GROUP ASSIGNMENT 2

Object Oriented Development Course –
FadiWedyan

Abstract

Effect of code bad smells on modularity

Ushaswini Renukunta Malreddy Pavani Reddy

https://github.com/PavaniMalreddy/OODgroupAssign-2.git

Section 1: Goal, Questions, and Metrics (GQM) Approach

Objective

The primary objective of this study is to empirically evaluate the impact of code bad smells on the testability of a software system. We aim to identify the correlation between the prevalence of code smells and the difficulty of writing, maintaining, and running effective tests.

Questions

Q1: What type and how frequent are bad smells in the system?

By identifying the types of smells that appear frequently, we can have a better understanding of the system's weaknesses, which can lead to testability issues.

Q2: How does the presence of these code smells affect the complexity of the system?

Bad smells often result in increased complexity, which can inhibit testability. We want to quantify this impact.

Q3: How does the presence of code smells influence the time taken to write and maintain tests?

Testability is not only about the capability to write tests but also about how straightforward and time-efficient it is to write them.

Q4: Does the removal or refactoring of bad smells lead to an increase in testability?

If the reduction of bad smells can result in improved testability, it would provide substantial evidence for prioritizing code quality in development efforts.

Metrics

To answer the above questions, we will need to collect specific data. For each bad smell identified, we will measure:

- **Frequency**: The number of instances of each bad smell in the system. This will help answer Q1.

- **Complexity Metrics**: Such as complexity or depth of inheritance. These metrics can be used to quantify the effect of bad smells on system complexity, helping answer Q2.

- **Testing Time:** The amount of time taken to write, maintain, and run tests. This data will be used to address Q3.

- **Post-refactoring Testability:** After bad smells are removed or code is refactored, we will measure changes in complexity and testing time. This will provide data to answer Q4.

Through this empirical study, we aim to establish a clear correlation between code smells and testability, providing insights for future development and testing efforts.

The Chidamber and Kemerer (CK) suite is a widely used collection of software metrics that helps to measure various aspects of software quality. The CK metrics suite includes six metrics, as follows:

- **Weighted Methods per Class (WMC):** The sum of the complexities of all methods in a class. It gives an idea about how much time and effort is required to develop and maintain the class.

- **Depth of Inheritance Tree (DIT):** The maximum length from the node to the root of the tree. This metric gives an idea about the potential reuse of inherited methods and the complexity added due to inheritance.

- **Number of Children (NOC):** The number of immediate successors or children of a class in the class hierarchy. The greater the number of children, the greater the likelihood of improper abstraction of the parent class.

- **Coupling Between Object Classes (CBO):** It counts the number of other classes to which a class is coupled. High coupling can complicate testing, as changes in one class may affect others.

- **Response for a Class (RFC):** It counts the number of methods that can potentially be executed in response to a message received by an object of that class. A high value indicates a high level of complexity, reducing the readability of the code.

- **Lack of Cohesion in Methods (LCOM):** Measures how well the methods of a class are related to each other. High cohesion (low LCOM) tends to be associated with a well-designed class.

So, based on the CK metrics suite, you could adapt the GQM approach as follows:

Questions:

Q1: What are the values of CK metrics in the system? And how are these values distributed across classes?

Q2: How do the CK metric values correlate with the frequency and severity of code smells?

Q3: How do code smells affect the CK metric values?

Q4: How do changes in CK metrics values, due to refactoring of code smells, influence the testability of the system?

Modularity is an important attribute in software design, as it allows for increased manageability, reusability, and maintainability of the software. The following Chidamber and Kemerer (CK) metrics can play a crucial role in measuring the modularity of a software system:

Coupling Between Object Classes (CBO): Coupling refers to the degree to which one class knows about another class. If a class has high coupling, it means that it interacts with a large number of other classes. High coupling can negatively impact modularity because changes in one class might affect many other classes. Therefore, lower values of CBO generally indicate better modularity.

Lack of Cohesion in Methods (LCOM): Cohesion measures the degree to which the responsibilities of a single class are interconnected. A class with high cohesion means that its methods are well-related and it is focused on what it should be doing. High cohesion (low LCOM) tends to indicate good modularity, as it means that each class has a single, well-defined role.

