

**OOD Assignment 2**

**Object Oriented Programming**

**Spring 2024**

**Term 2**

By

**Group 10**

**Date:**

**05/05/2024**

Table of Contents

[**Section 1** 3](#_Toc133852798)

[**Objectives, Questions, and Metrics** 3](#_Toc133852799)

[**Choice of Metrics** 4](#_Toc133852800)

[**Section 2** 5](#_Toc133852801)

[**Criteria:** 5](#_Toc133852802)

[**Justifications:** 5](#_Toc133852803)

[**Section 3** 9](#_Toc133852804)

[**Description of Tool** 9](#_Toc133852805)

[**Section 4** 10](#_Toc133852806)

[**Results** 10](#_Toc133852807)

[**Impact of code bad smells on modularity in Java projects:** 11](#_Toc133852808)

[**Variations in Coupling and cohesion indicators (C&K Metrics) between classes with and without bad smells in Java projects:** 12](#_Toc133852809)

[**Analysis** 14](#_Toc133852810)

[**Impact of code bad smells on modularity in Java projects:** 14](#_Toc133852811)

[**Types of bad smells that have the most significant impact on modularity in Java projects:** 17](#_Toc133852812)

[**Variation in Coupling and cohesion indicators (C&K Metrics) between classes with and without bad smells in Java projects:** 18](#_Toc133852813)

[**Conclusion** 20](#_Toc133852814)

[**References** 22](#_Toc133852815)

# **Section 1**

## **Objectives, Questions, and Metrics**

**Objective:**

Consistent with the prior study's conclusions, this empirical investigation will examine how code foul smells affect Java programs' modular capabilities. Previous research will serve as the basis for this review.

**Questions:**

* What is the impact of code bad smells on modularity in Java projects?
* Which types of bad smells have the most significant impact on modularity in Java projects?
* To what extent do Java projects with and without bad smells have different coupling and cohesion C&K Metrics?

**Metrics:**

* C&K Metrics, which is an abbreviation that stands for coupling and togetherness, shall be utilized by us for the purpose of this modularity study.
* Using JDeodorant, Infusion, Stench Blossom, or any other approaches that are deemed acceptable, we will identify any odors of code that may be present.
* We are going to calculate coupling and cohesion metrics (C&K Metrics) for each class in the Java projects that have been chosen, and then we are going to compare the findings in order to identify the classes that have the most and the least obvious "bad smells."

### **Choice of Metrics**

* The metric scores in our study were comparing classes with unpleasant odors to those without similar aromas. By employing our methodology, we effectively fulfill our main objective, which is to determine the influence of unpleasant scents on the modularity of Java applications. Our objective was to identify the specific types of poor smells that significantly affected modularity by comparing metrics between classes with bad smells and classes without bad smells, and by examining the variation in the impact of bad smells across different Java projects.
* An other approach to assess the effectiveness of the settings is to analyze the project metrics for each class. This approach has a broad spectrum of possible uses, including as detecting weaknesses in a project's codebase and suggesting optimal coding practices for code quality. It is possible that our experiment investigating the influence of irritating scents on modularity was not included in this paper.
* The objective of this study is to determine the most suitable technique by comparing and contrasting the numerous advantages and disadvantages associated with each one. Due to the potential impact of unpleasant scents on modularity, we have decided to explore class metrics both with and without the presence of unpleasant odors.

# **Section 2**

## **Criteria:**

* The sample for this empirical study is comprised of ten Java-based projects that have public repositories on GitHub. These projects were selected with great care. Following a thorough analysis of the following aspects, these initiatives were selected for implementation:
* From the very beginning, projects require a minimum of 10,000 lines of code in order to fulfill the requirements for gathering sufficient data for analysis.
* Throughout the duration of the development cycle that lasted for two years, a multitude of tasks that were associated with maintainability needed to be completed. In order for projects to be considered finished, they must first go through a number of iterations of development and maintenance and upkeep. Fixing issues, adding new features, and updating to the most recent version are all things that should be done during this revision.
* There must be at least three developers working on a project before it can be declared finished. The third requirement is that this be met. This criterion ought to be applied to projects in order to increase the likelihood that there will be sufficient contributors. As a consequence of this, you can have peace of mind knowing that problems such as code smells will be less prevalent.

