# What you See is What you Get: Exploring the Relation between Code Aesthetics and Code Quality

Theodoros Maikantis
Dept. of Applied Informatics
University of Macedonia
Thessaloniki, Greece
tmaikantis@uom.edu.gr

Iliana Natsiou
Dept. of Applied Informatics
University of Macedonia
Thessaloniki, Greece
iis20043@uom.edu.gr

Apostolos Ampatzoglou
Dept. of Applied Informatics
University of Macedonia
Thessaloniki, Greece
a.ampatzoglou@uom.edu.gr

Alexander Chatzigeorgiou
Dept. of Applied Informatics
University of Macedonia
Thessaloniki, Greece
achat@uom.edu.gr

Stelios Xinogalos
Dept. of Applied Informatics
University of Macedonia
Thessaloniki, Greece
stelios@uom.edu.gr

Nikolaos Mittas
Hephaestus Laboratory, Dept. of Chemistry, School of Science, International Hellenic University
Kavala, Greece
nmittas@chem.ihu.gr

## ABSTRACT

Software artifacts and source code are often viewed as pure technical constructs aiming primarily at delivering specific functionality to the end users. However, almost each line of a computer program is the result of developers' craftsmanship and thus reflects their skills and capabilities, but also their aesthetic view of how code should be written. Additionally, by nature, the code is not an artifact that is managed by a single person: the code is peer-reviewed, in some cases programmed in pairs, or maintained by different people. In this respect, the first impression for the quality of a code is usually a matter of "*reading*" the aesthetics of the code and then, diving into the details of the actual implementation. This "*first-look*" impression can psychologically bias the software engineer, either positively or negatively and affect his/her evaluation. In this article we investigate whether code beauty (or code aesthetics) must be valued in software programs, as a proxy to the quality of the code. Specifically, we attempt to relate the notion of code beauty with code quality metrics. For this purpose, we catalogued existing beauty measures (assessing the aesthetics of images, objects, and alphanumeric displays), tailored them to match code beauty, and correlated them to structural properties that are related to Technical Debt Interest (such as coupling, cohesion, etc.). The results of the study suggest that some code beauty metrics can be considered as correlated to Technical Debt Interest; and therefore, the "*first-look*" impression might to some extent be representative of the quality of the reviewed code chunk.



*TechDEBT' 24 April 2024, Lisboa, Portugal*
 978-1-4503-0000-0/18/06...$15.00
https://doi.org/XXXX

## CCS CONCEPTS

• Software and its engineering→Software creation and management→Software verification and validation→ Empirical software validation

• Software and its engineering→Software creation and management→Software post-development issues→Maintaining software

## KEYWORDS

Code beauty, Code aesthetics, Code Quality, TD Interest

**ACM Reference format:**

T. Maikantis, I. Natsiou, A. Ampatzoglou, A. Chatzigeorgiou, and S. Xinogalos, and P. Kyriakakis, "What you See is What you Get: Exploring the Relation between Code Aesthetics and TD Interest". In *Proceedings of ACM 7th International Conference on Technical (TechDEBT' 24). ACM, New York, NY, USA, 10 pages.*

## 1 INTRODUCTION

Modern software development methodologies rely heavily on the human-aspects of software engineering, dictating the use of practices (such as pair-programming [1] and code reviewing [2]) that require the cross-checking of code by engineers different than those that have originally written the code. On top of these inter-developer human-code interactions, maintenance tasks are in many cases assigned to developers, agnostically to who is the original contributor of the code. Reading, understanding, and changing the code that you have not authored is a task that is far more challenging than changing your own code [2], promoting the understandability of code as an important factor for keeping maintenance cost [3][4] at an affordable level. In the literature the "*wasting*" of maintenance effort, due to internal poor quality (such as readability and understandability) is communicated as Technical Debt (TD) Interest payments.

In the first minutes of code inspection, review, or maintenance, the developers' assessment on the anticipated effort for the task (and therefore the eagerness of the developer to start the task) can be biased by the "*first-look*" of the code [5][6]. This "*first-look*" is not related to the content or the quality of the code but is mostly biased by treating (looking at) the code in its entirety as an image or as a shape. Based on this assumption, in this paper, we aim at exploring if this inherent psychological bias that can be aroused by the "*first-look*" at the code can indeed mislead the software engineer, or if this first impression is in the correct direction. Inspired by the study of aesthetics in other scientific disciplines, we are exploring if "*code beauty*" can be correlated to the quality of the code, and more specifically to Technical Debt Interest.

The definition of "*beauty*" has been a matter of debate for many years and has been a distinct branch of philosophy dealing with the nature of art and beauty. We usually perceive something as beautiful when it is pleasing to the senses, especially eyesight. Many philosophers, psychologists and other scientists discuss if beauty can be objective, or if it is always subjective: "*Beauty is in the eye of the beholder"* [7]. Between these views, a common ground was found supporting that the aesthetic evaluation of an object is related to the observer's memories and feelings. Beauty is not only limited to objects or entities; it can also be observed in text and mathematical equations. There have been brain scans implying that seeing mathematical equations can sometimes evoke the same sense of beauty as masterpieces of painting and music suggesting that there is a neurobiological basis to beauty [8]. For example, Euler's identity is often cited as an example of deep mathematical beauty, due to its simplicity, involvement of only three arithmetic operations and five fundamental constants. Although constants such as $e$, $\pi$ and $i$ are complex concepts, they are beautifully linked by a simple and concise formula. Mathematicians usually describe a pleasing proof or technique as elegant, especially when it is concise and relies on a minimum number of previous assumptions and when it can be generalized to solve a variety of problems. As a result, we can assume that a similar case exists for source code aesthetics [9]—see Section 2.

## 2 RELATED AND BACKGROUND WORK

***Code Beauty***: A prominent example of discussing beauty in coding, is provided by the book "*Beautiful Code*" [9], which supports the importance of code appearance and studies the effects it has on its performance. In this book, the beauty of the source code is related to performance, elegance, simplicity, and understandability of the final software product [9].

Coleman et al. [10] explored the correlation of code beauty and software maintainability. Based on their findings, code beauty and maintainability seem to be two intricately connected aspects of software development. Beautiful code, characterized by clarity, simplicity, and adherence to best practices, inherently contributes to improved maintainability. When code is aesthetically pleasing, it becomes more readable and understandable for developers [10]. The relationship between code beauty and maintainability underscores the idea that writing elegant, clear, and well-structured code not only enhances the development process but also ensures that software remains adaptable and sustainable over time. Another study investigates the relation of beauty with the quality of UML diagrams. Specifically, it discussed various design criteria for UML class diagrams and emphasized the relation between the aesthetic quality of a diagram and the quality of the object-oriented design it represents [11]. Finally, Aldenhoven et al. [12] highlight the impactful relationship between beautiful software architecture and developer productivity, emphasizing the positive influence on team dynamics and product quality. It stresses the importance of beauty, since ugly software architecture tends to frustrate and demotivate developers, thus decreasing productivity.

