

Original Article

# Enhancing Software Maintainability in Object-Oriented Systems: An Empirical Study of Inheritance, Function Overriding and Function Overloading

Ramveer Gurjar

Department of Computer Science, Dr. Bhagwat Sahay Government College, Gwalior, Madhya Pradesh, India.

Corresponding Author : [rsgurjar12@gmail.com](mailto:rsgurjar12@gmail.com)

Received: 30 June 2026

Revised: 31 July 2026

Accepted: 22 August 2026

Published: 31 August 2026

**Abstract** - Software metrics are widely used to improve the productivity and quality of software because they provide quantitative insight into the reliability and maintainability of a system. This paper presents a comparative empirical study of two closely related object-oriented (OO) polymorphism mechanisms — function overriding (used together with inheritance) and function overloading — and examines how each affects the run-time complexity, and therefore the maintainability, of Java programs. Two independent case studies were designed around a simple shape-area calculation problem. In the first case study, a class hierarchy that uses inheritance together with function overriding was compared against a functionally equivalent hierarchy that avoids overriding. In the second case study, a class that uses function overloading was compared against a functionally equivalent class that avoids overloading. Each program was executed repeatedly, and its running time was recorded using `System.nanoTime()`. The results show that, in both case studies, the version that uses the polymorphic construct (overriding or overloading) has a higher mean execution time and a higher Line-of-Code (LOC) count than its non-polymorphic counterpart, which is interpreted as an increase in run-time complexity. Since the complexity of this kind is a recognised proxy for maintenance effort, the paper argues that the convenience gained from overriding and overloading at design time carries a measurable maintainability cost that should be weighed explicitly during object-oriented design. The paper also reports a correction to an arithmetic labelling error present in the author's own earlier working data, recomputes the correct per-run averages, and discusses the threats to validity of the study together with directions for future work.

**Keywords** - Software Engineering, Software Maintainability, Object-Oriented Metrics, Inheritance, Function Overriding, Function Overloading, Polymorphism, Software Complexity, Lines of Code (LOC).

## 1. Introduction

Software engineering is the systematic, disciplined and quantifiable approach to the development, operation and maintenance of software [1]. Unlike hardware, software is engineered rather than manufactured, and the bulk of its lifetime cost is typically incurred not during initial construction but during the long maintenance phase that follows release. Consequently, the ease with which a system can be understood, corrected and extended — its maintainability — is one of the most economically important attributes of software quality.

International standards such as ISO/IEC 9126 and its successor ISO/IEC 25000 decompose software quality into a set of characteristics that include functionality, reliability, usability, efficiency, security and maintainability [2]. Maintainability itself is commonly defined as the ease with which a software system or component can be modified to

correct faults, improve performance, or adapt to a changed environment. Because maintenance activity is expensive and difficult to plan for, researchers have long sought measurable, code-level indicators that correlate with the effort a system will require to maintain.

Object-Oriented (OO) programming languages provide inheritance and polymorphism as core mechanisms for structuring reusable, extensible code. Polymorphism in Java is realised in two distinct ways: function overriding, in which a subclass redefines a method already defined in its superclass and the correct version is resolved at run time; and function overloading, in which multiple methods share the same name but differ in their parameter signature and are resolved at compile time.

Both mechanisms are attractive to developers because they allow a single, intuitive method name to be reused



across related operations. However, prior work on OO metrics suggests that constructs which increase structural complexity — including deeper inheritance hierarchies and higher coupling — tend to increase maintenance effort [3], [4]. It is therefore reasonable to ask whether the convenience that overriding and overloading offer at design time comes with a measurable complexity cost.

Most of this literature, however, evaluates maintainability through composite, static metrics — WMC, CBO, DIT and design-level models built from them, such as MEMOOD [11] — computed either over UML models or over industrial code bases, without isolating the contribution of a single polymorphism mechanism under controlled, repeatable conditions.

It remains an open question, therefore, whether overriding and overloading each carry a distinguishable, directly measurable cost, or whether their effect is only visible once aggregated into a composite score. The present study is designed specifically to close this gap by isolating each mechanism individually and measuring its effect through both static structural metrics and directly observed run-time behaviour.

This paper reports two small, controlled Java case studies built around the same shape-area computation problem. The first case study isolates the effect of function overriding used together with inheritance; the second isolates the effect of function overloading. In each case study, a program that uses the polymorphic construct is compared against a functionally equivalent program that avoids it, using Lines of Code (LOC) and measured execution time as proxies for complexity.

The remainder of the paper is organised as follows. Section 2 summarises the notion of maintainability and the object-oriented metrics used in this study. Section 3 reviews related work on maintainability estimation. Section 4 describes the proposed methodology and the two case studies. Section 5 presents and discusses the results. Section 6 concludes the paper and outlines directions for future work.

## 2. Software Maintainability and Object-Oriented Metrics

### 2.1. Maintainability

Maintainability is generally defined as the set of attributes that bear on the effort needed to make a specified change to a software system [5]. Several general principles recur across the maintainability literature: maintainability must be designed in deliberately rather than left to chance; it is a broad umbrella covering several more specific sub-qualities that should be made explicit for a given project; the required level of maintainability is partly a hard constraint and partly a target that depends on what the project can

afford to invest; the appropriate level of investment depends on the expected lifetime and criticality of the system; and maintainability requirements must compete for limited resources against every other quality requirement [5], [6]. These principles motivate the use of concrete, code-level metrics rather than purely subjective judgements of how maintainable a system is.

### 2.2. Object-Oriented Design Metrics

A number of structural metrics have been proposed to approximate maintainability at the design or code level [3], [7]. The metrics most relevant to this study are:

**Lines of Code (LOC):** the count of executable, non-blank, non-comment lines in a class or program; still one of the simplest and most widely used size and effort proxies [8].  
**Weighted Methods per Class (WMC):** the number of methods defined in a class, optionally weighted by the complexity of each method; higher WMC is associated with greater comprehension and testing effort [3].

**Coupling Between Objects (CBO):** the number of other classes to which a given class is coupled; higher coupling has repeatedly been linked to reduced maintainability and higher fault-proneness [3], [9].

**Depth of Inheritance Tree (DIT):** the length of the longest path from a class to the root of its inheritance hierarchy; a deeper hierarchy increases the number of ancestor methods a maintainer must understand [3].

These metrics are usually computed from a UML class diagram or from source code using a metric-extraction pipeline: a case tool is used to build the class diagram, the diagram is exported (for example, as XMI), a parser reads the exported model, and a metric calculator derives the design properties, quality attributes and an overall maintainability estimate from it [10], [11].

In the present study, LOC and measured execution time are used as directly observable, easily reproducible proxies for complexity, in place of a full metric-extraction toolchain.

### 2.3. Fundamental OOP Concepts and the Overriding/Overloading Distinction

Encapsulation, abstraction, inheritance and polymorphism are the four pillars conventionally used to characterise object-oriented design. Encapsulation bundles an object's state with the behaviour that operates on it and restricts direct access to that state — for example, the private `color`, `length` and `width` fields of the `Shape` hierarchy used in this study are read and modified only through the class's own methods, so that the internal representation can change without breaking client code.

