East African Scholars Journal of Engineering and Computer Sciences
Abbreviated Key Title: East African Scholars J Eng Comput Sci
ISSN: 2617-4480 (Print) & ISSN: 2663-0346 (Online)
Published By East African Scholars Publisher, Kenya

Volume-4 | Issue-10 | Dec-2021 | DOI: 10.36349/easjecs.2021.v04i10.007

Review Article

Influence of Anti-Patterns Detection Techniques on Software Maintenance

Dr. Vadivel G[1*], Er. Abhinash Singla[2], Er. Sanju Kumari[2]
[1]Associate Professor IT, Bhai Gurdas Institute of Engineering & Technology, Sangrur, Punjab
[2]Assistant Professors IT, Bhai Gurdas Institute of Engineering & Technology, Sangrur, Punjab

Article History
Received: 18.10.2021
Accepted: 29.11.2021
Published: 30.12.2021

Journal homepage:
https://www.easpublisher.com

Quick Response Code

Abstract: Anti-patterns are flaws that adversely impact the quality of the system. The term "Code Smell" refers to a sign that there are anti-patterns in software, which prompts system restructuring. As a result, managing the maintenance becomes challenging. Refactoring is required more when the number of smells increases. Different methods for detecting anti-patterns in the system have been found. The research sought to examine how anti-patterns affect classes and which specific anti-patterns have a greater influence than others. A conclusion has been reached regarding the findings for upcoming research on open-source systems. The introduction is followed by the various anti-patterns in the first of the paper's four sections. Additionally, the associated work has been thoroughly analysed, followed by a succinct conclusion. Thus, the study presents many methods for identifying software system code smells. Therefore, odour detection will be useful in increasing reliability during testing and maintenance phases by foreseeing anti-patterns and errors before to product delivery. Additionally, the community of software engineers and managers will benefit from the identification of anti-patterns by using it to enhance software development maintenance efforts.

Keywords: Anti-patterns, Code Smells, Refactoring and Maintenance.

1. INTRODUCTION

Software systems need intelligent development and ongoing maintenance to be of high quality. Numerous quality assurance procedures, including formal technical reviews, testing, and standard enforcement, can improve quality. Every time new software needs to be created, existing resources (source code files, design templates) are used in the creation process. But most of the time, the performance and quality of the software suffer when software components can be reused. Software libraries, design patterns, frameworks, and systematic software reuse are a few typical examples of software reusability. The phrase "Design Patterns" was coined by software developers to describe situations in which a problem occurs repeatedly and a usual solution is required.

However, the design patterns frequently begin acting as "ANTI-PATTERNS." These design flaws are known as anti-patterns or code smells. If there are code smells, maintenance of the system is required. The presence of anti-patterns in the system is suggested by the term "code smell." The more scents there are, the more care is required. Anti-Patterns are characterised as a traditional design that offers a solution to a problem that could have unfavourable effects. Anti-patterns are therefore viewed as the bad solutions that create more issues than they solve. As a result, the system begins to demand Refactoring [6].

The goal of the study is to reveal the anti-patterns that frequently occur in systems and are normally described in [5, 6]. The work completed will benefit the community of software engineers and managers in order to enhance the activities involved in software development and maintenance.

2. ANTI-PATTERNS IN SOFTWARE SYSTEM

The design of the software is what defines its quality. As time goes on and adjustments are made to the structure as a result of shifting user demands, the design begins to fall apart. Anti-patterns or "Code Smells" are two ways that software flaws manifest themselves [6]. The distinction between a code smell and an anti-pattern is extremely subtle. Anti-patterns are viewed as poor programming practises, not mistakes. Due of the software developers' inexperience and lack of expertise, anti-patterns are being added into the literature to address certain issues. Code Smell is a symptom that identifies a software system issue [5]. It implies the admission of anti-patterns. Although technically correct, code smells point to delicate design. It could result in system failure and future bug risk. Refactoring is therefore required by the system, which involves modifying the current software code without changing the external behaviour. In [6], nearly 20 code smells are listed that are present in the source code and call for refactoring. The following author suggests the following categories of anti-patterns/code smells.

