Int. J. Nonlinear Anal. Appl. 14 (2023) 1, 1339–1345
ISSN: 2008-6822 (electronic)
http://dx.doi.org/10.22075/ijnaa.2022.7029

Code smells in software: Review

Farah Khiled AL-Jibory

Ministry of Education, Iraq

(Communicated by Javad Vahidi)

Abstract

A code smell is any display in a program's source code that may signal a broader issue, impeding software development and maintenance. Developers struggle to recognize code smells, which has resulted in several detection tactics, tools, and software analysis tools, commonly referred to as code smell detection approaches. These techniques are generally used to uncover programming abnormalities and poor practices. Code smell detection techniques are typically used to raise developer awareness of the internal quality of a development program, as different tools have different implications.

Keywords: code smell, Developers, detection, tool, techniques
2020 MSC: 03-04

1 Introduction

Code smells are symptoms in a program's source code that may imply a broader problem, inhibiting software maintenance and evolution. Developers struggle to discover code smells, leading to various methodologies and tools [14]. As the phrase implies, it signals a more significant condition that can be detected. The best smell is easy to find yet leads to a tricky problem, such as classes with data but no behaviour. Depending on an organization's design requirements, code smell varies per project and developer [11]. A code smell study can be used to examine and improve the software. Code smells indicate poor code quality, creating developers' maintenance issues. Code smells imply faulty coding, and design choices impair code quality (readability, changeability, defects) [17]. Challenges in maintenance can reflect and explain performance, product quality, and developer motivation. The analysis of maintenance concerns can [15]:

- A better understanding of the relative impact of numerous (product and process-related) variables on maintainability, and eventually
- Development of more precise causal models for maintainability.

If code smell maintenance issues are known, better code augmentation strategies can be made.

Software engineers often refactor Code. It's a way to improve a program's underlying structure without changing its behaviour, according to Martin Fowler [7]. Refactoring code regularly helps programmers make it more understandable, identify flaws, and prepare it for new additions. It enhances program design and product quality. Manual refactoring is possible. There is much literature and software for object-oriented restructuring code. Refactoring involves reverse engineering and agile software development [5].

Email address: csd1b0037@gmail.com (Farah Khiled AL-Jibory)

Received: August 2022 *Accepted:* October 2022

2 Code Smells Concept

Refactoring simplifies and modifies source code. The programmer must decide where to refactor until source code rewriting tools exist. In refactoring-enabled development environments, users must interact to find refactorable Codes and pick the proper refactoring. A code scent identifies a source code problem that refactoring can fix. Syntax problems and compiler warnings are not coded smells; compilers find such. Code smells indicate bad program design or programming techniques that could cause problems if the program is modified, such as ported to a different platform or added additional functionality. This complex Code can be refactored. Not all smells indicate danger. Therefore, we use 'could' It's just a hint. Fowler and Beck say human intuition can't be measured. Detecting destructive Code or design smells and refactoring as needed improves code quality. These smells make system maintenance harder and failure more likely [11].

3 Smell Code Considerations

A stench is a symptom, while restructuring is a treatment. Every smell may imply more than one approach because the same disease may have different remedies. There are a few things to think about [12]:

- **Existence:** Not all code smells must be erased; it depends on the system; in some cases, it's the best approach; Large Class is an example.
- **Cost:** Detecting smells difficult and expensive. As a result, it is better to remove them as quickly as possible when they must be removed. Tool support for code scent detection is essential because many codes smell unnoticed during the programmer's work.
- **Tool:** Detection tools employ a combination of combined metrics, object-oriented metrics, and smell-specific metrics. It must focus on information like class relationships and other smells that are neglected when detecting code smells other than metric combinations. Some detection technologies help with reengineering and maintenance, while others increase software code quality. Most tools can't restructure odours automatically.

3.1 Code Smell Categories

There are many classifications for the smell of the Code, but the following Figure 1 shows the most famous and most important ones.

