**Impact of Code Smells on Software Modularity: An Empirical Study**

***Presented by***

**Siva Reddy Potlapati -** **L30086362**

**Pavan Kalyan Teluguti – L30088205**

**Spring 2024-II**

**ABSTRACT**

This empirical study looks into how software modularity is affected by code smells. Code smells, which include feature envy, God classes, and excessive parameter lists, are obvious indicators of more serious problems inside the source. These problems may not immediately prevent programs from running, but they frequently make code harder to read, modify, and maintain, which weakens software modularity. By applying the Goal-Question-Metric (GQM) methodology, this study attempts to provide a thorough understanding of how much code smells impact software modularity. In order to detect and measure code smells in different software projects, the study analyzes such systems. It then determines how these bad smells affect software modularity measures and draws empirical inferences from the data. The study's findings offer insightful information on how software modularity and code quality are related, which can help in developing methods for enhancing code maintainability and system flexibility.

*Keywords*: code bad smells, software modularity, empirical study, feature envy, God classes, excessive parameter lists, Goal-Question-Metric (GQM) approach, code maintainability, system flexibility.

# **Introduction**

Building strong, adaptable software is key in today's ever-evolving development landscape. One powerful tool to achieve this is software modularity. Modularity breaks down complex systems into independent, reusable building blocks. This makes the software easier to grasp, adjust, test, and keep up to date over time.

Software modularity is hindered by code "bad smells," which are obvious signs of more serious systemic problems. These cover a wide range of issues, such as feature envy, in which classes depend too heavily on methods from other classes; God classes, which bear too many responsibilities; large parameter lists; lengthy methods, which have too many lines or statements; type checking, which uses conditional logic to ascertain object types; and duplicate code, which denotes identical or similar code segments that are present in multiple places. Although these issues might not always cause the program to crash, they seriously hinder the readability, maintainability, and changeability of the code, which weakens its modularity.

The purpose of this study is to empirically examine how software modularity can be influenced by code smells. This study uses the Goal-Question-Metric (GQM) paradigm to understand how much code smells affect software modularity.

The criteria used to choose the software programs, the programs studied, the metrics-gathering tool, our study findings, and conclusions drawn from the empirical data are covered in the following parts.

1. **Research Goals, Questions, and Metrics: Employing the GQM Approach**

## **2.1 Research Objective:**

The purpose of this study is to evaluate the association between software modularity and code smells using an empirical analysis. The aim of this study is to examine the effects of code smells on software systems' modularity, quality, and maintainability using the Goal-Question-Metric (GQM) paradigm.

## **2.2 Research Questions:**

The following research question will be investigated in this study:

**RQ1**: What effects do various code smells have on a software system's modularity?

Our examination into the precise influence of different code bad smells on software modularity will be guided by this question, allowing us to gain a more detailed knowledge of their implications on software quality and maintainability.

* 1. **Research Metrics:**

In order to address our research topic, the following measures will be employed:

a) **Coupling Between Objects (CBO):** This measure assesses how dependently different classes are on one another in a software system. Greater coupling is shown by higher CBO values, which suggests that a class depends more on other classes. Modularity and maintainability may suffer from such tight coupling since modifications in one class may have an impact on others. When a class overly relies on features or methods from other classes, it can cause a "feature envy" code smell that weakens cohesiveness and adds complexity.

b) **Lack of Cohesion in Methods (LCOM)**: LCOM gauges how cohesive or dissimilar a class's methods are from one another. A higher LCOM number points to a possible "god class" code smell by implying that a class's methods are not closely connected. A class that tries to do too much at once is known as a god class because it reduces modularity and makes it harder to comprehend and manage the codebase. Methods that are not cohesive can become more complex and make them less scalable and extensible.

c) **Response For a Class (RFC):** This metric counts the set of distinct responses that a class invokes and those that its methods invoke in order to evaluate the complexity and modularity of a class. A class with a high RFC value has more roles and interactions than others, which reduces the class's modularity. This could be a sign of a code smell called "shotgun surgery," in which adjustments to one class necessitate extensive alterations across the whole codebase. High RFC values can make a class harder to understand and edit, making errors more likely to occur. This can make a class less maintainable.

We will be able to assess software systems' modularity and determine how code smells affect the overall quality and maintainability of the systems thanks to these measurements.

## **Overview of Data Set**

This section provides an overview of the software systems chosen for the empirical analysis of code bad smells and their impact on software modularity, as well as the criteria employed for their selection.

## **3.1 Project Selection Criteria**

We have employed the below criteria for selecting our projects for our study.

1. **Requirement for Project Size:** Programs with a minimum of 5,000 lines of code (LoC) were chosen to meet the size requirement. This guarantees that the programs are complicated enough to possibly display several kinds of code bad smells, which enables a thorough examination of their effects on software modularity.
2. **Requirement for Project Age**: At least six years old projects are taken into consideration. By ensuring that the chosen programs have seen substantial development and maintenance over a considerable amount of time, this criterion offers insights into the long-term effects of code smells on software maintainability.
3. **Developer Involvement**: At least 3developers contributed to the programs that were taken into consideration. The presence of a larger development team is indicative of a diversity of coding styles, ways of cooperation, and potential conflicts in design decisions. These factors can affect software modularity by influencing the amount and impact of code smells.

Following these requirements, we were able to choose 10 Java projects from GitHub that are typical of the class and should offer deep insights into the relationship between code smells and modularity.

## **3.2 Justification for Project Selection Criteria**

* **Age Requirement**: It's likely that programs that are six years or older have experienced numerous rounds of development and maintenance. This age requirement makes sure that the programs have become sufficiently complicated and have experienced a range of maintainability tasks, which means there will be plenty of opportunity for code smells to appear and be examined.
* **Size Requirement**: Programs that have at least 5,000 lines of code are anticipated to be quite complex, which means that code smells may emerge in them. This kind of program analysis allows for a thorough investigation of the connection between software modularity and code smells.
* **Developer Involvement**: If a development team has at least three people involved, there is likely to be a great deal of collaboration and maybe different coding practices. Multiple developers raise the possibility of running into various kinds of code bad smells and highlight how crucial it is to fix them in order to preserve software modularity.