***Beauty on Objects, Mathematics, and Text***: Since code beauty and functionality are independent, we reuse measures for object, mathematical and text aesthetics. During the last centuries, several measures that define the beauty of objects have been proposed by scientists and artists. For example, balance was for the first time referred to as a beauty trait back in the 5$^{th}$ century when the famous Greek sculptor Polykleitos made his statue Doryphoros. A relation of beauty and harmony to symmetry was found during Renaissance after the creation of the Vitruvian Man, a painting by the Italian artist Leonardo Da Vinci. Other traits found in the 19$^{th}$ century include quality and the form of the whole object which is affected by its color, proportion, and size [13]. The elements of regularity, mathematical harmony, order, and shortness along with symmetry, size and quality which were also mentioned by artists [14]. Another sector of beauty, mathematical beauty, introduced the elements of understandability and simplicity which turned out to be crucial for the beauty evaluation of mathematical formulae [15]. These two factors were also mentioned by Wertheimer, i.e., the founder of the Gestalt theory [16], a theory which has been further extended by many other researchers.

## 3 DEFINING AND MEASURING CODE BEAUTY

For the aesthetic evaluation of code, we consider the screen as a generic interface and evaluate it as such. An important resource for the aesthetic evaluation of interfaces has been published by Ngo [17]. We note that in terms of contributions, our work goes to a different direction from Ngo [17], in the sense that his metrics targeted UI, whereas ours are tailored for source code assessment. Considering that we are interested in the overall aesthetic evaluation of code, we consider the complete length and height of a code file as our hypothetical interface borders. Also, another important aspect of the interface is its quadrants. In Figure 1, we see the screen split into the quadrants of Upper Right (UR), Upper Left (UL), Lower Right (UR) and Lower Left (LL), playing an important role in the calculation of our selected measures. All the selected measures are presented below briefly, and a detailed presentation is given in an online supplementary material[1].

[1] https://users.uom.gr/~a.ampatzoglou/aux_material/techdebt24.pdf

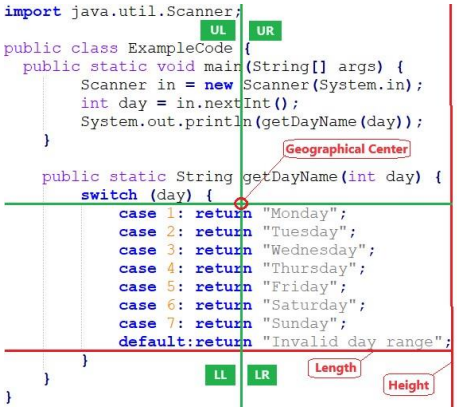


**Figure 1.** Quadrants, Axis of symmetry & geographical center

## 3.1 Simplicity (SMM)

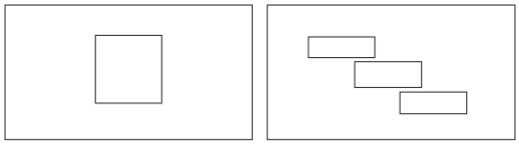


**Figure 2.** Left image is simpler because of fewer alignment points

Simplicity refers to the understandability of the code's layout based on its alignment points and non-blank lines. Simplicity is achieved when all non-blank lines and their alignment points are the least possible. Simplicity involves counting the number of different rows or columns on the screen that are used as starting positions of alphanumeric data items [17]. In Figure 2 we see two examples of entities with the left being simpler because of less elements and alignment points. Higher SMM values indicate greater simplicity, while lower values suggest more complex and potentially harder-to-read code. The code in the top part of Figure 3 scores 4.3% in SSM, whereas the code on the bottom scores 12%, due to its reduced line count and fewer number of vertical and horizontal alignment points.

```java
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner reader=new Scanner(System.in);
        System.out.println("Enter a day");
        int day=reader.nextInt();
        reader.close();

        if(day<4) {
            if(day<3) {
                if(day<2) {
                    System.out.println("Mon");
                }else {
                    System.out.println("Tue");
                }
            }else {
                System.out.println("Wed");
            }
        }else {
            if(day<7) {
                if(day<5) {
                    System.out.println("Thu");
                }else if(day<6){
                    System.out.println("Fri");
                }else {
                    System.out.println("Sat");
                }
            }else {
                System.out.println("Sun");
            }
        }
    }
}
```

```java
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        String[] days={"Mon","Tue","Wed","Thu","Fri","Sat","Sun"};
        Scanner reader=new Scanner(System.in);
        System.out.println("Enter a day");
        int day=reader.nextInt();
        reader.close();

        System.out.println(days[day-1]);
    }
}
```

**Figure 3**. Days of Week Code Examples

## 3.2 Symmetry (SYM)

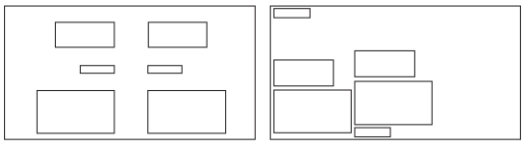


**Figure 4.** Left image is more symmetrical

Symmetry is axial duplication. It measures how well characters of a code file exhibit horizontal, vertical, and radial symmetry on the screen. To achieve Symmetry, all units must be perfectly mirrored vertically, horizontally, or diagonally on all screen quadrants. In Figure 4 we see two examples of simple drawings, with the left exhibiting higher symmetry.

```java
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {
        ArrayList<String> cars;
        cars = new ArrayList<String>();

        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");

        for (String i : cars) {
            System.out.println(i);
        }
    }
}
```

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");

        System.out.println(cars.get(0)+"\n"+cars.get(1)+"\n"+cars.get(2)+"\n"+cars.get(3));