Table 1: Types of Anti-Patterns

Author Name	Anti-pattern/Code Smells
M. Fowler [6]	Lazy Class, Large Class, Long Method, Long Parameter List, Message Chain, Duplicate Code, Divergent Change, Shot Gun Surgery, Feature Envy, Data Clumps, Primitive Obsession, Switch Statements, Parallel Inheritance Hierarchies, Middle Man, Speculative Generality, Inappropriate Intimacy, Temporary Field
B.F. Webster [1] and W. J. Brown *et al.*, [5]	Blob, Spaghetti Code, Conditional Complexity, Anti-Singleton, Class Data Should Be Private (CDSBP), Refused Parent Bequest (RPB), Swiss Army Knife

3. RELATED WORK

Work on patterns produces anti-patterns. These are seen as poor design decisions rather than mistakes. Code smells are a sign that anti-patterns are present. Andrew Koenig coined the term "anti-pattern" when he first discussed how patterns are perceived in software engineering [3]. Code smells and anti-patterns are phrases that are frequently used synonymously.

A number of authors investigated the impact of code smells and anti-patterns on software systems. The work investigates code smells and antipatterns in the context of software engineering activities. Webster [1], who also discusses the dangers associated with coding and quality assurance, wrote the first book on odours. According to [5], object-oriented systems require care. In [5], more than 35 odours were listed, including the well-known "BLOB" design smell. Due to nearly 20 code smells, the idea of refactoring was introduced in [6]. The word "refactoring," which refers to the process of updating a software system without changing its external behaviour, was explained in depth by the author. As examples of code smells, the author listed Lazy Class, Long Method, Long Parameter List, Shotgun Surgery, etc. The aforementioned authors offer in-depth knowledge on anti-patterns and code smells. However, the authors' strategy for identifying anti-patterns was entirely manual. For large projects, it was a labour-intensive and error-prone task. Thus, some studies suggested the statistical, visual, and automatic detection strategies.

3.1 Traditional Detection Techniques

The researchers introduced various strategies for manually identifying anti-patterns and code smells in the system in addition to the usual detection techniques. The development of automatic and semi-automatic anti-pattern detection methods began with this. The methods used range from software reading to metric-based and template-driven approaches. Here, a few of the methods are discussed:

Travassos and F. Shull made the manual detection approach [31] suggestion [7]. It was a method of reading software that aided in Object Oriented Systems' ability to detect odours. Defect-based reading, perspective-based reading, and use-based reading are only a few of the various reading techniques that have been used for detection. The team of students created a project that was evaluated on two criteria: horizontal review and vertical review. The process took a lot of time and was entirely manual.

Connie U. Smith and Lloyd G. William [8] looked into the system's impact of the anti-pattern God Class and provided solutions. Additionally, they suggested three additional performance antipatterns that are prevalent in software systems. A metric-based approach for the detection of anti-patterns was introduced by R. Marinescu [14]. The method was implemented on Tool plasma. Nearly the same amount of approaches were used to identify more than 8 anti-patterns. The threshold values were contrasted with metrics and set operator values.

A template-driven model was proposed by M.J. Munro [16] to identify anti-patterns. The template has three parts: the name of the fragrance, Textual description of code smell characteristics Heuristics for odour recognition. He looked at the product metrics for detecting "bad odours" in Java source code. The research sought to determine the peculiarity of code smell using metrics. The metrics findings that are applied to Java source-code were calculated using interpretation rules. Therefore, the locations of foul odours in the Java code may be easily determined based on the estimated results. On two case studies, a prototype tool has been employed for the implementation.

A language developed by H. Alikacem and H. Sahraoui [20] identifies quality characteristics that are overlooked and offers a mechanism for smelling out odours in object-oriented systems. With the assistance of fuzzy logic, metrics, association, and inheritance, the terminology offered the rules. However, it wasn't tested on any actual projects in the real world. R. Allen and D.

Garlan [2, 4], E.M. Dashofy and A. vander Hoek *et al.*, [17] all address further similar detection methods.

3.2 Visualization Based Detection Techniques

The researchers employed a variety of methods for visualization-based detection strategies, including Metric Based Visualization Technique, Visualized Design Defect Detection Strategy, Domain Specific Language, etc. Here, a few of the methods are discussed.

