Dissecting the Modularity of Java Projects: How Code Smells Influence Coupling and Cohesion

Andrew James Milligan
Jake Mikiewicz
Department of Computer and Mathematical Sciences
Lewis University
Romeoville, United States

***Abstract*—This study aims to investigate the impact of bad code smells on the modularity of software projects, a crucial attribute affecting maintainability and quality. The presence of bad smells within a codebase may significantly influence two key indicators of modularity, Coupling Between Objects (CBO) and Lack of Cohesion in Methods (LCOM). Using a dataset comprising ten open-source Java projects, this project seeks to analyze this relationship and provide insights that may guide better coding practices. Preliminary findings reveal higher CBO and lower LCOM values in classes with bad smells, indicating potential impairment of modularity, and emphasize the need for further research into other factors influencing modularity.**

The goal for this study is to explore the relationship between the presence of code smells in classes and their impact on the modularity of the software. Modularity refers to the degree to which a system's components can be separated and recombined, and it is a critical attribute in assessing the quality of the design in a piece of software. By exploring this relationship, the study aims to provide insights into how code smells may compromise the quality of a system's design, specifically, its modularity.

The first research question the study aims to answer is: "Do classes that are part of code smells exhibit higher coupling?" To explore this, the study utilizes the Coupling Between Objects (CBO) metric. CBO measures the number of classes a given class is coupled with. The idea behind this is that classes involved in code smells might exhibit higher coupling due to their complexity, dependency, or improper design, which could, in turn, reduce modularity. By comparing the CBO of classes with code smells against those without code smells it may be determined if there's a correlation between the presence of code smells and increased coupling. High coupling often signifies excessive dependencies between classes, which can decrease modularity. Taking this into account, it can be reasoned that classes with code smells would present higher coupling due to the nature of these smells leading to tangled, dependent code.

The second research question of the study is: "Does the presence of code smells in classes impact their cohesion?" In this context, the Lack of Cohesion in Methods (LCOM) metric will be employed. LCOM measures how related the methods within a class are. If classes exhibiting code smells tend to have a higher LCOM, it would suggest that these classes are doing too many unrelated things, hence decreasing modularity. High cohesion (or low LCOM) is desirable in object-oriented programming as it indicates that a class has a single, well-defined role. It's plausible that classes with code smells could have lower cohesion due to their possibly unorganized and complex nature, which negatively affects modularity. By comparing the LCOM scores of classes with code smells against those without, the information from the study can be used to assess the impact of code smells on cohesion, and in turn, modularity.

Both of these metrics were chosen due to their direct relevance to the research goal and their well-established status in the software engineering field as measures of software quality. These metrics will allow us to systematically and objectively assess the impact of code smells on the modularity of a software system, providing valuable insights that could guide developers in improving their code quality and design practices.

Two tools were used to collect data for this study, the “CK” tool and JDeodorant. The "CK" tool used in this study, developed by Maurício Aniche, is an efficient Java-based software metrics extraction tool. This tool calculates a wide array of metrics from Java source code, including the size of the class (in lines of code), the number of methods, the number of fields, and the complexity of the methods and constructors. It also computes Chidamber and Kemerer (CK) metrics, such as Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response for a Class (RFC), and Lack of Cohesion in Methods (LCOM).

Specifically for this study, the CK tool provides the metrics necessary to assess the coupling (CBO) and cohesion (LCOM) of the classes within the Java programs selected as subjects. These two metrics are critical in evaluating the modularity of the software under study. The tool's capacity to operate directly on Java code, its ease of use, and its capability to produce detailed and precise metrics data, particularly in the object-oriented software design context, make it an excellent choice for this research.

JDeodorant is an Eclipse plugin used to identify design problems in software, also known as "code smells," and recommend refactorings to improve the software's design. Code smells are indicators of poor design and programming style, often decreasing the understandability and modularity of a system, while increasing its complexity. Given that the objective of this study revolves around the impact of code smells on the modularity of software, JDeodorant is an excellent choice to evaluate the subject programs. It employs an advanced set of algorithms to identify five different types of code smells - Feature Envy, Type Checking, Long Method, God Class, and Duplicated Code. The ability of JDeodorant to pinpoint these issues facilitates the precise evaluation of the presence and impact of code smells in the selected Java programs.