Number of Children (NOC) and Depth of Inheritance Tree (DIT): These metrics are related to inheritance, which is one way of achieving modularity in object-oriented design. However, deep inheritance trees (high DIT) and broad inheritance hierarchies (high NOC) can lead to higher complexity and reduced modularity.

Of these metrics, CBO and LCOM are probably the most directly related to modularity, as they measure the degree of interaction between classes and the single responsibility principle, both of which are key aspects of modularity. However, all of these metrics provide valuable insights into the design of a software system and can be useful for assessing its modularity.

Tool Used:-

The PMD tool was used as a static code analysis tool for the empirical research of the impact of code foul odours on modularity. PMD is an open-source tool for finding possible problems, such as "code smells," in Java projects by applying a set of preset criteria. It analyses the source code statically to find places where the quality of the code might be enhanced.

Modularity is only one of many characteristics of code quality that may be examined by PMD thanks to its flexible rulesets. High coupling, poor cohesion, and convention violations are only some of the problems that may be uncovered by analysing the code's metrics and patterns. The study's goal was to detect and quantify the prevalence of code foul odours that might be indicative of decreased modularity by running PMD on the applications in question.

PMD's output revealed useful information about the prevalence and severity of code foul smells in the examined algorithms. We evaluated the effect of code foul odours on modularity by analysing these results and correlating them with modularity measures. Through this study, we were able to identify a correlation between certain "code smells" and the degree to which the software programs demonstrated modularity.

PMD not only has the capacity to identify code foul smells, but also to automatically apply code refactorings to fix the problems that have been found. In the context of an empirical study of modularity, this feature is invaluable. The research was able to detect code foul smells that affected modularity and conduct automatic refactorings to increase the modularity of the programmes under examination by using PMD's refactoring capabilities. This made it possible to examine in depth the connection between code foul odours, refactoring efficiency, and modularity. Applying refactorings inside the PMD tool itself was a simplified and effective method of increasing modularity in the examined software.

PMD: An Analyzer for Programming Source Code

Cite:- https://pmd.github.io/

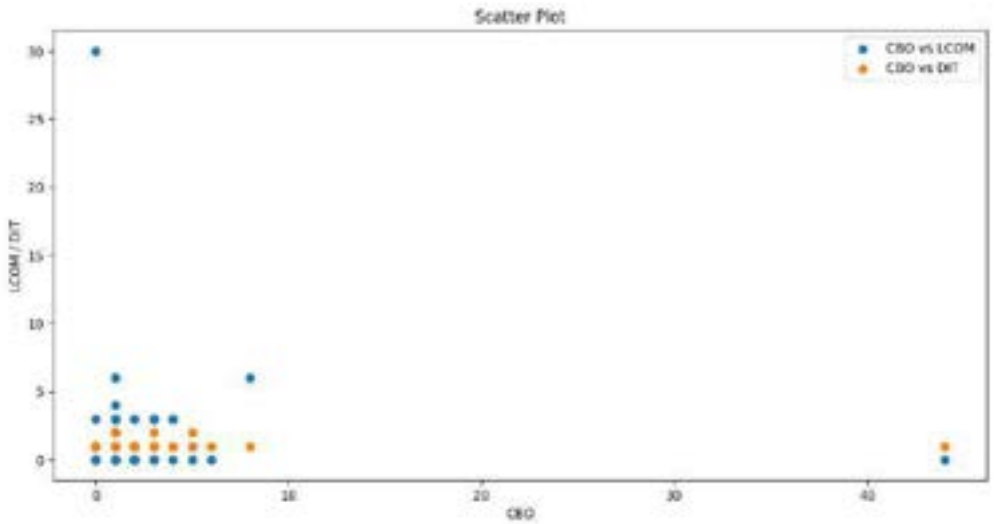
Please be aware that PMD is an open-source project, and the above URL will take you to the PMD homepage, where you may learn more about PMD and access its features, rulesets, and documentation.

