Operationalizing Software Antipatterns for Automated Detection Using Structural Metrics and Supervised Learning

Marcela Mosquera[1,2]

[1]Universidad Politécnica Salesiana
Cuenca - Ecuador

[2]Escuela Politécnica Nacional
Quito - Ecuador

emosquerae@est.ups.edu.ec

Abstract. *This doctoral research aims to develop and validate a machine-learning approach for detecting software antipatterns during the construction phase. The study addresses three central challenges: ambiguous taxonomies and definitions of antipatterns, elevated false-positive rates in detection, and limited reproducibility of learning-based results without standardized protocols. The proposed pipeline consolidates taxonomy and formal definitions, operationalizes antipatterns as logical compositions of code smells and measurable structural properties, and derives observable indicators from structural metrics. Supervised learning with explicit hyperparameter optimization and a metric–label correlation layer are planned to support robustness, interpretability, and traceable evidence.*

1. Introduction

In the software development life cycle (SDLC), antipatterns and code smells have been widely recognized as indicators of structural quality issues [Hübener et al. 2022, Lin et al. 2021]. Although they do not immediately introduce defects, they can reduce the quality and productivity of the project [Liu et al. 2024]. Antipatterns are considered recurring solutions that appear valid but generate negative consequences that outweigh the benefits they are intended to provide [Brown et al. 1998]. Code smells are structures in the code that suggest the possibility of refactoring and could indicate deeper design flaws [Fowler 2018]. Antipatterns and code smells can, in the long term, affect maintainability and security and degrade software performance, thereby increasing technical debt [Agrahari et al. 2023, Kermansaravi et al. 2021].

This paper presents an ongoing PhD research plan on automated detection of software antipatterns. Rather than reporting final empirical results, we focus on (i) a structured methodology to operationalize antipattern definitions into measurable indicators using structural metrics, (ii) a supervised-learning pipeline designed for reproducibility, and (iii) an evaluation plan aimed at reducing false positives and improving cross-project generalization. The expected outcome is a set of reusable operational definitions and an end-to-end framework that can be assessed against established detectors using standard performance metrics and project-wise validation.

2. Research questions and objectives

This research aims to propose and validate a machine-learning-based approach for detecting software antipatterns during the software construction phase.

1. What machine learning techniques have been used so far to detect antipatterns or code smells during the software construction phase, and which ones have demonstrated the greatest effectiveness or acceptance in the scientific literature?
2. What terms and classifications currently exist to describe antipatterns present during software construction, and how can they be unified or harmonized to promote interoperability across studies and tools?
3. What observable indicators in the source code may be related to the presence of antipatterns?
4. Which code characteristics are most relevant for training machine learning models that detect antipatterns during software construction, and how can they be used effectively in the implementation of such models?
5. To what extent does the proposed model outperform or complement existing approaches for antipattern detection?

2.1. General Objective

Propose and validate a machine learning approach for detecting antipatterns during the software construction phase.

2.2. Specific Objectives

1. Identify existing denominations of construction-phase antipatterns and unify equivalent concepts under a consistent terminology.
2. Operationalize antipatterns as a logical composition of code smells and measurable structural properties.
3. Define and extract a set of observable code indicators (metrics/features) suitable for learning-based detection.
4. Train and evaluate learning-based models for antipattern detection using the defined indicators and a reproducible evaluation protocol.
5. Compare the proposed approach against existing detectors to quantify improvements and limitations.

3. Current knowledge of the problem domain

Early foundational work defined antipatterns and code smells conceptually, but without providing explicit detection mechanisms [Mashiach et al. 2023], and several definitions remain broad or vague [Kovačević et al. 2022] in practice. As a result, part of the literature uses antipatterns and smells interchangeably as near-synonyms [Linares-Vásquez et al. 2014, Sobrinho et al. 2021, Ma'ayan et al. 2023] . In contrast, other authors argue for a conceptual distinction: antipatterns correspond to high-level erroneous decisions that may initially appear beneficial, whereas code smells represent low-level implementation symptoms and act as indicators of potential issues [Hallal et al. 2004, Moha and Guéhéneuc 2007, Rivera et al. 2025a, Rivera et al. 2025b].