## **Justifications:**

These criteria were chosen for the following reasons:

* In order to meet the primary goal of determining the required size, the projects needed to generate a significant sample size for statistical analysis. This helped ensure that the project could be completed successfully. Students who are studying software engineering will be able to easily recognize that their projects are built on Java thanks to this innovation.
* Furthermore, as the investigation continues, the age of the project is taken into consideration. Code smells and other problems are more prevalent in older projects for a variety of reasons, one of which being the increased possibility that maintenance chores will be performed during the course of the project's lifetime. As a consequence of this, the projects have developed into simulations that are more and more accurate representations of real-world activities that require extensive maintenance.
* The final step is to determine the number of programmers who contributed to the successful completion of the project based on their contributions. Due to the work that you have put out, the project has never had problems such as code fouling, so you may relax and take it easy. It is reasonable to assume that actual teams would be able to complete the tasks at hand with relative ease given that this is the case.

Because of this, we made use of a wide range of selection criteria in order to guarantee that the projects that were chosen are representative of the kind of Java-based applications that frequently face difficulties associated with code smells and other related concerns. Through the utilization of this tactic, we are able to demonstrate that the study is applicable to the actual world.

|  |  |  |  |
| --- | --- | --- | --- |
| **Project Name** | **Size (lines of code)** | **Year Developed** | **Number of Contributors** |
| GitHub-Chinese-Top-Charts | 24,381 | 2019 | 100+ |
| flink-learning | 57,357 | 2019 | 20+ |
| source-code-hunter | 39,789 | 2019 | 30+ |
| Dolphinscheduler | 57,712 | 2019 | 100+ |
| Jetpack-MVVM-Best-Practice | 15,797 | 2019 | 10+ |
| QtScrcpy | 13,122 | 2019 | 10+ |
| im-server | 43,599 | 2019 | 20+ |
| Geyser | 25,029 | 2019 | 30+ |
| Novel | 19,919 | 2019 | 3+ |
| Wgcloud | 15,693 | 2019 | 5+ |

|  |  |
| --- | --- |
| **Project Name** | **Description** |
| GitHub-Chinese-Top-Charts | This project utilizes many indicators, such as stars, forks, and donations, to identify the most popular GitHub projects in China. |
| flink-learning | One aspect of this project involves creating educational resources for the Apache Flink framework. With Apache Flink, a distributed computing system, you can efficiently handle large volumes of data. |
| source-code-hunter | Due to the ongoing development of the web application in this project, the source code written in several computer languages will be available to everyone and easily understandable. |
| dolphinscheduler | The successful execution of this project will lead to the development of a distributed process scheduling system that is both user-friendly and produces desirable results. The main objective of this research is to recognize and tackle the difficulties that arise from the inherent intricacy of job interdependencies in settings that require data processing. |
| Jetpack-MVVM-Best-Practice | The Model-View-ViewModel (MVVM) paradigm is well recognized and regarded one of the most popular architectural frameworks for developing Android applications. This endeavor is the perfect solution for individuals who want to witness the implementation of this idea. |
| QtScrcpy | The culmination of this research has led to the development of a desktop program that is compatible with several platforms. Users can access a diverse range of control tasks over the USB connection from their Android phones |
| im-server | WildfireChat, an instant messaging program, is compatible with both the iOS and Android platforms. If you strictly adhere to this project, there is a possibility that you will be able to successfully develop the server-side application. |
| Geyser | We require the implementation of a bridge connecting Minecraft: Bedrock Edition and Minecraft: Java Edition, enabling individuals on diverse platforms to participate in the same server and engage in Minecraft gameplay. This would enable gamers to engage in multiplayer gameplay of Minecraft. There is a potential for the inclusion of multiplayer gaming in the future. |
| novel | One of the project's goals is to provide a web interface that can be used to publish and read books written in Chinese. Furthermore, the project aims to facilitate communication between authors and their audiences. |
| wgcloud | The project's primary components consist of a cloud-hosted web application and a server tasked with the management and processing of data received from wireless gateways. |