Both tools, the "CK" tool and JDeodorant, are specifically tailored to work with Java, the language in which all the subject programs are written. Additionally, both tools have been widely used and validated in scientific research, which further substantiates their accuracy and effectiveness. The combination of the "CK" suite's robust object-oriented software measurement capabilities and JDeodorant's comprehensive code smell detection ensures a thorough evaluation, making them the best options for this study.

The criteria set for the selection of subject programs for this study were chosen to ensure a robust, meaningful, and practical analysis. The first criterion mandates that the programs be written in Java. This is primarily due to the utility of the “CK” tool and JDeodorant, where both tools are designed specifically to detect CK metrics and other information in addition to code smells only in Java code, which makes Java an essential language requirement for this study.

The second criterion stipulates that each program should be medium to large-sized, requiring a minimum of 5,000 lines of code (LOC). This is because larger projects typically exhibit varying degrees of modularity and contain a diverse variety of code smells. It is reasoned that small programs may not provide enough data for substantial analysis.

The age of the program also factors into the selection process, with a minimum age requirement of three years. This ensures that the program has undergone a series of maintainability tasks over time, making it more probable to contain code smells that have emerged due to modifications and updates.

Another important criterion is the number of contributors. A program with at least three contributors is preferred as when multiple developers work on a project, the potential for differences in coding styles and practices increases, possibly leading to a higher chance of introducing code smells.

The activity of the program also plays a crucial role in its selection. It is required that the program have active contributors and regular commits. Ideally, a program should have a minimum of 500 commits and at least one commit per week within the past year. Actively maintained projects are more likely to represent a real-world scenario of how code smells impact modularity.

Maven compatibility is another criterion to consider. The presence of a pom.xml file ensures that the program could be easily imported into the Eclipse IDE for analysis by JDeodorant. Open source programs are given preference to allow unrestricted access to the classes and code inside the program for an accurate evaluation.

The programs selected for this study align with the pre-established selection criteria, proving them to be suitable for this particular research.

Starting with Apache Commons IO, it is a utility library that assists in developing IO functionality. Given its extensive use, active maintenance, and broad utility, this program fits well within the parameters of the study.

Secondly, Apache Commons Lang serves as a provider of helper utilities for the java.lang API. It is capable of string manipulation, number conversion, and date manipulation. With its wide usage and constant updates, it becomes an ideal candidate for this research.

The third program, Guava, is a set of core libraries that extends to new collection types such as multimap and multiset, immutable collections, a graph library,

and utilities for concurrency, I/O, hashing, among others. Due to its extensive functionality and usage, it stands as a valid candidate for the study.

Next is HDRHistogram, which supports recording and analyzing sampled data value counts across a configurable integer value range with configurable value precision within the range. Given its active maintenance, it is aptly suitable for this study.

Java Websocket is a barebones WebSocket client and server implementation written entirely in Java. Given its use in real-time web applications, it is an excellent candidate for the analysis of code smells and modularity.

JFreeChart, a free Java chart library, is widely used in various sectors and can generate a multitude of charts. Its widespread use and consistent maintenance make it suitable for the study.

Joda Time serves as a quality replacement for the Java date and time classes, a function that is complex and frequently used. This program is an ideal candidate for the study.

Jsoup, a Java library that works with real-world HTML, provides a convenient API for extracting and manipulating data. It is frequently used and broadly functional, making it a fitting selection for the study.

JSQLParser is a SQL parser for Java that translates SQL into a customizable data structure for further processing. Due to its specific utility and active maintenance, it serves as a suitable candidate.

Lastly, Junit4, a common testing framework for Java, is used pervasively and regularly updated, making it ideal for inclusion in this study.

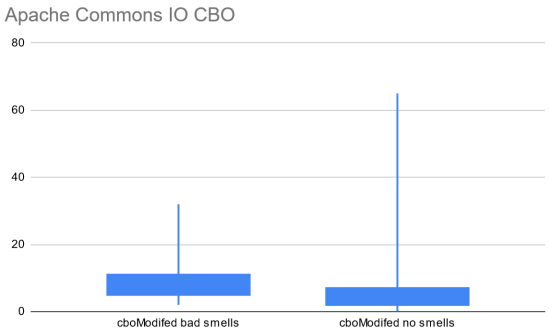
Below a quick reference table for each Java subject program can be found:

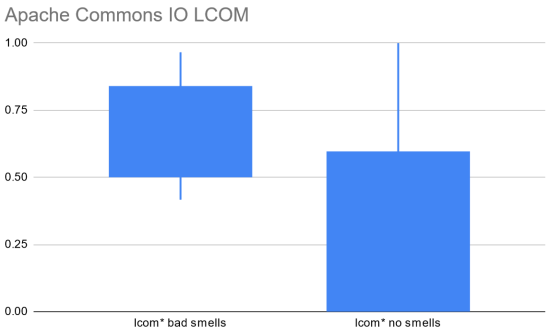
TABLE I. JAVA PROJECT SELECTION CRITERIA

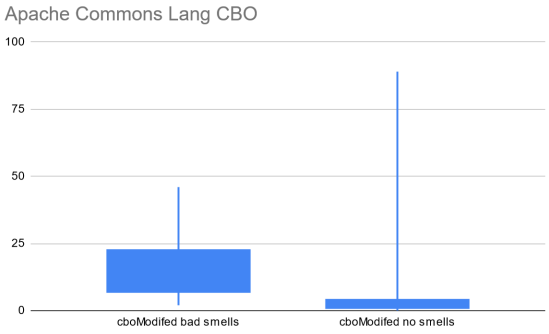
Program	Description	Age	Commits	Contributors
Apache Commons IO	A utility library assisting with IO functionality.	16+ years old	4.2k+	99
Apache Commons Lang	Provides helper utilities for the java.lang API, capable of string manipulation, number conversion, and date manipulation.	16+ years old	7.1k+	190
Guava	A set of core libraries that includes new collection types (e.g., multimap, multiset), immutable collections, a graph library, and utilities for concurrency, I/O, hashing, and more.	13+ years old	6.1k+	289
HDRHistogram	Supports recording and analyzing sampled data value counts across a configurable integer value range with configurable value precision within the range.	4+ years old	764	39
Java Websocket	A barebones WebSocket client and server implementation written entirely in Java.	13+ years old	1.1k+	77
JFreeChart	A free Java chart library, widely used in various sectors, can generate a multitude of charts.	7+ years	4.2k+	25

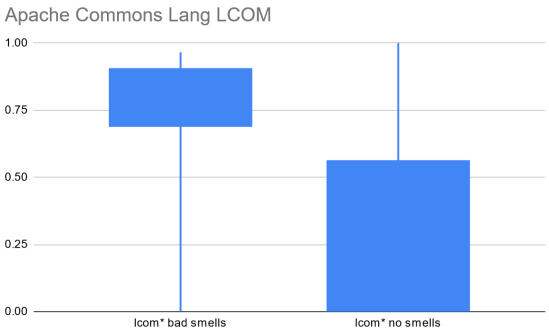
Joda Time	Serves as a quality replacement for the Java date and time classes.	7+ years old	2.2k+	90
Jsoup	A Java library that works with real-world HTML, providing a convenient API for extracting and manipulating data.	3+ years old	1.8k+	104
JSQLParser	A SQL parser for Java that translates SQL into a customizable data structure for further processing.	10+ years old	1.8k+	113
Junit4	A common testing framework for Java.	19+ years old	2.5k+	154

Candlestick charts served as an ideal visual representation for the data collected from the subject programs in this study due to their capacity to encapsulate and express significant amounts of data clearly and concisely. These charts display the distribution of data points, capturing the interquartile range within the "candle" and indicating the minimum and maximum values via the "wicks". They facilitated the comparison of the two distinct sub-groups in the study: classes with and without code smells. Furthermore, candlestick charts visually encode key statistics, allowing a quick grasp of the essential statistical information and potential outliers, and in the case of this study, a comprehensive view of the distribution of the metrics (CBO Modified and LCOM*) for the different categories. Below candlestick charts can be found representing the data:

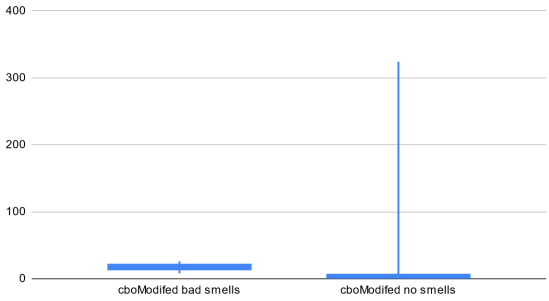




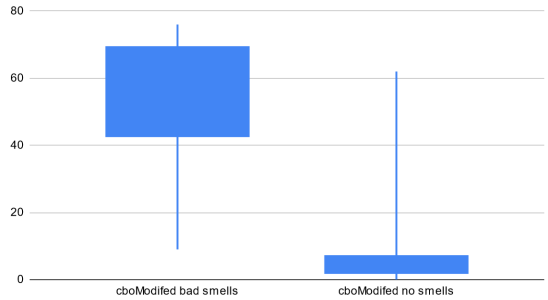




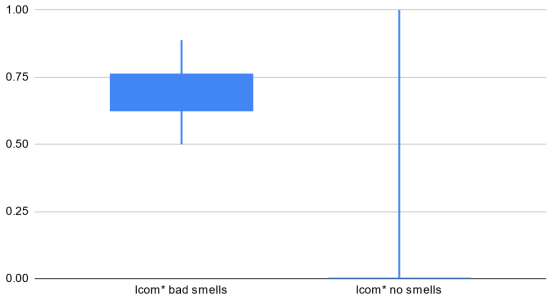
Guava CBO



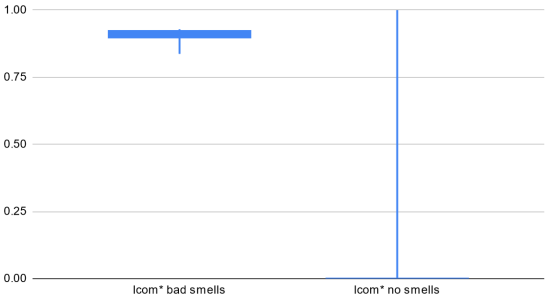
Java Websocket CBO



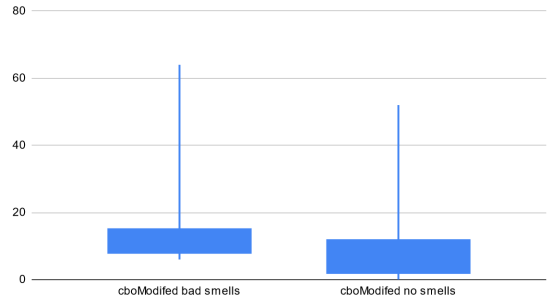
Guava LCOM



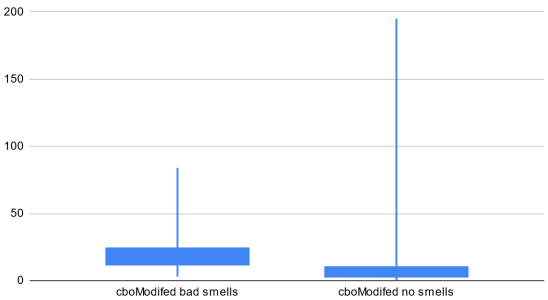
Java Websocket LCOM



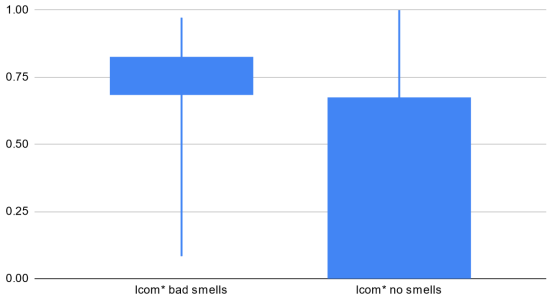
HDRHistogram CBO



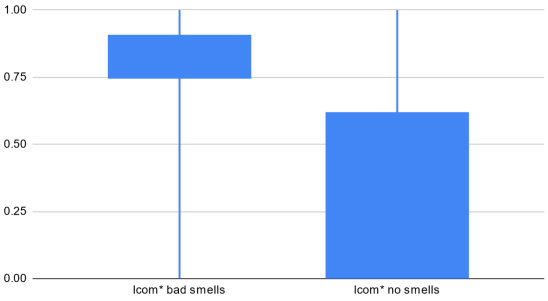
jfreechart CBO

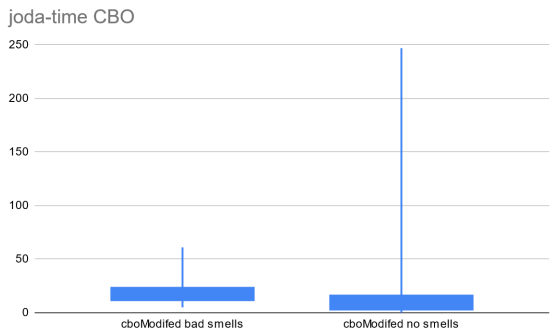


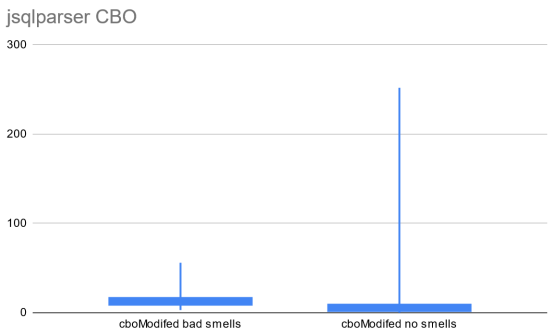
HDRHistogram LCOM

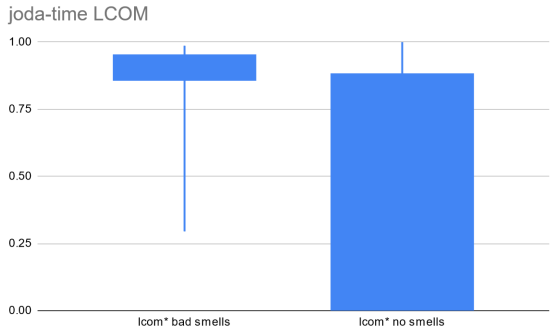


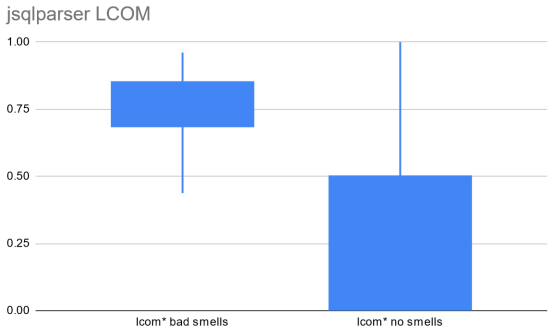
jfreechart LCOM

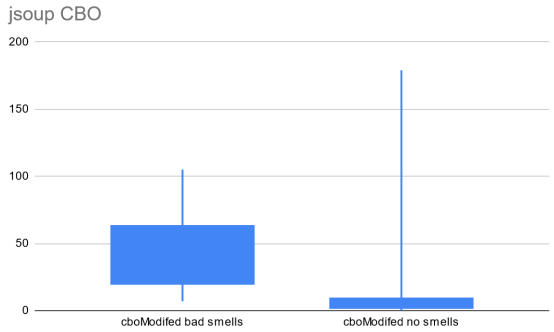


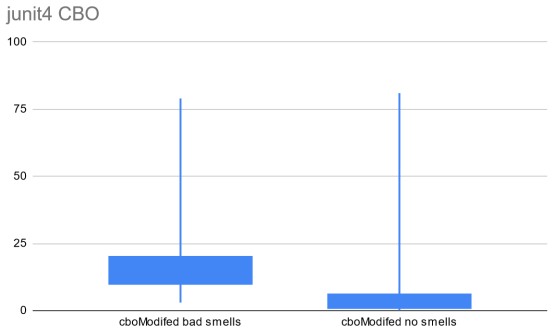


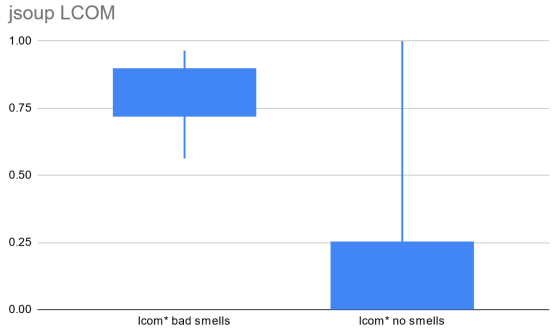


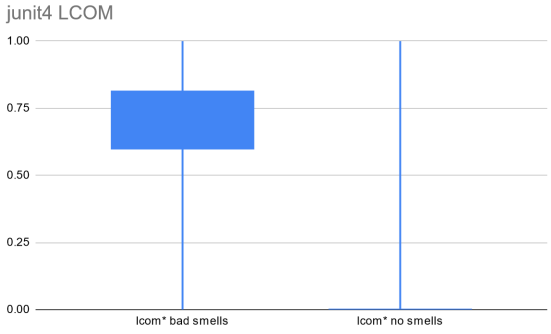












Upon a comprehensive examination of the provided data from ten open-source Java projects several important conclusions can be observed

It appears that classes identified with bad smells exhibit a trend of higher coupling and lower cohesion. In every single project under observation,

the minimum, first quartile, and third quartile of the CBO Modified score for classes flagged with bad smells consistently exceeded those of classes free from bad smells. This pattern signifies that classes with bad smells are likely to have a greater number of dependencies. Consequently, these classes demonstrate tighter coupling, which can complicate maintainability and modularity as changes in one class may necessitate modifications in the dependent classes.