Antipatterns and smells have been studied across the SDLC, covering multiple research lines. Within the construction phase, prior work has addressed antipatterns in microservices [Taibi et al. 2020, Tighilt et al. 2023], code smells in Android applications [Fawad et al. 2025], and smells in object-oriented systems [Sabir et al. 2019].

Additional empirical studies have explored relationships between smells and antipatterns and software quality-related aspects [Bavota et al. 2015, Saboury et al. 2017, Spadini et al. 2018].

From a detection perspective, both antipattern and smell detection have traditionally relied on source-code metrics combined with conditional rules derived from thresholds to decide whether an entity (e.g., method or class) exhibits a smell. Thresholds are commonly defined either as fixed constants derived from expert knowledge [Tsantalis et al. 2008] or obtained from statistical distributions of metrics within a project [Moha et al. 2010, Marinescu 2004].

Beyond metric-threshold heuristics, the literature also reports approaches based on code evolution and learning-based detection [Sharma and Spinellis 2018, Thakur et al. 2026]. In particular, recent work has increasingly adopted machine learning to learn complex patterns linking code metrics to smell labels [Ho et al. 2025], including techniques such as Support Vector Machines [Maiga et al. 2012], Bayesian Networks [Khomh et al. 2009], and Random Forest [Sarafim et al. 2022].

However, regardless of the selected technique (heuristic or machine learning), fundamental challenges remain and affect reproducibility and comparability. A key issue is the lack of a consensual taxonomy: authors often treat antipatterns, code smells, and bad smells as synonyms due to terminological overlaps, inconsistent boundaries, and incompatible operational criteria. This ambiguity complicates communication and hinders the definition of consistent internal guidelines.

In detection practice, the main limitation is the absence of consistent operational criteria that translate informal descriptions of antipatterns and smells into automated analysis rules. Many detectors still rely on heuristic metrics with ad-hoc thresholds tuned to a language, a domain, or even a specific project, which yields divergent outcomes: the same class may be flagged as a God Class by one tool but ignored by another, or strict rules may generate excessive false positives. Without standardized definitions and thresholds, detection remains subjective and poorly reproducible, and most studies focus on low-granularity smells rather than on antipatterns at the system level [Yu et al. 2023].

These limitations are reflected in recurring gaps reported in tools and empirical studies: (i) unclear taxonomies and definitions that lead to inconsistent detection results and hinder practical adoption and dataset construction, (ii) high false-positive rates that reduce effective tool usage [Yu et al. 2023], and (iii) promising yet still immature learning-based models constrained by multilingual dataset scarcity, class imbalance, and non-standardized data-preparation protocols, which limit reproducibility and generalization [Kumar et al. 2025, Prati et al. 2024, Zhang et al. 2024].

4. Research Methodology

The methodological approach of Design Science Research [Hevner and Chatterjee 2010] will be adopted within the framework of a sequential mixed-methods strategy (qualitative followed by quantitative). This combination enables the organization of existing knowledge about antipatterns and also to construct and empirically validate a machine learning artifact capable of detecting antipatterns during the software construction phase.

- **Systematic Literature Review:** through the review, it will be possible to identify

the machine learning techniques used to date to detect antipatterns and code smells during the construction phase.

- **Taxonomic Consolidation:** extraction of the terms used to refer to antipatterns, identifying a canonical term.
- **Specification of each Antipattern:** through quantifiable indicators in the source code.
- **Design and Implementation of the Machine Learning Model:** based on the CRISP-DM process, code characteristics will be extracted and various algorithms will be evaluated. The final result will be a software artifact capable of detecting antipatterns during the construction phase.
- **Evaluation and Comparative Validation:** through experimentation, the performance of the model will be compared with established detectors.