A potent method to examine the internal software quality using a metric-based visualisation approach was proposed by E. Simon *et al.*, [10]. Move function, move attribute, extract class, and inline class were the four types of source code restructuring that have been examined. Crocodile, an improved metric tool, had been employed. The method allowed the software engineers to spot the "code smells" by clicking the mouse and adhering to the visualisation guidelines.

A visualisation strategy was laid forth by E. Langelier *et al.*, [19] for the quality analysis of large-scale systems. A framework had been offered, and open source software had been used to implement it. The classes were represented using geometric 3D boxes. The metrics for coupling, cohesion, inheritance, and size complexity (CBO, LCOM5, DIT) have their values analysed. WMC metrics were applied.

A technique based on visualisation for design fault detection was presented by K. Dhambri *et al.*, [25]. Blob, functional decomposition, and divergent change are the three forms of anomalies for which the technique has been validated. In the near future, the study will expand to include an automatic detection-based approach.

The investigation of offensive odours during the designing process was undertaken by Cedric Bouhours *et al.*, For the purpose of identifying offensive odours, the spoiled patterns had been focused. Spoiled patterns are those that didn't give the system they were created for the required functionality, according to the definition of a spoiled pattern. There had been a comparison between design patterns and ruined patterns.

For the purpose of uncovering anti-patterns, Naouel Moha *et al.*, [31] established a domain-specific language based on DECOR.

By following the processes of Description analysis, Specification, Processing, Detection, and Validation, it is a method that provides a track for the description of antipatterns. It deploys a detection system called DETEX, which serves as a reference instance of DECOR. On 11 open source systems, more than 15 different types of code odours had been discovered.

3.3 Automatic Detection Techniques

Fully automatic detection tools has been applied in the automatic detection tactics. A couple of these methods are proven on real-world systems while various anti-pattern kinds are discovered. Here, a few of the methods are discussed:

Three categories of design flaws—intra-class (inside class), inter-class (among classes), and semantic nature—were established by Yann-Gael Gueheneuc *et al.*, [9]. Design patterns had been described using a meta model. Ptidej Tool could be used to quickly fix inter-class design flaws. Eva van Emden and Leon Moonen [11] outlined a method for raising the calibre of java source code software. The tool for automatic software inspection can also make use of the final results. To reveal the smells in Java source code, the jCOSMO code smell browser was created. The instrument has been approved for use with CHARTOON. A hierarchical prototype was suggested by Jagdish Bansiya and Carl G. Davis [12] for the assessment of quality attributes (reusability, flexibility, understandability, etc.) in object-oriented designs. Using design metrics such as DAM, DCC, CAM, etc., architectural and detectable equity of classes and their objects are computed. The model offered a method for quickly applying it to projects in the real world.

A tool set called Ptidej, developed by Yann-Gael Gueheneuc [15], can reliably translate multiple programming languages into UML class diagrams by reverse engineering. In order to identify code smells at a higher level of abstraction, PTIDEJ creates UML class diagrams. Different reverse engineering tools, including Rational Rose, ArgoUML version 0.14.1, Chava Fujaba version 4.0.1 IDEA, Borland Together, and Womble recover, were briefly described by the author.

The reformation of seven open source Java systems, including MegaMek, JasperReports, Antlr, Tyrant, PDFBox, Velocity, and HSQLDB, was documented by S. Counsell and Y. Hassoun [21]. The findings showed that renaming and moving fields and methods across the code are the two most frequently used re-engineering techniques for open source systems.

Design Motif Identification Multilayered Approach (DeMIMA) for the detection of micro structures was presented by Yann Gael Gueheneuc and Giuliano Antoniol [26]. (complementary to design motifs). A copy of the source code was provided by the first two layers, and design patterns were found by the third layer in this three-layered architecture. Using explanation-based constraint programming, the method provided 100% recall on both the open source and commercial systems.

To find instances of foul odours in the software, Stephane Vaucher *et al.*, [28] extensively studied the God. Classes. For the examination of God Classes, the open-source systems Xerces and Eclipse JDT had been explored.