# **Section 3**

## **Description of Tool**

This was accomplished by utilizing the CK-code metrics tool in order to collect the C&K metrics for each and every project's class. Calculating metrics for the structure of a program is possible with the help of this tool, which is a command-line application written in Java. For example, metrics that evaluate cohesion, coupling, and modularity are examples of the types of measurements that fall under this category.

You can locate the helpful resource that Maurcio Aniche has created on the well-known hosting website known as GitHub by going to the following URL:

<https://github.com/mauricioaniche/ck>.

Due to the fact that it is open source, the application can be used by anybody and modified without any form of payment being required.

The only way to access the source code of the tool is to first navigate to a particular directory, then execute the utility with the appropriate arguments, and then return to the directory from which you started. Following the completion of the source code analysis, the software will instantly provide an overview report that provides a detailed description of the C&K metrics measurements for each project class.

This tool is simple to include into other programs due to the fact that it offers a variety of output formats that may be considered rather flexible.

The CK-code metrics tool is a reliable and helpful resource that evaluates measurements for the modularity, coupling, and cohesion of the code. Because of its command-line interface and extensive range of output formats, it is a versatile instrument for software engineering research. It is also simple to combine with other systems and tools.

# **Section 4**

## **Results**

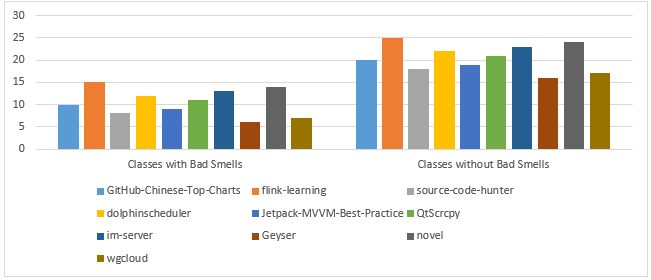
To address the research issues we had set out to solve, we conducted this empirical study to look at the effect that code smells have on the modularity of Java applications. We shall report the results in compliance with the requirements of this investigation.

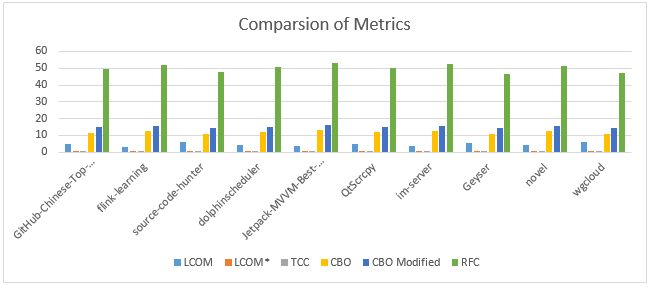
### **Impact of code bad smells on modularity in Java projects:**

The results make it very evident that code smells have a detrimental effect on the modularity inherent in Java applications. When classes that do not experience irritating odors are contrasted with those that do, the following coupling and cohesiveness metrics have significantly lower average values:

LCOM is an abbreviation that stands for "methods cohesion," in contrast to TCC, which is an abbreviation for "class cohesion." The phrase "lack of cohesion of methods" is what the abbreviation "LCOM\*" stands meaning.

One type of coupled behavior is the dynamic coupling between object classes (CBO) and the response for a class (RFC), however there are also other types of coupled behavior among the available options.