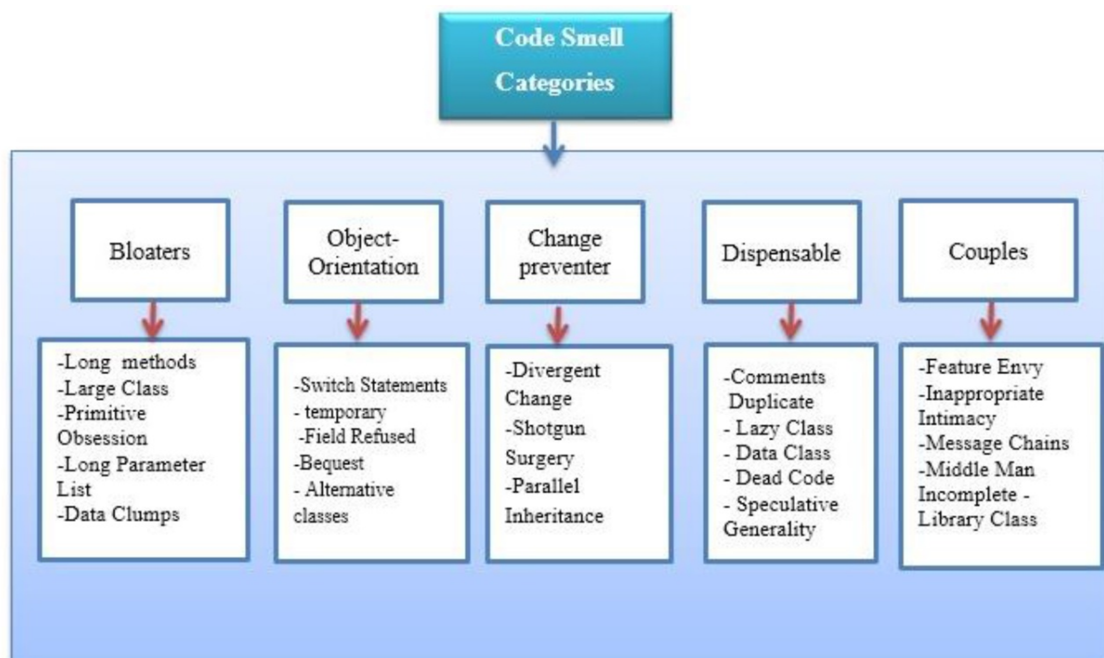

Figure 1: Code Smell Categories [1].

3.1.1 Bloaters

The Means code or classes in this category have been enlarged to the point where they are challenging to work with. These smells don't appear immediately; instead, they build up over time as the program progresses, especially if no one tries to eliminate them. It is divided as follows [16]:

- Long methods
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

The first code smell in this category is a Long Method; this method has an excessive number of Code lines, making it difficult to reuse, update, and understand. The easiest way to deal with the fragrance is to break apart this technique into component pieces. Another code smell to watch out for is Large Class. This arises when a single class has many instances and responsibilities. This class' scent makes reuse and maintenance harder. This problem can be solved by splitting and extracting the class. Third, in this category is Primitive Obsession.

In some cases, small classes should replace primitive types. Instead of tiny objects, simple operations use strings for phone numbers, ranges, and currencies. A long Parameter List is the fourth code smell. This smell makes parameter descriptions harder to understand and use when a method has more than four parameters. Code smells end with data clumps. In other circumstances, such as database connection parameters, the same variable groups occur in different Codes. These aggregates should be sorted.

3.1.2 Object-Orientation

Abusers, all of these smell result from insufficient or poor object-oriented programming. Divided [3]:

- Switch Statements.
- Refused Bequest.
- temporary field.
- alternative classes.

