emerging trends in machine intelligence and big data
ORIENT review is a scientific journal publisher based in Asia, dedicated to disseminating high-quality research across various academic disciplines. With a strong commitment to fostering scholarly advancement and promoting cross-cultural understanding, ORIENT REVIEW has established itself as a reliable platform for researchers, academics, and professionals across the globe.

Code Smells in Software Development Causes, Consequences, and Detection Strategies

Manish Raj Khatri

Department of Computer Science, Amrit Science Campus, Tribhuvan

University, Kathmandu, Nepal

Anushka Thapa

Department of Computer Science, North Campus, Tribhuvan University, Kathmandu, Nepal

abstract

Code smells, indicative of potential design weaknesses in software, have garnered attention due to their subtle yet impactful implications on software maintainability. These symptoms, unlike explicit bugs or errors, hint at deeper architectural or design issues. This study delves into the origins, key manifestations, repercussions, and strategies to pinpoint these smells. Primary causes include inexperienced development, looming deadlines, evolving or ambiguous requirements, a neglect of refactoring, and insufficient code review processes. Such origins can manifest as widely recognized code smells, such as Large Classes, Duplicated Code, and Primitive Obsession. The presence of these patterns, while not immediately problematic, can precipitate several negative outcomes. These include diminished maintainability, an increased propensity for bugs, stunted development processes, limited code reusability, and reduced code comprehensibility. To address these concerns, this research advocates for a multipronged detection approach. Regular manual code reviews are fundamental, augmented by automated static analysis tools like SonarQube and PMD. Metrics such as cyclomatic complexity offer quantitative insights into code health. Moreover, integrating these checks within Continuous Integration systems can preemptively identify and mitigate these smells.

Keywords: Code smells, Software maintainability, Design weaknesses, Static analysis tools, Refactoring

introduction

Code smells refer to patterns or characteristics in source code that signal potential problems, inefficiencies, or complexities. Although they don't necessarily represent errors or bugs, they often indicate areas of code that may require refactoring for improved readability, maintainability, or performance [1]–[3]. Code smells can also serve as early warnings for potential issues that could cause difficulties in the debugging process, the addition of new features, or the scalability of the software. There are various types of code smells such as "Long Method,"

ORIENT review is a scientific journal publisher based in Asia, dedicated to disseminating high-quality research across various academic disciplines. With a strong commitment to fostering scholarly advancement and promoting cross-cultural understanding, ORIENT REVIEW has established itself as a reliable platform for researchers, academics, and professionals across the globe.

where a method does more than it should, or "Data Clumps," where the same set of data appears together in numerous places. Another example is "Feature Envy," which occurs when an object excessively uses the methods of another object, suggesting a potential need for object reorganization [4].

The components that contribute to code smells can be categorized in several dimensions. One such dimension is the scope of the smell: is it localized within a single method (e.g., "Long Method" or "Duplicated Code") or does it pervade an entire class (e.g., "Large Class" or "God Class")? Another dimension is the symptom that the smell evokes. For instance, "Inappropriate Intimacy" between classes indicates a high degree of coupling, whereas "Primitive Obsession" denotes excessive use of primitive types instead of creating small classes to encapsulate related data and behaviors. Complexity is another dimension, encompassing smells like "Switch Statements" or "Conditional Complexity," which make the code harder to follow and maintain [5], [6].

Detecting code smells is often a mix of automated tooling and human expertise. While certain code smells can be identified through automated static code analysis, such as finding duplicated blocks of code or methods with too many lines, other smells like "Speculative Generality" (building functionality that isn't needed yet) require a human reviewer's contextual understanding of the software project's requirements and goals. Tools can flag potential issues, but human judgment is often needed to evaluate the trade-offs of refactoring the smelly code versus leaving it as is, especially considering the potential risks and costs associated with making changes to a codebase.

causes

One of the most common causes for code smells is a lack of experience on the part of the developers. Junior developers, or those new to a particular language or framework, may not be familiar with best practices or the idiosyncrasies that can make or break code quality [7], [8]. They may employ anti-patterns unknowingly, use incorrect data structures, or write convoluted logic that could have been simplified. Often, this is not due to a lack of intelligence or capability, but simply a lack of exposure to better methods. Over time, as these developers gain more experience and knowledge, they may be more likely to recognize these smells themselves, but initial ignorance can sow seeds of trouble in a codebase [9].