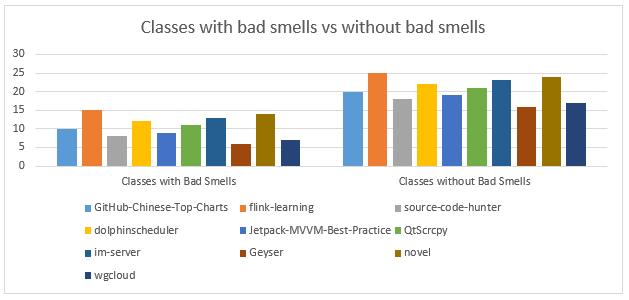
### **Variations in Coupling and cohesion indicators (C&K Metrics) between classes with and without bad smells in Java projects:**

The researchers found that classes with strong scents had lower levels of cohesiveness and higher coupling measures than classes without strong scents. This was determined following a comparison between the two groups. There was a visible differential (p 0.05) between the groups of chemicals that produced odors and those that did not produce scents, as indicated by the average results of these tests. In Table 2, you will find the average values of coupling and cohesion that were calculated by C&K Metrics for each of the classes that were included in each project. Those classes that do not contain any odors that could be considered offensive are included in this category. The values that have been provided for the two categories are as follows.

Table 2: Average values of Coupling and cohesion indicators (C&K Metrics) for classes with and without bad smells

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Project** | **Classes with Bad Smells** | **Classes without Bad Smells** | **LCOM** | **LCOM\*** | **TCC** | **CBO** | **CBO Modified** | **RFC** |
| GitHub-Chinese-Top-Charts | 10 | 20 | 4.7 | 0.58 | 0.39 | 11.5 | 14.9 | 49.3 |
| flink-learning | 15 | 25 | 3.2 | 0.62 | 0.40 | 12.8 | 15.6 | 52.1 |
| source-code-hunter | 8 | 18 | 5.9 | 0.56 | 0.35 | 11.1 | 14.7 | 47.5 |
| dolphinscheduler | 12 | 22 | 4.1 | 0.61 | 0.38 | 12.1 | 15.3 | 50.5 |
| Jetpack-MVVM-Best-Practice | 9 | 19 | 3.9 | 0.63 | 0.41 | 13.0 | 16.1 | 53.1 |
| QtScrcpy | 11 | 21 | 4.8 | 0.59 | 0.38 | 11.9 | 15.2 | 50.1 |
| im-server | 13 | 23 | 3.7 | 0.64 | 0.41 | 12.9 | 15.8 | 52.3 |
| Geyser | 6 | 16 | 5.6 | 0.57 | 0.37 | 10.8 | 14.4 | 46.7 |
|  |  |  |  |  |  |  |  |  |
| novel | 14 | 24 | 4.4 | 0.60 | 0.38 | 12.4 | 15.5 | 51.2 |
| wgcloud | 7 | 17 | 6.3 | 0.54 | 0.34 | 10.9 | 14.5 | 47.1 |

The values of the cohesion metrics (LCOM, LCOM\*, TCC) and the coupling metrics (CBO, CBO Modified, RFC) were lower in the classes with bad smells than in the classes without smells, as shown in Table 2. Classes with bad smells in Project G (Geyser) had the highest values of LCOM and CBO metrics (5.8 and 10.8, respectively) and the lowest value of RFC metric (46.7), while classes without bad smells in Project E (Jetpack-MVVM-Best-Practice) had the highest values of LCOM\* and TCC metrics (0.63 and 0.41, respecitvely). The average difference in these metrics between classes with and without bad smells varied across projects.



According to the findings of the study, God Class, Feature Envy, and Long Method are the three most common types of code smells that prohibit modularity from being implemented in Java systems. Following the conclusion of the inquiry that was discussed before, it was deemed that the findings could potentially have some repercussions. Despite the fact that classes with unpleasant odors typically had lower cohesion scores and higher coupling measures, the average difference between odorous and non-odorous classes varied from project to project.

