**AN EMPIRICAL INVESTIGATION OF BAD SMELLS IN CODE ON MAINTENANCE EFFORT**

**Abstract-**

In this article, we propose automated code scent detection and reworking tools for determining code quality by identifying code smells and raising code quality using refactoring techniques. Refactoring is a procedure for reorganising or enhancing a piece of software's internal structure without altering its functionality. Declarative programming is used in conjunction with object-oriented software metrics to accomplish this goal. Code smell detection is based on a number of facts and rules. This method was employed to identify offensive odours in case studies based on oops. The three categories (Hi, Low, and Medium) of the code maintainability index indicate the level of source code quality (Low, Hi, Medium).

**Keywords-**

Code smell detection, code quality, maintainability index, refactoring, and object-oriented metrics.

**Introduction-**

The phrase "code smell" [1] was initially used by Kent Beck2 to describe formatting issues in the software source code that are looked at by seasoned programmers. The following quote is from Kent Beck: "A code smell is a clue that there is some issue in somewhere in your code [1]."

Although the built structure might not currently be having a significant influence (in terms of bugs and failures [2]), this error must be having a significant impact on the system's fully built code source, which lowers the quality factors. Code smells can complicate a system's design, making it challenging to comprehend and maintain. Additionally, the existence of code smells results in the emergence of numerous new flaws [3] such as significant development issues like poor architectural decisions or even poor management techniques.

Our primary focus in this project is on code smell and the created automatic methods for detecting it [7]. Code smells are primarily described as a method [5] for examining structural aspects of software that indicate a problem with the code or the design (format). Additionally, it makes software challenging to understand and maintain. Fowler [5] introduced this idea after deriving 22 different categories of scents. Later, other odours were discovered by other authors (like Mäntylä [MVL04]), and new ones might be created. Code smells are solely associated with the procedure of refactoring software in order to assess its internal quality. In order to determine if the appearance of a scent is related to a fall in the system's source code structure or not, developers must address any foul smells they uncover in their code. If scent is determined to be detrimental to code, accurate refactoring is chosen to eliminate it. As we are all aware, difficulties are similar to the signs of any potential disease, and just as there are numerous procedures used to treat these diseases, the same rules are applied to the concepts of code smell and refactoring [9]. Software code [2] source can be exceedingly challenging to manage because it changes over time. Additionally, the designs are becoming increasingly difficult to understand, making it crucial to periodically or altogether restructure the code. The software must act in the same manner as it did before the reorganisation when the code is changed [9], which is a highly important consideration. Refactoring is the term for programme modifications that preserve semantics. Refactoring [7] is not seen as a primary front during the development process due to numerous disadvantages, including the possibility of inaccuracy and time wastage. It is obvious that not all developers can benefit from refactoring because it neither adds new functionality to the software nor enhances any inherent aspects of the product. Although it has been established that refactoring does not improve the internal qualities of software source code, namely reusability, maintainability, and readability [18], it does have an important function to aid in their improvement. Even if developers receive excellent scores in the refactoring study, it might be quite difficult for them to determine which parts of the source code can benefit from refactoring.

Software refactoring, according to an author by the name of Karnam Sreenu, is a process that alters various software artefacts without changing the software's exterior characteristics [2]. Additionally, it defined object-oriented metrics that could be computed to identify offensive odours. There are a few additional odd methods for detecting foul odours in addition to metric-based ones.

Using a Bayesian Belief Network, Khomh suggested a unique method for sniffing out offensive odours [5]. The approach calculates the effect likelihood rather than simply informing the user whether or not a code component is affected by a bad smell.

Software engineers can manually define the specification for bad smells detection using the taxonomy and vocabulary, and, if necessary, the context of the analysed systems, according to Moha's Décor proposal, which uses a Domain-Specific Language (DSL) [8] for specifying smells at a high abstraction level.

Marija Katic presents the key definitions and terms relating to how testing is directly related to software redesign. This article provides a quick overview of the methodology and the software redesign process [10]. Testing is necessary to make sure that the behaviour has not changed, even though one would argue that, for instance, redesigning the source code falls under the implementation phase.

Code smells are structural features of software that may point to a code or design issue that makes software difficult to evolve and maintain and may cause restructuring of code, according to Francesca Arcelli, Fontana Pietro Braione, and Marco Zanonia [6]. When the complexity of the code gets too large for manual examination, recent research is actively developing automatic detection techniques to assist humans in smelling odours.

D. B. This paper examines how coupling and cohesion [7] qualities are affected by refactoring and discusses how to spot refactoring opportunities that enhance these characteristics. On the other hand, coupling and cohesiveness are quality traits that are widely acknowledged as being among the most likely quantitative markers for software maintainability.

In this study, Pessoa introduces an Eclipse plug-in that automates a method for identifying and rating code smells in Java source code [8]. The use of a dynamic statistical procedure that depends on expert knowledge and can theoretically be applied to any smell sets the Smell Checker tool's detection method apart from all other known ideas.

