

Assessing the Construct Validity of Object-Oriented, Class-Level Code Quality Metrics

Hera Arif ✉ 

Dalhousie University, Halifax, Canada

Miikka Kuuttila ✉ 

LUT University, Lahti, Finland

Paul Ralph ✉ 

Dalhousie University, Halifax, Canada

Abstract

Background. Code quality metrics are intended to measure latent properties of software source code. Although numerous code metrics have been proposed and used, their construct validity is rarely evaluated. Thus, the extent to which code metrics actually measure what they claim to measure is often unclear.

Aim. Drawing from modern measurement theory, we investigate the construct validity of common class-level, object-oriented code quality metrics.

Method. As code quality metrics are intended to reflect latent attributes, such as cohesion and coupling, we identified the factor structure of code quality metrics using *Exploratory Factor Analysis* (EFA). The metrics were extracted from the Apache Maven project by three software tools: *Designite*, *JHawk*, and *Understand*. The factor structure was later verified using *Confirmatory Factor Analysis* (CFA) on 22 randomly selected open source projects meeting a predetermined eligibility criteria.

Results. 24 code quality metrics that correspond to six constructs: *Cohesion*, *In-Coupling*, *Out-Coupling*, *Size*, *Sub-Inheritance (related to subclasses)*, and *Sup-Inheritance (related to superclasses)* were revealed in the underlying factor structure. Ten metrics did not correspond to any known dimension of software quality and were removed in the exploratory analysis. Ten additional metrics exhibited low loadings in the confirmatory analysis, suggesting their removal from the final measurement model. *Size*, *Cohesion*, *Inheritance*, and *Coupling* were the constructs retained, with subcategories identified for *Inheritance* and *Coupling*.

Conclusions. Our results strongly support the construct validity of 24 code quality metrics. *Coupling* and *Inheritance* are revealed as multidimensional constructs, since they require measuring two different concepts, revealed as sub-categories in our analysis, and *Complexity* may be better explored in a multilevel model. Our results also corroborate the relationship between *Cohesion*, *Size*, and *Out-Coupling* which can be further explored in a structural model. Some metrics from the Chidamber & Kemerer metrics suite are found to perhaps be measuring different constructs than intended. Our results reveal the need for creating or integrating metrics that reflect existing constructs but measure fundamentally different properties to improve the content validity of the measurement model. Additionally, we provide useful recommendations for researchers, developers, and tool providers which stem from theoretical and empirical justification. Overall, our study demonstrates the value of applying modern measurement theory and latent variable modeling in validating software code quality metrics.

2012 ACM Subject Classification Software and its engineering → Empirical software validation

Keywords and phrases Code quality metrics, Factor analysis, Exploratory factor analysis, Confirmatory factor analysis, Software quality, Size, Inheritance, Coupling, Cohesion

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.10

Category Technical Track Paper

Supplementary Material Dataset: <https://doi.org/10.5281/zenodo.20278447>

Funding Paul Ralph: NSERC Discovery Grant RGPIN-2020-05001



© Hera Arif, Miikka Kuuttila, and Paul Ralph;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 10; pp. 10:1–10:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Code quality metrics provide quantitative measures that evaluate attributes of a software system. Here, code quality refers to the characteristics of source code that influence its overall quality, including complexity, modifiability, understandability, and testability. Accordingly, efforts to improve software quality generally target these quality attributes, either explicitly or implicitly [63]. Code metrics are used by developers to provide insights about the attributes of software, and are a popular topic of interest for software developers and researchers. Despite the proliferation of proposed metrics, metrics research suffers from several persistent challenges. The software development industry appears to rely on metrics in a very limited manner [63]; while measurements in other fields form the foundation of a successful system, software developers often favor popular, easy to compute metrics [52, 58]. More fundamentally, a key insight in both recent philosophy of science and theory of measurement is that observation is intrinsically problematic [51]. Most observations are mediated by instruments and measure the effects of “latent” structures rather than the structures themselves. In this vernacular, source code is a *latent structure* because we cannot directly observe its execution without instruments (e.g. computer screen, IDE, debugger, terminal window). Important aspects of code such as size, complexity, coupling, and cohesion are *constructs* because, again, we cannot observe them directly. We *operationalize* a construct using one or more *metrics* (e.g. a script that counts lines of code).

Thinking about code metrics this way highlights several problems:

- The instrument is defective or does not do what the researcher thinks it does (e.g. script for counting lines of Java code that accidentally counts commas instead of semicolons).
- The instrument measures the property correctly, but the property is not caused by the target construct (e.g. the researcher aims to measure complexity, but counts lines of code, which is caused by the system’s size rather than its complexity).
- A multidimensional property like coupling (which has direction, strength, and granularity [10]) is operationalized using a single instrument (e.g. Chidamber & Kemerer’s CBO [14]), which considers incoming and outgoing links as having equivalent strength, disregarding distinctions between these dimensions).
- An instrument correctly measures the target property, but the property is caused by multiple constructs (e.g. the McCabe [45] and Halstead [30] metrics are commonly used to measure complexity, but they are highly correlated with the lines of code in a system [26]).

Construct validity concerns problems like these—validation refers to the process of ascertaining if an indicator actually measures the aspect it proposes to measure (theoretical validation) and asserting that the indicator relates to other similar measures in an expected way (empirical validation) [52]. Assessing how well an indicator operationalizes a construct is crucial for the integrity of a measurement. Given the abundance of existing and newly proposed metrics, without a corresponding emphasis on validating them, it becomes challenging to assess how well a metric aligns with the underlying construct it intends to measure. Currently, metric’s measurement is mostly dependent upon what a software developer understands the metrics definition to be [57, 62]. For example, metrics with the same name are implemented by two tools differently and differ in the resulting measurement values if applied to the same software code (see § 3.1.2). Moreover, code quality metrics defined at the method-level are being used to infer information about class-level attributes—the results of a method level complexity metric (CC [45]) is being averaged to analyze the complexity of a class as a whole (see § 3.1.1). This results in a bad metric value being masked and compensated by good metric values, influencing the overall result. Another example is that of Maintainability Index

(MI) which is considered a proxy of software maintainability, however, a multifaceted concept like maintainability cannot be fully captured by a single metric [60]. In fact, metrics chosen to represent maintainability may influence the outcome more than the actual maintainability of code [59]. This may mislead professionals by providing them a false sense of confidence in their decisions, or discourage them from relying on code metrics when making important decisions altogether [16, 59, 63], undermining the very purpose of code metrics, which is to offer scientifically grounded evidence for the measurement of concepts.

Thus, the aim of this study is to investigate the validity of common code quality metrics and the research question of this study is as follows:

Research Question: Do object-oriented, class-level, code quality metrics measure the latent quality attributes they are supposed to measure?

To accomplish this, we conduct an *Exploratory Factor Analysis* (EFA)—a statistical approach used to discover “factors”, “constructs”, or “latent variables” from a set of variables (in our case code quality metrics) by analyzing the correlations between them [28]. This is followed by a *Confirmatory Factor Analysis* (CFA)—which tests the model revealed by the EFA to assess how well the specification of factors and variables represents actual data [29].

2 Related Work

In this section, we provide an overview on the theory of measurement, the current state of metrics literature and prior research related to our study.