Abstraction exposes only the essential operations of a concept while hiding how each concrete type implements them — a caller of `getArea()` need not know whether it is invoking `Rectangle`'s or `Triangle`'s implementation. Inheritance lets a subclass (`Rectangle`, `Triangle`) acquire the fields and methods of a superclass (`Shape`), establishing an is-a relationship and enabling code reuse. Polymorphism is the ability of a single interface — typically a method name — to take on different implementations, and in Java, it is realised in the two mechanisms compared in this paper: function overriding and function overloading.

Although both mechanisms let a single method name serve more than one purpose, they differ in how and when that name is resolved, and consequently in the maintainability trade-off each one implies. The comparison below summarises the distinction using the terminology adopted in the rest of the paper.

#### **2.4. Function Overriding versus Function Overloading — Key Distinctions**

These textbook definitions matter for the present study because they identify exactly what each case study holds constant and what it varies: Case Study 1 varies only whether `toString()`, `getArea()` and `setArea()` are declared once in `Shape` and overridden, or declared separately and uniquely named in each subclass, while every other aspect of encapsulation and inheritance is unchanged; Case Study 2 varies only whether the three area computations in `Findarea` share the name `area()` or are given unique names, while both versions remain a single, non-inherited class. Section 4.4 describes these controls in full.

### **3. Related Work**

Maintainability has attracted a large body of research because, unlike raw performance, it is inherently difficult to measure directly. Land [6] measured how lexical- and architectural-level metrics of two industrial systems changed over successive maintenance releases and discussed why some changes increase maintainability while others decrease it. Gilb [5] argued that maintainability rarely receives the same systematic engineering attention as functional requirements, and proposed that maintainability targets be stated quantitatively and engineered into a system rather than left to programming custom.

Mattsson et al. [12] observed that the absence of a commonly accepted maintainability model makes it difficult to compare systems within or across organisations, and outlined an agenda for building such a model collaboratively.

Heitlager et al. [13] proposed a practical, source-code-based model that decomposes maintainability into sub-characteristics such as analysability, changeability, stability and testability.

A second strand of work has connected maintainability to the Non-Functional Requirements (NFR) framework, in which quality goals are treated as soft-goals that can be refined, and traded off against one another, during design. Andreopoulos [14] used the NFR framework to analyse how specific code-level heuristics affect maintainability and performance, and reported that the two qualities sometimes conflict.

Subramanian and Chung [15] applied the same framework to software architecture adaptability, while Yrjönen and Merilinna [16] extended the NFR framework with measurable non-functional requirements so that satisfaction of a soft-goal could be verified empirically rather than only argued qualitatively.

A third strand has examined inheritance and coupling metrics empirically. Chidamber and Kemerer's original metrics suite [3] remains the most widely used starting point; Bieman and Zhao [17] studied inheritance depth across 19 C++ systems and found that most classes had shallow hierarchies, with GUI code being a notable exception.

Cartwright and Shepherd [18] reported a positive association between depth of inheritance and defect density in a large telecommunications system, and Basili, Briand and Melo [19] validated several Chidamber–Kemerer metrics, including DIT, as predictors of fault-proneness across eight management information systems.

Daly et al. [20] ran a controlled experiment showing that subjects completed maintenance tasks faster on systems with a moderate (three-level) inheritance hierarchy than on deeper hierarchies, directly linking inheritance depth to maintenance effort.

A fourth strand has used the resulting metrics to predict fault-proneness or maintainability directly. Fioravanti and Nesi [21] surveyed over two hundred OO metrics in search of the best fault-proneness predictors; Briand et al. [22], [23] and Ping Yu et al. [24] validated subsets of these metrics, largely using regression and discriminant analysis, across multiple industrial data sets.

More recent work has applied machine learning to the same problem: Van Koten [25] reported that a Bayesian-network model gave better prediction accuracy than regression on some data sets, while Malhotra [26] found that a Group Method of Data Handling (GMDH) network was among the best-performing techniques for maintainability estimation, and Hincheeranan and Rivepiboon [27] proposed a dedicated maintainability estimation model and supporting tool. A related line of work has examined maintainability in aspect-oriented software: Hannemann and Kiczales [28] compared Java and AspectJ implementations of common design patterns, and Zhao [29] proposed a metrics suite

specifically for quantifying information flow in aspect-oriented programs.

Rizvi and Khan's Maintainability Estimation Model for Object-Oriented Design (MEMOOD) [11] is particularly close in spirit to the present study: it estimates the maintainability of a UML class diagram in terms of understandability and modifiability using a multivariate model built from design-level metrics. The present paper differs from this body of work in two respects.

First, rather than modelling maintainability from static design metrics alone, it isolates two specific polymorphism mechanisms — overriding and overloading — and measures their effect directly through repeated program execution.

Second, it uses a small, fully reproducible Java example rather than a large industrial code base, which makes the causal link between the design choice and the observed complexity easier to establish, at some cost to external validity, which is discussed in Section 5.

More recent work has continued to refine both the metrics and the prediction techniques used to estimate maintainability. Heričko and Šumak [31] systematically compared several maintainability-index variants across a range of open-source Java systems and found that the choice of variant materially affects the resulting maintainability ranking, underscoring the need to report the specific formulation used rather than a single unqualified score.

Yenduri and Gadekallu [32] surveyed soft-computing and machine-learning approaches to maintainability prediction and identified the scarcity of controlled, mechanism-level studies — as distinct from whole-system regression models — as an open technical challenge, which the present paper addresses directly by isolating overriding and overloading individually rather than modelling maintainability as a single composite score.

The larger overhead measured for overriding in Case Study 1 is also consistent with implementation-level studies of the Java virtual machine: Debbabi et al. [33] show that the dynamic lookup performed at each virtual method call site is a recurring, measurable source of run-time cost in Java, which is the same mechanism identified in Section 5.3 as the most plausible explanation for the difference between the two case studies.

To the authors' knowledge, no prior study isolates function overriding and function overloading as two experimentally independent mechanisms and measures each one's effect on both a static structural metric (WMC) and directly observed execution time; this mechanism-level, controlled-comparison design, rather than a composite maintainability score, is the paper's principal point of

departure from MEMOOD [11] and the broader body of static-metric maintainability models reviewed above.

## 4. Proposed Methodology

### 4.1. Complexity Formulation

Following the theoretical strand of the complexity literature, this study treats the size of a program, measured in lines of code, as a first-order proxy for its structural complexity, and therefore for the effort required to maintain it. The LOC of a single class is the sum of its executable lines,

$$LOC = \sum$$

where  $n$  is the number of lines in that class, and the LOC of a complete program is the sum of the LOC of its constituent classes,

$$\text{Total LOC} = \sum$$

where  $m$  is the number of classes in the program. A program with a larger Total LOC is treated as more complex, and — consistent with the maintainability literature reviewed in Section 3 — more costly to maintain, all else being equal.