    }
}
```

**Figure 5**. Car Brands Example

To calculate Symmetry, we use a modified version of the formula proposed by Ngo [17]. An example is provided in Figure 5. While both codes share comparable levels of horizontal symmetry (0.353 for top / 0.156 for bottom), vertical (0.648 for top / 0.844 for bottom), and radial symmetries (0.648 for top / 0.844 for bottom), the first code surpasses the second in terms of the average value of symmetry (45% for top, against 38% for bottom).

## 3.3 Sequence (SQ)

Sequence is a metric that assesses the distribution of lines in the code and rates how well it follows the reading pattern commonly used in Western cultures (the eye, trained by reading, starts from the upper left and moves back and forth across the display to the lower right.). In Figure 6 we see two examples of entities with

"good" and "bad" sequence. Example (a) guides the viewer according to the desired reading pattern, whereas example (b) has a more irregular pattern.

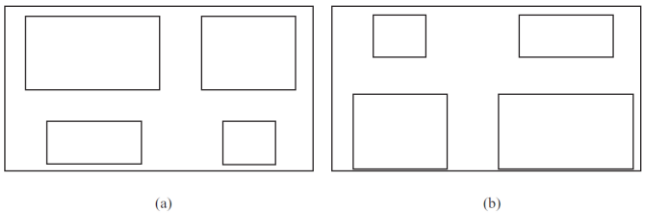


**Figure 6.** Left image follows a common reading sequence pattern

To calculate the value of Sequence, we use the formula given by Ngo [17]. The basic aspects that influence SQ are Quadrant Weighting and Vertical Alignment of lines within each quadrant. The former corresponds to the importance of each quadrant in reading, whereas the latter refers to the lines of code within each quadrant. A higher SQM value suggests that the code distribution is closer to the expected reading pattern. For the code in the bottom part of Figure 3, SQ scores 100%, whereas the code in the top part of Figure 3 scores 50%. The bottom code adeptly distributes characters across the four quadrants, aligning seamlessly with the specified reading pattern. In contrast, the top code exhibits a moderate Sequence value, attributed to a relatively consistent number of characters across the quadrants from upper left to lower right.

## 3.4 Balance (BM)

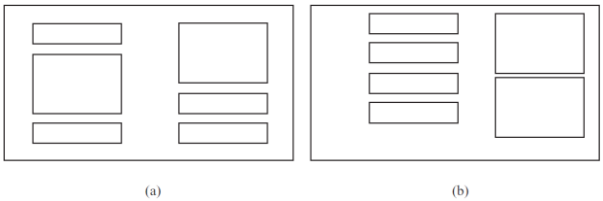


**Figure 7.** Left image is more balanced

Balance is a metric related to the visual weight of code, particularly how the length and positioning of lines affect the perceived visual balance. Larger blocks of code appear "heavier" than smaller ones, thereby changing the perception of the viewer. To achieve Balance, all elements located above and below the center of the frame on the y-axis must have the same weight. The same applies for the elements placed at both sides of the center on the x-axis [17]. In Figure 7 we see two examples of entities with "good" and "bad" balance. Example (a) seems to be balanced, whereas example (b) is clearly imbalanced towards the right.

A higher total BM value suggests that the code layout is visually balanced, with lines distributed in a way that creates an aesthetically pleasing and well-structured appearance. The top part of Figure 3 scores 75% in terms of BM with BMvertical = -0.15 and BMhorizontal = 0.35. These variables assess the visual weight distribution in code, indicating whether the code is relatively heavier on the left or right (BMvertical) or on the top or bottom (BMhorizontal). If either of those values is 0, it indicates a balance in that plane. The bottom part scores 53.5% with BMvertical = 0.78 and BMhorizontal = 0.15, because of a noticeable imbalance between characters placed to left and right of frame center.

## 3.5 Equilibrium (EM)

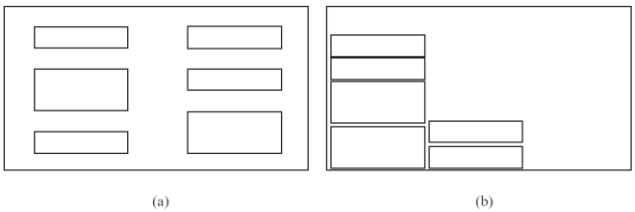


**Figure 8.** Left image exhibits a higher EM

Equilibrium measures how well the center of mass of the code aligns with the geographic center of the frame. To achieve ideal Equilibrium, this difference must be equal to zero [17]. In Figure 8 we see two images, where example (a) is centered to the frame, whereas example (b) has a mass shifted towards the lower left part of the frame. Equilibrium can be defined along the X-Axis and along the Y-Axis. Equilibrium along the X-Axis (EM_x) measures the difference between the center of mass of the code in the X direction (horizontal) and the center of the frame. Similarly, the equilibrium along the Y-Axis (EMy) assesses the equilibrium along the vertical axis (Y-axis). A higher EM value suggests that the code's center of mass aligns well with the center of the frame, which contributes to a more visually balanced and aesthetically pleasing code layout. To discuss EM, we use the example codes of Figure 3. The code on the top-side demonstrates minimal deviation from equilibrium on both the horizontal and vertical axes, indicating a proximity between the center of the code's mass and the geographic center of the frame. Consequently, the Equilibrium value for this code is notably high (EM = 93%, EMx = 0.04 and EMy = -0.1). In contrast, the bottom-side code exhibits lower equilibrium values on both the x-axis and y-axis, signifying a considerable distance between the center of mass and the geographic center (EM = 85.5%, EMx = -0.26 and EMy = -0.03).

## 3.6 Regularity (RM)

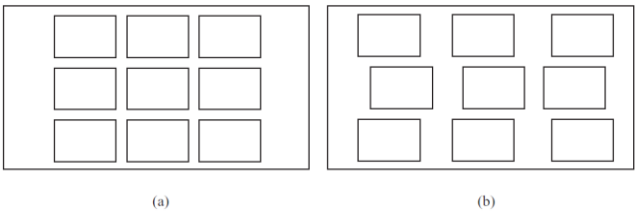


**Figure 9.** Left image is more regular

Regularity assesses the degree of consistency in alignment points and spacing among the distribution of lines. The basic aspects that influence RM are Alignment Regularity and Spacing Regularity. It aims to determine how well the lines align (Alignment Regularity) and are consistently spaced within the code (Spacing Regularity) [17]. In Figure 9 we see two examples of entities, where example (a) contains consistently spaced entities vertically and horizontally, whereas example (b) does not. Using the example of Figure 5, the code on the bottom-side scores 45% in RM (align-

ment 35% and spacing 55%), whereas the code on the top-side scores 36% (alignment 34% and spacing 37%).

## 3.7 Rhythm (RHM)

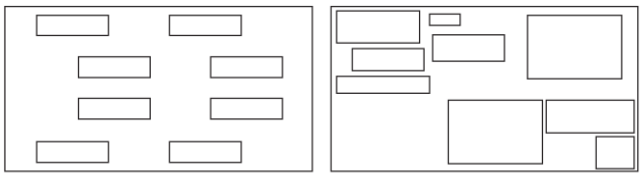


**Figure 10.** Left image has a higher rhythm

This measure evaluates whether the lines follow a distribution pattern and assesses the variety in both alignment points and line sizes. Unlike previous measures, good Rhythm value is achieved when there is diversity in the code layout as in a variety in both the alignment points and the lines sizes [17]. In Figure 10 we see two examples of entities with "good" and "bad" rhythm. Example (a) contains diverse, yet structured and aligned entities, whereas example (b) has a very disorganized structure.