Another major factor contributing to code smells is deadline pressures. Developers often work under tight schedules, and when the clock is ticking, it can be tempting to take shortcuts in order to get functionality out the door. This might involve hardcoding values, neglecting to separate concerns, or bypassing necessary validation checks. While these measures can expedite the immediate goal of meeting a deadline, they often lead to

ORIENT review is a scientific journal publisher based in Asia, dedicated to disseminating high-quality research across various academic disciplines. With a strong commitment to fostering scholarly advancement and promoting cross-cultural understanding, ORIENT REVIEW has established itself as a reliable platform for researchers, academics, and professionals across the globe.

technical debt that will need to be paid back later, often with interest, in the form of more time and resources spent on refactoring and debugging [10]. A third cause of code smells is poorly defined requirements. When a project's requirements are ambiguous, incomplete, or constantly changing, it can be difficult for developers to produce high-quality code [11], [12]. They may be forced to make assumptions or create workarounds that result in messy or confusing code. For example, if requirements change mid-way through a project, developers may have to "patch" existing code to fit the new specifications. This often leads to inconsistency and can make the code difficult to read and maintain [13].

Lack of refactoring is also a significant cause of code smells. Software development is a dynamic process. As new features are added, and existing ones are changed or removed, the code needs to evolve. When developers ignore this ongoing need for refactoring, code can easily become a tangled web of dependencies and hacks. In essence, what might have been a clean and effective solution at one point can deteriorate into a code smell if not updated to match the changing context or requirements [14].

Inadequate reviews constitute another reason why suboptimal code may proliferate. Code reviews are a critical aspect of software development, offering a venue for catching mistakes, improving code quality, and sharing knowledge among team members. When code reviews are rushed, infrequent, or superficial, problematic code can go undetected. Developers might miss opportunities to catch redundancies, unnecessary complexities, or other issues that make the code harder to maintain and understand [15].

Code smells often emerge as a result of a variety of factors that range from individual developer experience to systemic issues like deadlines and changing requirements. Addressing the root causes can be challenging but is necessary for maintaining a healthy codebase in the long run. Preventative measures, such as mentorship for less experienced developers, realistic scheduling, clear requirements, ongoing refactoring, and thorough code reviews, can all contribute to reducing the likelihood of code smells appearing in the first place [16], [17].

common code smells

Duplicated code is a pervasive issue in software development. When the same piece of code appears in multiple locations, it often suggests that the logic could be centralized or abstracted for reusability. The peril of duplicated code lies in the increased maintenance cost and the potential for errors. If a developer updates the logic in one location but forgets to make the corresponding change in another, inconsistencies arise. This creates a fertile ground for bugs and makes the codebase harder to manage over time.

ORIENT review is a scientific journal publisher based in Asia, dedicated to disseminating high-quality research across various academic disciplines. With a strong commitment to fostering scholarly advancement and promoting cross-cultural understanding, ORIENT REVIEW has established itself as a reliable platform for researchers, academics, and professionals across the globe.

A Long Parameter List is another code smell that often arises in evolving codebases. Methods with too many parameters can be difficult to understand, and they often indicate that the method is doing too much. Numerous parameters can also make it easy to introduce errors, as it becomes cumbersome to remember the correct order or purpose of each argument. Ideally, methods should operate with fewer arguments, and objects should encapsulate related sets of data that can be passed around together. This enhances readability and reduces the likelihood of mistakes [18].