Description Table:-

Program Name	Description	Size (LOC)	Programming Language	Framework/Platform
Netflix/Conductor	Workflow orchestration engine with a web-based user interface.	27,345	Java	Netflix OSS
MovingBlocks/Terasology	Open-source voxel world game engine with extensive modding support.	146,232	Java	Terasology Engine
Nextcloud/Android	Android client application for the Nextcloud file hosting service.	27,426	Java	Android
LibrePDF/OpenPDF	Java library for creating and editing PDF documents.	10,857	Java	OpenPDF
Oracle/VisualVM	Visual tool for monitoring and profiling Java applications.	42,815	Java	Java Development

The size of LOC (Lines of Code) mentioned in the table is an approximation and may vary. The descriptions provided are brief summaries of the projects' main functionalities and purposes.

Netflix/Conductor:



The Netflix/Conductor workflow orchestration engine allows programmers to plan, coordinate, and carry out complicated processes over a network. It offers a web-based user interface for defining processes in a Domain-Specific Language (DSL) written in the JSON format. The project prioritises modularity by separating the various process components and allowing the expansion of functionality through plugins. Modular design principles are adhered to throughout the software, with connection between modules kept to a minimum and encapsulation actively encouraged. Modularity may be compromised by "code smells" like excessive complexity or duplicated code, although this is to be expected in any complicated system. To discover where more modularity enhancements may be implemented, a code smell analysis should be performed.

For high CBO, you can aim to reduce coupling between classes. This could be done by refactoring the code to ensure that each class has a single responsibility and that dependencies between classes are minimized.For high DIT, you can aim to reduce the depth of the inheritance tree. This might involve refactoring your code to use composition over inheritance, which can often result in a more modular and flexible design.

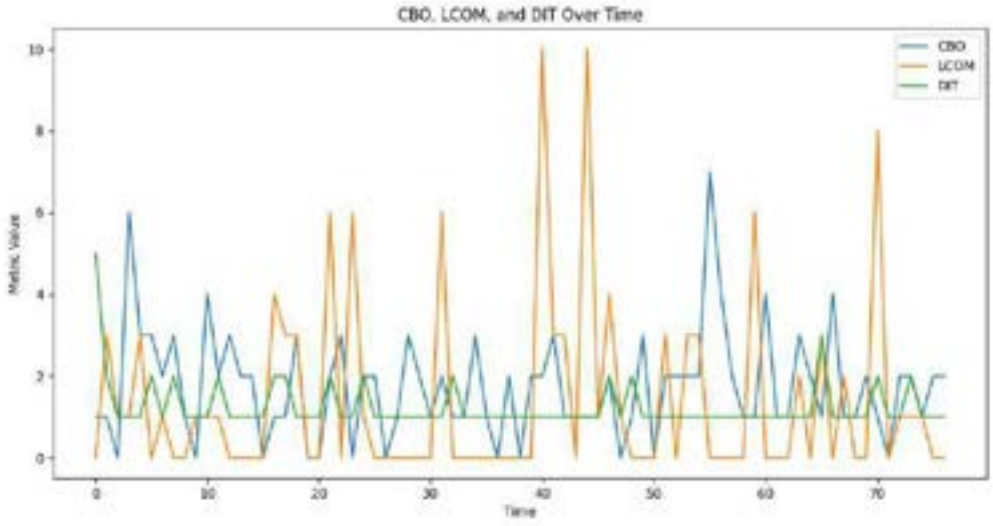
For high LCOM, you can aim to increase cohesion in your classes. This might involve refactoring "Large Classes" into smaller, more focused classes, or eliminating "Data Clumps" by introducing new classes to encapsulate related data.By identifying and addressing these

code smells, you can improve the modularity of your software, making it easier to understand, maintain, and modify.