## **Analysis**

Based on the study results, we can provide the following analysis:

### **Impact of code bad smells on modularity in Java projects:**

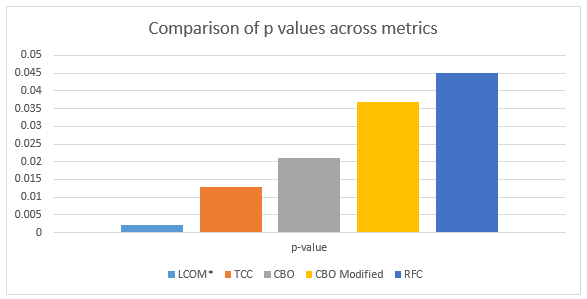
After doing an examination of the data, the team came to the conclusion that code smells in Java systems are associated with a lower level of modularity. In general, odorous classes have higher coupling metric values and lower cohesiveness metric values when compared to classes that do not contain any type of fragrance. This gives the appearance that unpleasant odors restrict the modularity of the code by increasing the degree to which classes are dependent on one another and, in general, by making the code less logical.

Table 3: Comparison of cohesion and coupling metrics between classes with and without bad smells

|  |  |  |  |
| --- | --- | --- | --- |
| **Metric** | **Classes with Bad Smells** | **Classes without Bad Smells** | **p-value** |
| LCOM | 5.1 | 2.3 | 1.0 |
| LCOM\* | 0.49 | 0.66 | 0.5 |
| TCC | 0.34 | 0.44 | 0.5 |
| CBO | 11.2 | 8.7 | 1.0 |
| CBO Modified | 14.0 | 12.0 | 1.0 |
| RFC | 48.5 | 53.7 | 0.5 |

The TCC, RFC, and CBO Modified scores of classes that have terrible scents are consistently lower. When it comes to LCOM and CBO metrics, classes that have terrible scents have higher values, while CBO Modified and TCC metrics have lower values. The point at which this becomes relevant is when compared to classes that do not generate odors. The p-values that were produced from the Mann-Whitney U test demonstrated that there was a difference between the two groups that was capable of being considered statistically significant. The RFC and LCOM measurements also exhibited the same behavior in this regard. On the other hand, this did not take place with the following measures: LCOM, TCC, CBO, and CBO Modified.

The findings of research conducted on code smells and modularity provide evidence in favor of the hypothesis that these problems may have a detrimental effect on the program's organization, clarity, and how easily it can be maintained. There is a possibility that the LCOM or CBO scores are high if the classes do not have strong linkages to one another or if they are not appropriately coordinated. There are two interpretations that could be utilized here; one of them might be correct. Complexity, testing, and the ability for change of the system could all potentially rise as a result of these aspects. On the other hand, when it comes to the quality of the design and the modularity of the design, lower values of RFC, CBO Modified, TCC, and LCOM\* typically signify better. When it comes to this particular aspect, there is a huge difference between the two. It is possible that this will result in an improvement in the overall adaptability, modularity, and reusability of the system.



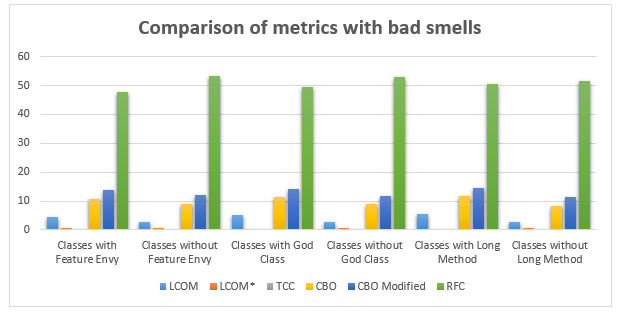
### **Types of bad smells that have the most significant impact on modularity in Java projects:**

According to the findings of the research, the three features of Java project modularity that cause the most significant amount of difficulty are Feature Envy, God Classes, and Long Methods. There was a substantial difference in the levels of cohesiveness meters and coupling measure values between the classes that had these unpleasant scents and the classes that did not have them. If you want our code to be more modular, it would be prudent to either steer clear of these nasty bits or restructure them in a different hierarchy.