2.1 Contemporary Theory of Measurement

Measurement is the principled assignment of numbers to reality [11]. A metric is the process of representing a phenomenon numerically. Operationalizing unobservable phenomena using a single metric is philosophically and methodologically outdated because it assumes measurement is unproblematic and ignores mono-method bias [51]. Instead, contemporary measurement theory suggests constructing measurement models that operationalize each phenomenon (latent variable) as the shared variance of multiple diverse metrics.

Software maintainability, flexibility, understandability are external quality dimensions that researchers acknowledge cannot be observed directly. Whereas, software complexity, coupling, cohesion, are specific internal quality criteria that researchers strive to measure directly using new or improved metrics. However, multiple metrics differ slightly and sometimes significantly when measuring the same construct (see § 3.1.2). This suggests that metrics are indirect measures and should not be used as a direct representation of the construct. Thus, code quality is a *latent structure* and metrics can be considered indicators that measure a theoretical concept that cannot be measured directly [33].

2.2 Current State of Code Quality Metrics Research

A code quality metric, in essence, measures attributes of a software by analyzing its source code. For example, LOC measures the lines of code of software code. Initially, research related to code quality metrics focused on size and complexity metrics like LOC, CC [45], and the Halstead metrics [30] but then shifted to object-oriented (OO) metrics like the C&K [14], MOOD [1], and Martin’s design metrics [44, 58]. Researchers are increasingly trying to determine factors that are holding back metrics research and how these problems can be fixed [63]. Some common criticisms are as follows. **Limited use:** Code quality

metrics' impact on software development is underexplored, with metrics like LOC and CC used commonly; while others ignored or used rarely [58]. **Lack of precise definitions and guidelines:** Every new metric is expected to provide a sound theoretical foundation and a precise definition, which seem to be lacking in the present literature [58]. Additionally, there are numerous discrepancies in the naming conventions of metrics. For example, Depth of Inheritance Tree (DIT) is also called Depth of Inheritance (DIH) but both these metrics measure the maximum length of a tree from node to root [56]. **Shift towards code smells:** More recently, code smells are used as a metaphor to depict patterns commonly linked with poor design and programming practices [57]. However, the idea of code smells departs significantly from contemporary theories of measurement. Code quality is a latent structure whose measurement requires a statistical measurement model that specifies the dimensions of code quality and operationalizes each of them as the shared variance of multiple indicators. The presence or frequency of one or more code smells could constitute some of the indicators in this measurement model. However, focusing only on code smells to measure code quality is insufficient and doesn't exempt the researcher from constructing a sound measurement model and quantitatively assessing its validity. **Rise of agentic code:** As software developers incorporate more agentic code generation, a simultaneous increase in assessing the quality of code generated by AI is observed. Recent studies have employed the use of traditional code quality metrics like MI, LOC, and CC to understand the quality of agentic code developed by various generative AIs like GitHub Copilot ¹, ChatGPT ², etc. [35,49]. These studies highlight generative AI's focus on accelerating programming, sometimes even at the expense of code quality. **Poor validation:** Over the years, researchers have proposed numerous approaches for validating software metrics. Some researchers propose a set of "validity criteria" to assess metrics [65]. Weyuker et al. [65] define a set of "properties" for complexity metrics; and while Cherniavsky et al. [13] and Fenton [21] disagree with this approach, Chidamber et al. [14] support it. Another example is the findings of a literature review by Meneenly et al. [46] that reports 47 validation criteria for software code quality metrics highlighting the ad-hoc nature of metrics validation as well as conflicting and subjective perspectives among researchers. Some studies interpret correlation as validation, considering metrics valid if they are correlated with other metrics measuring the same construct [21]; however, correlation and validation are not interchangeable. Some empirical studies consider metrics as predictors of external features and consider this relationship as evidence of metric validity. Basili et al. [3] investigated the C&K metrics suite's [15] effectiveness as predictors of fault proneness and Briand et al. [8] investigated the relationship between object-oriented metrics and the likelihood of fault detection in classes. They found that metrics were effective [3] but redundant [8] as they captured similar attributes of the software. Emam et al. [20] and Gil et al. [24] studied the relationship between external quality attributes and size metrics. They recommended that future studies take the confounding effect of size into consideration when investigating the validity of metrics. Researchers have also attempted to create unified frameworks for software quality measurement (such as [5–7,39]). These studies emphasized the need for stronger theoretical foundations for software measurement and the need for further research into scientifically sound methods for measuring software to ensure valid measurements. They also highlighted that metric's validation remains an unresolved challenge that must be addressed to achieve reliable measurement of software quality.

¹ <https://github.com/features/copilot>

² <https://chatgpt.com/>

The underlying *factor structure*—the relationships that exist between variables (code quality metrics) and the constructs they are supposed to be measuring, is not taken into consideration in most metrics research and unvalidated metrics are used as *direct measures* to measure code quality constructs (such as size, complexity, and cohesion) [52]. The metrics literature appears to overlook the distinction between direct measures and indirect proxy measures and largely focuses on identifying a single direct metric that can be applied to every objective [4, 21]. Consider two variants of C&K’s LCOM [14]: LCOM2 [15] and LCOM5 [32]; they are both meant to measure the cohesion in software code, so a strong correlation between them is expected. In fact, this is not the case (see *Discussion*). These issues are addressed by “construct validity”—the degree to which a warranted inference can be made for a construct using a measure. Construct validity includes assessing: *face validity*—does a measure “on its face” seem to be a good measure of the construct?, *content validity*—is the measure representative of relevant domain related to the construct?, *convergent validity*—how similar is the measure with other measures it should be theoretically similar to?, *discriminant validity*—how different is the measure with other measures it should theoretically be different to?, and *predictive validity*—does the measure predict something it should theoretically be able to predict? [64].

2.3 Prior Research on Factor Analysis and Code Metrics

Munson et al. [47] performed an EFA to reduce a number of complexity metrics into a smaller set of independent complexity domains. They analyzed 106 modules and 16 software complexity code metrics produced by their Ada Metric Analyzer to group complexity into five constructs: control, size, information content, modularity, and data structure. Al et al. [41] conducted a study to identify the efficiency of traditional metrics in measuring the complexity of OO code. They used the CPPOOM (C++ ObjectOriented Metric) tool to calculate metrics on four C++ systems, analyzing 43 trees and 222 classes, to reveal four complexity constructs. They concluded that traditional metrics loaded on a single factor and were correlated with other traditional metrics and encouraged further research on the topic. Cook et al. [17] conducted a study to analyze ten releases of a real-time telephone switching system written in a macro-assembly language. They calculated 18 complexity metrics and conducted a principal components analysis to reduce dimensionality. The results revealed four complexity domains: size, information flow into/out of functions, and control flow.

Prior research in this domain seems to focus on a single construct of complexity with the aim of finding sub-categories or performing dimension reduction [17, 41, 47]. These studies are conducted over 30 years ago with a small dataset of code quality metrics and lack implementation details. We aim to build upon this research by considering numerous metrics and constructs, and providing a publicly accessible dataset with a comprehensive replication package that can be used to re-implement our analysis and ensure accuracy and reliability.

3 Methodology

Assessing the construct validity of code quality metrics requires a (rudimentary, incomplete) theory of code quality that can be generated inductively from data by examining the factor structure of metrics using factor analysis. Briefly, our methodology involves selecting some source code to be fed into software tools that calculate code quality metrics. These metrics are screened and excluded if not appropriate for a factor analysis. The construct validity of the resulting metrics is then assessed iteratively using an exploratory and a confirmatory factor analysis revealing six constructs comprising 24 metrics. All statistical analysis is performed with R version 4.3.2, using the R packages *psych* [53] and *lavaan* [54, 55].