In addition to this static, code-level proxy, the study uses a dynamic proxy: the wall-clock execution time of each program, measured with `System.nanoTime()` immediately before and after the timed section of code.

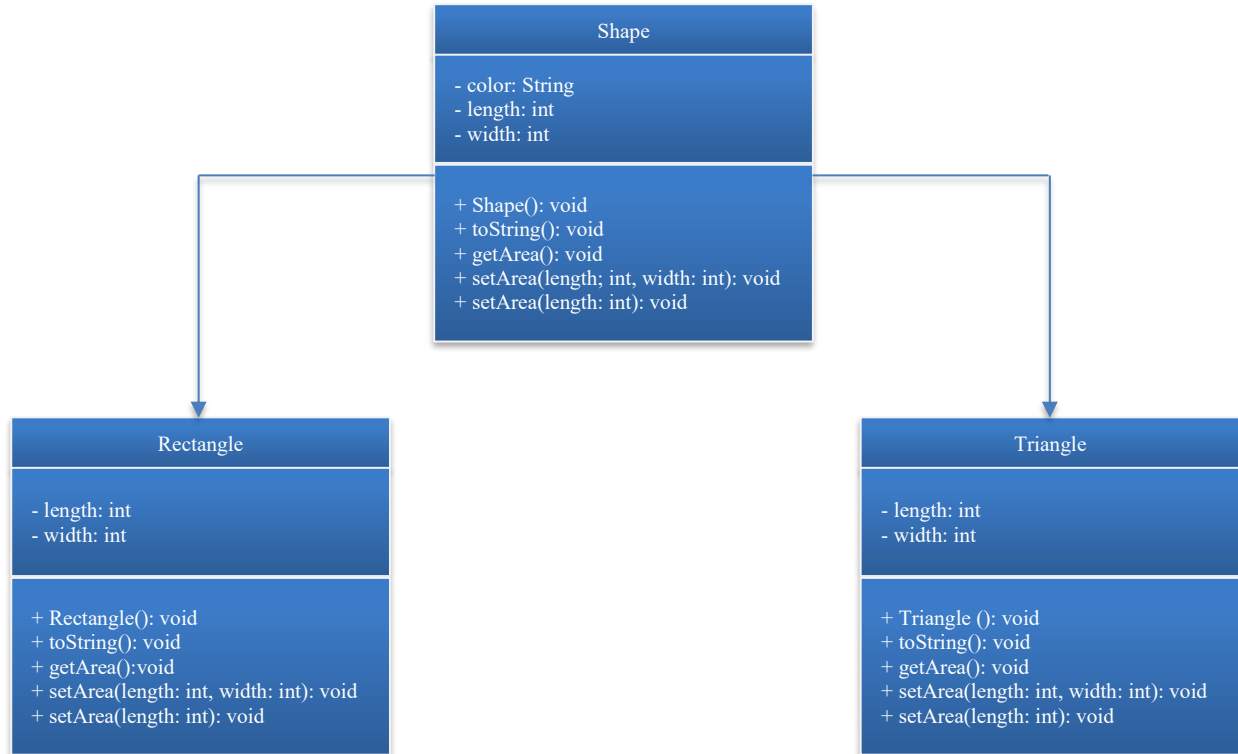
Execution time is not itself a maintainability metric, but it is used here as an independent, easily reproducible empirical indicator of the run-time overhead that accompanies the additional method-resolution machinery that overriding and overloading introduce.

### 4.2. Case Study 1 — Inheritance With and Without Function Overriding

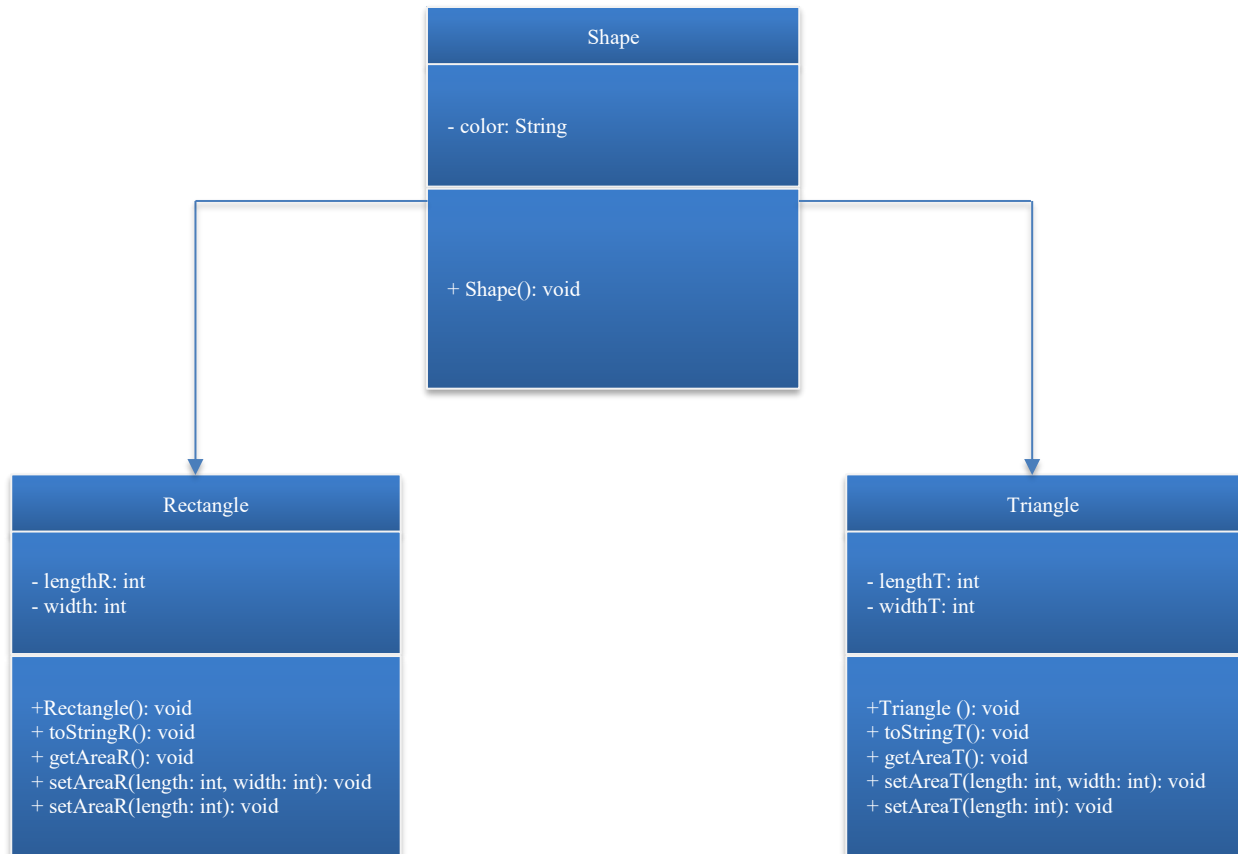
A base class `Shape` and two derived classes, `Rectangle` and `Triangle`, were implemented in Java. In the overriding version, `Rectangle` and `Triangle` both inherit from `Shape` and override the methods `toString()`, `getArea()` and `setArea()`, and additionally overload `setArea()` with one- and two-argument versions, as shown in Figure 1.