The basic aspects that influence RHM are Rhythm in the X-Axis, Y-Axis, and Covered Area. Rhythm in the X-Axis (RHMx) evaluates the variety in the x-distances between lines in different quadrants. Similarly, rhythm in the Y-Axis (RHMy) quantifies the variety in the y-distance between lines in different quadrants. A higher RHM value suggests that the code exhibits a more diverse and aesthetically pleasing layout, contributing to improved code readability and visual appeal. In order to attain a high RHM value, we need RHMx and RHMy to be as low as possible. To demonstrate RHM, we use the example of Figure 5. The bottom-side code exhibits minimal disruption to rhythm, since the RHM values on both the horizontal (36%) and vertical axes (37%), as well as across the entire frame area (25%) are low, This occurs because the lines of this code are thoughtfully distributed among the four quadrants, displaying a consistent and organized arrangement (RHM equals 66%). The top-side code registers higher values in the mentioned measures (62%, 58%, and 50% respectively), with its lines appearing less systematic (RHM = 42%).

## 3.8 Deviation of the Center of Mass (DCM)

The Deviation of the Center of Mass (DCM) refers to the distance between the geographic center of the code and its own center of mass [18]. The center of mass is determined by the distribution of lines within the frame. The DCM metric calculates the Euclidean distance between the normalized coordinates of the geographic center of the content and its center of mass [19]. Calculating DCM relies on the normalized values for the Center of Mass for the X (COMx) and Y (COMy) axis. For the final DCM measure, the Euclidean distance between the COMx, COMy and the ideal center of the frame (0.5, 0.5) is computed. In summary, the DCM metric gives a numerical measure of how well-distributed the lines are within the frame. It provides insights into the balance and symmetry of the code layout, with a lower DCM indicating a more centered and balanced distribution. Although DCM is very similar to EM, since they both measure the balance of a code's structure, they do so from slightly different perspectives. DCM measures the Euclidean distance from the geographic center to the center of mass of the code thus giving an absolute measure of how far the "weight" of the code (based online lengths) deviates from the center. EM, on the other hand, measures the difference between the center of mass and the geo-graphic center, but it normalizes this difference by the dimensions of the frame. This gives a relative measure of balance, indicating how much the "weight" of the code deviates from the center relative to the size of the frame. So, while both metrics use the same reference point (the geographic center), they provide different perspectives on the balance of the code's structure. DCM gives an absolute measure of deviation, while EM provides a relative measure of balance. This subtle difference can offer complementary insights when analyzing the structure of a code. Using the example of Figure 3, the top-side code scores 7% (COMx: 52% and COMy: 43%) and the bottom-side code scores 14% (COMx: 35% and COMy: 48%). The top-side code is well-mirrored in terms of the center of the frame, signifying that the center of mass is near the geographic center of the code.

## 3.9 Density (DEN)

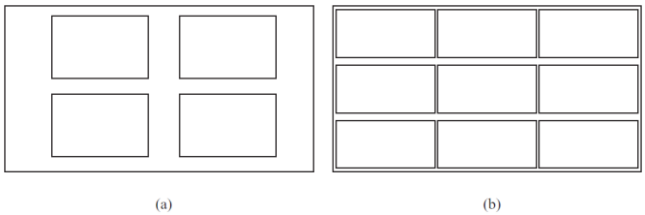


**Figure 11.** Left image has a higher density

Tullis [20] introduced a set of measures to evaluate text user interfaces, among them the Density Measure later used to evaluate the aesthetics of user interfaces. Density depicts the screen coverage with data, in the case of text is the percentage of the screen covered with characters. A simplified formulation by Ngo et al. [17] refers to objects in the frame. Density is calculated by the division of the Area Covered by Lines (a-covered) with the Area Occupied by the Frame (a-frame). The Area Covered by Lines represents the total area occupied by the lines in the code after removing any leading spaces or indentation. In other words, it calculates the space taken up by the actual content of the code. The Area Occupied by the Frame represents the total area covered by the frame that encloses the code. The frame refers to the boundary or container that holds the code, which, in our case, is the whole code file. This ratio provides an indication of how densely written the code is in relation to the available space provided by the frame. In the context of code aesthetics, lower density values are often preferred, as they suggest a more visually pleasant and readable layout. Code that is too densely packed may be challenging to read and understand. In Figure 11 we see two graphic examples with the left being substantially less dense. For the code of Figure 5, Density equals 23% for the top-side code (a-covered = 235 and a-frame = 1026), and 34% for the bottom-side code (a-covered = 254 and a-frame = 740).

# 4 CASE STUDY DESIGN

To explore the relation of the proposed beauty metrics with structural quality, we have performed an exploratory case study on open-source software projects. More specifically, we explore which of these metrics are related to code quality calculation parameters. The case study is conducted, and reported, based on the linear analytic structure [21].

## 4.1 Research Objectives and Research Questions

The goal of this study is to investigate several code beauty metrics and to ascertain if a correlation exists between code aesthetics and code quality. Based on this goal, we have set a main research question (***RQ***), as follows: "*To what extent does code beauty correlate to TD Interest parameters?*". To answer this research question, we explore if there are specific code beauty metrics that exhibit stronger correlations with TD Interest calculation parameters.

## 4.2 Cases and Units of Analysis

This case study is organized as an embedded single case study. The subjects for this study are open-source software projects, and the units of analysis are classes. The reporting is performed cumulatively for the complete dataset, and we do not separate per software project. The decision to not explore the project parameter does not influence the validity of the study in the sense that our work is focusing on specific files and the studied relation is not expected to change, due to organizational aspects of the project.