3.1 Exploratory Factor Analysis

As the name suggests, an EFA “explores” the underlying factor structure represented by variables to develop a measurement model [29]. Our objective is to assess the construct validity of common, object-oriented, class-level, code quality metrics. We followed the method outlined by Hair et al. [28] to conduct an EFA by performing the following steps.

3.1.1 Designing the Factor Analysis

To assess the factor structure of the code quality metrics, we must compute metrics for some code. Representative sampling is neither practical nor desirable for the initial, exploratory stage (see *Limitations*). Rather, we want a single project that is large enough for the factor structure to emerge but small enough that the codebase is comprehensible. To avoid becoming overwhelmed by complexity, we begin by focusing on class-level Java metrics because both the language and quality concepts would be familiar to many researchers interested in software metrics. This includes many popular metrics (e.g. C&K [14], L&K [42], MOOD metrics [31]).

- **Tools Selection Criteria:** The software tools must calculate a wide range of class-level metrics, and must be able to analyze large projects containing mainly Java code.
- **Project Selection Criteria:** In a preliminary, exploratory phase, we only require a project with enough observations for the factor structure to emerge—which is a property of the variables themselves and not the objects of the study (see *Limitations*). Additionally, the project should contain at least 10 observations per variable [28].
- **Class Selection Criteria:** Exclude any classes for which one or more selected tools cannot calculate *any* metrics.
- **Metrics Selection Criteria:** Metrics should be numerical, at an ordinal, interval, or ratio level. They should measure something about the code structure at class-level that corresponds to an identifiable construct, and should not be a function of method-level metrics.

The software tools selected were: Designite,³ Understand,⁴ and JHawk⁵ which calculate 11, 45, and 40 class-level code quality metrics respectively, yielding a total of 96 metrics. These tools are popular for metrics research because they each contain a wide range of metrics found in the literature and are calculated at different levels (e.g. file, class, method level). There could be other tools that meet our selection criteria. It is not necessary for our purposes to include every tool that meets the above criteria. We selected the *Apache Maven* project,⁶ a popular open-source Java tool used to manage external dependencies and automatically build software projects as the selected project should contain at least 960 classes (10*96). The Apache Maven source code was downloaded from its GitHub repository⁶ and compiled. It contains around 1,300 classes which are sufficient for our analysis. Special Java classes like enums, anonymous, or nested inner classes are not recognized by some tools so they are excluded from our analysis (e.g. Understand does not calculate LCOM metrics for enums). Non-numerical metrics such as NAME OF SUPER CLASS were excluded. Some metrics were calculated at the method-level and summed to produce a class-level measurement. For example, CC is calculated at the method-level to estimate class complexity. We exclude such metrics because there are more sophisticated ways of handling this situation

³ DesigniteJava Enterprise (v. 2.1.2), <https://www.designite-tools.com/>

⁴ Understand (v. 6.1), <https://www.scitools.com/>

⁵ JHawk (v. 6.1.7), <http://www.virtualmachinery.com/jhawkprod.htm>

⁶ Apache Maven (v. 2.0), <https://github.com/apache/maven>

(see *Future Work*). To determine which construct each metric ostensibly reflects, we refer to the documentation provided by the software metric tools. For metrics that were not categorized into any constructs, we reviewed the metrics literature. A complete list of the included and excluded metrics and their mapped constructs is provided in the replication package (see *Data Availability Statement*). Based on the selection criteria, 33 metrics out of the 96 metrics calculated were excluded. The final dataset contains measurements from 35 size metrics, 6 cohesion metrics, 11 inheritance metrics, and 11 coupling metrics—63 in total. We aim to classify these metrics into five factors: size, cohesion, inheritance, in-coupling, and out-coupling, following the classification provided by the software tools. Table 1 shows a brief explanation of these factors in terms of class-level.

■ **Table 1** Definition of factors at class-level.

Factor	Description
Size (S)	The size of the class
Cohesion (C)	Degree to which elements in the class belong together
Inheritance (I)	Degree of hierarchical class reuse and extension
In-Coupling (InC)	Degree to which the class is used by other classes
Out-Coupling (OutC)	Degree to which the class is dependant on or uses other classes

3.1.2 Test Assumptions of Factor Analysis

Factor analysis conceptually assumes that an underlying factor structure exists in the data that is to be analyzed. The purpose of this study is also to reveal this underlying factor structure and to assess its validity. Factor analysis assumes a homogeneous sample of measurements which our dataset complies to. The correlation matrix for our dataset has numerous correlations greater than 0.3 indicating a correlated set of variables appropriate for factor analysis [28]. Since the tools calculate some of the same metrics, some metrics (e.g. *Size.NOM.Understand* and *Size.NOM.JHawk*) have a perfect linear relationship (singularity) indicating a valid measurement. More surprisingly, seemingly different metrics are identical, e.g. *Inheritance.DIT.JHawk* and *Inheritance.CountSup.JHawk* have a perfect linear relationship and *Size.CountDeclMethodDefault.Understand* and *Size.CountComment.JHawk* are highly correlated ($r = 0.83$). Nine such metrics were removed as they caused a “not positive definite” (NPD) which is not factorable. Excluding these nine metrics has little effect because they are so tightly correlated with remaining metrics. The Kaiser–Meyer–Olkin (KMO) test to assess sampling adequacy found the dataset “meritorious” [37] ($KMO = 0.84$). Bartlett’s test of sphericity was significant ($p < 0.001$), indicating correlated variables. The final dataset consists of 1279 observations (classes) and 54 variables (code quality metrics)—a ratio of approximately 20 observations per variable, which is higher than the recommended minimum sample size and thus acceptable for conducting factor analysis [28].

3.1.3 Derive Factors and Assess Fit

We used multiple methods to determine the number of factors to extract: parallel analysis [34] suggests 22 factors, the scree plot [12] suggests 2 or 8 factors, and the Kaiser criteria [38] suggests 8 factors. These methods could overestimate the number of factors (an over-fit model) in very large datasets like ours and involve some degree of subjectivity [12]. A combination of methods can be used to make a definitive decision and prior theoretical knowledge of the expected number of factors can be very beneficial in making this decision [18]. According to theory, we predict five factors: size, cohesion, inheritance, in-coupling, and out-coupling

(Table 1). So, we begin with eight factors since that is the largest reasonable estimate of the number of factors (the parallel analysis estimate of 22 factors is *prima facie* unreasonable). The number of factors retained should have an explained variance of at least 60% [28]. Our initial eight-factor model explained 71% of the variance.

3.1.4 Interpret Factors

We rotate the solution using an *oblique*, “oblmin” rotation as it assumes factors are correlated [28]. The construct validity of the metrics is assessed iteratively using an exploratory factor analysis. The factor solution obtained needs refining as it is overcomplicated and contains numerous unacceptable factor loadings—a *factor loading* explains how much of a variable’s variance can be explained by the factor [28]. A high loading means that the variance explained for a variable is sufficient for it to have a considerable relationship with the factor. The factor solution also contains low communalities (h^2) which is the amount of variance in a variable that can be explained by the factor solution extracted by the EFA [22]. We perform the following steps to refine the model.