In conclusion, the majority of methodologies described in the literature rely primarily on structural data taken from a single software system snapshot. As far as we are aware, not much study has been done on the use of evolutionary data for the detection of code smells.

**Literature Survey-**

In this study, Pessoa introduces an Eclipse plug-in that automates a method for identifying and rating code smells in Java source code [8]. The use of a dynamic statistical procedure that depends on expert knowledge and can theoretically be applied to any smell sets the Smell Checker tool's detection method apart from all other known ideas.

In conclusion, the majority of methodologies described in the literature rely primarily on structural data taken from a single software system snapshot. As far as we are aware, not much study has been done on the use of evolutionary data for the detection of code smells.

**Bad Smells In The Code-**

Fowler and Beck provide the earliest explanation of the idea of foul odour. Any foul odour in the source code merely indicates that there must be a significant flaw in the system's code. Any layout problem that makes it difficult to collect and maintain the source code of the system is referred to as a "foul smell." Basically, a terrible odour in source code indicates the presence of a defect or error of some type. A bad odour can also refer to any kind of technical issue. The phrase became more common once it was used in Refactoring. Refactoring can be used to address some of the traits that bad code demonstrates. These are referred to as "Bad Smells," and they include the following:

* A long parameter list
* A long method
* A large class,
* A lazy class
* A switch statement
* Comment lines.

**A Long Parameter List-**

Long parameter lists can be confusing. A method that takes a lot of parameters has a long parameter list.

**A Long Method-**

When a method is too long, it refers to any method with a lot of coding lines.

**A Large Class-**

Classes with numerous instance variables and a lot of lines of code. Large classes that are occasionally used can also suffer from code duplication.

**A Lazy Class-**

Classes with no methods or classes that perform little work.

**A Switch Statement-**

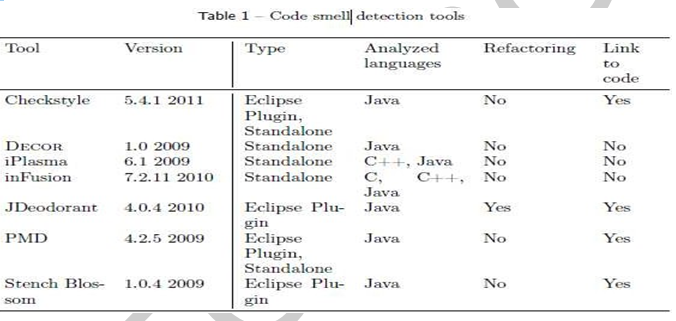
Switch statements could result in duplicate data. Similar switch statements are dispersed throughout the programme in various locations. Perhaps classes and polymorphism would be better choices.

**Comment Lines-**

If there are more comments in the code than there are lines of code.

**Tools For The Automatic Detection Of Code Smells-**

Although some of the tools mentioned above (JDeodorant is the only one that offers refactoring options) are still unable to perform code refactoring automatically in response to detected smells, modern integrated development environments (IDEs) are generally capable of doing so even when they require user input. It would be ideal for scent detector tools to assist the user in understanding at least the source of the smells, and, as stressed in the Murphy-Hill and Black [MHB10] standards, the tools shouldn't overload the programmer with smell information in the event of smell proliferation.



**Checkstyle** –

To assist programmers in creating Java code that adheres to a coding standard, Checkstyle2 has been created. It may identify the code smells known as Large Class, Long Method, Long Parameter List, and Duplicated Code.

**DECOR** –

An technique that enables the specification and automatic identification of code and design smells was defined by Moha et al. [MGDM10, MGM+10] (also called anti patterns). They created a proprietary language to specify six code smells, used templates to automatically create their detection algorithms, and evaluated the algorithms' precision and recall. Their Decor platform for software analysis uses this methodology. 3 The component created for code smell detection is referred to as Decor in the text below.

**InFusion**-

The most recent commercial iteration of iPlasma is inFusion4. More than 20 design errors and code smells, such as duplicate code, classes that violate encapsulation, such as the Data Class and God Class, tightly connected methods and classes, and poorly structured class hierarchies, are all detectable by inFusion.

**iPlasma** –

Model extraction through high-level metrics-based analysis are all supported by this tool [MMM+05], which is an integrated platform for quality assessment of object-oriented systems. 5 The authors categorise identification disharmonies, collaboration disharmonies, and classification disharmonies as code disharmonies that iPlasma is capable of detecting. [LM06] contains a thorough discussion of these disharmonies. Several code smells, such as God Class, Feature Envy, Refused Parent Bequest, and Duplicated Code (also known as Significant Duplication), are regarded as disharmonies.