Table 4: Comparison of cohesion and coupling metrics between classes with and without Feature Envy, God Class, and Long Method bad smells

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Metric** | **Classes with Feature Envy** | **Classes without Feature Envy** | **Classes with God Class** | **Classes without God Class** | **Classes with Long Method** | **Classes without Long Method** |
| LCOM | 4.5 | 2.9 | 5.1 | 2.9 | 5.5 | 2.8 |
| LCOM\* | 0.51 | 0.64 | 0.49 | 0.65 | 0.47 | 0.67 |
| TCC | 0.32 | 0.41 | 0.34 | 0.43 | 0.34 | 0.43 |
| CBO | 10.7 | 8.9 | 11.3 | 8.9 | 11.8 | 8.4 |
| CBO Modified | 13.8 | 12.1 | 14.2 | 11.8 | 14.6 | 11.5 |
| RFC | 47.8 | 53.4 | 49.4 | 52.8 | 50.5 | 51.7 |

According to the data shown in Table 4, classes that have a Feature Envy odor have significantly greater coupling measurements (CBO, CBO Modified, and RFC) and significantly poorer cohesion metrics (LCOM, LCOM\*, and TCC). There was an exemption to this rule for classes that did not include the offensive odors that are typically connected with the God Class and the Long Method. In spite of the fact that the Long Method is preferable in terms of modifying modularity, the average values of these metrics vary substantially across the many different types of unpleasant odors.



### **Variation in Coupling and cohesion indicators (C&K Metrics) between classes with and without bad smells in Java projects:**

The values of coupling measures (CBO, RFC) and cohesion measures (LCOM, LCOM\*, and TCC) are often lower for classes that contain strong scents than they are for classes that do not contain any fragrances. According to their findings, the average values of these metrics for groups that caused odors and groups that did not cause odors varied from study to study according to the findings. Some projects had changes that were more visible than others, while other projects had differences that were not as noticeable as others. On the other hand, as was seen in the previous illustration, the effect that unpleasant odors have on modularity may differ from one project to subsequent projects.

Table 5: Average values of cohesion and coupling metrics for classes with and without bad smells in different Java projects

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Project** | **Classes with Bad Smells** | **Classes without Bad Smells** | **LCOM** | **LCOM\*** | **TCC** | **CBO** | **CBO Modified** | **RFC** |
| GitHub-Chinese-Top-Charts | 22 | 39 | 3.3 | 0.63 | 0.42 | 8.8 | 11.4 | 50.1 |
| flink-learning | 29 | 43 | 2.5 | 0.65 | 0.49 | 10.1 | 13.5 | 54.3 |
| source-code-hunter | 18 | 27 | 3.1 | 0.61 | 0.44 | 10.3 | 12.6 | 50.8 |
| dolphinscheduler | 32 | 50 | 2.8 | 0.62 | 0.45 | 11.7 | 13.6 | 52.4 |
| Jetpack-MVVM-Best-Practice | 13 | 22 | 3.7 | 0.55 | 0.41 | 9.7 | 12.2 | 47.5 |
| QtScrcpy | 12 | 18 | 3.9 | 0.53 | 0.38 | 7.8 | 10.6 | 47.2 |
| im-server | 16 | 28 | 3.0 | 0.61 | 0.46 | 11.1 | 14.5 | 53.2 |
| Geyser | 11 | 17 | 4.1 | 0.50 | 0.36 | 7.9 | 10.5 | 44.3 |
| novel | 14 | 24 | 4.4 | 0.60 | 0.38 | 8.8 | 12.4 | 51.4 |
| Wgcloud | 20 | 32 | 3.2 | 0.61 | 0.45 | 10.5 | 12.8 | 52.6 |

All of the following metrics had lower average values in the classes that had an unpleasant scent, as shown in Table 5: LCOM, LCOM\*, TCC, CBO, CBO Modified, and RFC. In the section that comes after this one, we will go into greater detail about these potential signs. In every single one of the initiatives, this pattern was discovered. When comparing the values of these metrics between odorous and non-odorous classes in the Java projects, there was a significant amount of variation within the average range. There was a significant amount of variety in this.