5. Introduction of the proposed solution

Based on the identified requirements and the need to improve the consistency and practical usefulness of detection, we present a proposed solution aimed at operationalizing antipatterns and enabling their automatic detection based on observable information from the source code. The solution is conceived as an analysis pipeline that integrates conceptual knowledge (definitions and taxonomy) with quantitative evidence (metrics), so that the final result is not only a "detected/not detected" label, but also traceable technical evidence to support the decision.

5.1. Conceptual layer: operationalization of the antipattern

The proposed solution relies on an operational specification in which each antipattern is characterized by observable symptoms and measurable structural properties. This enables informal definitions to be translated into explicit criteria that support consistent labeling, empirical analysis, and comparison across studies. To further strengthen this process, the approach incorporates a taxonomic consolidation step aimed at unifying denominations and reducing terminological overlap, thus improving conceptual consistency and label stability throughout training and evaluation.

Each target antipattern is described through a formal structure composed of four elements: the analysis granularity at which it is observed, the related symptoms or constituent smells associated with its manifestation, the structural metrics used as observable indicators, and the decision criterion adopted for label construction or prediction. This structure is expressed as follows:

$$AP_i = \langle G_i, S_i, M_i, D_i \rangle \tag{1}$$

where G_i denotes the analysis granularity, S_i the set of related symptoms or constituent smells, M_i the associated structural metrics, and D_i the decision criterion adopted for empirical detection. Under this formulation, an antipattern is no longer viewed only as a narrative notion, but as an analyzable construct grounded in explicit observable evidence.

5.2. Measurement layer: structural metrics as observable variables

The measurement layer connects the conceptual definition of an antipattern with the information that can actually be observed in source code. In this study, structural metrics are not used as a generic collection of code measures, but as indicators selected according to the expected manifestation of the antipattern under analysis.

The Blob antipattern, for instance, is often associated with an excessive concentration of responsibilities, low cohesion, and strong dependence on other classes. At the empirical level, these characteristics can be approximated through structural metrics extracted from the analyzed entities. For a Blob antipattern, aspects such as size, complexity, dependence on surrounding classes, and low cohesion can be approximated through metrics extracted from the code. In this case, size can be reflected by Lines of Code (LOC) and Number of Methods (NOM), complexity by Weighted Methods per Class (WMC), dependence by Coupling Between Objects (CBO) and Response for a Class (RFC), and cohesion problems by Lack of Cohesion in Methods (LCOM). These metrics are considered because they offer measurable signals that fit the expected manifestation of the antipattern.

This does not imply that the chosen metrics fully capture the conceptual meaning of the antipattern. Rather, the metric vector is treated as an empirical approximation of its structural manifestation in the dataset. When thresholds are needed for initialization, comparison, or label construction, they will not be assumed to be universal. Instead, they will be derived from prior studies, distribution-based criteria computed from the analyzed systems, and empirical calibration during training and evaluation. Under this view, the measurement layer serves as a reproducible link between the conceptual definition of the antipattern and its representation in the learning process [Mosquera et al. 2025].

5.3. Predictive layer: supervised model with hyperparameter optimization

Regarding the metric vector, the solution incorporates supervised learning as the classification mechanism. The central contribution in this layer is not limited to the use of a particular algorithm, but rather to the explicit incorporation of a hyperparameter optimization scheme, given that the performance of metric-based detection is sensitive to parameter configuration and may affect the reproducibility and comparability of results. As an initial modeling line, the study will consider Logistic Regression, Support Vector Machines, and Random Forest as controlled baselines for metric-based detection. This choice is not intended to define the full set of algorithms to be explored, but rather to provide a comparable starting point for experimentation. Additional models may be incorporated later if warranted by the characteristics of the dataset or by empirical results. To improve robustness and reproducibility, the approach also incorporates hyperparameter search strategies such as grid, random, and hybrid search [Mosquera et al. 2025].