1. **Remove variables with low loadings (< 0.5) and low communalities (< 0.5) in the eight-factor model:** We iteratively removed such variables, rerunning the factor analysis after each removal until no such variables remain (the loadings and communalities change every time a factor analysis is run). Eight variables were removed, increasing the variance explained by the factor model to 83%.
2. **Reduce from eight to six factors:** In the eight-factor model, the *Inheritance* and *Size* variables each loaded on more than two factors, suggesting an over-fit, so we re-ran the factor analysis with fewer factors to investigate. Despite some cross-loadings, we could label the factors more easily and clearly in the six-factor model, which map well into our a-priori constructs: *Size*, *Cohesion*, *In-Coupling*, *Out-Coupling*, and two types of inheritance—*Sub-Inheritance* and *Sup-Inheritance* that measure the characteristics of a class related to its sub-classes and super-classes respectively. The variance explained by the factor model decreased to 74%.⁷
3. **Remove variables with low loadings (< 0.5) and low communalities (< 0.5) in the six-factor model:** Similar to before, we iteratively remove such variables, rerunning the factor analysis after every removal. Two variables were removed, increasing the variance explained by the factor model to 76%.
4. **Remove variables loading on the incorrect factor:** We analyzed the model for variables loading on the wrong construct, deleting them in order of lowest communalities, and ignoring cross-loadings hoping they would integrate into a single high loading. Ten metrics were removed, increasing the variance explained by the model to 85%.

3.1.5 Exploratory Factor Analysis Results

Table 2 shows the final solution, which explains 85% of the total variance with the six factors extracted, which is above the recommended minimum threshold [28]. Each factor contains at least three variables, and correspond to the theorized classification (Table 1), with sub-categories revealed for *Inheritance* i.e. *Sub-Inheritance* (related to sub-classes),

⁷ The variance explained by a factor analysis increases monotonically with the number of factors extracted, so reducing the number of factors will *not* increase the variance explained even if the model is better in other ways.

and *Sup-Inheritance* (related to super-classes). The Cronbach's α for the overall model is $\alpha = 0.92$, which is considered "excellent". The Cronbach's α for each individual factor in the model also has acceptable values indicating internally consistent variables in the model [19].

■ **Table 2** Exploratory Factor Analysis Model.

	C	InC	OutC	S	SubI	SupI	h2
Cohesion.LCOM.Understand	0.90						0.85
Cohesion.LCOMModified.Understand	0.91						0.82
Cohesion.YALCOM.Designite	0.70						0.5
In-Coupling.CBO.JHawk		0.85					0.96
In-Coupling.FANIN.Designite		0.98					0.93
In-Coupling.FANIN.JHawk		0.97					0.92
Out-Coupling.CBO.Understand			0.78				0.86
Out-Coupling.FANOUT.Designite			0.81				0.89
Out-Coupling.FANOUT.JHawk			0.92				0.96
Size.CountDeclMethodPrivate.Understand				0.76			0.68
Size.CountLine.Understand				0.97			0.98
Size.CountLineBlank.Understand				0.52			0.74
Size.CountLineCodeDecl.Understand				0.81			0.87
Size.CountLineCodeExe.Understand				0.97			0.97
Size.CountLineComment.Understand				0.70			0.49
Size.CountLocalMethodCalls.JHawk				0.67			0.57
Size.CountSemicolon.Understand				0.96			0.94
Size.CountStmt.JHawk				0.98			0.97
Size.CountStmt.Understand				0.99			0.98
Size.CountStmtDecl.Understand				0.86			0.95
Size.CountStmtExe.Understand				1			0.96
Size.LOC.Designite				0.99			0.91
Size.LOC.JHawk				0.98			0.97
Size.LOC.Understand				0.98			0.99
Size.NOM.Designite				0.66			0.79
Size.NOM.Understand				0.68			0.79
Size.UWCS.JHawk				0.59			0.82
Sub-Inheritance.CountSub.JHawk					0.89		0.79
Sub-Inheritance.NC.Designite					0.62		0.49
Sub-Inheritance.NC.Understand					0.77		0.68
Sub-Inheritance.SpecializationRatio.JHawk					0.86		0.76
Sup-Inheritance.CountSup.JHawk						0.98	0.96
Sup-Inheritance.DIT.Understand						0.92	0.88
Sup-Inheritance.ReuseRatio.JHawk						0.95	0.91
Cronbach's Alpha	0.74	0.95	0.91	0.94	0.83	0.87	

3.2 Confirmatory Factor Analysis and Results

We followed the method outlined by Hair et al. [29] to test the pre-specified measurement theory obtained from the EFA and assess how well the theory fits population data. The steps followed are described below.

3.2.1 Define Individual Constructs

Considering the 34 code quality metrics measuring six constructs revealed as the result of the EFA, the conceptual definitions of constructs match well with the item wordings, and the relationships are corroborated by previous literature, scholarly peer-review, and tool documentation, thus establishing reasonable face validity. Some metrics seem to clearly be measuring a specific construct (for example, COUNT LINE measures *Size*); and some metrics seem to be conflated (for example, CBO measures both incoming and outgoing dependencies and seems to be conflated between *In-Coupling* and *Out-Coupling*)—therein lying the motivation of this paper.

3.2.2 Develop the Overall Measurement Model

The six latent constructs may correlate with each other as they would in real software where quality dimensions are congeneric. An indicator variable loads on a single construct and error variances between indicators exist but are not allowed to correlate. Every construct is identified with a minimum of three indicators per construct asserting an ideal “over-identified model”—a model with more degrees of freedom than paths to be estimated ($dof = 512$). In the proposed model, all constructs are hypothesized as *reflective* making the direction of causality from the latent construct to the measured indicator variables.

3.2.3 Design a Study To Produce Empirical Results

We used the SEART GitHub Search Engine⁸ to collect a sample of Java projects by applying minimum selection thresholds. These criteria were intended to identify professionally developed and actively maintained projects, rather than hobby or dummy projects. We obtained 71 Java projects, out of which, 64 projects could be compiled using the popular build tools Maven,⁹ Gradle,¹⁰ or Ant¹¹. Any project used in the exploratory analysis cannot be used in the confirmatory phase; thus, one project was excluded. Since we need compiled code to conduct the analysis, we selected the 63 projects and shuffled them using Python’s `random.shuffle` method, and downloaded and compiled them. We need a minimum of 340 classes to conduct the CFA ($34 * 10$) [29]. Since we can get a much larger dataset, we decided to use workload as the deciding factor of sample size. We aim to complete data collection in two weeks (80 hours). Downloading, compiling, and preparing each project takes approximately 3.75 hours.

$$Numberofprojects = \frac{80hours}{3.75hours/project} = 21.33 \quad (1)$$

Thus, 22 projects (1) were included in our analysis.¹² The smallest project included had 506 classes, and the largest project included had 19,833 classes ($M = 3063$). A list of the parameters set and additional details about the included projects can be found in the replication package (See *Data Availability Statement*). A process similar to the one described in § 3.1.1 was followed to extract the code quality metrics collected by Designite, JHawk, and Understand and combine them into a single spreadsheet. Creating a combined dataset does not affect the results of the analysis since we are performing instrumentation not prediction. Given the number of indicators and constructs, our sample size of approximately 100,000 is more than sufficient for analysis.