Feature Envy is an anti-pattern where a class excessively uses methods from another class [19], indicating that the behavior might belong in the latter. Feature envy can compromise the principles of object-oriented programming, particularly encapsulation and cohesion. It can make the code harder to understand and modify, as behavior that logically belongs in one class is spread across multiple locations. This dispersed logic increases the risk of errors during updates or refactoring, as changes in one class may inadvertently affect another [20].

Switch Statements present a unique kind of problem. While sometimes necessary, an overreliance on switch statements often indicates that a codebase could benefit from polymorphism or other object-oriented principles. Extensive use of switch statements makes the code less flexible and harder to extend. For example, adding a new case would require modifying existing switch structures, risking unintended consequences. Using polymorphism allows for more easily extendable and maintainable code, as new behaviors can be added without altering existing code.

Lazy Class refers to classes that don't do enough to justify their existence. Such classes can increase the complexity of a codebase without adding significant value, making it harder to navigate and understand the system as a whole [21]. They often result from incomplete refactoring or premature optimization efforts. Similarly, Data Clumps, where the same group of variables is passed around in multiple places, indicate a lack of structure or abstraction. This can make the code more error-prone, as changes to one part of the data clump will likely necessitate changes in all places where it appears [22].

Finally, Primitive Obsession is a code smell that indicates an overreliance on primitive types like integers and strings to represent complex ideas. For example, using a string to represent a date instead of a dedicated Date object. This can lead to type errors, decreased readability, and lost opportunities for encapsulating behavior. Encouragingly, many of these code smells can be mitigated through proper design patterns, disciplined coding practices, and regular refactoring.

consequences

The impact of code smells on a software project is not just theoretical; it manifests in several tangible ways that can slow down

development and compromise code quality. One of the most significant consequences is decreased maintainability. A codebase riddled with smells is often a labyrinth that is hard to navigate and understand. This makes it more time-consuming to implement even minor changes or to add new features. The convoluted nature of such a codebase could also mean that developers are afraid to make changes for fear of introducing new bugs, essentially freezing the project's ability to evolve effectively [23].

Speaking of bugs, another serious consequence of code smells is the increased likelihood of errors cropping up [24], [25]. Complex, redundant, or poorly structured code provides ample hiding places for bugs. These issues can range from simple, easily detected errors to more insidious bugs that only surface under particular conditions. As these accumulate, the stability of the application becomes compromised, which might not only affect the user experience but could also pose security risks depending on the nature of the project [26].

Slower development speed is another by-product of code smells. When a codebase becomes harder to understand, new features take longer to implement. This can create a feedback loop where deadline pressures lead to more code smells, further reducing development speed. This is particularly detrimental in today's fast-paced software development environments, where the ability to quickly adapt and release new features can be a critical competitive advantage. Any slowdowns can lead to missed market opportunities and reduced profitability.

Reduced code reusability is another issue that arises from code smells. Ideally, a well-designed codebase will allow for pieces of code to be reused in different parts of the application or even in entirely different projects. However, the presence of code smells often means that the code is too specialized, redundant, or entangled to be easily reused. This not only impacts the current project but can also limit a team's future efforts, forcing them to reinvent the wheel each time they need functionality that could have been reused from an earlier project.

Another less obvious but equally impactful consequence is decreased readability, which affects the onboarding of new team members. In a codebase full of smells, the learning curve becomes significantly steeper. New developers may need more time to become productive members of the team, as they have to wrestle with understanding the intricacies of a complex codebase [27], [28]. This can lead to longer development cycles and increased costs in terms of both time and money [29].