This scent is caused by multiple if statements or a sophisticated switch operator. Temporary fields stink. Many-parameter approaches use temporary fields. The programmer builds class fields with various parameters. Inaccessible algorithm fields. It stinks forever. It improves readability and organization. Finally, Refused the Bequest. This scent develops when a subclass uses a few superclass methods and attributes. This smell prefers delegation over the inheritance. Fourth, different class interfaces. A code smell is two similar classes with different method names [3].

3.1.3 Change Preventers

When you need to update something in one place in your Code and then alter many other things, these smells appear. As a result, the software development process is more complicated and costly. There are three distinct sorts of putrid smells that fall under this category, as described below [8]:

- Divergent Change
- Shotgun Surgery
- Parallel Inheritance

Divergent Change is the group's first stench. Multiple class changes cause a terrible smell. The best way to eliminate this stench is to divide the class. Inheritance should be used to unify classes with similar behaviours. This will organize and reduce duplicate code. Second in this group is Shotgun Surgery. This happens when multiple classes are updated simultaneously. This stench results from dividing a task among multiple classes. Moving class behaviours into a single class eliminates this stench. This will organize, reduce, and maintain the Code. Parallel Inheritance Hierarchies are the third stench. When you construct a class subclass, you get this aroma and realize you need to build another.

3.1.4 Dispensable

Appear When parts of the Code are no longer needed and can be deleted, the Code becomes cleaner, more efficient, and easier to understand. This category includes six different parts, as shown in the following [13]:

- Comments
- Duplicate
- Lazy Class
- Data Class
- Dead Code
- Speculative Generality

Comments are the first sort of foul smell in this group. This smell appears when the application contains a lot of explanatory comments. The Duplicate Code is the second form of foul smell in this category. This smell emerges when the identical Code appears in multiple program portions, resulting in a huge code. This foul smell can be eliminated by encapsulating the duplicated Code in a new method. A worthless class, the Lazy Class, is the third form of foul smell in this category. Every class that is created needs time and effort to comprehend and manage. Eliminating these classes is the most excellent way to eliminate the smell. Data Class is the fourth type of foul smell in this category. This is a class that merely contains fields and rarely has any logic. Fields have getters and setter methods in the Data Class. Dead Code is the seventh sort of foul smell in this group. When a piece of Code is never run, something happens. Speculative Generality is the sixth sort of foul smell in this category. A parameter, field, method, or class isn't used. This foul smell is that Code is occasionally written to support anticipated future features never deployed. As a result, the Code becomes more challenging to comprehend and maintain.

3.1.5 Couplers

The smells in this category either contribute to or exemplify what happens when excessive delegation replaces class coupling. The sections are as follows [10]:

- Feature Envy
- Inappropriate Intimacy
- Message Chains
- Middle Man
- Incomplete Library Class

Feature Envy is the first form of foul smell in this category. When a method accesses the data of another object more than its data, it releases a foul smell. When fields are moved to a Data Class, this happens frequently. If this occurs, data operations should also be shifted to this class. Inappropriate Intimacy is the second form of foul smell in this group. This happens when one class uses another class's internal methods and properties to complete its tasks. Message chains are the third sort of foul smell in this group. When a client requests anything, that object makes the request, which is then fulfilled. Middle Man is the fourth sort of offensive smell in this category. When a class performs one activity while delegating responsibilities to another, its existence becomes redundant. This smell could be the effect of too hastily deleting Message Chains. An unfinished library class is this category's sixth form of unpleasant smell. This happens when libraries can no longer meet their patrons' requirements.

3.1.6 Others

This group contains the Code smells that do not correspond to any of the six categories above due to their unique characteristics. The following essential point concerns subdivisions [10]:

- Comments
- incomplete library class

The following Table 1 shows the smell code summary of types in code smell.

Table 1: Types Summary Of Categories Smell Code [8, 10].