**Table I**. Selected Projects

| Name | Lines of Code | Number of Classes |
|---|---|---|
| Antlr4 | 44,613 | 421 |
| Conductor | 53,488 | 507 |
| DD-trace-java | 175,482 | 2,686 |
| Dolphin Scheduler | 107,772 | 1,873 |
| Druid | 981,231 | 7,140 |
| Dubbo | 200,404 | 3,407 |
| Incubator Sea Tunnel | 92,462 | 1,797 |
| Pulsar | 527,708 | 3,899 |
| Rocket MQ | 169,725 | 1,698 |
| Sky-walking | 75,137 | 1,722 |
| Stream-pipes | 60,712 | 2,144 |

Our dataset consists of more than 27,000 classes, which correspond to the complete set of classes from 11 Java open-source projects. The selected projects along with some basic descriptive statistics are presented in Table I. The selection of projects has been reused from the work of Nikolaidis et al. [22]: i.e., the projects are written in Java to enable our static analysis, they are having active development to ensure that they are up-to-date subjects, and they are having substantial history that uses a structured committing process to ensure that they are mature projects.

## 4.3 Data Collection

The data collection process can be split into two parts, the first one for calculating the Code Beauty metrics, and the second for the collection of the TD Interest parameters.

*Part A-Code Beauty Metrics*: The first part of data collection aimed at calculating the metric scores of code beauty for the classes of the dataset. Since the presented metrics are novel, we needed to develop a tool to automate their calculation. The tool is available online[2] and has been well-tested in various settings: (a) small-scale projects for which the beauty metrics were calculated manually and contrasted to the automatically extracted scores; (b) large-scale projects for identifying abnormal (outside metric range) score; and (c) checking common static analysis mistakes that we have catalogued over the years in other source code parsers. The code beauty metrics have been calculated in the last version of the selected projects.

*Part B-TD Interest Parameters*: According to Arvanitou et al. [23] the calculation of TD Interest is an open research problem, and no established way for quantifying TD Interest exists in the state-of-practice. To this end, we have preferred not to correlate Code Beauty with TD Interest, but with the parameters of its calculation, and more specifically with structural quality metrics. According to FITTED [24], TD Interest is related to the major quality characteristics that are maintainability predictors: i.e., coupling, complexity, cohesion, and size. To quantify these quality characteristics, we have selected the following metrics:

- **Cyclomatic Complexity** (CC): This metric measures the complexity of a program by counting the number of linearly independent paths through the code. Cyclomatic complexity is considered as the state-of-the-art complexity metric because it is well-established and well-tested in terms of its relation to maintainability. CC, when compared to other complexity metrics, considers the internals structure of a method, enabling the capture of the actual complexity of the class.
- **Lack of Cohesion of Methods** (LCOM): LCOM measures the lack of cohesion among methods within a class. It quantifies the number of method pairs that do not share any instance variables. This metric has been selected since: (a) high cohesion is one of the most important principles of object-orientation, and (ii) lack of cohesion directly implies the existence of the large class "bad smells", which urges for the application of well-known refactoring.
- **Message Passing Coupling** (MPC): MPC measures the number of distinct methods called from a class. MPC has been selected since it is the only coupling metric that captures both coupling volume (number of relationships) and coupling intensity (how closely connected the two classes are). An additional characteristic of MPC is that it counts coupling intensity using the discrete count function, and therefore is not biased from the number of times one method is being called.
- **Lines of Code** (LoC): This is a simple metric that counts the number of lines in the source code. It is a basic measure of the size of the codebase. LoC can give an indication of the scale of a class or project. LoC is used in almost all maintainability studies and is accredited as a top predictor of maintenance load, which is a basic component of TD Interest calculation.

[2] https://github.com/teomaik/Code_Beauty_Calculator

The selected metrics have been indicated by previous as the optimal maintainability predictors: Riaz et al. [25] have performed a quality assessment of maintainability models, through a quantitative checklist, to identify studies that provide reliable evidence on the link between metrics and maintainability. Among the studies with the highest scores were those of van Koten and Gray [26] and Zhou and Leung [27]. Both studies have been based on two metric suites proposed by Li and Henry [28] and Chidamber et al. [29]. The employed suites contain metrics that can be calculated at the source-code level, and can be used to assess well-known quality properties, such as inheritance, coupling, cohesion, complexity, and size. To calculate the metric scores, we used Metrics Calculator, a well-tested and stable tool for calculating quality metrics for Java code.

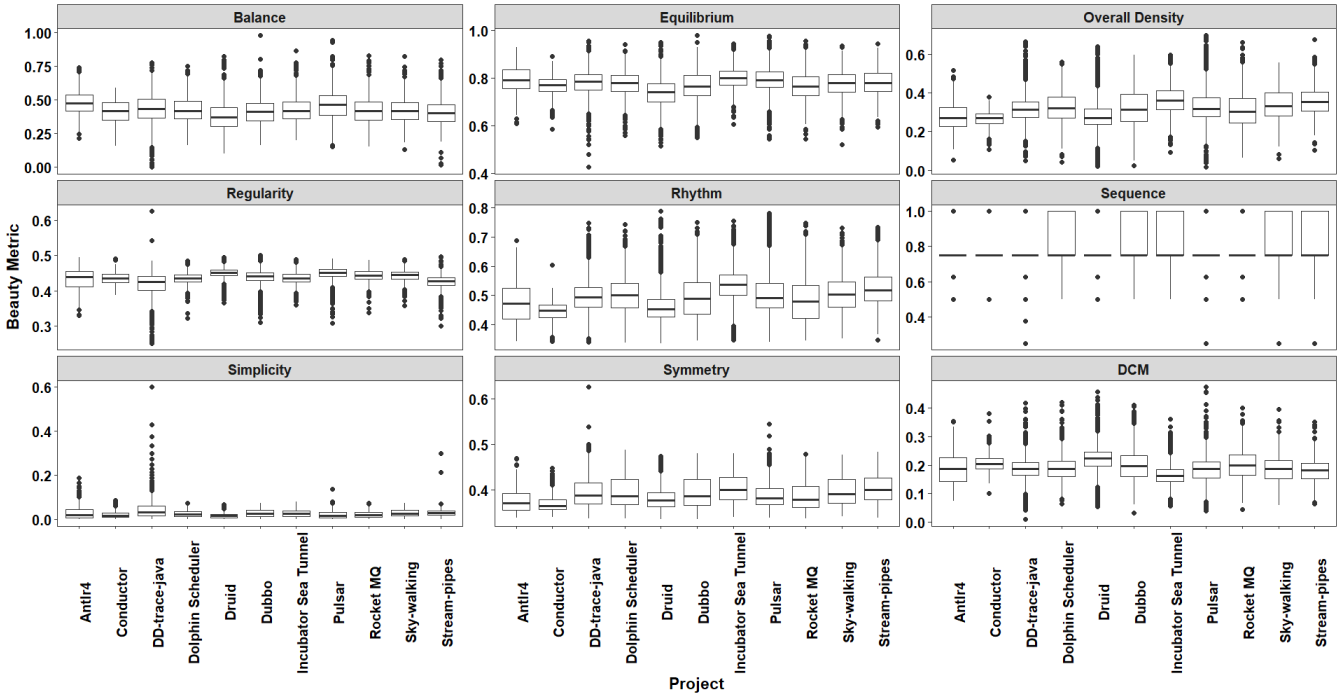


**Figure 12.** Distributions of beauty metrics per Open-Source Projects

## 4.4 Data Analysis

To answer the RQs, we performed correlation analysis. We computed the Spearman's correlation [30] for each beauty and quality metric to verify their relationship. Spearman's correlation is particularly effective when dealing with non-linear associations and is less sensitive to outliers. It also helps mitigate potential concerns of metric outliers that can skew results. In parallel, we calculated the p-value of each correlation to determine the evidence against a null hypothesis [31]. For interpreting the importance of the correlations, we implemented the rule of thumb, proposed by Hinkle [32] (0 to ±0.30 negligible/ ±0.30 to ±0.50 low/ ±0.50 to ±0.70 moderate/ ±0.70 to ±0.90 high/ ±0.90 to ±1.00 very high). To visualize metric scores, we have used Violin plots.