IDS (Immune based Detection Strategy), a machine learning approach powered by the human body's immune system, was introduced by Salima Hassaine *et al.,* [32]. If code smells and anti-patterns were present, the system might be quickly detected. We personally examined Xerces v2.7.0 and Gantt Project v1.10.2 for the presence of smells. Bayesian Detection Expert, a Goal Question Metric (GQM) based method to build Bayesian Belief Networks (BBNs) from the descriptions of anti-patterns, was proposed by Foutse Khomh *et al.,* [34]. Whether a class is an anti-pattern or not was studied by BBN. Three anti-patterns, including the Gantt Project and Xerces open source systems, Blob, Functional Decomposition, and Spaghetti Code, are validated for BDTEX. Eclipse and JHotDraw are two commercial projects to which the method is also applied.

Software metrics and encapsulation have been found to be important for exposing code smells, according to Satwinder Singh and K.S. Kahlon [35]. The classification of smells in the code was made possible by the introduction of a software metric model. The open source Firefox system had been looked into to verify the outcomes.

A metric approach was devised by Satwinder Singh and K.S. Kahlon [36] for analysing the stinky classes in the system. The study found that identifying bad classes and bad code may be done with the use of the metrics' results. Various software analysis code scent detection techniques are allegedly available on the market, according to Francesca Arcelli Fontana *et al.,* [37], although it is still unclear how accurate this claim can be made. As a result, six different Gantt Project iterations have been examined in order to find four different sorts of code smells utilising more than six different tools.

Source code changes (SCC) obtained from 16 Java open source systems were taken into account by Daniele Romano, Paulius Raila, and colleagues when studying the system [38]. Complex Class, Spaghetti Code, and Swiss Army Knife are three examples of anti-patterns. The amount of code changes in antipattern classes have been found to be higher than the number of changes without an antipattern.

Foutse Khomh *et al.,* [40] looked into how antipatterns affected classes. Four systems—ArgoUML, Eclipse, Mylyn, and Rhino—had more than 50 releases that were taken into consideration. Anti-patterns come in 13 different varieties. Investigated is the relationship between the presence of anti-patterns and the change- and fault-tendencies. Classes that participate in anti-patterns are found to be more flawed than others.

Hui Liu *et al.,* [41]. 's goal was to find foul odours in the code. The effort required to detect offensive odours has been reduced by a ratio of 17 to 20% according to a newly developed detection approach.

SMURF, an Anti-pattern Detection Approach, was described by Abdou Maiga, Nasir Ali, and others [42]. On three different systems, namely ArgoUML, Xerces, and Azureus, more than 290 experiments have been carried out. Blob, Spaghetti Code, Functional Decomposition, and Swiss Army Knife are four examples of anti-patterns. The accuracy rate of SMURF for detecting anti-patterns in the system is higher than that of DETEX and BDTEX, according to the author.

By integrating the tools required for refactoring with the code smells, Kwankamol Nongpong [43] conducted the investigation. A programme called JcodeCanine had been developed, and it was capable of quickly identifying code smells and letting users know where refactoring was required.

Design patterns and anti-patterns are related, according to Fehmi Jaafar, Yann Gael Gueheneuc, and others [44]. For the evaluation of relationships, three open source systems—ArgoUML, JFreeChart, and XercesJ—had been taken into consideration. It was determined that there is a tenuous relationship between design patterns and anti-patterns. More mistake tendency, which is found in such anti-patterns, was present in the classes.

The open source systems were investigated by Harshpreet Kaur Saberwal *et al.,* [45] in order to find code smells in the classes. For the purpose of smelling things in the system, an empirical model had been created. The work is verified using versions of a real-world project called JfreeChart.

Pandiyavathi and Manochandar [47] proposed some techniques for identifying systemic code smells. A time-saving overview of refactoring techniques had been put forth. The refactoring methods has an algorithm presented for implementation.

In business processes, Francis Palma *et al.,* [48]. 's description of antipattern detection is given. In order to improve the efficiency of BPEL (Business Process Execution Language) operations and identify BP antipatterns, the rule-based approach has been identified. Three example BPEL processes have been used to identify four of the seven BP anti-patterns that have been specified. Francis Palma *et al.,* [49] proposed that the employment of anti-patterns has an impact on the quality of service-based systems. According to data gathered from the SBS FraSCAti, it has been demonstrated that services that appear to be anti-pattern-suspicious require more maintenance than non-pattern-suspicious services.