## **3.3 Studied Projects:**

Based on the aforementioned standards, the 10 projects listed below were chosen to offer a variety of systems and applications for analysis:

**Table 1: *Summary of Studied Projects***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Program Name** | **Developers** | **Program Age (Years)** | **Size (LOC)** | **Description** |
| **Java SAML Toolkit** | 42 | 10 | 37,121 | A Java implementation of the SAML protocol that makes single sign-on authentication possible is called java-saml. It provides tools for creating, interpreting, and modifying SAML claims. The integration of SAML-based authentication into Java applications is made easier by this project. |
| **Slf4j** | 78 | 18 | 40,225 | A logging framework called SLF4J (Simple Logging Facade for Java) offers a straightforward abstraction over other logging frameworks. It provides parameterized logging for better speed and marker support for classifying log messages. Changes to application code are not necessary when moving between logging technologies thanks to SLF4J. |
| **Elasticsearch-SQL** | 31 | 9 | 55,020 | Elasticsearch SQL is a plugin that makes data retrieval and analysis activities easier by allowing Elasticsearch queries to be written in SQL syntax. It easily connects with Elasticsearch, giving users a recognizable query interface for Elasticsearch indices. By improving accessibility and usability, the plugin serves users who are familiar with SQL querying techniques. |
| **Spring-Cloud-Platform** | 4 | 6 | 6715 | A complete framework for creating distributed systems using Spring Cloud and other technologies is called Spring-Cloud-Platform. For creating cloud-native applications, it offers an extensive collection of tools and parts, such as distributed tracing, configuration management, and service discovery. This platform facilitates microservices architecture development and deployment by integrating seamlessly with the Spring environment. |
| **spring-boot-examples** | 8 | 7 | 28,603 | The GitHub repository "spring-boot-examples" offers a thorough instructional project that teaches Spring Boot via real-world examples. It provides a range of examples, from basic to complex, all of which are intended to be simple and uncomplicated so that students may rapidly understand various facets of Spring Boot. |
| **Razorpay Java SDK** | 14 | 7 | 23,930 | The official Java SDK for using the Razorpay API and integrating its payment gateway functionality into Java applications can be found on GitHub under the "razorpay-java" project. It is intended for developers that want reliable, scalable financial processing in their Java applications, and it handles tasks like payments, refunds, and settlements. |
| **logstash-forwarder-java** | 3 | 9 | 5280 | Log shipping is a Go utility that has a Java equivalent called "logstash-forwarder-java". Because of its lightweight and portable nature, it may be used in settings like AIX systems where Go is not accessible. Because it utilizes less resources and is compatible with the Logstash-forwarder configuration files, this tool is a good option for integrating with the ELK stack. |
| **Vert.x Core** | 256 | 10 | 97786 | The goal of the JVM toolkit Vert.x is to facilitate the development of scalable and performant reactive applications. It works well with modern reactive programming paradigms, supports a broad variety of network protocols, and provides tools for asynchronous communication. The essential capabilities manage file system operations, TCP, HTTP, and much more. |
| **JavaPoet** | 81 | 11 | 14123 | JavaPoet is a Java API that simplifies code generation using a programmatic interface by creating.java source files. This removes the need to create boilerplate code by enabling the dynamic production of Java code during runtime, such as when managing metadata or processing annotations. |
| **JavaCV** | 75 | 11 | 57,151 | Popular multimedia and computer vision libraries like FFmpeg and OpenCV may be used from Java programs thanks to JavaCV, an interface to these libraries. Processing images, identifying features, and manipulating videos are made simpler, and it incorporates utility classes for simpler interface with Java systems, such Android. |

**3.4** **Comprehensive Project Details and their applicability to current research**

Here is a thorough summary of the chosen projects that will help us gain a deeper grasp of each while showcasing its special qualities and applicability to this research:

1. **Java SAML Toolkit:** A Java implementation of the SAML protocol The Java SAML Toolkit is a crucial Java implementation of the Security Assertion Markup Language (SAML) protocol. It is housed in the "java-saml" repository on GitHub. It provides an extensive set of tools and libraries designed only for handling SAML assertions, encompassing operations like SAML message generation, parsing, and modification. Given its complexity and importance in web authentication, this toolkit makes an interesting case study within the current research on software modularity and code smells. Determining whether code smells exist in the toolkit's codebase and evaluating how they affect modularity might provide important information about how code quality and software design interact.
2. **Slf4j:** The ability of SLF4J to offer a uniform logging interface for various logging frameworks can greatly aid in comprehending how code "bad smells" affect program modularity. Code smells associated with different logging approaches are less common because of SLF4J's smooth integration with several logging implementations, which guarantees consistency in logging practices. Furthermore, SLF4J mitigates common code smells like excessive logging or improper log message formatting by supporting markers and providing parameterized logging options, which encourage simpler and more maintainable code. Examining how SLF4J is used in codebases that are having problems with modularity might provide information about how good logging techniques enhance overall code quality and modularity, which is directly related to the goals of the present study**.**
3. **Elasticsearch-SQL**: Elasticsearch-SQL improves data retrieval and analysis by symbiotically fusing classical SQL querying with Elasticsearch's distributed search capabilities. Its user-friendly syntax encourages broader acceptance among a variety of user groups, and features like support for sophisticated queries make work easier. Examining how it affects modularity and maintainability of the code provides information about how to integrate third-party plugins successfully, deal with typical code smells, and encourage structured querying. With Elasticsearch SQL as a case study, this study attempts to identify methods for preserving code quality during external component integration.
4. **Spring-Cloud-Platform:** A sophisticated RBAC management system using microservices architecture with components from Alibaba Cloud, Spring Cloud, and Spring Boot is called the "Spring-Cloud-Platform" on GitHub. It uses contemporary technologies like Sentinel for fault tolerance, Nacos for service discovery, and JWT for security, along with features like user and permission management. This platform is very important for research on how code smells affect the structural integrity and maintainability of software systems since it is perfect for examining the impacts of architectural choices and code quality on software modularity.
5. **Spring-boot-examples:** Many Spring Boot apps with varying degrees of complexity are available from the "spring-boot-examples" repository. These illustrations serve as a great tool for analysing how coding patterns and ideas are used in practice in a real-world setting. This repository provides a diverse dataset for empirical analysis in research focusing on the effect of code smells on software modularity. In a controlled, instructional environment, researchers may investigate how various coding methods impact modularity, recognizing certain code smells and their effects on maintainability and system structure. Because of this, it's a priceless resource for comprehending the real-world effects of software design decisions.
6. **Razorpay Java SDK:** With the "razorpay-java" SDK, Java developers may easily include Razorpay's payment services, which include features like invoices, refunds, and payments via a strong API. Its modular design enables the addition of sophisticated payment features without sacrificing maintainability. Because it provides a practical framework for examining how industry-grade software is designed to improve modularity and lower maintenance challenges, the structure of this SDK is highly relevant for research on the effects of code smells on software modularity, with findings being applicable to current industry practices.
7. **logstash-forwarder-java:** This Java-based log shipper might be useful for studying how software modularity is affected by code smells. Researchers may investigate how the modular architecture contributes to low coupling and strong cohesion—characteristics that reduce the emergence and propagation of code smells—by looking through the source. This might provide guidance on how to create scalable, maintainable logging solutions in Java, which is essential for preserving the integrity of large-scale systems.
8. **Vert.x Core:** A JVM toolkit called vert.x facilitates the development of reactive applications and offers scalable, high-performance solutions for a range of network protocols. It effectively handles TCP and HTTP by using reactive programming principles and asynchronous communication. This project is perfect for researching how code smells affect software modularity because of its modular architecture and usage of reactive paradigms. It is relevant for empirical software engineering research because it offers a strong case study for analysing how architectural choices affect maintainability and scalability.
9. **JavaPoet:** Research on how code smells affect software modularity may benefit greatly from the use of JavaPoet. Because of its structured and programmable code construction technique, researchers may test different code creation tactics and see how they affect the occurrence or avoidance of code smells. Furthermore, applying it to automated code generation scenarios offers useful knowledge on how to preserve code quality and modularity in dynamically produced software settings.
10. **JavaCV:** Beyond its interface functionality, JavaCV constantly works to improve Java's ability to handle real-time image and video processing tasks, which makes Java an excellent choice for applications needing sophisticated multimedia manipulation and analysis. The project's capacity to connect native libraries and Java may be essential to understanding how these integrations either increase or decrease structural complexity and code smells. Researchers may get a more profound comprehension of the effects of cross-language library integration on modularity and overall software health by exploring its architectural principles and implementation approaches. This investigation may provide crucial information for enhancing software design in such difficult settings**.**