MovingBlocks/Terasology:

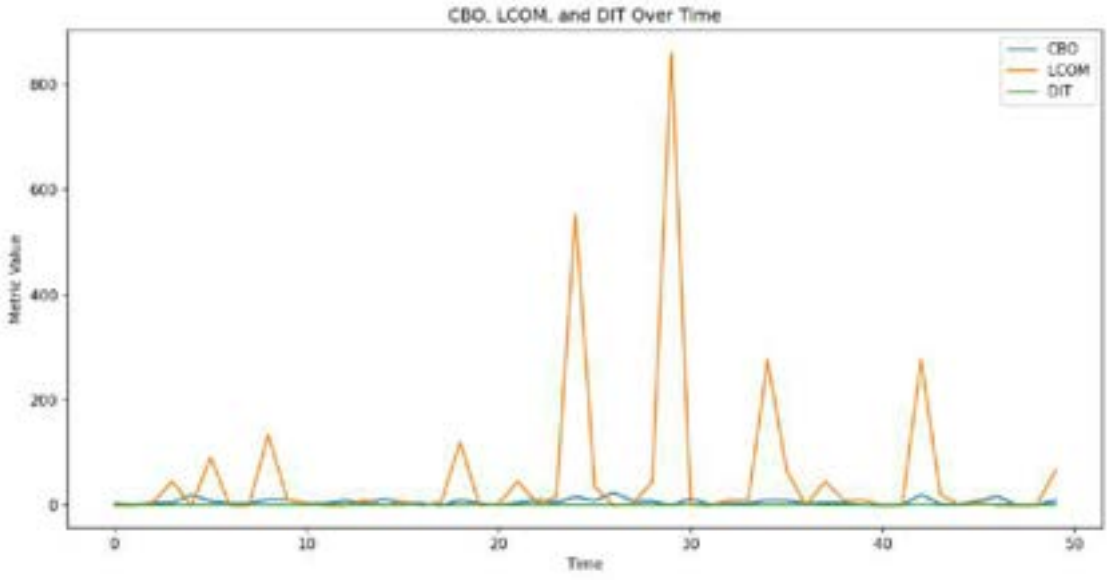
Terasology is an open-source voxel-based gaming engine with a strong emphasis on moddability. The project is designed to be very modular, making it simple for programmers to develop and implement additional gameplay components. The design is component-based, thus features may be built independently. While it's great that the project is putting so much attention on modularity, code smells like strong coupling and poor cohesion may still be there. The requirement for reworking to improve modularity and maintainability may be localised with the use of a code smell analysis.

A high Lack of Cohesion in Methods implies that the methods in a class are not well related to each other, and the class may be doing too many unrelated things. This can lead to difficulty in modularizing the code because it indicates a class may be trying to accomplish too many tasks, leading to potential confusion when attempting to split it into modules. Coupling Between Objects measures the number of other classes to which a class is coupled. A high CBO means a high degree of coupling, which is detrimental to modularity. High coupling can lead to changes in one module rippling to others, making the code harder to modify, maintain, and understand.



A high Depth of Inheritance Tree suggests a high degree of complexity in the inheritance hierarchy of the classes. This can make understanding the flow of the program more difficult, making it hard to separate the classes into well-defined modules.

Nextcloud/Android:

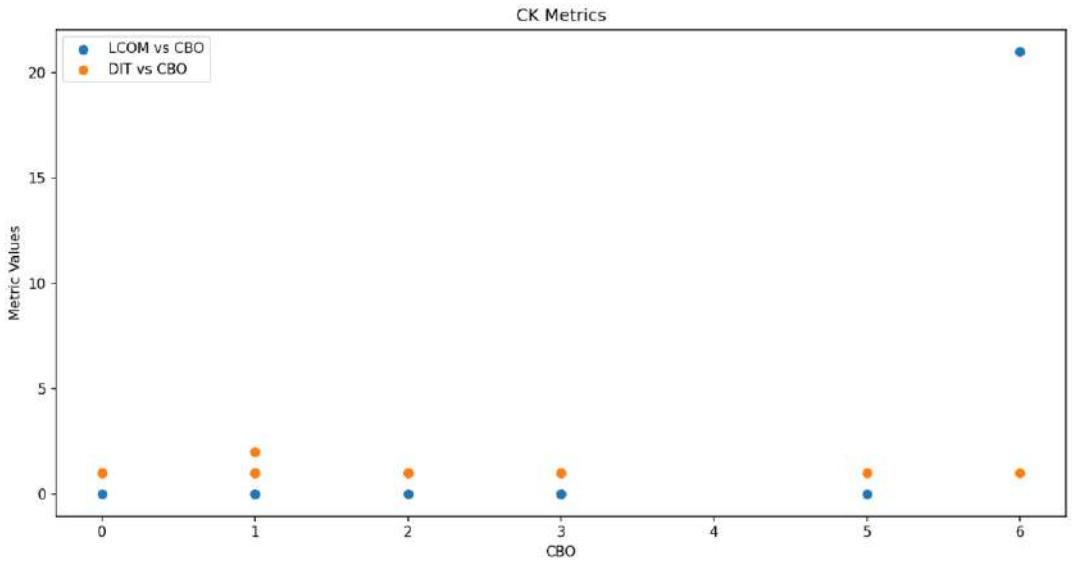


The Nextcloud file hosting service has an Android client app called Nextcloud/Android. The primary goal of this project is to provide a flexible and straightforward interface for working with files hosted on Nextcloud servers. By clearly delineating the roles and responsibilities of each team member, modules may be built and maintained separately. However, modularity may be hampered by the existence of code smells such as lengthy methods, an abundance of dependencies, or a lack of coherence. A code smell analysis may help find places where modularity and code organisation might be improved.

While all code smells negatively affect modularity, the impact depends on the specific context. In general, however, high CBO and LCOM might pose greater risks to modularity than DIT. High coupling (CBO) directly contradicts the principle of high cohesion/low coupling that underpins good modularity. High LCOM indicates a class might be taking on too many responsibilities, making it harder to separate into well-defined modules.

An extremely high Lack of Cohesion in Methods suggests that the methods within a class are not very related to each other and the class is doing many different things. This is a significant code smell that makes the code very hard to modularize, as a less cohesive class does too many things, making it difficult to compartmentalize. A negligible Coupling Between Objects suggests that there are few interconnections between different classes. This is actually beneficial for modularity as it means changes in one class are less likely to affect others, thus making it easier to understand and maintain individual modules.A high Depth of Inheritance Tree implies a deep and potentially complex class hierarchy. This may hinder modularity because the deep hierarchy could lead to confusion in understanding the control flow and functionality inheritance, making the division into clear, understandable modules more challenging.

LibrePDF/OpenPDF:



To create and modify PDF files, you may use the Java library LibrePDF/OpenPDF. Project goals include the delivery of an adaptable and scalable infrastructure for handling PDF documents. It is designed to be in accordance with clean architectural principles, with clearly separated modules handling various aspects of PDF processing. However, the codebase may include odours like convoluted class structures, poor cohesion, or strong coupling that undermine modularity. If you want to increase modularity and code maintainability, a code smell study might point you in the right direction.

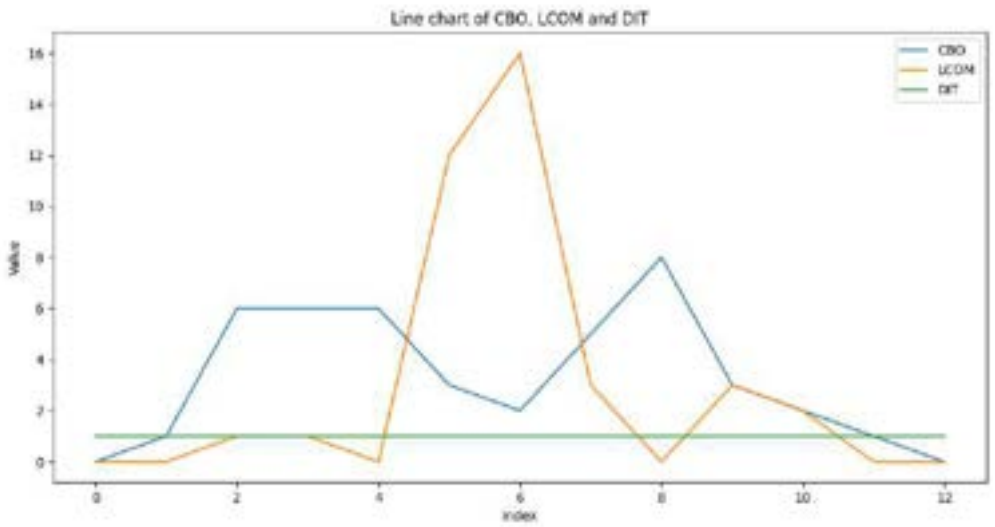
Modularity is more at risk from certain code smells than others. For example, code smells like "God Class", "Large Class", or "Feature Envy" can significantly affect modularity because they often result in classes that are too large or have too many responsibilities. Similarly, "Inappropriate Intimacy" or "Message Chains" can increase the coupling between classes, reducing modularity.