3.2.4 Assess Measurement Model Validity

A confirmatory factor analysis is conducted on the dataset and a six construct measurement model is produced (Table 3). The purpose of the factor analysis performed in this study was primarily to evaluate the relationship between commonly used software code metrics and their hypothesized constructs, and not to optimize a measurement instrument with a maximized global fit. Thus, some of the metrics showing weaker loadings were intentionally

⁸ <https://seart-ghs.si.usi.ch/>

⁹ <https://maven.apache.org/>

¹⁰ <https://gradle.org/>

¹¹ <https://ant.apache.org/>

¹² All the data collection was completed by May, 2025

retained during model evaluation to assess their behavior within the proposed construct structure. Consequently, we also do not report overall fit indices, as we are investigating whether metrics measure the latent quality attributes they are supposed to measure, and not what is the most accurate way to do so.

■ **Table 3** Six Construct CFA Result.

	Size	Out-C	In-C	C	Sub-I	Sup-I
Size.CountDeclMethodPrivate.Understand	0.29					
Size.CountLine.Understand	0.97					
Size.CountLineBlank.Understand	0.74					
Size.CountLineCodeDecl.Understand	0.95					
Size.CountLineCodeExe.Understand	0.97					
Size.CountLineComment.Understand	0.36					
Size.CountLocalMethodCalls.JHawk	0.55					
Size.CountSemicolon.Understand	1.00					
Size.CountStmt.JHawk	0.84					
Size.CountStmt.Understand	1.00					
Size.CountStmtDecl.Understand	0.97					
Size.CountStmtExe.Understand	1.00					
Size.LOC.Designite	0.39					
Size.LOC.JHawk	0.95					
Size.LOC.Understand	0.99					
Size.NOM.Designite	0.36					
Size.NOM.Understand	0.36					
Size.UWCS.JHawk	0.54					
Out-Coupling.CBO.Understand		0.78				
Out-Coupling.FANOUT.Designite		0.93				
Out-Coupling.FANOUT.JHawk		0.95				
In-Coupling.CBO.JHawk			0.99			
In-Coupling.FANIN.Designite			0.81			
In-Coupling.FANIN.JHawk			0.98			
Cohesion.LCOM.Understand				1.00		
Cohesion.LCOMModified.Understand				0.96		
Cohesion.YALCOM.Designite				0.50		
Sub-Inheritance.CountSub.JHawk					0.99	
Sub-Inheritance.NC.Designite					0.24	
Sub-Inheritance.NC.Understand					0.42	
Sub-Inheritance.SpecializationRatio.JHawk					0.92	
Sup-Inheritance.CountSup.JHawk						1.00
Sup-Inheritance.DIT.Understand						0.89
Sup-Inheritance.ReuseRatio.JHawk						0.88
Average Variance Extracted (AVE)	0.61	0.80	0.86	0.73	0.51	0.86
Construct Reliability (CR)	0.96	0.92	0.95	0.88	0.77	0.95
Correlation Matrix*						
Size	1.00	0.10	0.02	0.03	0.00	0.00
Out-C	0.31	1.00	0.05	0.11	0.00	0.02
In-C	0.15	0.23	1.00	0.01	0.01	0.00
C	0.16	0.33	0.10	1.00	0.00	0.00
Sub-I	0.02	0.06	0.10	0.05	1.00	0.00
Sup-I	0.04	0.13	0.03	0.01	0.02	1.00

*Values below the diagonal depict correlations among constructs and values above the diagonal depict squared correlations

We now assess the validity of the measurement model obtained by assessing the following.

1. **Face, Nomological, and Predictive Validity:** Face validity was established before the factor analysis was conducted (see § 3.2.1). Nomological validity can be estimated by looking at the correlation between constructs and concluding if they make sense. All correlations between the constructs in the model are statistically significant at $p < 0.05$, as

- expected. There is a moderate positive correlation between *Out-Coupling* with *Cohesion* ($r = 0.33$) and *Size* ($r = 0.31$). Other correlations are positive and low. This also satisfies predictive validity in a way that constructs posit predicted causal relationships.
2. **Content Validity:** We are limited to the metrics provided by Designite, JHawk, and Understand. These tools are frequently used in research related to software code quality, so we presume that these metrics cover the breadth of the constructs.
 3. **Convergent Validity:** Convergent validity is estimated using several techniques (Table 3). The *standardized factor loadings* are acceptable for most variables (> 0.7). Variables with low loadings load majorly on *Size* suggesting further investigation into their inclusion in the model. Two loadings are greater than one (have an error variance less than zero)—*Cohesion.LCOM.Understand* ($e = -0.004$) and *Sup-Inheritance.CountSup.JHawk* ($e = -0.007$), a “heywood case”. This can occur as a compensatory mechanism for high residual correlation between highly correlated variables that causes an inflated loading [9,36]. The *AVE* and *CR* of all constructs in the model have acceptable values—(> 0.5) and (> 0.7) respectively, indicating good convergence and internally consistent variables in constructs [29]. Overall, problematic variables do not appear to be significantly harming model fit or internal consistency and sufficient evidence of convergent validity is provided.
 4. **Discriminant Validity:** We compared the squared correlations between constructs with the AVE values for all construct pairs in the model and found that the AVEs were greater, indicating good discriminant validity (Table 3).

4 Results

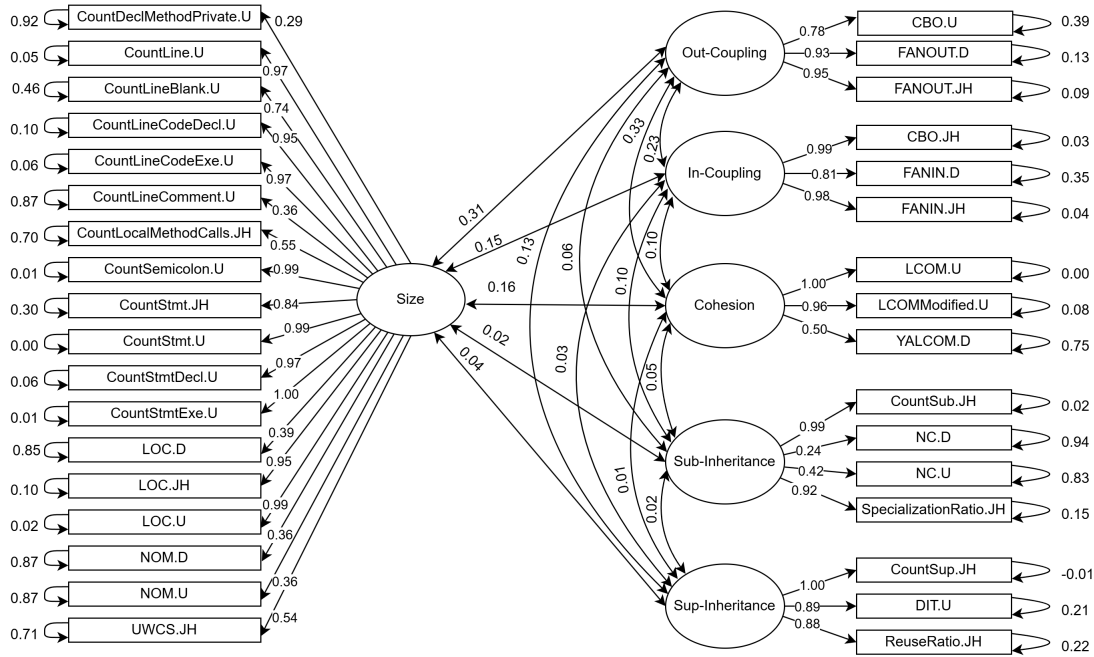
In this section we answer our research question directly:

Research Question: Do object-oriented, class-level, code quality metrics measure the latent quality attributes they are supposed to measure?