In software quality assurance, Satwinder Singh and K. S. Kahlon [50] discussed the significance of metrics and threshold values. Software system risk analysis was investigated in comparison to threshold values for the detection of offensive odours. Therefore, incorrect classes could be quickly discovered based on threshold values. The three open source versions of Mozilla Firefox are used to validate the study.

Jiang Dexun, Ma Peijun, and others [51] hypothesised that functionally unrelated classes could cause issues with software maintenance. Therefore, it is necessary to identify and refactor such classes. The author identified Functional over related classes as a foul smell (FRC). To identify the offensive odour, a detection approach was provided. Four open source systems—HSQLDB, Tyrant, ArgoUML, and JfreeChart—were used to validate the work.

3.4 Empirical Detection Techniques

The following empirical detection methods examine the research on code odours and anti-patterns. Different authors have thought about several anti-pattern categories. Here is some of the suggested work: The research on offensive code smells was provided by Mika Mantyla *et al.*, in their article [13]. The report offered a taxonomy to help readers better grasp the odours. The author described various classifications for offensive odours, including Bloaters, Encapsulators, Disposables, Couplers, etc. At a Finnish software company, a survey was conducted, and the results showed a connection between the odours.

By preventing the usage of hazardous antipatterns, Foutse Khomh *et al.*, [24] created the notion of software quality maintenance. The study found that using anti-patterns had an impact on the software's quality.

S. Olbrich *et al.*, [27] took into account the Lucene and Xerces historical data. Classes having the antipatterns Blob and Shotgun Surgery have been found to change more frequently than classes without them.

Min Zhang, Tracy Hall, and colleagues [33] examined more than 300 studies on code foul odours published since 2000 to provide a thorough understanding of the field. The report made it clear that more research is required to fully understand the impact of code smells. It had been determined that the smell of duplicated code was more thoroughly researched than the others.

The effect of anti-patterns on the creation of open source software was noted by Rabia Bashir [39]. The paper described the anti-patterns that can be found in the creation of open source software as well as ways to prevent them.

Various anti-pattern identification methods, including manual (metric-based approach, metric-based heuristics, ad hoc domain specific language), semi-automated (DCPP matrix), and SVM-based methods, were studied by Harvinder Kaur and Puneet Jai Kaur [46]. (DTEX, BTEX, SMURF).

4. CONCLUSION

An extensive literature review has been conducted in this study to highlight the impact of anti-patterns on the source code. Our research identifies many techniques for identifying antipatterns and code smells. It has been determined that the research community has only evaluated its findings in light of internal corporation projects. It has been determined that it is necessary to evaluate the outcomes for various business undertakings. Additionally, it has been determined that few scholars have worked on significant initiatives to identify the anti-patterns. Few have conducted research on a significant number of anti-patterns to reveal the significance of these odours.

As a result, there is a need to identify anti-patterns and the various types of anti-patterns with respect to how they affect classes in object-oriented open source projects. As a result, it will be feasible in the future to identify the often recurring anti-patterns in open source systems and to look at how they affect software metrics. As a result, by foreseeing the problematic classes during the testing phase, the results would help the software industry improve the quality of the software system.

By foreseeing errors and anti-patterns before the product is delivered, it contributes to greater reliability during testing and maintenance phases. Engineers will therefore be interested in the outcomes as well since they can more accurately estimate which classes will be examined. By identifying code smells, the study will help managers and software engineers improve their maintenance tasks.

REFERENCES

1. Webster, B. F. (1995). *Pitfalls of object-oriented development*. M & T Books.
2. Garlan, D., Allen, R., & Ockerbloom, J. (1995). Architectural mismatch: Why reuse is so hard. *IEEE software*, *12*(6), 17-26.
3. Koenig, A. (1998). Patterns and antipatterns. In *The patterns handbooks: techniques, strategies, and applications* (pp. 383-389).
4. Allen, R., & Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *6*(3), 213-249.
5. Brown, W. H., Malveau, R. C., McCormick, H. W. S., & Mowbray, T. J. (1998). *AntiPatterns:*

refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc..
6. Beck, K., Fowler, M., & Beck, G. (1999). Bad smells in code. *Refactoring: Improving the design of existing code*, *1*(1999), 75-88.
7. Travassos, G., Shull, F., Fredericks, M., & Basili, V. R. (1999). Detecting defects in object-oriented designs: using reading techniques to increase software quality. *ACM sigplan notices*, *34*(10), 47-56.
8. Smith, C. U., & Williams, L. G. (2000, September). Software performance antipatterns. In *Proceedings of the 2nd international workshop on Software and performance* (pp. 127-136).
9. Guéhéneuc, Y. G., & Albin-Amiot, H. (2001, July). Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39* (pp. 296-305). IEEE.
10. Simon, F., Steinbruckner, F., & Lewerentz, C. (2001, March). Metrics based refactoring. In *Proceedings fifth european conference on software maintenance and reengineering* (pp. 30-38). IEEE.
11. Van Emden, E., & Moonen, L. (2002, October). Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* (pp. 97-106). IEEE.
12. Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, *28*(1), 4-17.
13. Mantyla, M., Vanhanen, J., & Lassenius, C. (2003, September). A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* (pp. 381-384). IEEE.
14. Marinescu, R. (2004, September). Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* (pp. 350-359). IEEE.
15. Guéhéneuc, Y. G. (2004, November). A systematic study of UML class diagram constituents for their abstract and precise recovery. In *11th Asia-Pacific Software Engineering Conference* (pp. 265-274). IEEE.
16. Munro, M. J. (2005, September). Product metrics for automatic identification of" bad smell" design problems in java source-code. In *11th IEEE International Software Metrics Symposium (METRICS'05)* (pp. 15-15). IEEE.
17. Dashofy, E. M., Hoek, A. V. D., & Taylor, R. N. (2005). A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *14*(2), 199-245.
18. Guéhéneuc, Y. G. (2005, July). Ptidej: Promoting patterns with patterns. In *Proceedings of the 1st ECOOP workshop on Building a System using Patterns. Springer-Verlag* (pp. 1-9).
19. Langelier, G., Sahraoui, H., & Poulin, P. (2005, November). Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (pp. 214-223).
20. Alikacem, E. H., & Sahraoui, H. (2006). Generic metric extraction framework. In *Proceedings of the 16th International Workshop on Software Measurement and Metrik Kongress (IWSM/MetriKon)* (pp. 383-390).
21. Counsell, S., Hassoun, Y., Loizou, G., & Najjar, R. (2006, September). Common refactorings, a dependency graph and some code smells: an empirical study of Java OSS. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* (pp. 288-296).
22. Guéhéneuc, Y. G. (2007, October). Ptidej: A flexible reverse engineering tool suite. In *2007 IEEE International Conference on Software Maintenance* (pp. 529-530). IEEE.
23. Moha, N., Guéhéneuc, Y. G., Meur, A. F. L., Duchien, L., & Tiberghien, A. (2010). From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, *22*, 345-361.
24. Khomh, F., & Guéhéneuc, Y. G. (2008, April). Do design patterns impact software quality positively?. In *2008 12th European conference on software maintenance and reengineering* (pp. 274-278). IEEE.
25. Dhambri, K., Sahraoui, H., & Poulin, P. (2008, April). Visual detection of design anomalies. In *2008 12th European Conference on Software Maintenance and Reengineering* (pp. 279-283). IEEE.
26. Guéhéneuc, Y. G., & Antoniol, G. (2008). Demima: A multilayered approach for design pattern identification. *IEEE transactions on software engineering*, *34*(5), 667-684.
27. Olbrich, S., Cruzes, D. S., Basili, V., & Zazworka, N. (2009, October). The evolution and impact of code smells: A case study of two open source systems. In *2009 3rd international symposium on empirical software engineering and measurement* (pp. 390-400). IEEE.
28. Vaucher, S., Khomh, F., Moha, N., & Guéhéneuc, Y. G. (2009, October). Tracking design smells: Lessons from a study of god classes. In *2009 16th working conference on reverse engineering* (pp. 145-154). IEEE.
29. Bouhours, C., Leblanc, H., & Percebois, C. (2009). Bad smells in design and design patterns. *The Journal of Object Technology*, *8*(3), 43-63.
30. Moha, N., Guéhéneuc, Y. G., Meur, A. F. L., Duchien, L., & Tiberghien, A. (2010). From a

domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, *22*, 345-361.