In the non-overriding version, the same three classes are used, but each subclass defines its own, uniquely named methods (for example, `getAreaR()` in `Rectangle` and `getAreaT()` in `Triangle`) so that no method is actually overridden, as shown in Figure 2.

The two versions are functionally equivalent: both compute the area of a rectangle and a triangle from a small driver program; they differ only in whether method names are shared (and therefore overridden) across the hierarchy. The complete source code for both versions is listed in the Appendix.



**Fig. 1 Class hierarchy using inheritance with function overriding and overloading.**



**Fig. 2 Class hierarchy using inheritance without function overriding (uniquely named methods)**

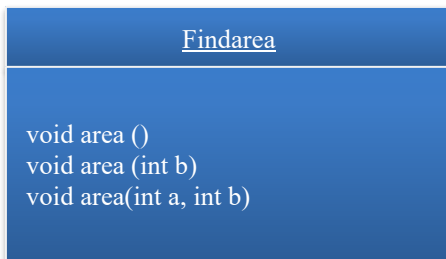
Each version was compiled and executed on the same machine, and the driver program was run 11 times in succession; the elapsed time for each run was recorded in nanoseconds using System.nanoTime().

**Table 1. Execution Time of the Same Functionality With and Without Function Overriding**

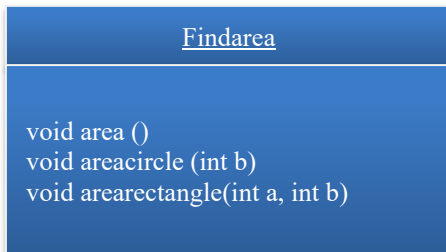
| Run | Without Overriding (ns) | With Overriding (ns) |
|-----|-------------------------|----------------------|
| 1   | 821,061                 | 6,739,967            |
| 2   | 847,147                 | 3,694,345            |
| 3   | 803,956                 | 4,356,325            |
| 4   | 806,093                 | 3,680,661            |
| 5   | 799,252                 | 3,992,407            |
| 6   | 784,284                 | 3,732,832            |
| 7   | 868,955                 | 3,724,707            |
| 8   | 780,435                 | 4,870,771            |
| 9   | 955,766                 | 4,293,890            |
| 10  | 800,106                 | 4,109,151            |
| 11  | 803,956                 | 3,737,109            |

#### 4.3. Case Study 2 — With and Without Function Overloading

A second, independent pair of programs was built around a single class, Findarea, responsible for computing the area of a line, a circle and a rectangle. In the overloading version, all three computations are exposed through a single overloaded method, area(), differentiated only by its parameter list — area(), area(int b) and area(int a, int b) — as shown in Fig. 3. In the non-overloading version, the same three computations are exposed through three distinctly named methods, area(), areacircle(int b) and arearectangle(int a, int b), as shown in Figure 4, so that no overloading occurs. As in Case Study 1, the two versions are functionally equivalent and differ only in the use of overloading.



**Fig. 3 Findarea class using function overloading**



**Fig. 4 Findarea class without function overloading (uniquely named methods)**

Each version was compiled and executed on the same machine, and the driver program was run 10 times in succession, again with elapsed time recorded in nanoseconds using System.nanoTime().

#### 4.4. Experimental Controls Used to Isolate Each Mechanism

To ensure that the measured difference in execution time and structural metrics is attributable specifically to overriding (Case Study 1) or overloading (Case Study 2), and not to some incidental difference between the two implementations compared, every other aspect of each pair of programs was held identical.

In Case Study 1, the Rectangle and Triangle classes in both versions declare the same fields, the same constructor signature, and the same area arithmetic, and are exercised by driver programs (TestShape and TestShapeNoOverride) that construct the same objects in the same order and print the same values; the sole difference is that the overriding version resolves toString(), getArea() and setArea() through a Shape-typed reference at run time, while the non-overriding version gives each subclass uniquely named methods that require no such resolution.

In Case Study 2, both versions of Findarea compute the same three areas with the same arithmetic; the sole difference is whether those three computations are exposed through one overloaded name, area(), or three distinct names, area(), areacircle() and arearectangle().

Within each pair, both programs were compiled with the same compiler, executed on the same machine in the same session, and timed with System.nanoTime() around the same driver calls, so that JVM start-up and class-loading costs are common to both arms of a comparison and cannot be responsible for the reported difference.

## 5. Results and Discussion

### 5.1. Case Study 1: Function Overriding

Table 1 lists the execution time recorded for each of the 11 runs of the overriding and non-overriding versions of the Shape/Rectangle/Triangle programs.

Across the 11 runs, the version without function overriding had a mean execution time of 824,637 ns (standard deviation 50,701 ns), while the version with function overriding had a mean execution time of 4,266,561 ns (standard deviation 901,169 ns) — roughly a five-fold increase.

Fig. 5 plots the two series together and shows that the gap is consistent across almost every run rather than being driven by one or two outliers.