Answer: At least 24 object-oriented, class-level metrics appear to measure the construct they are intended to measure, while the inclusion of 10 metrics needs further investigation.

Figure 1 illustrates the measurement model with six latent constructs measured reflectively by 34 indicator variables. To be more precise, of the 63 metrics we assessed:

- 24 metrics seem to reflect the construct they are intended to reflect; i.e. they measure what they are supposed to measure.
- 10 metrics do not reflect any of the constructs in the model (including the construct they are intended to measure); i.e. they do *not* measure what they are supposed to measure.
- 10 metrics loaded on the wrong factor; i.e. they do *not* measure what they are supposed to measure.
- 9 metrics caused an NPD matrix; i.e. our analysis is inconclusive regarding whether they measure what they are supposed to measure.
- 10 metrics load low in the CFA, i.e. their inclusion in the model needs further investigation.



Note: The acronym at the end of the metrics corresponds to the tool being used, with U corresponding to Understand, JH to JHawk and D to Designite

Figure 1 Measurement Theory Model for Code Quality Metrics.

5 Discussion

Our results are surprising in several ways. First, numerous widely cited code quality metrics appear not to measure the construct they purport to measure. For example, multiple metrics proposed by Chidamber et al. [15] were excluded from the final measurement model—*Coupling.RFC.Understand* was removed from the EFA because of low communality and *Cohesion.LCOM2.JHawk* was removed from the EFA because it loaded on *Size*. Additionally, *Sub-Inheritance.NC.Understand* and *Sub-Inheritance.NC.JHawk* loaded low in the CFA suggesting their removal. The fact that so many metrics could be proposed, implemented, and used without anyone demonstrating that they reflect the underlying construct of interest suggests a worrisome lack of concern for construct validity in code metrics research. Researchers should shift their perspective of *proposing* software metrics as the ultimate research objective to instead focusing on *validating* software metrics by adhering to the science of the theory of measurement to ensure that reliable and valid measurements are produced [27, 51, 63].

Second, the *lack* of multicollinearity problems is surprising. Highly (or perfectly) correlated variables prevents the factor structure from computing (the dreaded “NPD matrix”). Meanwhile, the three tools we studied compute many of the same metrics; for instance, all three tools compute LOC, NC, and NOM. Retaining two implementations of the same metric *should* make the *factor analysis* fail, and in some cases it did, but in many cases it did not. The fact that the model includes, for example, numerous LOC metrics, suggests non-trivial differences in the way the three tools compute LOC. As another example, *Inheritance.DIT.Designite* is not highly correlated with *Inheritance.DIT.JHawk* ($r = 0.271$) or *Inheritance.DIT.Understand* ($r = 0.332$). Investigating these differences is beyond the scope of the present study, but something seems wrong here. We retained seven metrics that were calculated by at least two tools: LOC, NC, COUNT STMT, NOM, FANIN, FANOUT, and CBO.

Third, the relationships between the code quality constructs identified in our analysis support our theory that the constructs are correlated with each other, but not to the extent that they measure the same underlying concept. We found a moderately positive correlation between *Out-Coupling* with *Cohesion* ($r = 0.33$) and *Size* ($r = 0.31$). These relationships are also observed in the literature [52] motivating further research, particularly when extending the measurement model to include inter-construct relationships to form a structural model.

Fourth, some metrics are inherently conflated but are popularly used to measure a singular concept. We have established from our analysis that *Coupling* is a multidimensional concept and popular measures (like *Out-Coupling*, *CBO*, *Understand*) are conceptually conflated as they measure the incoming and outgoing connections between code entities. Thus, CBO measures two different concepts and cannot be combined in an attempt to capture both aspects at once [52]. Our previous discussion (see § 2.1) on code quality metrics becoming increasingly detached from science and theory of measurement seems applicable here: measurements should have clear objectives and specify the entities of interest and their significant attributes to ensure reliable and valid measurements [4, 21, 27]. In the case of software code quality, perhaps the high interrelation of code quality constructs often makes it difficult to determine which indicators correspond to which constructs. This observation has also been highlighted by researchers in the past who notice a focus on empirical validation over theoretical validation:

- Fregnan et al. [23] and Gomez et al. [25] discovered that most metrics research primarily focused on empirical validation through comparisons with existing indicators or by assessing correlations; disregarding theoretical validation.
- Kitchenham et al. [40] highlighted that researchers repeatedly empirically validate popular metrics proven to be theoretically invalid; rendering their analysis meaningless.
- Stevanetic et al. [61] noted that validating a metric's relevance requires a theoretical justification supported by empirical evidence, which is reportedly missing in current metrics research.
- Briand et al. [7, 8] emphasized the need for stronger theoretical foundations in software measurement to ensure that metrics accurately measure their intended constructs.

Thus, theoretical *and* empirical validation of the relationship between metrics and constructs using appropriate techniques such as factor analysis is both essential and valuable, and should be encouraged [63].

Fifth, as AI agents become more widely used and AI generated code grows increasingly sophisticated, the need for humans to evaluate, assess, and understand this code also increases. This highlights the need for code that is understandable and readable, with code architecture that is easy for humans to interpret. Researchers are using code quality metrics as one way to assess the quality of AI generated code. For example, Murthy et al. [48] analyze the understandability, maintainability, and complexity of AI generated code using CC [45]. In such studies, metrics are treated as dependent or outcome variables and their effect on a code quality attribute is estimated / predicted. A measurement is only meaningful if it accurately reflects what it is meant to measure. If the measurement lacks validity, the conclusions might be misleading. This highlights the importance of developing valid metrics, benchmarks, and measurement models to support human effort in understanding code quality, which is an objective our research aims to address.

5.1 Recommendations

In light of the findings of this study and an understanding of the theory of measurement which emphasizes that the operationalization of a construct must stem from a theoretical and empirical justification, we offer the following recommendations for researchers, developers, and tool providers.

1. **DO** measure *Cohesion*, *In-Coupling*, *Out-Coupling*, *Size*, *Sup-Inheritance* and *Sub-Inheritance* using the corresponding metrics shown in Table 3. Both professionals and scientists should prefer these metrics.
2. **DO NOT** use the metrics recommended for exclusion from the model due to low / incorrect loadings or low communality; or use them with discretion.
3. **DO NOT** use new, unvalidated metrics; rather encourage validation of newly proposed metrics.
4. **DO** Estimate constructs using factor scores from a confirmatory factor analysis or similar technique.
5. **DO NOT** use individual metrics to estimate constructs.
6. **DO NOT** estimate constructs by averaging / aggregating metrics because metrics vary in distribution and relative importance to their corresponding construct. Use factor scores.
7. **DO** ask providers of software metrics tools to present factor scores and **DO** present factor scores in any tools you build.
8. **DO NOT** propose new metrics without quantitatively assessing construct and measurement validity using factor analysis or a similar technique.
9. **DO NOT** accept for publication manuscripts proposing more unvalidated metrics, or metric proposals without permanently archived source code (because written descriptions of metric algorithms are often incomplete).
10. **DO** use evidence standards (e.g. [50]) to design studies and assess manuscripts; **DO** ensure evidence standards require construct and measurement validity assessment for new metrics.