# **Analytical Tools Utilized in This Study**

We used two well-known software analysis tools, CKJM and JDeodorant, to accurately extract and analyze the C&K metrics we picked as well as to identify code smells in the programs we chose.

**4.1 CK Metric Analysis Tool (CKMAT):** Six essential object-oriented metrics are part of the Chidamber and Kemerer (CK) metrics package, which is used to evaluate software architecture. These include Number of Children (NOC), which counts subclasses; Coupling Between Objects (CBO), which assesses interdependencies between classes; Response For a Class (RFC), which totals unique method responses; Weighted Methods per Class (WMC), which measures complexity based on the quantity and type of methods; Depth of Inheritance Tree (DIT), which measures a class's inheritance levels; and Lack of Cohesion in Methods (LCOM), which measures method cohesion within a class. Together, these metrics aid in the analysis of software structures' coherence, coupling, and complexity.

**4.2 Code Smell Detection with JDeodorant:** We used JDeodorant, an Eclipse plug-in that is well-known for its ability to use static analysis to find different types of code smells, in our investigation. This tool is good at spotting Type Checking, Duplicated Code, Long Method, God Class, and Feature Envy smells. These problems make it more difficult to maintain and scale programs, which has a substantial impact on software modularity. JDeodorant detects these scents and offers reworking actionable information that are critical to enhancing the software's performance and structural integrity.

**4.3 Methodology**

First, we used the CK metrics tool on all the projects I have selected focussed on measuring Response For a Class (RFC), Coupling Between Objects (CBO), and Lack of Cohesion in Methods (LCOM). These metrics illuminated the software's modularity by providing crucial information on the cohesion and coupling levels inside it. In parallel, we used the JDeodorant plug-in to find and categorize code smells in the same codebases, determining the kinds and quantities of these scents that were important for our investigation. To make the next stages of analysis go more quickly, every piece of information gathered—from code smells to software metrics—was methodically documented and assembled.

In the context of this investigation, three measures stand out as particularly important:

* **Coupling Between Objects (CBO)**: CBO counts the number of classes with which a given class directly interacts in order to assess the interdependence between classes in a software system. Strong dependencies are indicated by high CBO values, which point to a strongly connected system architecture. Over-coupling can make maintenance more difficult and limit the adaptability of the system, which could lead to more mistakes when making modifications. As a result, CBO is an essential indicator of system flexibility that also captures the connectivity and maybe vulnerability of software designs.
* **Response For a Class (RFC):** RFC counts the set of distinct responses from a class, including methods that the class can call and methods that the class can call. This method is used to determine how complex a class is. Higher complexity and lower modularity are indicated by a high RFC value, which may make the codebase more difficult to comprehend, test, and maintain. To guarantee manageable and maintainable software systems, RFC management is crucial.
* **Lack of Cohesion in Methods (LCOM):** LCOM measures how closely connected a class's methods are to one another. Greater LCOM values imply a lack of cohesiveness and an excessive number of tasks being attempted by the class. When one class takes on too many responsibilities, it frequently leads to a "God class" code smell, which reduces modularity and increases complexity. Improving program modularity and code maintainability requires addressing high LCOM values.

**4.4 The Justification for Tool Selection**

The decision to use CKJM and JDeodorant was made in light of their track record of correctness, dependability, and standing in the software engineering research community. Because both tools are particularly made for Java, accuracy and compatibility are guaranteed. CKJM makes it easier to monitor important software metrics precisely, which is essential for assessing modularity. In a similar vein, JDeodorant's capacity to recognize and classify common code smells offers invaluable insights, expanding our knowledge of the variables influencing software quality and modularity.

# **Results**

Based on our examination of particular software projects, this section presents the findings of our empirical investigation on the effect of code smells on software modularity. Together with identified code smells, we provide the calculated metrics for each project's Coupling Between Objects (CBO), Lack of Cohesion in Methods (LCOM), and Response For a Class (RFC). Our objective is to identify classes that have anomalous metric values that can lead to the presence of code smells and to illustrate the patterns in these metrics across different classes within the projects.