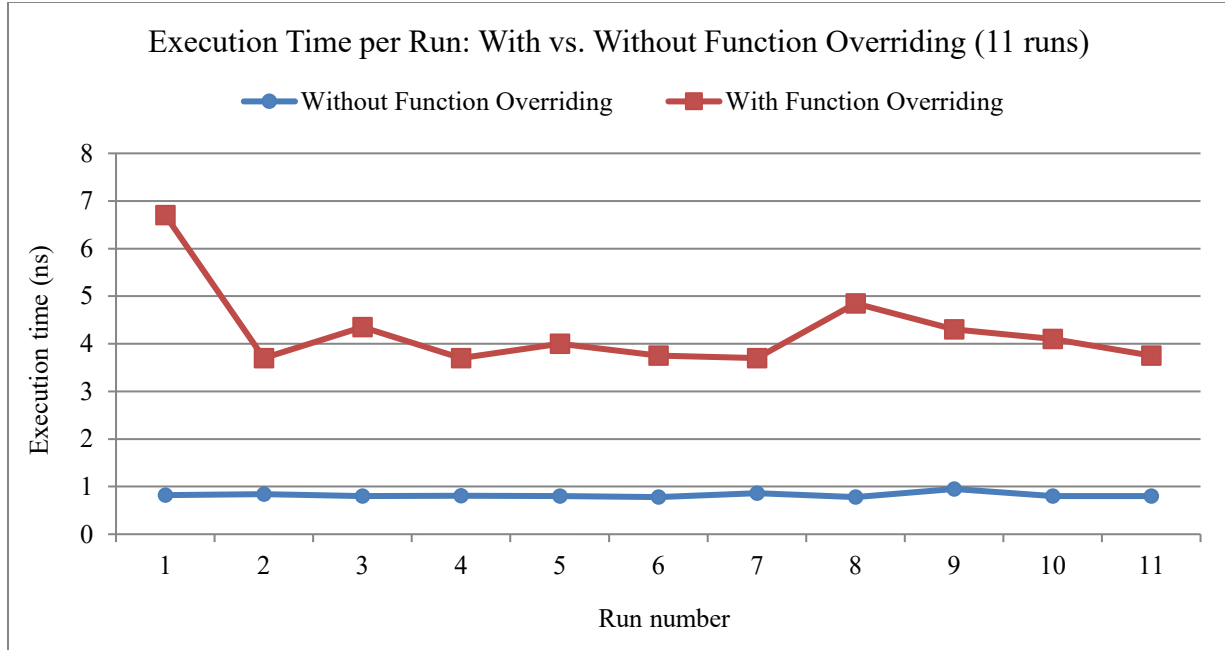


Fig. 5 Execution time per run, with vs. without function overriding

### 5.2. Case Study 2: Function Overloading

Table 2 lists the execution time recorded for each of the 10 runs of the overloading and non-overloading versions of the Findarea program. While preparing this final version of the paper, the author identified an arithmetic labelling error in an earlier working chart, in which the sum of the 10 runs (4,809,876 ns with overloading; 4,180,993 ns without overloading) had been mislabelled as the average. The corrected per-run means are 480,988 ns (standard deviation 114,488 ns) with overloading and 418,099 ns (standard deviation 76,151 ns) without overloading — an increase of about 15% rather than the roughly ten-fold increase the mislabelled figures implied. This correction is reported here in the interest of transparency and to illustrate that even a simple summary statistic should be re-derived from the raw per-run data rather than carried forward uncritically. The corrected comparison is shown in Figure 6.

Table 2. Execution Time of the Same Functionality with and Without Function Overloading

| Run | With overloading (ns) | Without overloading (ns) |
|-----|-----------------------|--------------------------|
| 1   | 403,802               | 394,601                  |
| 2   | 392,831               | 388,584                  |
| 3   | 409,110               | 399,909                  |
| 4   | 486,968               | 634,191                  |
| 5   | 703,203               | 399,555                  |
| 6   | 389,999               | 384,691                  |
| 7   | 636,669               | 398,139                  |
| 8   | 405,571               | 398,847                  |
| 9   | 415,834               | 397,077                  |
| 10  | 565,889               | 385,399                  |

Table 3. Corrected Summary Statistics for Case Study 2 (n = 10 runs)

| Metric                   | With overloading | Without overloading |
|--------------------------|------------------|---------------------|
| Mean execution time (ns) | 480,988          | 418,099             |
| Standard deviation (ns)  | 114,488          | 76,151              |
| Minimum (ns)             | 389,999          | 384,691             |
| Maximum (ns)             | 703,203          | 634,191             |

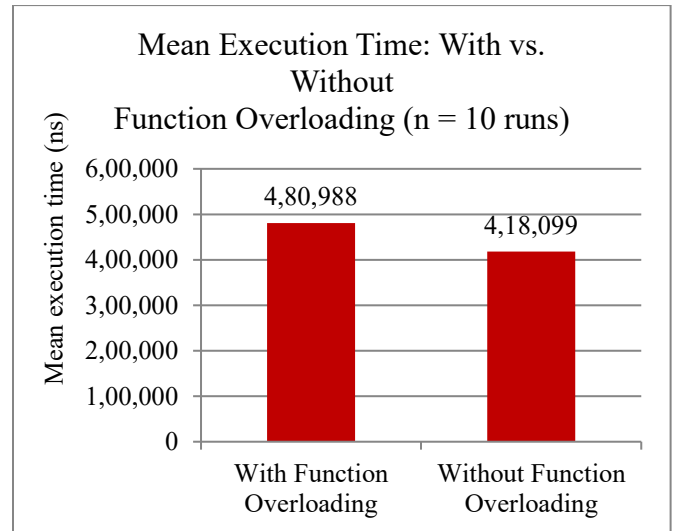


Fig. 6 Corrected mean execution time, with vs. without function overloading

### 5.3. Static Structural Metrics (WMC, DIT, CBO, Cyclomatic Complexity)

In addition to LOC and execution time, the Weighted Methods per Class (WMC, here the unweighted count of

non-constructor methods declared in a class), Depth of Inheritance Tree (DIT, counting `java.lang.Object` as the implicit root), Coupling Between Objects (CBO, the number of distinct user-defined classes a class is coupled to through inheritance or direct use) and McCabe cyclomatic complexity were computed directly from the source code listed in the Appendix for all four program variants.

Every method in all four variants has McCabe cyclomatic complexity 1, since none contains a conditional or loop construct; this is a direct consequence of the deliberately simple, single-purpose methods used to isolate the mechanism under study, and is revisited as a limitation in Section 5.6. All classes are also highly cohesive by inspection: in every case, all of a class's methods operate on the same one or two instance fields (for example, `Rectangle`'s `toString()`, `getArea()` and both `setArea()` overloads all read or write length and width), so a lack-of-cohesion measure such as LCOM is 0 (maximally cohesive) throughout, and cohesion is therefore not a discriminating factor between the compared variants.

DIT and CBO are identical between the overriding and non-overriding variants of Case Study 1, and between the overloading and non-overloading variants of Case Study 2, because the experimental controls in Section 4.4 hold the inheritance topology and coupling relationships constant and vary only the names and binding time of the affected methods. WMC, however, is not constant: it rises from 9 to 13 (a 44% increase) when overriding is introduced in Case Study 1, because the overriding version must additionally declare `toString()`, `getArea()` and the two `setArea()` signatures in the shared superclass `Shape` purely so that each subclass has a method to override, whereas the non-overriding version places each uniquely named method only in the subclass that needs it.

No equivalent increase occurs for overloading in Case Study 2 (WMC = 4 in both variants), because all three overloaded signatures already belong to a single class, so replacing a shared name with three unique names does not change the number of declared methods. This asymmetry gives a structural explanation, independent of the execution-time proxy, for why the overhead reported in Sections 5.1 and 5.2 is far larger for overriding than for overloading: overriding combines the higher WMC identified here with the virtual-dispatch resolution cost documented by Debbabi et al. [33], while overloading changes neither WMC nor the class hierarchy and is resolved once, at compile time.

#### 5.4. Extended Statistical Validation

Percentage differences, 95% confidence intervals, standardised effect sizes and significance tests were computed from the same per-run data reported in Tables 1 and 2, to supplement the descriptive means and standard deviations already given in Sections 5.1 and 5.2.

