int. Conf. Program Comprehension*, 2011, pp. 81–90.

[39] A. Stoianov and I. Sora, "Detecting patterns and antipatterns in software using Prolog rules," in *Int. Joint Conf. Computational Cybernetics and Technical Informatics (ICCC-CONTI 2010)*, 2010, pp. 253–258.

[40] C. Trubiani and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models," in *ACM SIGSOFT Software Engineering Notes*, 2011, vol. 36, no. 5, pp. 19–30.

[41] K. Dhambri, H. Sahraoui, and P. Poulin, "Visual detection of design anomalies," in *Proc. European Conf. Software Maintenance and Reengineering, CSMR*, 2008, pp. 279–283.

[42] G. Szoke, C. Nagy, R. Ferenc, and T. Gyimóthy, "Designing and developing automated refactoring transformations: An experience report," in *IEEE 23rd Int. Conf. Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016, vol. 1, pp. 693–697.

[43] H. Kaur and P. J. Kaur, "A GUI based unit testing technique for antipattern identification," in *5th Int. Conf.-The Next Generation Information Technology Summit (Confluence)*, 2014, pp. 779–782.

[44] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: A multi-objective approach," *Autom. Softw. Eng.*, vol. 20, no. 1, pp. 47–79, 2013.

[45] A. Maiga *et al.*, "Support vector machines for anti-pattern detection," in *Proc. 27th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, 2012, pp. 278–281.

[46] Z. Han, P. Gong, L. Zhang, J. Ling, and W. Huang, "Definition and detection of control-flow anti-patterns in process models," in *IEEE 37th Annu Computer Software and Applications Conf. Workshops (COMPSACW)*, 2013, pp. 433–438.

[47] Z. Ujhelyi *et al.*, "Anti-pattern detection with model queries: A comparison of approaches," in *IEEE Conf. Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week*, 2014, pp. 293–302.

[48] R. Fourati, N. Bouassida, and H. Ben Abdallah, "A metric-based approach for anti-

pattern detection in UML designs," in *Computer and Information Science 2011*, Springer, 2011, pp. 17–33.

[49] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proc. 24th ACM SIGSOFT Int. Symposium Foundations of Software Engineering*, 2016, pp. 727–738.

[50] R. Morales, "Towards a framework for automatic correction of anti-patterns," in *IEEE 22nd Int. Conf. Software Analysis, Evolution and Reengineering (SANER),* 2015, pp. 603–604.

[51] N. Moha, J. Rezgui, Y.-G. Guéhéneuc, P. Valtchev, and G. El Boussaidi, "Using FCA to suggest refactoring to correct design defects," in *Concept Lattices and Their Applications*, Springer, 2008, pp. 269–275.

[52] M. T. Llano and R. Pooley, "UML specification and correction of object-oriented anti-patterns," in *Fourth Int. Conf. Software Engineering Advances*, 2009, pp. 39–44.

[53] D. K. Saini, L. A. Hadimani, and N. Gupta, "Software testing approach for detection and correction of design defects in object-oriented software," *J. Comput.*, vol. 3, no. 4, 2011.

[54] N. Moha, "Detection and correction of design defects in object-oriented designs," in *Companion to the 22nd ACM SIGPLAN Conf. Object-oriented Programming Systems and Applications Companion*, 2007, pp. 949–950.

[55] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Gueheneuc, "A new family of software anti-patterns: Linguistic anti-patterns," in *17th European Conf. Software Maintenance and Reengineering (CSMR 2013),* 2013, pp. 187–196.

[56] Caprile and Tonella, "Restructuring program identifier names," in *Proc. Int. Conf. Software Maintenance ICSM-94*, 2000, pp. 97–107.

[57] F. Deisenbock and M. Pizka, "Concise and consistent naming," in *13th Int. Workshop on Program Comprehension (IWPC'05)*, 2005, pp. 97–106.

[58] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innov. Syst. Softw. Eng.*, vol. 3, no. 4, pp. 303–318, 2007.

[59] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *14th IEEE Int. Conf. Program Comprehension (ICPC),* 2006, pp. 3–12.

[60] E. Merlo, I. McAdam, and R. De Mori, "Feed-forward and recurrent neural networks for source code informal information analysis," *J. Softw. Maint. Evol. Res. Pract.*, vol. 15, no. 4, pp. 205–244, 2003.

[61] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Modeling the evolution of topics in source code histories," in *Proc. 8th Working Conf. Mining Software Repositories*, 2011, pp. 173–182.