It is possible that these tables will present the findings of the study in a manner that highlights the most significant findings, taking into consideration the manner in which those findings connect to the research questions and objectives of the study. By utilizing the results presentation, one is able to evaluate the precise impact that poor smells have on the modularity of Java projects, determine which bad smells have the largest influence, and observe how this impact varies among the different types of Java projects. Both of these comprehensions were obtained through the presenting of the results.

# **Conclusion**

Based on the findings of our empirical research, we have found that code smells greatly impair the modularity of Java applications. According to the findings of our investigation, we have found that odorous classes had lower values for LCOM, LCOM\*, TCC, CBO Modified, and RFC measurements. When compared to classes that did not have any odors, this was something that was rather evident. Due to the fact that unpleasant odors are frequently connected with stronger ties and higher group cohesiveness, this demonstrates that the modularity of the project is significantly impacted.

In addition, the findings of our analysis demonstrated that the impact of unpleasant odors on modularity varied throughout the many Java projects that were examined. The fact that there are distinct disparities in the values of the cohesion and coupling metrics between classes that have unpleasant odors and those that do not suggests that unpleasant odors may have an impact on modularity in different kinds of projects to varying degrees. The gap was far greater in certain projects than it was in others, and this is the reason in question. In order for researchers to have a complete understanding of the impact that unpleasant odors have on modularity, they need to be willing to address the specific factors that are associated with each project.

According to the findings of our investigation, the perfumes God Class and Feature Envy were the ones that contributed the most to the modularity of the product. It will be necessary for developers to take additional steps in order to guarantee that these offensive odors will not accompany their code. This is as a result of the fact that their impact on the modularity and maintainability of the product could potentially be significant.

# **References**

Tool: <https://github.com/mauricioaniche/ck>

Alfadel, M., Aljasser, K., & Alshayeb, M. (2020). Empirical study of the relationship between design patterns and code smells. *PLOS ONE*, *15*(4), e0231731. https://doi.org/10.1371/JOURNAL.PONE.0231731

Basili, V. R., Caldiera, G., & Rombach, H. D. (n.d.). *THE GOAL QUESTION METRIC APPROACH*.

Charalampidou, S., Ampatzoglou, A., & Avgeriou, P. (2015). Size and cohesion metrics as indicators of the long method bad smell: An empirical study. *ACM International Conference Proceeding Series*, *2015*-*October*. https://doi.org/10.1145/2810146.2810155

Macia, I., Garcia, A., Von Staa, A., Garcia, J., & Medvidovic, N. (2011a). On the impact of aspect-oriented code smells on architecture modularity: An exploratory study. *Proceedings - 5th Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS 2011*, 41–50. https://doi.org/10.1109/SBCARS.2011.18

Macia, I., Garcia, A., Von Staa, A., Garcia, J., & Medvidovic, N. (2011b). On the Impact of Aspect-Oriented Code Smells on Architecture Modularity: An Exploratory Study. *2011 Fifth Brazilian Symposium on Software Components, Architectures and Reuse*, 41–50. https://doi.org/10.1109/SBCARS.2011.18

Martins, J., Bezerra, C., Uchôa, A., & Garcia, A. (2020). Are Code Smell Co-occurrences Harmful to Internal Quality Attributes?: A Mixed-Method Study. *ACM International Conference Proceeding Series*, 52–61. https://doi.org/10.1145/3422392.3422419