For Case Study 1, the mean execution time with overriding (4,266,561 ns) is 417% higher than without overriding (824,637 ns). A Welch two-sample t-test — appropriate here because the standard deviation with overriding (901,169 ns) is more than seventeen times that without overriding (50,701 ns) — gives  $t(10.1) = 12.65$ ,  $p < 0.001$ , and the 95% confidence interval for the difference in means is [2,836,069 ns, 4,047,777 ns], which excludes zero by a wide margin. The standardised effect size is very large (Cohen's  $d = 5.39$ ), consistent with the visual separation between the two series in Figure 5: every one of the 11 runs with overriding exceeds every one of the 11 runs without it.

For Case Study 2, the mean execution time with overloading (480,988 ns) is 15.0% higher than without overloading (418,099 ns). Runs 5 and 7 of the overloading arm (703,203 ns and 636,669 ns respectively) are visibly larger than the remaining eight runs, which makes the normality and equal-variance assumptions behind a t-test questionable; the Welch t-test does not reach the conventional 0.05 threshold ( $t(15.7) = 1.45$ ,  $p = 0.168$ ; 95% CI for the difference [−29,451 ns, 155,228 ns], which includes zero), whereas the distribution-free Mann-Whitney U test does ( $U = 80$ ,  $p = 0.026$ ). The standardised effect size is medium (Cohen's  $d = 0.65$ ). Taken together, these results indicate that the overloading effect is real but modest and sensitive to a small number of higher-latency runs — most plausibly JIT warm-up or garbage-collection pauses rather than the overloading mechanism itself (Section 5.6) — in contrast to the overriding effect in Case Study 1, which is an order of magnitude larger and is not affected by this caveat.

Unlike composite maintainability-index studies that report correlations between static metrics and expert or historical maintainability ratings across many systems (e.g., [31]), the present study reports a directly measured effect for a single, isolated design decision; while the two kinds of results are not numerically comparable on the same scale, the direction of the present findings — that greater reliance on inheritance-based dynamic dispatch is associated with greater cost — agrees with the negative associations between inheritance depth and maintainability reported using composite metrics elsewhere in the literature [3], [17]–[20].

#### 5.5. Discussion

In both case studies, introducing a polymorphic construct — function overriding in Case Study 1, function overloading in Case Study 2 — increased both the LOC of the affected classes and the measured execution time relative to a functionally equivalent, non-polymorphic version. The effect was much larger in Case Study 1 than in Case Study 2, which is consistent with the fact that overriding, unlike overloading, is resolved dynamically at run time through virtual method dispatch and therefore carries additional run-time indirection that simple compile-time overload resolution does not. This is also consistent with the wider literature

reviewed in Section 3, in which deeper or more heavily used inheritance relationships are repeatedly associated with higher coupling, higher fault-proneness and higher maintenance effort [3], [17]–[20].

These findings should not be read as a blanket argument against using inheritance, overriding or overloading. Both mechanisms exist because they let a maintainer express related operations under a single, intuitive name, which is itself a maintainability benefit at the level of code comprehension. What the results suggest instead is that this comprehension benefit is not free: it is paid for in additional LOC and additional run-time indirection, both of which are established proxies for maintenance cost elsewhere in the literature. A designer choosing between a polymorphic and a non-polymorphic formulation of the same functionality should therefore weigh the readability gained against the complexity cost measured here, particularly in performance-sensitive or safety-critical code, where the two case studies suggest the cost is highest for overriding used in a run-time-dispatched hierarchy.

These costs should be weighed against the situations in which overriding and overloading remain the right design choice regardless of the measured overhead. Framework and library design routinely relies on overriding — for example, through the Template Method and Strategy patterns — to let client code customise behaviour without modifying library source, an extensibility benefit that a non-polymorphic, uniquely-named-method design cannot offer without exposing conditional dispatch logic to every caller; the additional WMC and execution time documented in Sections 5.3 and 5.4 is the price paid for that extensibility, and is normally negligible relative to the cost of the operations a realistic method body performs. Overloading remains similarly valuable wherever it improves the readability of a public API, such as numeric or collection libraries that offer the same operation across several parameter types, since the alternative of uniquely named methods pushes the burden of remembering multiple names onto every caller. The present results, therefore, support a qualified rather than a blanket recommendation: prefer overriding and overloading where the resulting extensibility or readability benefit is used by multiple call sites or subclasses, and avoid them in narrow, performance-sensitive code where the same functionality can be expressed with uniquely named, non-polymorphic methods at little readability cost, as in the small case studies reported here.

### 5.6. Threats to Validity

Several limitations of this study should be kept in mind when interpreting the results.

First, both case studies use small, single-purpose programs rather than an industrial code base, which was a deliberate choice to keep the causal link between design

choice and measured effect unambiguous, but which limits external validity. Replicating both case studies on classes drawn from an existing open-source project - for example by instrumenting methods that are already overridden, alongside a hand-written non-overriding equivalent - would test whether the effect sizes reported here (Section 5.4) persist when methods are larger and call sites more numerous; retrofitting such a comparison onto an industrial codebase without altering its observable behaviour is non-trivial and is left for future work rather than attempted here.

Second, execution time on a general-purpose operating system is subject to noise from the Java Virtual Machine's just-in-time compiler, garbage collector and OS scheduler. None of the reported runs included a JVM warm-up period, multiple independent JVM invocations, or explicit control of garbage-collection pauses; the study instead reports the mean, standard deviation, confidence interval and a non-parametric test alongside the t-test (Section 5.4) to compensate partially, and the two higher-latency runs identified in Case Study 2 are consistent with exactly this kind of noise. A rigorous replication should instead follow an established microbenchmarking protocol such as JMH (Java Microbenchmark Harness), which runs a warm-up phase before the timed iterations, forks a fresh JVM per configuration, and reports iteration-level variance.

Third, execution time is reported here strictly as an indicator of the run-time cost of the method-resolution mechanism itself, not as a maintainability metric in its own right, and should not be read as a measurement of how long a maintainer would take to understand, modify or extend the code. The static WMC/DIT/CBO metrics in Section 5.3 are closer in spirit to established maintainability indicators, but even these remain structural proxies rather than a direct measurement of maintenance effort.

A controlled maintenance-task experiment, in the style of Daly et al. [20] - in which participants are timed while performing an identical modification on the overriding and non-overriding versions of the same program - remains the most direct way to test whether the differences reported here translate into a difference in actual maintenance effort, and is identified as the single most valuable extension of this study.

## 6. Conclusion and Future Work

This paper presented two small, controlled Java case studies that isolate the effect of function overriding and function overloading on program size and measured execution time. In both cases, the version that used the polymorphic construct required more lines of code and more execution time than an equivalent version that avoided it, with the effect substantially larger for overriding than for overloading. Since LOC and run-time complexity are established, if imperfect, proxies for maintenance effort, the results support the view that the design-time convenience of

overriding and overloading carries a measurable, quantifiable complexity cost that should inform, rather than be ignored in, object-oriented design decisions. As part of finalising this paper, the author also identified and corrected an arithmetic labelling error in an earlier working chart for Case Study 2, underscoring the value of re-deriving summary statistics from raw data before publication.