**JDeodorant-**

JDeodorant [TC11] is an Eclipse plugin that detects code smells in Java projects such as Feature Envy, God Classes, Long Methods, and Switch Statements (in its Type Checking variation). 6 By identifying potential refactoring transformations that address the identified problems, ranking them according to their impact on the design, presenting them to the developer, and automatically applying the one the developer chooses, the tool helps the user decide on an appropriate sequence of refactoring applications.

**PMD-**

When PMD7 scans Java source code, it looks for things like duplicated or dead code, empty try/catch/finally/switch statements, unused local variables, and unused local arguments. Large Class, Long Method, Long Parameter List, and Duplicated Code smells are all detectable by PMD, and the user may define the threshold values for the metrics that are being exploited.

**Refactoring-**

Refactoring is similar to a fresh research chapter and is less well defined than other areas. There are numerous justifications for refactoring, but particular ones are still pending. The following definitions are mentioned:

* Refactoring is the process of taking an object's goal and reorganising it in various ways to make the design more adaptable and reusable. There are numerous justifications for doing this, with competency and maintainability likely being the most important.
* Refactoring encoding code entails rewriting it and "cleaning it up."
* Refactoring is the process of moving functional units around within your software**.**
* Receiving each piece of functionality to live in precisely one place in the software is the fundamental goal of refactoring.

**Refactoring Process-**

The refactoring process can be broken down into several parts, as seen below:

1. Identify and categorise the software coding areas that require reworking.

2. Choose which refactoring should be applied to which system code part.

3. Confirmation that the restructuring used preserves behaviour

Apply the refactoring [7] in step 4.

5. Examine how the reworking has affected the software's or the process's unique quality.

6. Keep the other software artefacts and the refectory programme code consistent.

**Metrics-**

Since everyone is aware that there are a virtually infinite number of quality metrics for object-oriented programmes (i.e., one alone describes more than 200 complexity metrics), I will focus only on those programme metrics that are frequently used, such as the number of methods, cyclomatic complexity, number of children, coupling between objects, response for a class, and lack of cohesion among methods.

**These Metrics' definitions are:**

• Number of Methods: This tells us the total number of methods in a class. It is sometimes described as a measure of a class's functional size.

• Cyclomatic Complexity: This metric is intended to quantify how many different ways an algorithm can be implemented. It serves as a gauge of a program's logical complexity and is based on the quantity of flow graph edges and nodes.

• Number of Children: This enables us to identify a class's direct descendants. It serves as a gauge of the class's generality.

• Lines of Code: This tells us how many lines of code there are in a class or a function. It serves as a measure of the coding's structural size.

• Comment lines: This information reveals how many lines of code are not used during execution.

• Coupling between Objects is a metric for the quantity of class-wide partnerships. It serves as a gauge of how sophisticated the conceptual functionality used in the class is.

• Response for a Class: It is used to determine the number of methods that are defined and inherited by a class, as well as the number of methods that are called by these methods from other classes. It serves as a sign of the class's susceptibility to change propagation.

• The inverse cohesion measure is used to indicate a lack of coherence among methods (high value means low cohesion). We utilise LCOM1 as defined by Henderson-Sellers, one of the several LCOM versions, which counts the number of method pairs in a class that don't share any attribute references. It serves as a gauge of how effectively the class's methods mesh.

**Conclusion And Future Work-**

This paper outlines the refactoring filtering criteria that programmers (developers) can use to identify applicants for reworking that will lessen or eliminate Code Smells. At the basic level of the programme, these circumstances are examined. They assist programmers in determining which portion of a software has to be refactored. Additionally, we set up the rules.

Can aid in choosing the refactoring method that produces the most maintainability. The experiment's findings imply that our cutting-edge method can categorise refactoring that is more effective than refactoring literature. However, our sample only includes two refactorings (extracts method and restore temp with query). We intend to do an additional experiment that examines six refactoring for shortening long techniques in our upcoming work.

References:

C. Y. Baldwin and K. B. Clark. Design Rules, Vol. 1: The Power of

Modularity. MIT Press, 2000.

[2] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice.

Addison-Wesley, 3rd edition, 2012.

[3] Y. Cai and K. J. Sullivan. Modularity analysis of logical design mod- els. In Proc. 21st IEEE/ACM International Conference on Automated Software Engineering, pages 91–102, Sept. 2006.

[4] Y. Cai, H. Wong, S. Wong, and L. Wang. Leveraging design rules to improve software architecture recovery. In Proc. 9th International ACM Sigsoft Conference on the Quality of Software Architectures, pages 133–142, June 2013.

[5] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. IEEE Transactions on Software Engineering, 35(6):864–878, July 2009.

[6] D. Falessi, G. Cantone, R. Kazman, and P. Kruchten. Decision-making techniques for software architecture design: A comparative survey. ACM Computing Surveys, 43(4):1–28, Oct. 2011.

[7] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, July 1999.

[8] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In Proc. 14th IEEE International Conference on Software Maintenance, pages 190–197, Nov. 1998.