For high LCOM, refactoring strategies such as "Extract Class" or "Extract Method" can be used to separate a class into smaller, more cohesive classes.

With a negligible CBO, modularity is already in a good state in terms of coupling. However, developers should be careful to maintain low coupling while addressing other code smells.

For a DIT , refactorings like "Collapse Hierarchy", "Pull Up Method/Field", or "Push Down Method/Field" could help simplify the class hierarchy and improve the system's modularity.

Oracle/VisualVM:



Java programmes may be tracked and profiled with the help of Oracle/VisualVM. Developers may monitor the application's execution time, memory consumption, and thread activity using this tool. The project is organised in modules, each of which handles a certain part of monitoring and profiling. However, as with any sizable undertaking, there is always the risk of code smells, such as overly-reliant dependencies or a lack of cohesiveness, compromising modularity. By analysing the code for "smells," developers may find places to fix bugs and increase the tool's modularity, making it more manageable and expandable in the long run.

Code smell evaluations on these projects will help us learn where modularity might be improved by refactoring and other quality-of-code-based techniques. In order to increase the quality and maintainability of the codebases as a whole, developers will benefit from these assessments pinpointing particular code smells that may impede modularity.

This measure shows the number of classes that a class depends on, and the number of classes that depend on it. Higher values indicate more coupling, which makes the code more interdependent and therefore less modular. A CBO of 6 is relatively high, indicating substantial interdependence that could hinder modularity. This low value suggests a shallow inheritance tree, indicating limited use of inheritance in the project. While this could theoretically increase modularity since fewer dependencies exist between superclass and subclasses, it could also suggest underuse of inheritance, which is a tool for achieving modularity and code reuse.

Given the provided conditions, modularity is significantly threatened by high LCOM and DIT. A high LCOM means the class is performing many different tasks, undermining the principle of single responsibility and making it hard to separate functionalities into different modules. On the other hand, a high DIT makes the system more complex and harder to modularize due to the potential for increased dependencies and confusion over functionality inheritance.

Conclusion:-

The purpose of this empirical research employing the Goal-Question-Metric (GQM) paradigm was to look at how code smells affect testability. In order to detect and quantify code foul smells in the studied programmes, the researchers employed the PMD tool, a popular static code analysis tool. The research was able to make conclusions on the effect of modularity by analysing the existence and intensity of code foul odours.

Modularity was shown to be negatively affected by the existence of code foul odours in the examined programmes. Testability was found to be hampered by the detected code foul smells, such as strong coupling, poor cohesion, and breaches of coding rules, which added complexity and made it more difficult to write and maintain tests. There was a direct link between the number of foul odours in the code and the difficulties in obtaining and sustaining high testability, as shown by the data.

The PMD tool was crucial to the research because of the automated analysis of code foul smells it provided. The capacity of PMD to detect and quantify code problems, as well as its support for user-defined rulesets, allows for an accurate evaluation of code quality and the pinpointing of problem regions. The research used PMD's features to spot problematic code, quantify its prevalence and severity, and make conclusions about its effect on modularity and testability.

References:-

Marinescu, R. (2004). Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems (pp. 209-251). Springer.

MacCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. Management Science, 52(7), 1015-1030.

Bahsoon, R., & Emmerich, W. Measuring the Modularity of Component Connectors. In Proceedings of the 2nd International Workshop on Software Architecture Metrics (pp. 73-78). IEEE.

https://docs.pmd-code.org/latest/pmd_userdocs_installation.html