The study also reports static structural metrics - WMC, DIT and CBO - computed directly from the case-study source code, which show that overriding, but not overloading, measurably increases WMC (Section 5.3), and confirms the significance of the execution-time differences using confidence intervals, Cohen's *d* and both parametric and non-parametric significance tests rather than descriptive statistics alone (Section 5.4).

Future work will extend this study in three directions: replicating both case studies on a larger, industrial-scale code base, using a full UML/XMI metric-extraction pipeline [10] to test whether the WMC asymmetry identified in Section 5.3 persists at scale, since the WMC/DIT/CBO metrics reported here were computed directly from the two small case studies rather than through such a pipeline; adopting a JMH-based microbenchmarking protocol with JVM warm-up and repeated independent trials to control for the run-to-run variance discussed in Section 5.6; and running a controlled human-subject maintenance-task experiment, similar in design to Daly et al. [20], to test directly whether the complexity differences reported here translate into measurable differences in the time required to understand, modify and correct the two versions of each program.

## References

- [1] V.S. Alagar, Li Qiaoyum, and O.S. Ormandjieva, "Assessment of Maintainability in Object-Oriented Software," *Proceedings of the 39<sup>th</sup> International Conference on Technology of Object-Oriented Languages and Systems (TOOLS39)*, Santa Barbara, CA, USA, pp. 194-205, 2001. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [2] ISO/IEC 9126: Software Engineering, 2025. [Online]. Available: <https://www.geeksforgeeks.org/software-engineering/iso-iec-9126-in-software-engineering/>
- [3] S.R. Chidamber, and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, 1994. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [4] Fernando Brito e Abreu, and Rogerio Carapuça, "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework," *Journal of Systems and Software*, vol. 26, no. 1, pp. 87-96, 1994. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [5] Tom Gilb, "Designing Maintainability in Software Engineering: A Quantified Approach," *International Council on Systems Engineering*, 2008. [[Google Scholar](#)]
- [6] Rikard Land, Measurements of Software Maintainability, *Proceedings of the 4<sup>th</sup> ARTES Graduate Student Conference*, pp. 1-7, 2002. [Online]. Available: [https://www.artes.uu.se/events/gskonf02/papers/Land\\_Maintainability.pdf](https://www.artes.uu.se/events/gskonf02/papers/Land_Maintainability.pdf)
- [7] Brian Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*, Prentice Hall, United States, 1995. [[Google Scholar](#)] [[Publisher Link](#)]
- [8] D.P. Darcy, and C.F. Kemerer, "OO Metrics in Practice," *IEEE Software*, vol. 22, no. 6, pp. 17-19, 2005. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [9] L.C. Briand, J.W. Daly, and J.K. Wüst, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91-121, 1999. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [10] S.W.A. Rizvi, and R.A. Khan, "Maintainability Estimation Model for Object-Oriented Software in Design Phase (MEMOOD)," *Journal of Computing*, vol. 2, no. 4, pp. 26-32, 2010. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]

## Conflicts of Interest

The author declares that there is no conflict of interest regarding the publication of this article.

## Funding Statement

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

## Acknowledgments

The author thanks the reviewers of this journal for their comments on earlier drafts of this manuscript.

## Declaration on the Use of Artificial Intelligence

During the preparation of this manuscript, the author used a large language model (Claude, Anthropic) to assist with reorganising and rewriting draft material originally prepared by the author, checking internal consistency between the text, tables and figures, and drafting formatting consistent with the journal template. The author reviewed, verified and edited all AI-assisted content, takes full responsibility for the accuracy and integrity of the final manuscript, and confirms that all reported data, figures and code originate from the author's own experiments.

## Author Contribution Statement

Ramveer Gurjar is the sole author of this article and is responsible for all aspects of the work. Conceptualization, study design, implementation of the Java case studies, data collection, formal analysis, and writing of the original draft and its revisions were carried out solely by the author. The author read and approved the final manuscript.

- [11] K. El Emam, "A Primer on Object-Oriented Measurement," *Proceedings of the Seventh International Software Metrics Symposium*, London, UK, pp. 185-187, 2001. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [12] Mira Kajko-Mattsson et al., "A Model of Maintainability-Suggestion for Future Research," *Proceedings of 2006 International Conference on Software Engineering Research and Practice*, vol. 1, pp. 436-441, 2006. [[Publisher Link](#)]
- [13] Ilja Heitlager, Tobias Kuipers, and Joost Visser, "A Practical Model for Measuring Maintainability," *6<sup>th</sup> International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, Lisbon, Portugal, pp. 30-39, 2007. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [14] Bill Andreopoulos, "Achieving Software Quality using the NFR Framework: Maintainability and Performance," *Proceedings of the 3<sup>rd</sup> International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications*, pp. 1-6, 2004. [[Google Scholar](#)]
- [15] Nary Subramanian, and Lawrence Chung, "Software Architecture Adaptability: An NFR Approach," *Proceedings of the 4<sup>th</sup> International Workshop on Principles of Software Evolution (IWPSE)*, New York, United States, pp. 52-61, 2001. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [16] Anton Yrjönen, and Janne Merilinnä, "Extending the NFR Framework with Measurable Non-functional Requirements," *Proceedings of the 2<sup>nd</sup> International Workshop on Non-functional System Properties in Domain Specific Modeling Languages (NFPinDSML)*, United States, 2010. [[Google Scholar](#)] [[Publisher Link](#)]
- [17] James M. Bieman, and Josephine Xia Zhao, "Reuse through Inheritance: A Quantitative Study of C++ Software," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. SI, pp. 47-52, 1995. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [18] M. Cartwright, and M. Shepperd, "An Empirical Investigation of an Object-oriented Software System," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 786-796, 2000. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [19] V.R. Basili, L.C. Briand, and W.L. Melo, "A Validation of Object-oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751-761, 1996. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [20] John Daly et al., "Evaluating Inheritance Depth on the Maintainability of Object-oriented Software," *Empirical Software Engineering*, vol. 1, no. 2, pp. 109-132, 1996. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [21] F. Fioravanti, and P. Nesi, "A Study on Fault-proneness Detection of Object-oriented Systems," *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal, pp. 121-130, 2001. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [22] L.C. Briand, W.L. Melo, and J. Wüst, "Assessing the Applicability of Fault-proneness Models Across Object-oriented Software Projects," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 706-720, 2002. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [23] Lionel C. Briand et al., "Exploring the Relationships between Design Measures and Software Quality in Object-oriented Systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245-273, 2000. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [24] Ping Yu, T. Systä, and H. Müller, "Predicting Fault-proneness using OO Metrics: An Industrial Case Study," *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, BudaPest, Hungary, pp. 99-107, 2002. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [25] C. Van Koten, and A.R. Gray, "An Application of Bayesian Network for Predicting Object-oriented Software Maintainability," *Information and Software Technology*, vol. 48, no. 1, pp. 59-67, 2006. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [26] Ruchika Malhotra, "A Systematic Review of Machine Learning Techniques for Software Fault Prediction," *Applied Soft Computing*, vol. 27, pp. 504-518, 2015. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [27] Alisara Hincheeranan, and Wanchai Rivepipoon, "A Maintainability Estimation Model and Tool," *International Journal of Computer and Communication Engineering*, vol. 1, no. 2, pp. 143-146, 2012. [[CrossRef](#)] [[Publisher Link](#)]
- [28] Jan Hannemann, and Gregor Kiczales, "Design Pattern Implementation in Java and AspectJ," *Proceedings of the 17<sup>th</sup> ACM SIGPLAN Conference Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seattle, Washington, USA, pp. 161-173, 2002. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [29] Jianjun Zhao, "Data-Flow-based Unit Testing of Aspect-Oriented Programs," *Proceedings 27<sup>th</sup> Annual International Computer Software and Applications Conference COMPAC 2003*, Dallas, TX, USA, pp. 188-197, 2003. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [30] D. Mishra, and A. Mishra, "Efficient Software Review Process for Small and Medium Enterprises," *IET Software*, vol. 1, no. 4, pp. 132-142, 2007. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [31] Tjasa Heričko, and Bostjan Šumak, "Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems," *Applied Sciences*, vol. 13, no. 5, pp. 1-36, 2023. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [32] Gokul Yenduri, and Thippa Reddy Gadekallu, "A Review on Soft Computing Approaches for Predicting Maintainability of Software: State-of-the-art, Technical Challenges, and Future Directions," *Expert Systems*, vol. 40, no. 7, 2023. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [33] M. Debbabi et al., "Method Call Acceleration in Embedded Java Virtual Machines," *Computational Science-ICCS 2003: International Conference, Melbourne, Australia and St. Petersburg, Russia*, pp. 750-759, 2003. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]