5.4. Evidence layer: interpretive support through metric correlation

To promote practical usefulness and reduce the effect of unexplained alerts, the solution adds an evidence layer based on the correlation between structural metrics and the binary label. This analysis makes it possible to identify a reduced set of metrics showing stronger positive or negative associations with the detected phenomenon, thereby providing analytical support for the model's decisions and facilitating technical inspection of the findings.

In this way, the proposed approach does not only return a detected/not-detected outcome, but also complements prediction with traceable evidence derived from the measurable properties of the analyzed source code.

References

Agrahari, V., Shanbhag, S., Chimalakonda, S., and Rao, A. E. (2023). A catalogue of game-specific anti-patterns based on github and game development stack exchange. *Journal of Systems and Software*, 204:111789.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., and Binkley, D. W. (2015). Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094.

Brown, W. J., Malveau, R. C., McCormick Hays, W. S., and Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley.

Fawad, M., Rasool, G., and Palma, F. (2025). Android source code smells: A systematic literature review. *Software: Practice and Experience*, 55(5):847–882.

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Hallal, H. H., Alikacem, E., Tunney, W. P., Boroday, S., and Petrenko, A. (2004). Antipattern-based detection of deficiencies in java multithreaded software. In *Proceedings - Fourth International Conference on Quality Software (QSIC 2004)*, pages 258–267.

Hevner, A. and Chatterjee, S. (2010). *Design Research in Information Systems*. Springer US.

Ho, A., Bui, A. M. T., Nguyen, P. T., Di Salle, A., and Le, B. (2025). Ensesmells: Deep ensemble and programming language models for automated code smells detection. *Journal of Systems and Software*, 224:112375.

Hübener, T., Chaudron, M. R. V., Luo, Y., Vallen, P., van der Kogel, J., and Liefheid, T. (2022). Automatic anti-pattern detection in microservice architectures based on distributed tracing. In *Proceedings (ACM)*, pages 75–76.

Kermansaravi, Z. A., Rahman, M. S., Khomh, F., Jaafar, F., and Guéhéneuc, Y. G. (2021). Investigating design anti-pattern and design pattern mutations and their change- and fault-proneness. *Empirical Software Engineering*, 26(1):1–47.

Khomh, F., Vaucher, S., Guéhéneuc, Y. G., and Sahraoui, H. (2009). A bayesian approach for the detection of code and design smells. In *Proceedings - International Conference on Quality Software*, pages 305–314.

Kovačević, A., Slivka, J., Vidaković, D., Grujić, K. G., Luburić, N., Prokić, S., and Sladić, G. (2022). Automatic detection of long method and god class code smells through neural source code embeddings. *Expert Systems with Applications*, 204:117607.

Kumar, L., Tummalapalli, S., Murthy, L. B., Misra, S., and Krishna, A. (2025). An empirical analysis on webservice antipattern prediction in different variants of machine learning perspective. *Scientific Reports*, 15(1):1–28.

Lin, T., Fu, X., Chen, F., and Li, L. (2021). A novel approach for code smells detection based on deep learning. In *Proceedings (Springer)*, pages 171–174.

Linares-Vásquez, M., Klock, S., McMillan, C., Sabané, A., Poshyvanyk, D., and Guéhéneuc, Y. G. (2014). Domain matters: Bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. In *22nd International Conference on Program Comprehension (ICPC 2014) - Proceedings*, pages 232–243.

Liu, A., Lefever, J., Han, Y., and Cai, Y. (2024). Prevalence and severity of design anti-patterns in open source programs—a large-scale study. *Information and Software Technology*, 170:107429.

Ma'ayan, D., Maoz, S., and Ringert, J. O. (2023). Anti-patterns (smells) in temporal specifications. In *Proceedings - International Conference on Software Engineering (ICSE-NIER)*, pages 13–18.

Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y. G., and Aimeur, E. (2012). Smurf: A svm-based incremental anti-pattern detection approach. In *Working Conference on Reverse Engineering (WCRE) - Proceedings*, pages 466–475.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 350–359.

Mashiach, T., Sotto-Mayor, B., Kaminka, G., and Kalech, M. (2023). Clean++: Code smells extraction for c++. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR 2023) - Proceedings*, pages 441–445.

Moha, N. and Guéhéneuc, Y. G. (2007). Decor: A tool for the detection of design defects. In *ASE'07 - 2007 ACM/IEEE International Conference on Automated Software Engineering*, pages 527–528.

Moha, N., Guéhéneuc, Y. G., Duchien, L., and Le Meur, A. F. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Mosquera, M., Bojorque, R., and Flores, P. (2025). Understanding code smell detection through hyperparameter optimization and metric correlation analysis. *IEEE Access*, 13:217750–217768.

Prati, A., Vickovic, L., Braovi, M., Singh Yadav, P., Singh Rao, R., Mishra, A., and Gupta, M. (2024). Machine learning-based methods for code smell detection: A survey. *Applied Sciences*, 14(14):6149.

Rivera, R., Flores, P., Anchundia, C., Mosquera, M., Jiménez, A., and Carpio, X. (2025a). Unraveling software antipatterns and smells definitions. In *Proceedings of the 28th Ibero-American Conference on Software Engineering*, pages 388–389.

Rivera, R., Mosquera, M., Flores, P., Anchundia, C. E., Jiménez, A., and Carpio, X. (2025b). Smells vs. software antipatterns: Do definitions matter? In *2025 IEEE/ACIS 23rd International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 169–174.

Sabir, F., Palma, F., Rasool, G., Guéhéneuc, Y. G., and Moha, N. (2019). A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. *Software: Practice and Experience*, 49(1):3–39.

Saboury, A., Musavi, P., Khomh, F., and Antoniol, G. (2017). An empirical study of code smells in javascript projects. In *SANER 2017 - Proceedings*, pages 294–305.

Sarafim, D. S., Delgado, K. V., and Cordeiro, D. (2022). Random forest for code smell detection in javascript. In *Encontro Nacional de Inteligência Artificial e Computacional (ENIAC)*, pages 13–24.

Sharma, T. and Spinellis, D. (2018). A survey on software smells. *Journal of Systems and Software*, 138:158–173.

Sobrinho, E. V. D. P., De Lucia, A., and Maia, M. D. A. (2021). A systematic literature review on bad smells-5 w's: Which, when, what, who, where. *IEEE Transactions on Software Engineering*, 47(1):17–66.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., and Bacchelli, A. (2018). On the relation of test smells to software code quality. In *ICSME 2018 - Proceedings*, pages 1–12.

Taibi, D., Lenarduzzi, V., and Pahl, C. (2020). Microservices anti-patterns: A taxonomy. In *Microservices: Science and Engineering*, pages 111–128. Springer.

Thakur, P. S., Chouhan, S. S., Rathore, S. S., and Parmar, J. (2026). Systematic literature review on software code smell detection approaches. *Journal of Systems and Software*, 235:112784.

Tighilt, R., Abdellatif, M., Trabelsi, I., Madern, L., Moha, N., and Guéhéneuc, Y. G. (2023). On the maintenance support for microservice-based systems through the specification and the detection of microservice antipatterns. *Journal of Systems and Software*, 204:111755.

Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2008). Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 329–331.

Yu, P., Wu, Y., Peng, J., Zhang, J., and Xie, P. (2023). Towards understanding fixes of sonarqube static analysis violations: A large-scale empirical study. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2023) - Proceedings*, pages 569–580.

Zhang, F., Zhang, Z., Keung, J. W., Tang, X., Yang, Z., Yu, X., and Hu, W. (2024). Data preparation for deep learning based code smell detection: A systematic literature review. arXiv.