31. Moha, N., Guéhéneuc, Y. G., Duchien, L., & Le Meur, A. F. (2009). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, *36*(1), 20-36.
32. Hassaine, S., Khomh, F., Guéhéneuc, Y. G., & Hamel, S. (2010, September). IDS: An immune-inspired approach for the detection of software design smells. In *2010 Seventh International Conference on the Quality of Information and Communications Technology* (pp. 343-348). IEEE.
33. Min, Zhang., Tracy, Hall. & Nathan, Baddoo. (2011). Code Bad Smells: a review of current knowledge,‖ *JournalSoftware Maintenance Evol*. Res. Pract., 179– 202.
34. Khomh, F., Vaucher, S., Guéhéneuc, Y. G., & Sahraoui, H. (2011). BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, *84*(4), 559-572.
35. Singh, S., & Kahlon, K. S. (2012). Effectiveness of refactoring metrics model to identify smelly and error prone classes in open source software. *ACM SIGSOFT Software Engineering Notes*, *37*(2), 1-11.
36. Singh, S., & Kahlon, K. S. (2011). Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. *ACM SIGSOFT Software Engineering Notes*, *36*(5), 1-10.
37. Fontana, F. A., Braione, P., & Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, *11*(2), 5-1.
38. Romano, D., Raila, P., Pinzger, M., & Khomh, F. (2012, October). Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In *2012 19th working conference on reverse engineering* (pp. 437-446). IEEE.
39. Kaur, S., & Singh, S. (2015). Influence of anti-patterns on software maintenance: A review. *International Journal of Computer Applications*, *975*, 8887.
40. Khomh, F., Penta, M. D., Guéhéneuc, Y. G., & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, *17*, 243-275.
41. Liu, H., Ma, Z., Shao, W., & Niu, Z. (2011). Schedule of bad smell detection and resolution: A new way to save effort. *IEEE transactions on Software Engineering*, *38*(1), 220-235.
42. Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y. G., & Aimeur, E. (2012, October). Smurf: A svm-based incremental anti-pattern detection approach. In *2012 19th Working Conference on Reverse Engineering* (pp. 466-475). IEEE.
43. Nongpong, K. (2012). *Integrating" Code Smells" Detection with refactoring tool support* (Doctoral dissertation, The University of Wisconsin-Milwaukee).
44. Jaafar, F., Guéhéneuc, Y. G., & Hamel, S. (2014). Analysing anti-patterns static relationships with design patterns. *Electronic Communications of the EASST*, *59*.
45. Saberwal, H. K., Singh, S., & Kaur, S. (2013). Empirical Analysis Of Open Source System For Predicting Smelly Classes,‖ Inter. *Journal of Engineering Research & Technology*, *2*(3), 1-6.
46. Kaur, H., & Kaur, P. J. (2014). A study on detection of anti-patterns in object-oriented systems. *International Journal of Computer Applications*, *93*(5).
47. Pandiyavathi, T. (2014). Usage of optimal restructuring plan in detection of code smells. *arXiv preprint arXiv:1407.1257*.
48. Palma, F., Moha, N., & Guéhéneuc, Y. G. (2013, September). Detection of process antipatterns: A BPEL perspective. In *2013 17th IEEE International Enterprise Distributed Object Computing Conference Workshops* (pp. 173-177). IEEE.
49. Palma, F., An, L., Khomh, F., Moha, N., & Guéhéneuc, Y. G. (2014, November). Investigating the change-proneness of service patterns and antipatterns. In *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications* (pp. 1-8). IEEE.
50. Singh, S., & Kahlon, K. S. (2014). Object oriented software metrics threshold values at quantitative acceptable risk level. *CSI transactions on ICT*, *2*, 191-205.
51. Jiang, D., Ma, P., Su, X., & Wang, T. (2014). Distance metric based divergent change bad smell detection and refactoring scheme analysis. *International Journal of Innovative Computing, Information and Control*, *10*(4), 1519-1531.

Cite This Article: Vadivel G, Er. Abhinash Singla, Er. Sanju Kumari (2021). Influence of Anti-Patterns Detection Techniques on Software Maintenance. *East African Scholars J Eng Comput Sci*, *4*(10), 166-172.