## Appendix 1. Case Study 1 — Java Source Code With Function Overriding

```
// Define superclass Shape
class Shape {
    private String color;
    private int length;
    private int width;
    public Shape(String color) { this.color = color; }
    @Override
    public String toString() { return "Shape of color=\"" + color + "\""; }
    public double getArea() {
        System.err.println("Shape unknown! Cannot compute area!");
        return 0;
    }
    public void setArea(int length, int width) { this.length = length; this.width = width; }
    public void setArea(int length) { this.length = length; this.width = length; }
}

// Define Rectangle, subclass of Shape
class Rectangle extends Shape {
    private int length;
    private int width;
    public Rectangle(String color, int length, int width) {
        super(color); this.length = length; this.width = width;
    }
    @Override
    public String toString() {
        return "Rectangle of length=" + length + " and width=" + width + ", subclass of " +
super.toString();
    }
    @Override
    public double getArea() { return length * width; }
    public void setArea(int length, int width) { this.length = length; this.width = width; }
    public void setArea(int length) { this.length = length; this.width = length; }
}

// Define Triangle, subclass of Shape
class Triangle extends Shape {
    private int base;
    private int height;
    public Triangle(String color, int base, int height) {
        super(color); this.base = base; this.height = height;
    }
    @Override
    public String toString() {
        return "Triangle of base=" + base + " and height=" + height + ", subclass of " + super.toString();
    }
    @Override
    public double getArea() { return 0.5 * base * height; }
    public void setArea(int base, int height) { this.base = base; this.height = height; }
    public void setArea(int base) { this.base = base; this.height = base; }
}

// Driver program for Shape and its subclasses
public class TestShape {
    public static void main(String[] args) {
        long startTime = System.nanoTime();
        Rectangle s1 = new Rectangle("red", 4, 5);
        System.out.println("Area is " + s1.getArea());
        Triangle s2 = new Triangle("blue", 4, 5);
        System.out.println("Area is " + s2.getArea());
        long endTime = System.nanoTime();
        System.out.println("Execution time (ns): " + (endTime - startTime));
    }
}
```

## Appendix 2. Case Study 1 — Java Source Code Without Function Overriding

```
class Shape {
    private String color;
    public Shape(String color) { this.color = color; }
}

class Rectangle extends Shape {
    private int lengthR, widthR;
```

```

public Rectangle(String color, int lengthR, int widthR) {
    super(color); this.lengthR = lengthR; this.widthR = widthR;
}
public String toStringR() { return "Rectangle of length=" + lengthR + " and width=" + widthR; }
public double getAreaR() { return lengthR * widthR; }
public void setAreaR(int lengthR, int widthR) { this.lengthR = lengthR; this.widthR = widthR; }
public void setAreaR(int lengthR) { this.lengthR = lengthR; this.widthR = lengthR; }
}
class Triangle extends Shape {
    private int baseT, heightT;
    public Triangle(String color, int baseT, int heightT) {
        super(color); this.baseT = baseT; this.heightT = heightT;
    }
    public String toStringT() { return "Triangle of base=" + baseT + " and height=" + heightT; }
    public double getAreaT() { return 0.5 * baseT * heightT; }
    public void setAreaT(int baseT, int heightT) { this.baseT = baseT; this.heightT = heightT; }
    public void setAreaT(int baseT) { this.baseT = baseT; this.heightT = baseT; }
}
public class TestShapeNoOverride {
    public static void main(String[] args) {
        long startTime = System.nanoTime();
        Rectangle s1 = new Rectangle("red", 4, 5);
        System.out.println("Area is " + s1.getAreaR());
        Triangle s2 = new Triangle("blue", 4, 5);
        System.out.println("Area is " + s2.getAreaT());
        long endTime = System.nanoTime();
        System.out.println("Execution time (ns): " + (endTime - startTime));
    }
}

```

### Appendix 3. Case Study 2 — Java Source Code With Function Overloading

```

class Findarea {
    void area() { System.out.println("a: "); }
    void area(int b) {
        double area = 22.7 * b * b;
        System.out.println("Area of Circle:" + area);
    }
    void area(int a, int b) {
        double area = a * b;
        System.out.println("Area of Rectangle:" + area);
    }
}
public class TestOverload {
    public static void main(String[] args) {
        long startTime = System.nanoTime();
        Findarea obj = new Findarea();
        obj.area();
        obj.area(5);
        obj.area(4, 5);
        long endTime = System.nanoTime();
        System.out.println("Execution time (ns): " + (endTime - startTime));
    }
}

```

### Appendix 4. Case Study 2 — Java Source Code Without Function Overloading

```

class Findarea {
    void area() { System.out.println("a: "); }
    void areacircle(int b) {
        double area = 22.7 * b * b;
        System.out.println("Area of Circle:" + area);
    }
    void arearectangle(int a, int b) {
        double area = a * b;
        System.out.println("Area of Rectangle:" + area);
    }
}
public class TestNoOverload {
    public static void main(String[] args) {
        long startTime = System.nanoTime();
    }
}

```

```
Findarea obj = new Findarea();  
obj.area();  
obj.areacircle(5);  
obj.arearectangle(4, 5);  
long endTime = System.nanoTime();  
System.out.println("Execution time (ns): " + (endTime - startTime));  
}  
}
```