5.2 Implications

This study serves as a foundational step in the largely under-explored field of assessing the construct validity of software code quality metrics and paves the way for new research opportunities. This study serves as a guiding framework and encouragement for researchers to replicate, generalize, challenge, and extend our findings. To facilitate this, all data used in this study is openly accessible for replication and additional analysis (see *Data Availability Statement*). Researchers may also adopt this approach when developing new metrics to provide evidence of construct validity. The findings of this study encourage further exploration into the relationships between various software code quality constructs in addition to the relationships between variables measuring these constructs. Certain aspects of code quality are inherently subjective yet crucial for developers working with code written by others. For instance, Mannan et al. [43] reported the need for better measurement and modeling of readability as perceived readability is difficult to quantify and integrate into a measurement model. Therefore, further research is needed to evaluate such attributes and their potential incorporation into a structural model.

Software developers and professionals may prefer qualitative assessments such as experience, expert judgment, intuition, or code smells, over quantitative measures like code quality metrics because of the uncertainty in metrics' implementations and measurements. For example, our results find that C&K metrics like RFC, NC, LCOM2 [15] may not accurately capture the quality aspects they are intended to measure. On the other hand, developers and professionals are more likely to trust and adopt code metrics that show evidence of empirical validation (such as through a technique like *factor analysis*) to accurately measure what they claim to measure. Thus, the results of this study can assist professionals and developers in understanding their code and how code changes impact software quality, providing them with clear quantitative assessments and measures to support their decisions.

For tool developers, we recommend recalibrating the inclusion of metrics and thresholds. Tools must provide clear definitions and detailed implementations of metrics they include including specifying what variant of the metric is being used which is not the current practice. For example, Ardito et al. [2] report several software tools do not state what variant of the LCOM metric they calculate. Given the multitude of metrics proposed to measure the same code quality attribute – often yielding conflicting results for the same code – the inclusion of metrics should not be arbitrary nor based solely on their popularity, rather, the evidence of their validity must be the deciding factor for inclusion. Only metrics that reliably measure their intended construct should be used to establish appropriate thresholds.

5.3 Limitations

The recommendations above should be considered in light of several important limitations. Code metrics are not the only way of assessing code quality. Our focus on metrics is not intended as a criticism of any other approach to software quality (e.g. code smells, peer code review, automated testing). We make no claims about the validity of the nine metrics excluded to avoid an NPD matrix (see § 3.1.2). Some of these may be interchangeable with their corresponding, included metrics. We also had to exclude classes for which the tools we studied could not generate data. The usefulness of a metric in practice has other aspects that can be estimated using additional steps. We only focus on construct validity in this study. We need to determine that a measurement model has construct validity before we can assess whether measuring the target construct is practically useful since usefulness presupposes accurate measurement.

For high content validity, we need metrics that count *different* things that are all driven by the same construct. For example, for *Size*, the model includes related but different things like LOC and NOM, which is desired. In contrast, the three *Cohesion* metrics are very similar—*Cohesion* is measured using LCOM and its variants; which raises concerns about the content validity of the measure. Using overly similar metrics can create the illusion of convergent validity which is further corroborated by the existence of a heywood case in the CFA. Our current analysis, however, is limited to metrics that Understand, JHawk, and Designite calculate. Future work may involve solutions such as: adding or removing variables from the model, or increasing the sample size. We have established an extremely large sample size compared to the requirement of a factor analysis, so we suggest the addition of variables in the measurement model (removing variables in this case is not recommended since that would violate the three variables per construct rule).

Regarding generalization, we studied one programming language, three metrics tools, and object-oriented, class-level metrics only. We make no claims about other languages, tools, or abstractions. The metrics and tools studied are frequently used in research, but they might not represent the state of the practice. In terms of representativeness, our sample for the EFA consisted exclusively of the Apache Maven project because applying EFA on a huge dataset is not practical. This is not as serious as it seems because, in the exploratory phase, we only require enough observations (classes) for the factor structure to emerge—the factor structure is a property of the variables (metrics) not the objects of the study (classes). Our sample for the CFA included multiple Java projects. However, these projects may not fully represent all types of code, as they were open-source projects, that met a predetermined eligibility criteria. Thus, we suggest and encourage replicating our analysis with additional, closed-source and industrial software to examine any potential differences in the results.

Our study also has many strengths. As shown in Tables 2 & 3, reliability is high. Verifiability / replicability is also high because we studied a public repository and provide a comprehensive replication package (see *Data Availability Statement*). We demonstrate

novelty by highlighting surprising findings at the beginning of this section. Moreover, while applying the common factor model to code metrics is not entirely new, no prior studies have attempted as comprehensive a model as this one.

5.4 Future Work

This paper reports the first step in a larger research program aiming to improve the validity of code quality measurement. Subsequent steps in this research program include:

- Improving the content validity of the measurement model by including or developing new metrics that reflect existing constructs, but which measure fundamentally different properties from existing metrics. This could include integrating static and dynamic code quality metrics in conjunction with repository mining.
- Extending our analysis to different levels of abstraction (e.g. method, package) using a multi-level model.
- Replicating our analysis on other types of code (e.g. closed-source and industrial software).
- Exploring code smell's integration into the model.
- Extending the model to additional programming languages.

6 Conclusion

To summarize, we used exploratory and confirmatory factor analysis to investigate the construct validity of object-oriented, class-level code metrics generated by three software tools; Designite, JHawk, and Understand. We provide a measurement model with six constructs—*Cohesion*, *In-Coupling*, *Out-Coupling*, *Size*, *Sub-Inheritance*, and *Sup-Inheritance*—each of which is operationalized by at least three metrics. The analysis supports construct validity for 24 of the studied metrics, while the evidence for construct validity was lacking for 30 additional metrics. The metrics identified through this study offer empirically supported, quantitative measures to support informed decision making regarding code quality. Using the factor model shown in Table 3 in its present form is scientifically defensible—especially compared to estimating constructs from individual metrics. However, more research is needed to improve content validity by developing broadened, and more diverse sets of reflective indicators. Taken together, our findings suggest that future software code metrics research should focus on measurement models, instead of interpreting individual metrics as direct measurements in isolation. This study offers the following contributions to the software code quality literature:

- Exploratory and confirmatory analysis related to software code quality metrics, providing a comprehensive and novel investigation of their underlying factor structure.
- Modeling software quality attributes as latent constructs and applying the common factor model to software code metrics. The resulting measurement model and the replication package provide a foundation for future work in researching construct validity of software metrics.
- Highlighting notable findings regarding the relationship between widely cited code quality metrics and the constructs they are intended to measure.

Data Availability Statement

We provide a comprehensive replication package¹³ which includes the detailed study protocol, scripts, results, and raw code quality metrics data for performing the EFA and CFA.