In a world where software projects are often complex and always evolving, the consequences of code smells can be severe. They may not always be immediately visible, but over time, they can significantly affect a team's productivity, the quality of the

ORIENT review is a scientific journal publisher based in Asia, dedicated to disseminating high-quality research across various academic disciplines. With a strong commitment to fostering scholarly advancement and promoting cross-cultural understanding, ORIENT REVIEW has established itself as a reliable platform for researchers, academics, and professionals across the globe.

software, and ultimately, the success of the project. Given these potential pitfalls, investing in good coding practices and regular refactoring becomes not just a matter of professional pride but a critical business imperative.

detection strategies

Manual Code Reviews are structured evaluations where developers examine each other's code to identify issues, including code smells. This method incorporates the expertise and judgment of multiple team members, making it possible to catch subtle or context-specific problems that automated tools might overlook. It fosters a culture of collective code ownership and shared responsibility for the codebase's quality. The downside is that manual reviews can be time-consuming and their effectiveness is often dependent on the skill level of the reviewers. Nonetheless, they serve as an invaluable tool for both detecting issues and for educational purposes, helping team members improve their coding skills through peer feedback [30].

Automated Static Analysis Tools are software applications designed to scan a codebase without executing it, looking for specific patterns that are indicative of code smells, security vulnerabilities, or other issues. Tools like SonarQube, PMD, and Checkstyle have pre-configured rules to identify common smells such as duplicated code or long methods. The advantage of these tools is their ability to quickly analyze large volumes of code, providing a first line of defense against deteriorating code quality. While they are efficient, these tools can sometimes generate false positives or lack the nuance to understand context-specific requirements, making human oversight essential [31].

Metrics Analysis is the practice of collecting and evaluating numerical data related to code quality. Various metrics like cyclomatic complexity, which measures the number of independent paths through a block of code, or depth of inheritance, which counts the levels of inheritance in object-oriented languages, can provide objective indicators of code health. These metrics can be tracked over time to measure the impact of changes and to flag potential areas of concern. They serve as a valuable supplement to other review methods, providing quantifiable data that can guide refactoring efforts [32], [33]. However, metrics alone cannot capture every nuance of code quality, and incorrect interpretation can lead to misguided refactoring efforts [34].

Continuous Integration (CI) Systems offer a continuous approach to code smell detection. In a CI pipeline, code is automatically built and tested every time a change is made, providing immediate feedback to developers. By integrating code smell detection into this process, it's possible to catch problematic patterns before they are merged into the main codebase. This real-time feedback enables teams to address issues promptly, reducing the technical debt that can accumulate when problems are left unaddressed [35], [36].

ORIENT review is a scientific journal publisher based in Asia, dedicated to disseminating high-quality research across various academic disciplines. With a strong commitment to fostering scholarly advancement and promoting cross-cultural understanding, ORIENT REVIEW has established itself as a reliable platform for researchers, academics, and professionals across the globe.

While highly effective for catching a wide array of issues early, CI systems require proper configuration and maintenance, and they cannot entirely replace the nuanced understanding that human reviewers bring to the table [37].

Refactoring Sessions are dedicated time periods where the primary objective is to improve the codebase, without the pressure of adding new functionalities or fixing existing bugs. These sessions provide an opportunity to focus solely on eliminating code smells and improving code quality. They are particularly useful for tackling more complex refactoring tasks that require a deeper understanding of the code and its architecture. Through regular refactoring sessions, teams can systematically reduce technical debt, making the codebase easier to work with and less prone to bugs. They also offer an educational experience, allowing less experienced developers to learn better coding practices from their more experienced peers [38].

Detecting code smells is the first crucial step in improving the quality of a software project. Manual code reviews are a traditional but effective approach for identifying problematic code. In this process, team members regularly review each other's code, looking for signs of bad practices or areas that need improvement. Because human judgment is involved, this method can catch subtle issues that automated tools might miss [39], [40]. However, it's also time-consuming and relies on the expertise of the reviewers, which can vary from person to person. Automated Static Analysis Tools like SonarQube, PMD, or Checkstyle offer a more automated approach to identifying code smells. These tools scan the codebase for known patterns that are likely to be problematic. The benefit of using automated tools is that they can quickly analyze large codebases and identify issues with consistency, saving human reviewers valuable time. However, they are often not as nuanced as a human reviewer and might produce false positives or overlook context-specific issues [41]

Metrics Analysis provides a quantitative approach to identifying code smells. By monitoring various code metrics such as cyclomatic complexity, depth of inheritance, or class cohesion, developers can get an objective measure of code quality. High cyclomatic complexity, for example, could indicate that a function is doing too much and might be a candidate for refactoring. This approach is particularly useful for large projects where manual reviews are impractical due to the sheer size of the codebase. Metrics can flag potential problem areas that warrant closer examination, although interpreting these metrics correctly does require expertise [42].