## **5.1 Project Specific Results**

**5.1.1** **Java SAML Toolkit Results**

This project has around 85 classes. For every class, the CBO, LCOM, and RFC metrics were computed, displaying differing levels of coupling, cohesion, and complexity.

Based on the line graph and CK Metrics we can figure out the following trends in the project:

1. Classes like com. onelogin. saml2.test.AuthTest (24), com.onelogin.saml2.util.Util, com.onelogin.saml2.Auth has the highest values of **CBO**. This shows that a large number of other classes in the software system depend heavily on these classes. Their high CBO values suggest that they interact with other components extensively. Whereas classes like SamlResponseStatus (0), Constants, AuthnRequestTest$Anonymous1, AuthnRequestTest$Anonymous2, AuthnRequestTest$Anonymous3have lower values of CBO which indicates only a few classes depending on them.
2. Greater Response For a Class (**RFC)** values are shown in the classes SettingsBuilder, AuthTest, and Util (213), which may indicate decreased modularity and increased complexity. On the other hand, AuthTest's nested classes and SamlMessageFactory (0) exhibit lower RFC values, indicating simpler and more concentrated tasks.
3. The classes Saml2Settings, AuthTest, and AuthnResponseTest (7171) show high levels of Lack of Cohesion in Methods **(LCOM)**, indicating that there may not be a strong relationship between their methods inside the class. On the other hand, classes with lower LCOM values, such as AuthnRequestTest$Anonymous1(0), AuthnRequestTest$Anonymous2, and others mentioned, show a more concentrated and well-organized arrangement of methods.

From the Jdeodrant the code smells like Feature Envy,Long Method,God Class are present in these classes

Util,AuthnRequest,AuthUtil,SamlResponse,LogoutResponse,LogoutRequest,IdPMetadataParser,SettingsBuilder,Metadata

**Table 1: Tabular details of CBO, RFC, LCOM values for Java SAML ToolKit having code smells**

|  |  |  |  |
| --- | --- | --- | --- |
| **Class** | **CBO** | **RFC** | **LCOM** |
| com.onelogin.saml2.util.Util | 24 | 213 | 2420 |
| com.onelogin.saml2.authn.AuthnRequest | 7 | 39 | 14 |
| com.onelogin.saml2.authn.SamlResponse | 15 | 124 | 726 |
| com.onelogin.saml2.logout.LogoutResponse | 15 | 81 | 137 |
| com.onelogin.saml2.logout.LogoutRequest | 13 | 90 | 662 |
| com.onelogin.saml2.settings.SettingsBuilder | 8 | 132 | 314 |

From these we can draw the conclusion that code smells indicating potential complexity, maintenance challenges, and reduced modularity in the codebase are present in Classes Util, AuthnRequest, SamlResponse, LogoutResponse, and SettingsBuilder. These code smells are characterized by high Coupling Between Objects (CBO), Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM). Improving the quality and maintainability of the code requires addressing these problems.

## **5.1.2 Slf4j Results**

There are around 278 classes in this project Out of these below are the classes that have higher or lower values of RFC, LCOM, and CBO.

i) Classes like Reload4jLoggerAdapter have the value of 19, CallerInfoTest, InvocationTest, and LoggerTest exhibit higher values of Coupling Between Objects (CBO), indicating increased interdependencies between classes. Conversely, classes such as MessageFormatterTest$Anonymous1, OptionHandler, and SilentPrintStream have CBO values of 0.

ii) Classes such as EventRecordingLoggerTest, LoggerFactory, and SubstituteLogger display higher values of Response For a Class (RFC), indicating increased complexity and potentially higher coupling within these classes. Conversely, classes like InvokeJCLTest$TestMessage and Reporter$TargetChoice have RFC values of 0, suggesting simpler class structures with fewer method invocations.

iii) Classes such as **Logger**, **NOPLogger**, and **AbstractLogger** exhibit lower Lack of Cohesion in Methods (LCOM) values, indicating that their methods are more cohesive and focused on specific tasks within the class and around 30 classes have lower LCOM values.

From the Jdeodrant LongMethod, Feature Envy code smells are reported.

Here are the classes with the greatest number of code smells.

**Table 2: Tabular details of CBO, RFC, LCOM values for Slf4j having code smells**

|  |  |  |  |
| --- | --- | --- | --- |
| **Class** | **CBO** | **RFC** | **LCOM** |
| org.slf4j.reload4j.Reload4jLoggerAdapter | 19 | 26 | 0 |
| org.slf4j.issue.CallerInfoTest | 14 | 32 | 0 |
| org.slf4j.reload4j.InvocationTest | 13 | 52 | 0 |
| org.slf4j.eventTest.EventRecordingLoggerTest | 11 | 84 | 300 |
| org.slf4j.LoggerFactory | 11 | 81 | 311 |
| org.slf4j.helpers.SubstituteLogger | 9 | 76 | 3035 |
| org.slf4j.migrator.internal.MigratorFrame | 3 | 66 | 162 |
| org.slf4j.helpers.MessageFormatter | 12 | 43 | 208 |

From this, we can figure out that there are differences in the complexity and interdependencies of classes that have code smells recognized. While some show a low Lack of Cohesion of Methods (LCOM) and a moderate Coupling Between Objects (CBO) and Response For Class (RFC), others, like org.slf4j.helpers, show a high Lack of Cohesion of Methods (LCOM).ReplaceLogger together with org.slf4j.helpers.Higher values for MessageFormatter are seen across all metrics, indicating possibly more complex and integrated architectures. These variations highlight possible effects on modularity and maintainability.

**5.1.3** **Elasticsearch-SQL Results**

There are around 150 Classes in this project and CBO, RFC, LCOM values are extracted for all these classes, and below are the trends.

1. The classes ElasticSqlStatementParser (304),ResponseConverter,RequestConverter classes are having higher values of CBO conversely, the classes such as ElasticSearchDruidDataSource$Anonymous2,CsvExtractorException,ElasticSearchDruidDataSource$Anonymous1 have lower values of CBO.
2. Higher RFC values are observed in classes like ElasticSqlStatementParser (570) ElasticSearchDruidDataSource (412), indicating greater interconnections and dependencies. In contrast, lower RFC values are found in classes such as CsvExtractorException,ObjectResultsExtractException,.GetIndexRequestRestListener$Fields, and MetaSearchResult, all recording an RFC value of 0.
3. The class with the highest LCOM value (17212) is " ElasticSearchResultSet", indicating potential method cohesion issues. On the other hand, classes like " SelectParser", " CSVResult", and " Point" have the lowest LCOM values of 0, suggesting strong method cohesion within these classes.