# 5 RESULTS AND DISCUSSION

## 5.1 Answering the Research Question

As a first step of the analysis, we performed descriptive analytics on the total set of the examined classes for the eleven Java open-source projects (Table II), whereas Figure 12 visualizes the distributions of the beauty metrics for each project.

**Table II**. Descriptive statistics for beauty metrics for the total set of examined classes of Open-Source Projects

| **Beauty Metric** | ***M*** | ***SD*** | ***Mdn*** | ***min*** | ***max*** |
|---|---|---|---|---|---|
| Balance | 0.42 | 0.10 | 0.41 | 0.00 | 0.98 |
| Equilibrium | 0.77 | 0.06 | 0.77 | 0.42 | 0.98 |
| Overall Density | 0.32 | 0.09 | 0.31 | 0.01 | 0.70 |
| Regularity | 0.44 | 0.02 | 0.44 | 0.25 | 0.62 |
| Rhythm | 0.49 | 0.07 | 0.48 | 0.34 | 0.79 |
| Sequence | 0.78 | 0.14 | 0.75 | 0.25 | 1.00 |
| Simplicity | 0.03 | 0.03 | 0.03 | 0.00 | 0.60 |
| Symmetry | 0.39 | 0.03 | 0.38 | 0.34 | 0.62 |
| DCM | 0.20 | 0.05 | 0.20 | 0.01 | 0.47 |
| CC | 1.41 | 2.09 | 1 | 0 | 107.53 |
| LCOM | 79.79 | 1007.25 | 4 | 0 | 63,190 |
| MPC | 19.66 | 42.09 | 7 | 0 | 1,574 |
| LoC | 91.18 | 259.92 | 32 | 0 | 13,816 |

To provide an answer to the posed research question in the study, we evaluated the correlation coefficient for each pair of the set of the code beauty metrics and the quality metrics. We note that: (a) with this study, we do not aim extracting causal relations, but only assess correlation; and (b) that we explore only one direction of the relation; although by definition correlation analysis is bi-

directional. Given the fact that the vast majority of pairwise correlation coefficients are statistically significant (more than 95% of the pairs), we can proceed with exploring the correlation coefficients. To aggregate the results at the dataset level, we constructed violin plots, organized by quality metric. The violin plots depict the variation in the correlation coefficients, between the quality metrics and the proposed beauty metrics. For a correlation to be consistent and very strong, we anticipate a "*short*" violin, with values concentrated close to either -1.0 or 1.0. For other cases: e.g., "*long*" violins, or values close to 0.0 the relation is inconclusive or very weak, respectively.

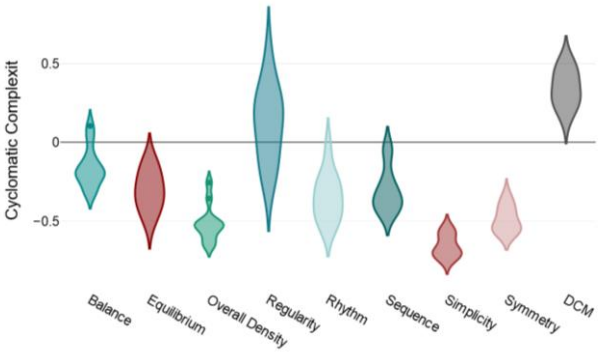


**Figure 13.** Correlation of CC and Beauty Metrics

Cyclomatic complexity and Simplicity (Figure 13) exhibit a robust negative correlation ranging from -0.744 to -0.552, indicating that as code simplicity increases, the cyclomatic complexity tends to decrease. This negative correlation is reinforced by a very low deviation, emphasizing the consistency of this relationship. Similarly, Density and Equilibrium showcase significant negative correlations, with medium deviations. These findings suggest that well-dense and equally structured code tends to exhibit lower cyclomatic complexity. On the other hand, metrics like Balance, Equilibrium, and Rhythm, and Sequence demonstrate negative correlations with CC, but with higher deviations, indicating a more inconclusive relationship. This (indicatively) implies that while the more balanced a code is, generally the less complexity it has; however, there are instances where this correlation is less pronounced. Regularity and DCM exhibit varied correlations with CC, emphasizing the multi-faceted nature of the relationship between beauty metrics and code complexity.

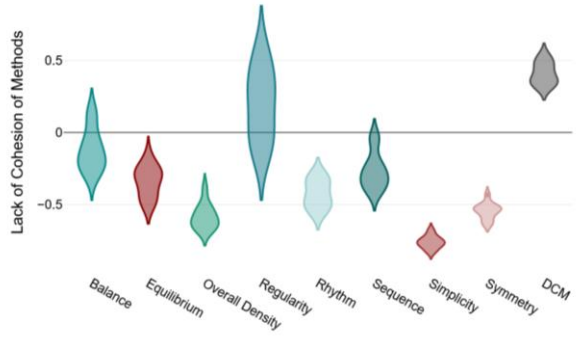


**Figure 14.** Correlation of LCOM and Beauty Metrics

The LCOM correlation with Simplicity (Figure 14) stands out with an exceptionally strong negative value, ranging from -0.831 to -0.675. This suggests a clear trend wherein as code simplicity increases, the Lack of Cohesion of Methods tends to decrease. Symmetry also demonstrates a noteworthy negative correlation, ranging from -0.632 to -0.432 with very low deviation, indicating that codebases exhibiting greater symmetry tend to have lower LCOM values. Equilibrium, Rhythm, and Density present substantial negative correlations, suggesting that well-balanced, rhythmically structured, and dense code tends to have lower LCOM values. The correlation with Balance and Sequence, while negative, presents a more varied picture, emphasizing the intricate nature of the interplay between these measures and LCOM. In contrast, Deviation of the Center of Mass (DCM) shows a positive correlation with LCOM, highlighting a potential trade-off between structural coherence and the distribution of code mass. Finally, yet again the results on Regularity are inconclusive.

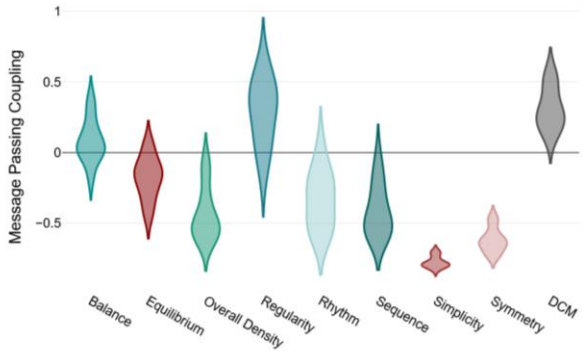


**Figure 15.** Correlation of MPC and Beauty Metrics

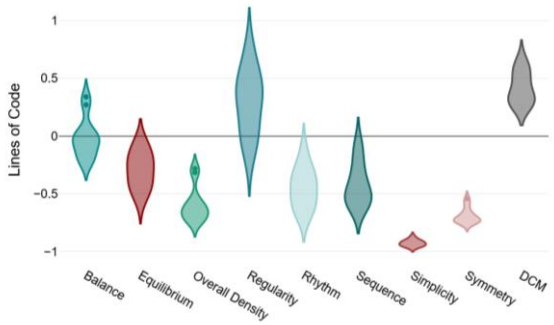


**Figure 16.** Correlation of LoC and Beauty Metrics

The MPC correlation with Simplicity (Figure 15) emerges as a key highlight, showcasing an extremely low deviation ranging from -0.831 to -0.701. This indicates a robust negative correlation, signifying that as code simplicity increases, the concerns related to coupling tend to decrease consistently. Symmetry, with a value between -0.735 and -0.459 and a low deviation, also exhibits a noteworthy negative correlation, suggesting that well-symmetric code structures are associated with improved maintainability. The correlations with Equilibrium, Density, Rhythm, and Sequence display high to very high deviations, although the sign of the cor-

relation is in most of the cases negative. For the rest of beauty metrics, the results are clearly inconclusive.

Finally, in terms of the LoC correlation with beauty metrics (Figure 16), again Simplicity stands out as a striking observation, showcasing an extremely low deviation and measured values ranging from -0.967 to -0.871. This negative correlation implies that as the simplicity of the code increases, the number of lines of code decreases significantly. Regularity, with a very high deviation and a variety of values ranging from -0.157 to 0.746, demonstrates a complex and diverse correlation, suggesting that the impact of structural aesthetics on code size is multifaceted. Symmetry exhibits a very low deviation between and measured values between -0.764 and -0.542, indicating a negative correlation, wherein well-symmetric code structures are associated with a reduction in code size. The correlations with Balance, Equilibrium, Rhythm, and Sequence are inconclusive, due to the existence of both positive and negative relations. Similarly, to the previous metrics, Density seems to present for most cases a negative correlation to LoC, suggesting that dense code is usually of small size. Deviation of the Center of Mass (DCM) showcases a medium to high deviation, however being concentrated always on the positive side for the sign of the correlation coefficients.