Continuous Integration (CI) Systems provide an ongoing strategy for catching code smells. By integrating smell detection tools into a CI pipeline, you can automatically scan for

ORIENT review is a scientific journal publisher based in Asia, dedicated to disseminating high-quality research across various academic disciplines. With a strong commitment to fostering scholarly advancement and promoting cross-cultural understanding, ORIENT REVIEW has established itself as a reliable platform for researchers, academics, and professionals across the globe.

issues every time code is committed. This ensures that problems are caught early, before they become deeply ingrained in the codebase [43]. CI systems often include a suite of tests that the code must pass before being merged, and adding smell detection to this suite can make the process even more robust. However, CI can only catch issues that it's configured to look for, so it's not a complete substitute for other forms of review [44].

Regularly scheduled Refactoring Sessions serve as another valuable strategy for detecting and eliminating code smells. In these sessions, the sole purpose is to clean up the code, rather than to add new features or fix bugs. This allows developers to focus entirely on improving code quality, making it easier to spot and remove smells. Refactoring sessions also provide an opportunity for less experienced team members to learn from more seasoned developers, promoting better coding practices across the team [45]. Each of these detection strategies has its own strengths and weaknesses, and they are often most effective when used in combination. Manual reviews provide nuance, automated tools offer speed and consistency, metrics offer quantifiable data, CI systems provide ongoing checks, and refactoring sessions allow for focused improvement. By employing a mix of these strategies, teams can significantly improve their ability to detect and eliminate code smells, leading to cleaner, more maintainable codebases [46].

conclusion

Code smells are symptomatic of underlying issues in a software project that, while not breaking the functionality, can lead to problems in readability, maintainability, and scalability. They serve as red flags that warn developers of sections in the code that may require attention or restructuring. Although these indicators are not outright errors, they do highlight weak spots that might make future adjustments or debugging more challenging. Code smells can be as simple as an excessively long function that tries to do too much, or as complex as a class that has assumed too many responsibilities, thereby violating the Single Responsibility Principle, a key tenet of object-oriented programming [47].

The identification of code smells involves several components and methods. First, there's the scope of where the smell occurs. Some smells are localized, affecting only a single method or function. Others might span an entire class or even multiple classes, suggesting architectural issues. The second component is the type of issue that the smell is signaling. For instance, "Duplicated Code" is often a sign that a particular logic has been used in more than one place and may be better suited as a separate method or class. On the other hand, a "Large Class" might indicate that a single class is doing too much and needs to be broken down into smaller, more focused classes. Finally, the severity of the smell is another component to consider. While some smells may be more of an

ORIENT review is a scientific journal publisher based in Asia, dedicated to disseminating high-quality research across various academic disciplines. With a strong commitment to fostering scholarly advancement and promoting cross-cultural understanding, ORIENT REVIEW has established itself as a reliable platform for researchers, academics, and professionals across the globe.

annoyance, others may hint at structural issues that can significantly affect the project's long-term viability [48]. Detecting and resolving code smells is usually a collaborative effort between automated tools and human intervention. Static code analysis tools can automatically scan a codebase and flag potential smells, like methods that are too long or variables that are poorly named. However, not all code smells can be caught by these tools. For example, identifying a "Data Clump," or a set of variables that are always used together, often requires contextual understanding that a tool can't provide. Human reviewers bring this context to the table, employing their knowledge of the project's requirements and potential future changes to determine the real impact of a code smell and the necessity of refactoring [49].