NO	Divisions	Definition	Subdivisions
1	Bloaters	Bloaters are smells that have gotten too large: long Method, Large Class, Primitive Obsession, Long Parameter List, and Data Clumps. New features contribute bloaters to a system's Code.	• Switch Statements. • Temporary field. • Refused Bequest. • Alternative classes.
2	Object-Orientation	These smells go against object-oriented principles. Switch Statements, Temporary Fields, Refused Bequests, Alternative Classes with Different Interfaces, and Parallel Inheritance Hierarchies are included.	• Switch Statements • temporary field • Refused Bequest • alternative classes with
3	Change Preventers	Change preventers make maintenance harder, owing to the current implementation. This odour is present when classes or methods are responsible for multiple features or functions. The Single Responsibility Principle states that a class that provides database connectivity functionality should only change if the database type changes. Divergent Change and Shotgun Surgery are here.	• Divergent Change • Shotgun Surgery • Parallel Inheritance
4	Dispensable	This stench indicates applications with unnecessary Code. This fragrance is used every day in responsible classes.	• Feature Envy • Inappropriate Intimacy • Message Chains • Middle Man • Incomplete Library Class
5	Couplers	The smells in this category either contribute to or exemplify what happens when the class coupling is replaced by excessive delegation.	• Feature Envy • Inappropriate Intimacy • Message Chains • Middle Man • Incomplete Library Class

3.2 Detection Tool in Smell Code

Software analysis tools, often known as code smell detection techniques, are primarily used to identify programming abnormalities and bad practices. However, because different tools have varying meanings, code scent detection techniques are generally used to raise a developer's awareness of the internal quality of a development program [1]. A developer or code scent detection tools can detect Code smells manual.

Manual code smell detection is time-consuming and error-prone. As a result, code scent detection tools may detect Code smells automatically or semi-automatically, assisting the developer in searching for (smelly) entities [6]. The following Table 2 shows the most important tools used in detecting smell and description in detail.

Table 2: Detection Tools Of Code Smell [12].

No.	Tool	Domain
1	Squale	Vague documentation
2	Soot	Unresolved errors in Code and, therefore, could not be executed
3	Infer	Unable to examine its versions
4	SourceMeter	Not available for download
5	SonarQube	Selected
6	WekaNose	Unresolved errors in Code and, therefore, could not be executed
7	JDeodorant	Selected
8	iPlasma	Vague documentation and no version history
9	PMD	Selected
10	Checkstyle	Selected
11	DECOR	Not available for download
12	UCDetector	Selected
13	SpotBugs	Selected
14	FindBugs	Selected

3.3 Topic Related to The Code Smell

Code smells have primarily been discussed by the software engineering community and industry practitioners. Several techniques help detect Code smells in programs written in various languages [30]. As stated in the preceding section. Similarly, there are numerous bug problem-tracking programs available. Bugs and vice versa influence the occurrence of the smell code. When it comes to detecting code smells, most tools use one of the six strategies listed below [27]:

- manual
- symptom-based detection
- metric-based
- probabilistic
- search-based
- cooperative-based approaches.

Many strategies use software metrics to detect code smells, some of which are metrics taken from third-party tools and then applied to threshold values. When used frequently, this method of automatic detection can be appealing. Though numerous research has looked into code smells, only a few have looked into the impact of Code smells on the occurrence of software problems [2, 9]. Whereas code smells don't always defect, it is also not technically incorrect and doesn't stop a program from working. Instead, they point to design flaws that could slow development or increase the likelihood of future faults or failures [4]. Table 3 shows the definition of two topic.

Table 3: Description Of Two Topic In Smell Conception [28]

Topic	Description
Bug	An issue that represents a flaw in the Code. It will, most likely at the worst possible time, if it hasn't already. This needs to be addressed.
foul smell	There is a problem with maintainability in the Code. If the Code is left alone, maintainers will have a more challenging time than they should. In the worst-case scenario, they'll be so perplexed by the existing condition of the Code that they'll introduce new mistakes as they work.
Commits	Commits contain a tremendous amount of metadata, including the contents and message, such as the author, timestamp, and more. These commits are timestamped snapshots of the repository as a whole. Should regularly commit, depending on logical units of Change. Commits should gradually explain the repository's evolution and how it arrived at its current state.