## 5.2 Interpretation of the Results

In this section, we summarize the most important findings and interpret them. First, by contrasting the results per beauty metric, we can observe an almost perfect consistency, in the sense that:

- *Simplicity* is negatively correlated to Complexity, Lack of Cohesion, Coupling, and Size. Thus, it is linked to code of better quality.
- The same applies for *Symmetry* and *Density*, but the relation is less strong (Moderate for Density).
- *Equilibrium*, *Rhythm*, and *Sequence* are in most of the cases negatively correlated with all quality attributes / metrics, but the correlation is very weak.
- *Balance* and *Regularity* produce only inconclusive results.
- *DCM* is the only beauty metric with a consistently positive correlation to Complexity, Lack of Cohesion, Coupling, and Size. We note that for DCM low values are desirable. Thus, it is linked with low code quality. However, this relation is also of moderate strength.

Since the presented code beauty metrics can be perceived by anyone regardless of his / her programming background and technical knowledge, even non-expert stakeholders can assess code quality only by viewing the code and without needing to know its functionality.

> We claim that Simplicity, Symmetry, and Density can be used as quite safe predictors of good structural quality.

This finding can be considered intuitive in the sense that code, fragments which are simple, symmetrical, and dense usually do not include many control statements (if, for, while, case, etc.) that would disturb the symmetry, and yield for more indentation. However, at this point we need to explain that this finding does not suggest the omission of indentation to enhance beauty, but we observe that the specific values for these beauty metrics are reflected to structural quality. Thus, any attempt to use such metrics shall be made on code fragments that obey the basic code formatting principles (e.g., after running a code beautifier).

Given the above finding, we can claim that ***practitioners'*** "*first look*" on a code fragment can act as a quite reliable approximation of the quality, as long as, basic formatting standards are obeyed. In that sense, we believe that code beauty must be a concern of the developer, while writing the code, especially targeting on writing small, modular, and less complex methods. This rule of thumb follows some basic principles of object-orientation, such as the Open-Close Principle, the use of Polymorphism, the adoption of the Single Responsibility Principle, etc.

## 5.3 Future Work Opportunities

From a ***researchers'*** point of view, we can conclude that beauty metrics seem to be useful for quality assurance purposes. Therefore, we champion their further investigation in future studies. However, an important first step before generalizing our results from the level of the specific metrics that we have used for the concept of beauty, there is a need for additional research on the perception of software practitioners as '*beautiful*' code. More specifically there is a need for validating the fact that the proposed metrics capture the perception of developer for code beauty. This can be performed either through a questionnaire study, or through psycho metrics captured along a maintenance task or code reading. It would also be interesting to investigate if the perception of code beauty differs between developers and non-programmers, in the sense that developers, consciously or subconsciously might look for specific 'anchors' in the code they read before assessing the overall picture.

Additionally, there is a need for studying how the specific beauty metrics are changing when coding standards are applied, and when they are not. This study will be important for differentiating cases where lack of symmetry is due to lack of formatting and necessary lack of blank spaces, e.g., due to reduced complexity of the code chunk. To our perception, such a study would require the combination of code styling and code beauty metric, under a common model or tool. Finally, it is important to study possible causality between beauty and quality, as well as tentative third causal factors, e.g., style or skills of the developer.

# 6 THREATS TO VALIDITY

While our study endeavors to explore the correlation between code aesthetics and quality metrics comprehensively, it is essential to acknowledge potential threats to the validity of findings.

First and foremost, the generalizability of our results may be limited by the specific set of projects and codebases chosen for analysis, as well as the use of only one programming language. The characteristics and coding practices of these projects might not be representative of the broader software development landscape posing threats to external validity. Further replication studies

would be needed to validate the identified correlations between code beauty measures and quality attributes.

The construct validity of the study is threatened by the choice of code quality metrics, as certain aspects of code quality may not be fully captured by the selected indicators. Similarly, the concept of beauty is inherently subjective and as a result the employed measures of code beauty reflect only some aspects of aesthetics in text/code. It would be reasonable to assume that the subjective nature of aesthetic evaluations may introduce inter-rater variability (in case of human assessments). The metrics employed for aesthetic evaluation might not encompass all dimensions of code beauty, and different stakeholders may have diverse opinions on what constitutes "beautiful" code. While the subjective evaluation has not been included as an independent parameter in our study, we attempted to mitigate this threat by substituting subjective evaluation of beauty by established aesthetic metrics. Finally, we note that this study is not aiming at identifying causal relations, but only provide an initial exploration of correlations.