Code smells often arise due to a variety of factors that contribute to less-than-ideal coding practices. One common reason is the lack of experience among junior developers, who may not be familiar with the best practices to avoid certain issues like overuse of primitives or passing around the same group of variables in multiple places. Deadlines also play a significant role; the pressure to deliver on time can lead to hastily written, suboptimal code that leaves behind problematic areas that are hard to maintain or extend. Ambiguous or frequently changing project requirements can further complicate matters, as developers may then produce quick, temporary solutions that eventually become permanent, leading to lingering smells. Another contributing factor is the lack of regular refactoring, which means that code can accumulate issues over time as it evolves. Finally, inadequate reviews or the absence of peer review processes can result in code smells going unnoticed and unaddressed.

Common code smells can vary in nature and complexity but often fall into recognizable patterns that developers should be wary of. A class or method that has grown too large can be a clear indicator of a section of code that is trying to do too much, potentially making it harder to understand and maintain. Duplicated code, where the same code structure is found in more than one location, can make future modifications cumbersome and error-prone. Long parameter lists in methods can make the code confusing and challenging to work with, while classes that excessively use methods from another class, known as Feature Envy, may signal responsibilities that are not well-distributed. Other smells like extensive use of switch statements, classes that do very little, or multiple places where the same group of variables is used can also indicate issues that require attention [50].

The consequences of ignoring code smells can be detrimental over time, affecting various aspects of software development. One immediate impact is the decrease in code maintainability; the more smells present, the harder it becomes to modify, extend or debug the software. This lack of maintainability also

increases the likelihood of bugs creeping into the system, as convoluted or overly complex code sections can become hard to test effectively. Development speed can suffer as well, especially as new team members struggle to understand a codebase riddled with smells, slowing down the addition of new features. Moreover, the presence of code smells can significantly reduce the reusability of code, limiting the potential for components to be used in different parts of the application or even in different projects. Lastly, code readability takes a hit, making it difficult for new or even existing team members to understand the code's logic, thereby steepening the learning curve.

Identifying code smells can be accomplished through various strategies, each with its advantages and limitations. Manual code reviews remain a tried-and-true method, wherein peers review each other's code to spot any potential issues. This approach brings the benefit of human intuition and contextual understanding but can be time-consuming. Automated static analysis tools can scan a codebase for known code smell patterns and are especially useful for quickly identifying common smells like duplicated code or long methods. However, these tools might lack the contextual understanding that a human reviewer would have. Metrics analysis can offer a more quantitative approach, focusing on aspects like cyclomatic complexity or depth of inheritance to flag potential problems [51]–[53]. Continuous Integration systems can also incorporate code smell detection to catch issues before they get merged into the main codebase, thereby acting as a preventative measure. Lastly, scheduling dedicated refactoring sessions allows developers to focus solely on cleaning up the code, which not only helps in identifying existing smells but also in preventing the introduction of new ones.

references

[1] S. Slinger, “Code smell detection in eclipse,” *Delft University of Technology*, 2005.

[2] A. Yamashita and L. Moonen, “Do developers care about code smells? An exploratory survey,” in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 242–251.

[3] A. Hamid, M. Ilyas, and M. Hummayun, “A comparative study on code smell detection tools,” *International Journal of*, 2013.

[4] Y. Huang *et al.*, “Behavior-driven query similarity prediction based on pre-trained language models for e-commerce search,” 2023.

[5] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, “Quantifying the Effect of Code Smells on Maintenance Effort,” *IEEE Trans. Software Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

[6] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, “Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection,” in *2019 IEEE/ACM 27th International*

Conference on Program Comprehension (ICPC), 2019, pp. 93–104.

[7] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proceedings of the 5th international symposium on Software visualization*, Salt Lake City, Utah, USA, 2010, pp. 5–14.

[8] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *J. Syst. Softw.*, vol. 176, p. 110936, Jun. 2021.