Conversely, an inspection of LCOM* scores reveal that classes with bad smells generally show lower cohesion amongst their methods. This is evident from the higher LCOM* scores associated with classes featuring bad smells compared to those without. Yet, it's noteworthy that high LCOM* scores, which denote lower cohesion, are also observed in classes without bad smells. This observation extends up to the maximum score of 1.00 in some projects, indicating that factors other than bad smells can also contribute to reduced cohesion.

While these overarching trends are consistent across the studied projects, it's important to highlight that the extent of their impact significantly differs. This variation is particularly visible in the disparity of the CBO Modified scores between classes with and without bad smells. For instance, in the Guava project, the maximum CBO Modified score for classes without bad smells significantly overshadows that of classes with bad smells. In contrast, the "jsoup" project's maximum CBO Modified score for classes with bad smells notably surpasses that for classes devoid of bad smells. These discrepancies suggest that the influence of bad smells on modularity may be moderated by other project-specific factors.

While the study offers valuable insights, it's prudent to recognize its potential limitations. Firstly, the limited dataset, drawn from only ten open-source Java projects, may affect the generalizability of the results to diverse software projects or those in other languages. Secondly, the binary classification of smells does not consider the severity or type of bad smells, potentially oversimplifying the real-world complexity. Thirdly, the reliance on CBO and LCOM metrics offers only approximations rather than precise measurements of coupling and cohesion, possibly missing nuances of software modularity. Fourthly, the data can be used to identify correlations between bad smells and modularity characteristics, but this does not equate to causation, leaving room for unexplored confounding factors. Lastly, the study is a snapshot, lacking temporal analysis, which prevents us from understanding how the relationships might evolve over time as the software is maintained and evolved.

This study investigated the influence of bad code smells on the modularity of ten open-source Java projects, providing valuable insights. The data revealed that classes with code smells generally exhibit higher coupling and lower cohesion, suggesting an overall decrease in modularity. However, these results demonstrated variability across projects, highlighting the influence of project-specific factors. While the conclusions hold significance for software design and maintenance strategies, the study's limitations must be recognized, including the limited dataset, binary classification of code smells, reliance on specific metrics, and the lack of temporal analysis. Despite these, the findings of this study represent an important step towards understanding the complex relationship between code smells and software modularity, encouraging further research to refine these insights and augment the understanding of effective software development practices.

REFERENCES

[1] Maurício Aniche, “Java code metrics calculator (CK)”, 2015, Available in https://github.com/mauricioaniche/ck

[2] Nikolaos Tsantalis, “JDeodorant”, 2022, Available in https://marketplace.eclipse.org/content/jdeodorant

[3] Gary Gregory, “Apache Commons IO”, 2023, Available in https://github.com/apache/commons-io

[4] Gary Gregory, “Apache Commons Lang”, 2021, Available in https://github.com/apache/commons-lang

[5] Chris Povirk, “Guava”, 2023, Available in https://github.com/google/guava

[6] Gil Tene, “HDRHistogram”, 2019, Available in https://github.com/HdrHistogram/HdrHistogram

[7] Marcel Prestel, “Java Websocket”, 2022, Available in https://github.com/TooTallNate/Java-WebSocket

[8] David Gilbert, “jfreechart”, 2023, Available in https://github.com/jfree/jfreechart/releases/tag/v1.5.4

[9] Stephen Colebourne, “joda-time” 2023, Available in https://github.com/JodaOrg/joda-time

[10] Jonathan Hedley, “jsoup”, 2023, Available in https://github.com/jhy/jsoup

[11] Tobias, “JSQLParser”, 2023, Available in https://github.com/JSQLParser/JSqlParser

[12] Marc Philipp, “junit4”, 2021, Available in https://github.com/junit-team/junit4