By using Jdeodrant there are code smells in multiple classes. Code smells likeLongMethod, Duplicated Code, God Class are present .But however the classes havig multiple code smells reported here.

**Table 3: Tabular details of CBO, RFC, LCOM values for Elastic-search-sql having code smells**

|  |  |  |  |
| --- | --- | --- | --- |
| **class** | **cbo** | **rfc** | **lcom** |
| org.elasticsearch.aggregations.pipeline.MovFnPipelineAggregationBuilder | 18 | 32 | 44 |
| org.elasticsearch.aggregations.pipeline.BucketSelectorPipelineAggregationBuilder | 13 | 33 | 13 |
| com.alibaba.druid.pool.ElasticSearchDruidPooledPreparedStatement | 14 | 23 | 8 |
| org.elasticsearch.join.aggregations.ChildrenAggregationBuilder | 20 | 14 | 64 |
| org.elasticsearch.join.query.HasChildQueryBuilder | 17 | 45 | 0 |
| org.nlpcn.es4sql.CSVResultsExtractorTests | 8 | 22 | 378 |
| org.nlpcn.es4sql.domain.Field | 3 | 5 | 14 |

An interesting pattern in this specific project is the identification of code smells in classes with modest levels of metrics such as CBO, RFC, and LCOM. Examples of such moderate values that are flagged as having code smells are "org.elasticsearch.aggregations.pipeline.MovFnPipelineAggregationBuilder", "org.elasticsearch.aggregations.pipeline.BucketSelectorPipelineAggregationBuilder", and "com.alibaba.druid.pool.ElasticSearchDruidPooledPreparedStatement". On the other hand, classes with extreme values—for example, "org.elasticsearch.join.query.HasChildQueryBuilder" with an RFC value of 0—generally do not exhibit code smells. This suggests that, in this particular situation, there is a complex relationship between these indicators and code quality.

## **5.1.4 Spring-Cloud-Platform**

In this Project we have around 178 classes. Here are the values of LCOM,CBO and RFC

1. The degree of interdependence between classes is shown by the Coupling Between Objects (CBO) measure. As an example, high CBO values for PermissionService, AccessGatewayFilter, and LuceneDao indicate a high degree of dependency on other classes. In contrast, the lower CBO values of UserController and BaseController suggest less dependencies and maybe simpler class hierarchies. These results imply that reworking some classes might be necessary to lessen their dependencies and enhance their modularity.
2. A class's interaction with other classes is measured by the Response For a Class (RFC) metric. With high RFC values in our dataset, GroupBiz, PermissionService, and AccessGatewayFilter stand out and may have interacted with other classes extensively. Conversely, classes with RFC values of 0 (such as UserTokenException and RestCodeConstants) indicate little interactions. Knowing RFC makes it easier to discover classes that might need further attention.
3. The cohesion of a class's methods is gauged by the Lack of Cohesion in Methods (LCOM) metric. GroupController and GroupBiz are two classes with high LCOM values that might have dispersed or unrelated methods, which could be a symptom of design flaws. On the other hand, low LCOM values are seen in classes like AuthServiceImpl and GeneratorRest, which indicate high method cohesion and well-organized code.

From the Jdeodrant we have found LongMethod,God Class and Feature Envy fro multiple classes. We have listed below the classes with most occurrences of code smells.

**Table 4: Tabular details of CBO, RFC, LCOM values for Spring-Cloud Platform classes which are having code smells.**

|  |  |  |  |
| --- | --- | --- | --- |
| **class** | **cbo** | **rfc** | **lcom** |
| com.gihub.wxiaoqi.security.gate.filter.AccessGatewayFilter | 31 | 64 | 8 |
| com.github.wxiaoqi.security.modules.admin.rpc.service.PermissionService | 29 | 104 | 27 |
| com.github.wxiaoqi.search.lucene.LuceneDao | 23 | 28 | 41 |
| com.github.wxiaoqi.security.modules.auth.service.impl.AuthServiceImpl | 20 | 39 | 0 |
| com.github.wxiaoqi.security.modules.admin.rpc.service.PermissionService | 29 | 104 | 27 |
| com.github.wxiaoqi.security.gate.filter.AccessGatewayFilter | 31 | 64 | 8 |
| com.github.wxiaoqi.security.generator.utils.GeneratorUtils | 9 | 59 | 21 |
|  |  |  |  |

In this project, it has been evident that the classes which have higher CBO,RFC,LCOM values are having more code smells. The code smell pattern with high Coupling Between Objects (CBO), Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM) values in the classes AccessGatewayFilter, PermissionService, and GeneratorUtils. These metrics point to potential problems in the codebase that may benefit from restructuring to increase readability and maintainability, such as tight coupling, excessive interactions with other classes, and a lack of method coherence.

**5.1.5 Spring-boot-examples**

In this project, we have around 308 classes. The higher and lower values of RFC, CBO, and LCOM are represented here.

i) A variety of classes with differing degrees of Coupling Between Objects (CBO) are displayed in the dataset. Classes with low CBO values, such as com.neo.util.NeoProperties, com.neo.dao.impl.UserDaoImpl, and com.neo.service.MailService, suggest a more modular and loosely linked architecture. On the other hand, substantially larger CBO values are shown by classes like com.neo.entity.SysPermission, com.neo.model.UserInfo, and com.neo.repository.CustomerRepositoryTest, indicating stronger interdependencies between objects. This variation highlights how crucial dependency management is to the flexibility and maintainability of programs.

ii) There is variation in Response For a Class (RFC) values throughout the sample. Some classes, like com.neo.repository.CustomerRepositoryTest, com.neo.model.SysPermission, and com.neo.entity.UserInfo, have noticeably higher RFC values, suggesting complex functionalities and possibly broader responsibilities. Meanwhile, classes like com.neo.util.NeoProperties, com.neo.config.PrimaryMongoConfig, and com.neo.web.HelloController have low RFC values, indicating simpler and focused responsibilities.