[9] H. Vijayakumar, A. Seetharaman, and K. Maddulety, "Impact of AIServiceOps on Organizational Resilience," 2023, pp. 314–319.

[10] J. Gesi, H. Wang, B. Wang, A. Truelove, J. Park, and I. Ahmed, "Out of Time: A Case Study of Using Team and Modification Representation Learning for Improving Bug Report Resolution Time Prediction in Ebay," *Available at SSRN 4571372*, 2023.

[11] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-Smell Detection as a Bilevel Problem," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 1–44, Oct. 2014.

[12] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "A Novel Approach for Code Smell Detection: An Empirical Study," *IEEE Access*, vol. 9, pp. 162869–162883, 2021.

[13] R. S. S. Dittakavi, "Deep Learning-Based Prediction of CPU and Memory Consumption for Cost-Efficient Cloud Resource Allocation," *Sage Science Review of Applied Machine Learning*, vol. 4, no. 1, pp. 45–58, 2021.

[14] F. A. Fontana, J. Dietrich, and B. Walter, "Antipattern and code smell false positives: Preliminary conceptualization and classification," *2016 IEEE 23rd*, 2016.

[15] J. Gesi, X. Shen, Y. Geng, Q. Chen, and I. Ahmed, "Leveraging Feature Bias for Scalable Misprediction Explanation of Machine Learning Models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023.

[16] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "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*, 2009, pp. 390–400.

[17] F. Palomba, "Textual Analysis for Code Smell Detection," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, vol. 2, pp. 769–771.

[18] J. Gesi *et al.*, "Code smells in machine learning systems," *arXiv preprint arXiv:2203.00803*, 2022.

[19] M. Tufano *et al.*, "When and Why Your Code Starts to Smell Bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, vol. 1, pp. 403–414.

[20] H. Vijayakumar, "Revolutionizing Customer Experience with AI: A Path to Increase Revenue Growth Rate," 2023, pp. 1–6.

[21] F. Palomba, G. Bavota, and M. Di Penta, "Do they really smell bad? a study on developers' perception of bad code smells," *2014 IEEE*, 2014.

[22] H. Vijayakumar, "The Impact of AI-Innovations and Private AI-Investment on U.S. Economic Growth: An Empirical Analysis," *Reviews of Contemporary Business Analytics*, vol. 4, no. 1, pp. 14–32, 2021.

[23] H. Vijayakumar, "Impact of AI-Blockchain Adoption on Annual Revenue Growth: An Empirical Analysis of Small and Medium-sized Enterprises in the United States," *International Journal of Business Intelligence and Big Data Analytics*, vol. 4, no. 1, pp. 12–21, 2021.

[24] P. Danphitsanuphan and T. Suwantada, "Code Smell Detecting Tool and Code Smell-Structure Bug Relationship," in *2012 Spring Congress on Engineering and Technology*, 2012, pp. 1–5.

[25] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, "Developer-Driven Code Smell Prioritization," in *Proceedings of the 17th International Conference on Mining Software Repositories*, Seoul, Republic of Korea, 2020, pp. 220–231.

[26] A. Groce *et al.*, "Evaluating and improving static analysis tools via differential mutation analysis," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 207–218.

[27] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, pp. 1–10, Jul. 2018.

[28] A. Yamashita and L. Moonen, "To what extent can maintenance problems be predicted by code smell detection?–An empirical study," *Information and Software Technology*, 2013.

[29] S. Khanna, "Brain Tumor Segmentation Using Deep Transfer Learning Models on The Cancer Genome Atlas (TCGA) Dataset," *Sage Science Review of Applied Machine Learning*, vol. 2, no. 2, pp. 48–56, 2019.

[30] S. Khanna and S. Srivastava, "Patient-Centric Ethical Frameworks for Privacy, Transparency, and Bias Awareness in Deep Learning-Based Medical Systems," *Applied Research in Artificial Intelligence and Cloud Computing*, vol. 3, no. 1, pp. 16–35, 2020.