4 Conclusion

Numerous ways employ software metrics to detect Code smells, some of which are derived from third-party technologies and applied to threshold values. This method of automatic detection can be enticing when employed regularly. Although various studies have examined code smell, few have examined the impact of code smell on the occurrence of software bugs. This paper presents a detection tool and what code smell means in software engineering.

References

[1] A. Al-Shaaby, H. Aljamaan and M. Alshayeb, *Bad smell detection using machine learning techniques: A systematic literature review*, Arab. J. Sci. Eng. *45* (2020), no. 4, 2341–2369.

[2] A.S. Cairo and G.D.F. Carneiro, *The impact of code smells on software bugs: A systematic literature review*, MDPI **3** (2018), 1–22.

[3] A. Chatzigeorgiou and A. Manakos, *Investigating the evolution of bad smells in object-oriented code*, Proc.-7th Int. Conf. Qual. Inf. Commun. Technol. QUATIC **2010** (2010), 106–115.

[4] P. Danphitsanuphan and T. Suwantada, *Code smell detecting tool and code smell-structure bug relationship*, Spring World Congr. Eng. Technol. SCET 2012-Proc. **2012** (2012), 1–5.

[5] E. Fernandes, J. Oliveira, G. Vale, T. Paiva and E. Figueiredo, *A review-based comparative study of bad smell detection tools*, Proc. 20th Int. Conf. Eval. Assess. Software Engin., 2016, pp. 1–12.

[6] F.A. Fontana, V. Ferme, M. Zanoni and R. Roveda, *Towards a prioritization of code debt: A code smell Intensity Index*, IEEE 7th Int. Work. Manag. Tech. Debt, MTD 2015-Proc, 2015, pp. 16–24.

[7] A. Hamid, M. Ilyas, M. Hummayun and A. Nawaz, *A comparative study on code smell detection tools*, Int. J. Adv. Sci. Technol. **60** (2013), 25–32.

[8] M.S. Haque, J. Carver and T. Atkison, *Causes, impacts, and detection approaches of code smell: A survey*, Proc. ACMSE 2018 Conf. 2018, no. 1.

[9] S. Jain and A. Saha, *Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection*, Sci. Comput. Program **212** (2021), 102713.

[10] A. Kaur and G. Dhiman, *A review on search-based tools and techniques to identify bad code smells in object-oriented systems*, N. Yadav, A. Yadav, J. Bansal, K. Deep and J. Kim (eds), Harmony search and nature inspired optimization algorithms, Adv. Intell. Syst. Comput. Springer, Singapore, 2019.

[11] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu and L. Zhang, *Deep learning based code smell detection*, IEEE Trans. Softw. Eng. **47** (2021), no. 9, 1811–1837.

[12] A. Löwe, S. Bampovits and F. Palma, *Do software code smell checkers smell themselves? A self reflection*, http://lnu.diva-portal.org/smash/record.jsf?pid=diva2:1458340, (2020).

[13] M.V. Mäntylä and C. Lassenius, *Subjective evaluation of software evolvability using code smells: An empirical study*, Empir. Software Engin. **11** (2006), no. 3, 395–431.

[14] T. Paiva, A. Damasceno, E. Figueiredo and C. Sant'Anna, *On the evaluation of code smells and detection tools*, J. Softw. Eng. Res. Dev. **5** (2017), no. 1, 1–28.

[15] D. Sahin, M. Kessentini, S. Bechikh and K. Deb, *Code-smell detection as a bilevel problem*, ACM Trans. Softw. Eng. Method. **24** (2014), no. 1.

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

[17] A. Yamashita and L. Moonen, *To what extent can maintenance problems be predicted by code smell detection? –An empirical study*, Inf. Software Technol. **55** (2013), no. 12, 2223–2242.