1. The dataset displays variations in the LCOM (Lack of Cohesion in Methods) scores between classes. As an example, the low LCOM values of com.neo.service.MailService, com.neo.dao.impl.UserDaoImpl, and com.neo.web.ThymeleafController suggest improved method coherence and targeted functionality. On the other hand, classes with much higher LCOM scores include com.neo.entity.UserInfo, com.neo.repository.CustomerRepositoryTest, and com.neo.model.SysPermission; these indicate a lack of method coherence and dispersed responsibilities. This version emphasizes how crucial coherent class architectures are to better understand and maintain code.

In Jdeodrant there are no code smells detected.

## **5.1.6 Razorpay Java SDK**

In this project, we have 99 classes. The highest,lowest values of RFC,LCOM,CBO are represented below.

i) com.razorpay.RazorpayClient,com.razorpay.ApiUtils,com.razorpay.BaseTest are the three classes which are having high values of CBO whereas

com.razorpay.ApiUtils$Method,com.razorpay.ValidationType,com.razorpay.RazorpayException have the lower values of CBO .

ii) com.razorpay.ApiUtils,com.razorpay.ApiClient,com.razorpay.BaseTest are the classes that are reportedly having higher values of RFC and there are multiple classes which are having RFC value as 0.

iii)com.razorpay.PaymentClient,com.razorpay.ApiClient,com.razorpay.SubscriptionClient are the classes having higher LCOM values. The highest LCOM value is 366 and there are multiple classes whose LCOM value is zero.

Jdeodrant has reported only one Code smell i.e. God Class and we can observe that the class having code smells has a higher value of RFC,lcom and even cbo value is moderately higher.

**Table 5: Tabular details of CBO, RFC, LCOM values for RazorPay Java SDK classes having code smells**

|  |  |  |  |
| --- | --- | --- | --- |
| **class** | **cbo** | **rfc** | **lcom** |
| com.razorpay.ApiClient | 9 | 46 | 183 |

## **5.1.7 logstash-forwarder-java**

The project has around 30 classes.

1. The Classes such as info.fetter.logstashforwarder.Forwarder has the highest value of CBO which is 25 followed by info.fetter.logstashforwarder.FileWatcher which s having a value of 16 and then info.fetter.logstashforwarder.FileReaderTest has the value of 11.And the classs such as info.fetter.logstashforwarder.util.AdapterException and info.fetter.logstashforwarder.Multiline$WhatType have lower values of CBO which is 0.
2. The Classes such as info.fetter.logstashforwarder.FileWatcher,info.fetter.logstashforwarder.util.RandomAccessFile,info.fetter.logstashforwarder.Forwarder have higher values of RFC the highest being 68 and the other classes such as info.fetter.logstashforwarder.util.AdapterExcep,info.fetter.logstashforwarder.Multiline$WhatType has lower values of RFC which are reported to be 0.
3. The classes such as info.fetter.logstashforwarder.util.RandomAccessFile have largest value of LCOM which is 2891 followed by info.fetter.logstashforwarder.FileState which reportedly has the value of 432 and there are multiple classes which are having the value of 0.

TheJDeodrant has reported there are four code smells of FeatureEnvy and GodClass for this project.

**Table 6: Tabular details of CBO, RFC, LCOM values for logstash-forwarder-java classes having code smells**

|  |  |  |  |
| --- | --- | --- | --- |
| **class** | **cbo** | **rfc** | **lcom** |
| info.fetter.logstashforwarder.Reader | 6 | 15 | 13 |
| info.fetter.logstashforwarder.FileWatcher | 16 | 68 | 0 |
| info.fetter.logstashforwarder.util.RandomAccessFile | 10 | 63 | 2891 |

## **5.1.8 Vert.x Core**

There are around 386 classes in this project.Here are the results of highest and lowest values of CBO,RFC ad LCOM values in each class.

1. The classes such as io.vertx.example.web.angular\_realtime.Server is having higher value of CBO reporting at 15 and followed by io.vertx.examples.webapiservice.WebApiServiceExampleMainVerticle which is having a value of 14 followed by io.vertx.example.web.sqlclient.Server.There are multiple other classes which are having values 0.
2. Some classes like **io.vertx.examples.http2.Http2ServerVerticle** and **io.vertx.example.mqtt.app.Server** show high RFC values (48), suggesting complex functionalities. Conversely, classes like **io.vertx.example.camel.rmi.HelloServiceImpl** and others have RFC values of 0, indicating simpler responsibilities and clearer code structures.
3. Classes like io.vertx.examples.webapiservice.models.Transaction and io.vertx.examples.service.rxjava3.ProcessorService exhibit higher LCOM values (39 and 35 respectively), suggesting lower cohesion and potentially more scattered responsibilities within these classes. On the other hand, several classes, including io.vertx.example.reactivex.services.serviceproxy.reactivex.SomeDatabaseService, have LCOM values of 0, indicating better method cohesion and a more focused design.

Jdeodrant has reported code smells like Duplicated code,Long Method,Feature Envy and God classs smell in multiple class. It’s clearly evident that all the classes which has exhibited higher values of RFC,CBO, or LCOM have more code smells when compared to ther classes.

**Table 7: Tabular details of CBO, RFC, LCOM values for Vert.x.Core classes having code smells**

|  |  |  |  |
| --- | --- | --- | --- |
| **class** | **cbo** | **rfc** | **lcom** |
| io.vertx.examples.webapiservice.models.Transaction | 4 | 7 | 39 |
| io.vertx.examples.service.rxjava3.ProcessorService | 6 | 14 | 35 |
| io.vertx.example.reactivex.services.serviceproxy.reactivex.SomeDatabaseService | 6 | 8 | 25 |
| io.vertx.example.sqlclient.template\_mapping.User | 6 | 1 | 18 |
| io.vertx.example.web.angular\_realtime.Server | 15 | 41 | 0 |
| io.vertx.examples.webapiservice.WebApiServiceExampleMainVerticle | 14 | 27 | 6 |
| io.vertx.example.proton.server.HelloServer | 13 | 46 | 10 |
| io.vertx.examples.http2.Http2ServerVerticle | 11 | 48 | 13 |
| io.vertx.example.mqtt.app.Server | 5 | 48 | 1 |
| io.vertx.example.proton.server.HelloServer | 13 | 46 | 10 |