## 7 CONCLUSIONS

Confirming the correlation between code beauty and code quality represents a pivotal finding in software engineering research. Through a meticulous analysis of various aesthetic metrics and their correlation with code quality measures, our study validates the intuitive belief that beautifully crafted code aligns with higher overall software quality. By employing Spearman correlation, our research reveals the relationship between factors like code simplicity, symmetry, balance, and metrics indicative of maintainability, performance, and reliability. The confirmation of this correlation underscores the importance of aesthetics in code development. It implies that codebases exhibiting elegance in design and structure are not only visually appealing but also tend to harbor qualities associated with robustness and maintainability. This insight has profound implications for software practitioners, highlighting that investments in code beauty can yield tangible benefits in terms of enhanced code quality, fostering a paradigm where aesthetics and functionality coalesce to form higher quality code.

## ACKNOWLEDGMENTS

This research is funded by the University of Macedonia Research Committee as part of the "*Principal Research 2023*" funding program.

## REFERENCES

[1] Begel, A., & Nagappan, N. (2008, October). Pair programming: what's in it for me? In Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement (pp. 120-128).
[2] Bacchelli, A., & Bird, C. (2013, May). Expectations, outcomes, and challenges of modern code review. In 2013 35th International Conference on Software Engineering (ICSE) (pp. 712-721). IEEE.
[3] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," Annual Reliability and Maintainability Symposium. 2002 Proceedings, Seattle, WA, USA, 2002, pp. 235-241.
[4] N. A. Al-Saiyd, "Source code comprehension analysis in software maintenance," 2017 2nd International Conference on Computer and Communication Systems (ICCCS), Krakow, Poland, 2017, pp. 1-5.
[5] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," Annual Reliability and Maintainability Symposium. 2002 Proceedings, Seattle, WA, USA, 2002, pp. 235-241.
[6] N. A. Al-Saiyd, "Source code comprehension analysis in software maintenance," 2017 2nd International Conference on Computer and Communication Systems (ICCCS), Krakow, Poland, 2017, pp. 1-5.
[7] T. Duchess, Molly Bawn. Dodo Press, 2008.
[8] S. Zeki, J. P. Romaya, D. M. T. Benincasa, and M. F. Atiyah, "The experience of mathematical beauty and its neural correlates," Front. Hum. Neurosci., vol. 8, p. 68, 2014.
[9] Oram, A. and Wilson, G. (2007). Beautiful Code. Newton: O' Reily.
[10] R. Coleman, "Beauty and Maintainability of Code," 2018 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 2018, pp. 825-828.
[11] H. Eichelberger. 2003. Nice class diagrams admit good design? In Proceedings of the 2003 ACM symposium on Software visualization (SoftVis '03). Association for Computing Machinery, New York, NY, USA.
[12] C. M. Aldenhoven and R. S. Engelschall, "The beauty of software architecture," 2023 IEEE 20th International Conference on Software Architecture (ICSA), L'Aquila, Italy, 2023, pp. 117-128.
[13] Santayana G. (1955). The Sense of Beauty. New York: Dover Publications
[14] Loukaki, A. (2008). Living Ruins. Value Conflict. Farnham: Ashgate.
[15] Zeki S., Romaya J. P., Benincasa D. M. T. and Atiyah M. F. (2014). The experience of mathematical beauty and its neural correlates, Frontiers in Human Neuroscience, vol. 8, article 68. DOI: 10.3389/fnhum.2014.00068
[16] Wertheimer M, Riezler K. (1994). Gestalt Theory, Social Research, vol. 11, no. 1, pp 78–99
[17] Ngo D. C. L., Teo L. S. and Byrne J. G. (2002). Evaluating Interface Esthetics, Knowledge and Information Systems, vol. 4, no. 1, pp. 46–79
[18] Khan Academy. What is the center of mass? [Accessed June 2023] https://www.khanacademy.org/science/physics/linear-momentum/center-of-mass/a/what-is-center-of-mass
[19] Danielsson, P. E. (1980). Euclidean distance mapping. Computer Graphics and image processing, 14(3), 227-248.
[20] Tullis, T. (1984). Predicting the Usability of Alphanumeric displays. (Ph.D.). Rice University, Houston
[21] Runeson, P., Host, M., Rainer, A., & Regnell, B. (2012). Case study research in software engineering: Guidelines and examples. Wiley & Sons.
[22] Nikolaidis, N., Mittas, N., Ampatzoglou, A., Arvanitou, E. M., & Chatzigeorgiou, A. (2023). Assessing TD Macro-Management: A Nested Modeling Statistical Approach. IEEE Transactions on Software Engineering, 49(4), 2996-3007.
[23] E. M. Arvanitou, P. Argyriadou, G. Koutsou, A. Ampatzoglou, and A. Chatzigeorgiou, "Quantifying TD Interest: Are we Getting Closer, or Not Even That?", 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 22), IEEE Computer Society, August 2020, Gran Canaria, Spain
[24] Ampatzoglou, A., Mittas, N., Tsintzira, A. A., Ampatzoglou, A., Arvanitou, E. M., Chatzigeorgiou, A., ... & Angelis, L. (2020). Exploring the relation between technical debt principal and interest: An empirical approach. Information and Software Technology, 128, 106391.
[25] Riaz, M., Mendes, E., & Tempero, E. (2009, October). A systematic review of software maintainability prediction and metrics. In 2009 3rd international symposium on empirical software engineering and measurement (pp. 367-377). IEEE.
[26] Van Koten, C., & Gray, A. R. (2006). An application of Bayesian network for predicting object-oriented software maintainability. Information and Software Technology, 48(1), 59-67.
[27] Zhou, Y., & Leung, H. (2007). Predicting object-oriented software maintainability using multivariate adaptive regression splines. Journal of systems and software, 80(8), 1349-1361.
[28] Li, W., & Henry, S. (1993). Object-oriented metrics that predict maintainability. Journal of systems and software, 23(2), 111-122.
[29] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object-oriented design. IEEE Transactions on software engineering, 20(6).
[30] De Winter, J. C., Gosling, S. D., & Potter, J. (2016). Comparing the Pearson and Spearman correlation coefficients across distributions and sample sizes: A tutorial using simulations and empirical data. Psychological methods, 21(3), 273.
[31] Thiese, M. S., Ronna, B., & Ott, U. (2016). P value interpretations and considerations. Journal of thoracic disease, 8(9), E928.
[32] Hinkle D.E., Wiersma W., Jurs S.G. (2003). Applied Statistics for the Behavioral Sciences 5th ed. Boston: Houghton Mifflin.