[31] H. Vijayakumar, "Business Value Impact of AI-Powered Service Operations (AIServiceOps)," *Available at SSRN 4396170*, 2023.

[32] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 612–621.

[33] J. Schumacher, N. Zazworka, and F. Shull, "Building empirical support for automated code smell detection," *Proceedings of the*, 2010.

[34] J. Gesi, J. Li, and I. Ahmed, "An empirical examination of the impact of bias on just-in-time defect prediction,"

in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–12.

[35] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, 2002, pp. 97–106.

[36] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell Intensity Index," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015, pp. 16–24.

[37] S. Khanna, "COMPUTERIZED REASONING AND ITS APPLICATION IN DIFFERENT AREAS," *NATIONAL JOURNAL OF ARTS, COMMERCE & SCIENTIFIC RESEARCH REVIEW*, vol. 4, no. 1, pp. 6–21, 2017.

[38] S. Khanna, "A Review of AI Devices in Cancer Radiology for Breast and Lung Imaging and Diagnosis," *International Journal of Applied Health Care Analytics*, vol. 5, no. 12, pp. 1–15, 2020.

[39] F. Arcelli Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, Jul. 2017.

[40] F. Arcelli Fontana, M. V. Mäntylä, and M. Zanoni, "Comparing and experimenting machine learning techniques for code smell detection," *Empir. Softw. Eng.*, 2016.

[41] R. S. S. Dittakavi, "Evaluating the Efficiency and Limitations of Configuration Strategies in Hybrid Cloud Environments," *International Journal of Intelligent Automation and Computing*, vol. 5, no. 2, pp. 29–45, 2022.

[42] S. Khanna, "EXAMINATION AND PERFORMANCE EVALUATION OF WIRELESS SENSOR NETWORK WITH VARIOUS ROUTING PROTOCOLS," *International Journal of Engineering & Science Research*, vol. 6, no. 12, pp. 285–291, 2016.

[43] Z. Feng, D. Guo, D. Tang, N. Duan, and X. Feng, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv*, 2020.

[44] F. Jirigesi, A. Truelove, and F. Yazdani, "Code Clone Detection Using Representation Learning," 2019.

[45] F. N. U. Jirigesi, "Personalized Web Services Interface Design Using Interactive Computational Search." 2017.

[46] R. S. S. Dittakavi, "Dimensionality Reduction Based Intrusion Detection System in Cloud Computing Environment Using Machine Learning," *International Journal of Information and Cybersecurity*, vol. 6, no. 1, pp. 62–81, 2022.

[47] H. Vijayakumar, "Unlocking Business Value with AI-Driven End User Experience Management (EUEM)," in *2023 5th International Conference on Management Science and Industrial Engineering*, 2023, pp. 129–135.

[48] S. Khanna, "Identifying Privacy Vulnerabilities in Key Stages of Computer Vision, Natural Language

Processing, and Voice Processing Systems," *International Journal of Business Intelligence and Big Data Analytics*, vol. 4, no. 1, pp. 1–11, 2021.

[49] R. S. S. Dittakavi, "An Extensive Exploration of Techniques for Resource and Cost Management in Contemporary Cloud Computing Environments," *Applied Research in Artificial Intelligence and Cloud Computing*, vol. 4, no. 1, pp. 45–61, Feb. 2021.

[50] S. Khanna and S. Srivastava, "AI Governance in Healthcare: Explainability Standards, Safety Protocols, and Human-AI Interactions Dynamics in Contemporary Medical AI Systems," *Empirical Quests for Management Essences*, vol. 1, no. 1, pp. 130–143, 2021.

[51] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, Apr. 2019.

[52] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection," *IEEE Trans. Software Eng.*, pp. 1–1, 2021.

[53] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, "Code Smell Detection: Towards a Machine Learning-Based Approach," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 396–399.