## **5.1.9 JavaPoet**

The JavaPoet has 100 Classes and here are the values of highest and Lowest RFC,CBO,LCOM values

i)The classes suh as com.squareup.javapoet.MethodSpecTest have higer value of CBO which is reported to be 28 which followed by

com.squareup.javapoet.TypeSpecTest which is having the value of 27 followed by com.squareup.javapoet.JavaFileTest class which has 25. There are multiple classes which are having lower values and the lowest beig reported is 0.

ii) The classes such as com.squareup.javapoet.TypeSpecTest com.squareup.javapoet.MethodSpec,com.squareup.javapoet.CodeWriter are having the highest values of RFC where the highest RFC is 150 and there are multiple classes which are having lower RFC values reportedly zero.

iii)com.squareup.javapoet.TypeSpecTest, com.squareup.javapoet.JavaFileTest, and com.squareup.javapoet.CodeBlockTest exhibit remarkably high LCOM values, indicating significant dispersion among their methods. Conversely, there are multiple classes with low LCOM values, such as those with a value of 0, showcasing more cohesive and concentrated method structure.

By using Jdeodrant I have found that there are code smells related to Feature Envy, TypeChecking and God Class.In the tabular form, I have represented the classes having multiple code smells.

**Table 8: Tabular details of CBO, RFC, LCOM values for JavaPoet classes having code smells**

|  |  |  |  |
| --- | --- | --- | --- |
| **class** | **CBO** | **RFC** | **LCOM** |
| com.squareup.javapoet.TypeSpec$Builder | 14 | 75 | 591 |
| com.squareup.javapoet.AnnotationSpec | 8 | 55 | 46 |
| com.squareup.javapoet.FieldSpec$Builder | 9 | 14 | 45 |
| com.squareup.javapoet.TypeSpecTest | 27 | 150 | 7354 |
| com.squareup.javapoet.ParameterSpecTest | 10 | 32 | 54 |
| com.squareup.javapoet.CodeBlock$CodeBlockJoiner | 3 | 5 | 0 |
| com.squareup.javapoet.CodeWriter | 10 | 85 | 415 |

It is evident that all the classes that have code smells either have higher RFC, CBO or LCOM values.

## **5.1.10 JavaCV**

There are around 223 classes in the JavaCV project. Here are the classes that are having high and lower values of RFC,CBO,LCOM

1. The classes org.bytedeco.javacv.FFmpegFrameGrabber, org.bytedeco.javacv.FFmpegFrameRecorder, and org.bytedeco.javacv.JavaCVCL exhibit notably high values of Coupling Between Objects (CBO), indicating extensive interdependencies between different classes and components within the system.
2. The classes org.bytedeco.javacv.FFmpegFrameGrabber, org.bytedeco.javacv.FFmpegFrameRecorder, and org.bytedeco.javacv.RealSense2FrameGrabber demonstrate the highest values for Response For a Class (RFC) in the dataset. This indicates that these classes may have a significant number of methods and operations associated with them, potentially leading to more complex functionalities.
3. The classes with the greatest Lack of Cohesion in Methods (LCOM) values in the dataset are org.bytedeco.javacv.FrameGrabber, org.bytedeco.javacv.RealSenseFrameGrabber, and org.bytedeco.javacv.FrameRecorder. This suggests that there might be a lot of interaction and close relationship between the methods in these classes, which could result in greater complexity and less encapsulation. Elevated LCOM scores indicate that reworking is necessary to enhance the codebase's modularity and maintainability.

Jdeodrant has reported there are code smells related to TypeChecking, Feature Envy and God Class at multiple classes.

**Table 9: Tabular details of CBO, RFC, LCOM values for Java CV classes having code smells**

|  |  |  |  |
| --- | --- | --- | --- |
| **class** | **CBO** | **RFC** | **LCOM** |
| org.bytedeco.javacv.CameraDevice | 14 | 49 | 25 |
| org.bytedeco.javacv.CameraDevice | 14 | 49 | 25 |
| org.bytedeco.javacv.ColorCalibrator | 6 | 16 | 0 |
| org.bytedeco.javacv.ProjectiveColorTransformer$Parameters | 5 | 25 | 33 |
| org.bytedeco.javacv.MarkerDetector | 19 | 47 | 0 |
| org.bytedeco.javacv.HandMouse | 13 | 52 | 6 |
| org.bytedeco.javacv.ProCamGeometricCalibrator | 13 | 52 | 113 |
| org.bytedeco.javacv.IPCameraFrameGrabber | 10 | 26 | 59 |
| org.bytedeco.javacv.AndroidFrameConverter | 5 | 24 | 2 |

**5.2 Result Summary**

Below is the tabular representation of all our 10 projects and the code smells identified.

**Table 10: Summary details of code smells of 10 Projects**

|  |  |
| --- | --- |
| **Project Name** | **Code Smells** |
| Java SAML Toolkit | Feature Envy, Long Methods, God Class. |
| Slf4j | Long Methods, Feature Envy. |
| Elasticsearch-SQL | LongMethod, Duplicated Code, God Class |
| Spring-Cloud-Platform | LongMethod,God Class and Feature Envy |
| spring-boot-examples | No Code Smells |
| Razorpay Java SDK | God Class |
| logstash-forwarder-java | FeatureEnvy and GodClass |
| Vert.x Core | Duplicated code,Long Method,Feature Envy and God classs |
| JavaPoet | Feature Envy, TypeChecking and God Class |
| JavaCV | TypeChecking, Feature Envy and God Class |

Slf4j, Elasticsearch-SQL, Spring-Cloud-Platform, Spring-boot-examples, Razorpay Java SDK, logstash-forwarder-java, Vert.x Core, JavaPoet, and JavaCV are just a few of the Java projects that have been analyzed. Several trends and findings about the metrics of Coupling Between Objects (CBO), Response For a Class (RFC), Lack of Cohesion in Methods (LCOM), and detected code smells are revealed.

Classes like Util, AuthnRequest, SamlResponse, LogoutResponse, and SettingsBuilder in the Java SAML Toolkit project have high values of CBO, RFC, and LCOM, suggesting possible complexity, difficulties with maintenance, and decreased modularity in the codebase. Similar trends are seen in other projects, such as God Class, Long Method, and Feature Envy, and classes with higher CBO, RFC, and LCOM values also have reported code smells, including Slf4j, Elasticsearch-SQL, and Spring-Cloud-Platform. On the other hand, although these metrics differ amongst classes in projects like Spring-boot-examples and JavaCV, no code smells have been observed, indicating a comparatively cleaner codebase.

Furthermore, several projects exhibit a relationship between high metric values and the existence of code smells, suggesting possible areas for code quality improvement and refactoring. For example, classes with higher RFC values typically have more sophisticated functionalities, whereas classes with higher LCOM values could show problems with method coherence and distributed responsibilities. Moreover, higher metric values are frequently associated with code smells such as God Class, Long Method, and Feature Envy, which point to possible areas for modularity, readability, and maintainability enhancement.

In conclusion, the research shows that classes with higher CBO, RFC, and LCOM metrics frequently have code smells present. This suggests possible places for code optimization and reworking to improve the overall quality and maintainability of the codebase. Improved software design, greater code maintainability, and increased developer productivity can result from identifying and fixing these problems.

## **Key Findings**

* **Metric Analysis and Code Smells**: Metrics that indicate program complexity, interdependencies, and method cohesion include Coupling Between Objects (CBO), Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM).Code smells like Long Method, Feature Envy, and God Class are frequently seen in classes with higher metric values, suggesting possible problems with code quality and modularity.
* **Impact on Software Modularity**: Higher metric values are frequently associated with code smells such as God Class, Long Method, and Feature Envy, which point to possible places where modularity may be violated. By adding complexity, tight coupling, and dispersed responsibilities, code smells can impede the modularity of software by making it more difficult to comprehend, expand, and maintain the codebase.
* **Refactoring for Better Modularity**: Improving software modularity requires locating and fixing code smells and high metric value areas.Code-smell and metric-value reduction refactoring can improve modularity and make the software more scalable, maintainable, and flexible.
* **Project-Specific Patterns and Trends**: Based on its particular architecture, complexity, and development methods, every project shows distinct patterns and trends in terms of metric values and code smells. To increase software modularity and overall code quality, focused refactoring solutions must take into account the interaction between metrics and code smells.

# **Conclusion**

This study's empirical analysis shows a strong correlation between software modularity and code smells. High metric values are a sign of problems with program quality and maintainability, as are code smells like God Class, Long Method, and Feature Envy. To improve software modularity and address these problems, targeted refactoring techniques are necessary, which raises the overall quality and maintainability of the code. Project-specific considerations highlight the significance of customized strategies for reducing code smells that take into account the particular features of every codebase. In summary, this research offers significant perspectives for researchers and software engineers who aim to enhance software quality and maintainability by using efficient code smell detection and refactoring techniques.

## **6.1 Implications**

* **Software Development Practices**: Early detection and mitigation of code smells for better code quality and maintainability are made possible by developers' understanding of the relationship between code smells and software modularity.
* **Refactoring Strategies**: By concentrating on areas with high metric values and code smells, targeted refactoring efforts improve software modularity and optimize the codebase for improved long-term maintainability.
* **Quality Assurance**: Software quality is maintained by preventing the accumulation of code smells by integrating code detection into quality assurance procedures through routine code reviews and automated analysis techniques.

## **6.2 Limitations and Future Recommedations**

This study's possible lack of general application is one of its limitations. Because software projects and domains vary inherently in terms of project characteristics and development techniques, findings might not transfer perfectly across all of them. An additional limitation is the precision of the measures that are employed, including CBO, RFC, and LCOM. These measures could be impacted by things like size and code complexity, which could lead to inaccurate assessments of software modularity. Furthermore, the efficacy of the code smell detection techniques used in this study may differ, which could have an impact on how code smells are identified and categorized inside the codebase.

In the future, investigations into various domains may be conducted in order to gain a deeper understanding of the connection between software modularity and code smells. Improvements in metric analysis approaches have the potential to improve assessment precision and finding reliability. Examining the long-term effects of code smells on maintenance efforts and program modularity may help determine if software systems can be sustained over time. Additionally, research into automated refactoring strategies customized for certain code smells has the potential to expedite the reworking process and lead to more effective software modularity improvements.

**References**

* Chidamber, S. R., & Kemerer, C. F. (1994). *A metrics suite for object-oriented design. IEEE Transactions on Software Engineering*, 20(6), 476-493.
* Tsantalis, N., & Chatzigeorgiou, A. (2009*). Identification of extract method refactoring opportunities for the decomposition of methods. Journal of Systems and Software*,84(10), 1757-1782.
* jDeodorant: Eclipse Plug-in. [Online] Available at: <http://jdeodorant.com/>
* Aniche, M. (n.d.). CK: Tool for calculating Chidamber and Kemerer Java metrics. Retrieved from <https://github.com/mauricioaniche/ck>
* SAML-Toolkits. (n.d.). Java SAML Toolkit [GitHub Repository]. Retrieved from <https://github.com/SAML-Toolkits/java-saml.git>
* SLF4J. (n.d.). Simple Logging Facade for Java (SLF4J) [GitHub Repository]. Retrieved from <https://github.com/qos-ch/slf4j>
* NLPchina. (n.d.). Elasticsearch SQL [GitHub Repository]. Retrieved from <https://github.com/NLPchina/elasticsearch-sql>
* wxiaoqi. (n.d.). Spring-Cloud-Platform [GitHub Repository]. Retrieved from <https://github.com/wxiaoqi/Spring-Cloud-Platform>
* ityouknow. (n.d.). Spring-boot-examples [GitHub Repository]. Retrieved from <https://github.com/ityouknow/spring-boot-examples>
* Razorpay. (n.d.). Razorpay Java SDK [GitHub Repository]. Retrieved from <https://github.com/razorpay/razorpay-java>
* didfet. (n.d.). Logstash Forwarder for Java [GitHub Repository]. Retrieved from <https://github.com/didfet/logstash-forwarder-java>
* Square, Inc. (n.d.). JavaPoet [GitHub Repository]. Retrieved from <https://github.com/square/javapoet>
* ByteDeco. (n.d.). JavaCV [GitHub Repository]. Retrieved from <https://github.com/bytedeco/javacv>
* Eclipse Foundation. (n.d.). Vert.x Core [GitHub Repository]. Retrieved from <https://github.com/eclipse-vertx/vert.x>