¹³<https://doi.org/10.5281/zenodo.20278447>

References

- 1 Fernando Brito Abreu and Rogério Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *Proc. of the 4th Int. Conf. on Soft. Quality*, volume 186, 1994.
- 2 Luca Ardito, Riccardo Coppola, Luca Barbato, and Diego Verga. A tool-based perspective on software code maintainability metrics: A systematic literature review. *Scientific Programming*, 2020(1):8840389, 2020. doi:10.1155/2020/8840389.
- 3 V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. on Soft. Eng.*, 22(10):751–761, 1996. doi:10.1109/32.544352.
- 4 Eric Bouwers, Joost Visser, and Arie Deursen. Getting what you measure. *Communications of The ACM - CACM*, 55:54–59, May 2012. doi:10.1145/2209249.2209266.
- 5 L. C. Briand, J. W. Daly, and J. Wust. A unified framework for cohesion measurement in object-oriented systems. In *Proc. Fourth Int. Soft. Metrics Symp.*, pages 43–53, 1997. doi:10.1109/METRIC.1997.637164.
- 6 L. C. Briand, J. W. Daly, and J. K. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999. doi:10.1109/32.748920.
- 7 L. C. Briand, S. Morasca, and V. R. Basili. Property-based software engineering measurement. *IEEE Trans. on Soft. Eng.*, 22(1):68–86, 1996. doi:10.1109/32.481535.
- 8 Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *J. of Sys. & Soft.*, 51(3):245–273, 2000. doi:10.1016/S0164-1212(99)00102-8.
- 9 T. A. Brown. *Confirmatory Factor Analysis for Applied Research*. Guilford Publications, 2015.
- 10 Rachel Burrows, Alessandro Garcia, and François Taïani. Coupling metrics for aspect-oriented programming: A systematic review of maintainability studies. In Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski, editors, *Eval. of Novel Approaches to Soft. Eng.*, pages 277–290. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- 11 N.R. Campbell. *Physics: The Elements*. Cambridge University Press, 2013.
- 12 R. B. Cattell. The scree test for the number of factors. *Multivariate Behavioral Research*, 1(2):245–276, 1966.
- 13 J. C. Cherniavsky and C. H. Smith. On weyuker’s axioms for software complexity measures. *IEEE Trans. on Soft. Eng.*, 17, June 1991. doi:10.1109/32.87287.
- 14 S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994. doi:10.1109/32.295895.
- 15 Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Proc. on OOPSLA*, pages 197–211. ACM, New York, USA, 1991. doi:10.1145/117954.117970.
- 16 Shaiful Chowdhury, Reid Holmes, Andy Zaidman, and Rick Kazman. Revisiting the debate: Are code metrics useful for measuring maintenance effort? *Empirical Softw. Eng.*, 27(6), November 2022. doi:10.1007/s10664-022-10193-8.
- 17 Curtis R. Cook and Andreas Roesch. Real-time software metrics. *J. of Sys. & Soft.*, 24(3):223–237, 1994. Oregon Workshop on Software Metrics. doi:10.1016/0164-1212(94)90065-5.
- 18 A. B. Costello and Jason Osborne. Best practices in exploratory factor analysis: Four recommendations for getting the most from your analysis. *Practical Assessment, Research & Evaluation*, 10:1–9, January 2005.
- 19 Lee J. Cronbach. Coefficient alpha and the internal structure of tests. *Psychometrika*, 16:297–334, 1951. doi:10.1007/BF02310555.
- 20 K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. on Soft. Eng.*, 27(7):630–650, 2001. doi:10.1109/32.935855.
- 21 Norman Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. on Soft. Eng.*, 20:199–206, April 1994. doi:10.1109/32.268921.

- 22 A Field. *Discovering Statistics Using IBM SPSS Statistics*. Sage, 5th edition, 2017.
- 23 Enrico Fregnan, Tobias Baum, Fabio Palomba, and Alberto Bacchelli. A survey on software coupling relations and tools. *Inf. & Soft. Technol.*, 107:159–178, 2019. doi:10.1016/j.infsof.2018.11.008.
- 24 Joseph Gil and Gal Lalouche. On the correlation between size and metric validity. *Empirical Soft. Eng.*, 22:2585–2611, 2017. doi:10.1007/S10664-017-9513-5.
- 25 Oswaldo Gómez, Hanna Oktaba, Mario Piattini, and Félix García. A systematic review measurement in software engineering: State-of-the-art in measures. In *Soft. & Data Technol.*, pages 165–176. Springer, Berlin, Heidelberg, 2008.
- 26 T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. on Soft. Eng.*, 26(7):653–661, 2000. doi:10.1109/32.859533.
- 27 Daniel Graziotin, Per Lenberg, Robert Feldt, and Stefan Wagner. Psychometrics in behavioral software engineering: A methodological introduction with guidelines. *ACM Trans. on Soft. Eng. & Methodol.*, 31:7, January 2022. doi:10.1145/3469888.
- 28 J. F. Hair, W. C. Black, B. J. Babin, and R. E. Anderson. *Multivariate Data Analysis*. Always learning. Pearson Education Limited, 2013.
- 29 J. F. Hair, W. C. Black, B. J. Babin, and R. E. Anderson. *Multivariate Data Analysis*. Cengage, 2018.
- 30 Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., USA, 1977.
- 31 Rachel Harrison, Steve J. Counsell, and Reuben V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Trans. on Soft. Eng.*, 24(6):491–496, 1998. doi:10.1109/32.689404.
- 32 Brian Henderson-Sellers, Larry L Constantine, and Ian M Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object oriented systems*, 3(3):143–158, 1996.
- 33 J. Henseler. *Composite-Based Structural Equation Modeling: Analyzing Latent and Emergent Variables*. Methodology in the Social Sciences Series. Guilford Publications, 2020.
- 34 J. L. Horn. A rationale and test for the number of factors in factor analysis. *Psychometrika*, 30:179–185, 1965.
- 35 Baskhad Idrisov and Tim Schlippe. Program code generation with generative ais. *Algorithms*, 17:62, January 2024. doi:10.3390/a17020062.
- 36 Karl G. Jöreskog. Structural equation modeling with ordinal variables. *Lecture Notes-Monograph Series*, 24:297–310, 1994. URL: <http://www.jstor.org/stable/4355811>.
- 37 H. F. Kaiser and J. Rice. Little jiffy, mark iv. *Educational & Psychological Measurement*, 34(1):111–117, 1974.
- 38 Henry F. Kaiser. The application of electronic computers to factor analysis. *Educational & Psychological Measurement*, 20(1):141–151, 1960.
- 39 B. Kitchenham, S.L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Trans. on Soft. Eng.*, 21(12):929–944, 1995. doi:10.1109/32.489070.
- 40 Barbara Kitchenham. What’s up with software metrics? – A preliminary mapping study. *J. of Sys. & Soft.*, 83(1):37–51, 2010. doi:10.1016/j.jss.2009.06.041.
- 41 Al Lake and Curtis R. Cook. Use of factor analysis to develop oop software complexity metrics, 1994.
- 42 Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.
- 43 U. A. Mannan, I. Ahmed, and A. Sarma. Towards understanding code readability and its impact on design quality. In *Proc. of the 4th ACM SIGSOFT Int. Workshop on NLP for Software Engineering*, NL4SE 2018, pages 18–21, NY, USA, 2018. ACM. doi:10.1145/3283812.3283820.
- 44 Robert Cecil Martin. *Designing object-oriented C++ applications*. Prentice Hall, 1995.

- 45 T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi:10.1109/TSE.1976.233837.
- 46 A. Meneely, B. Smith, and L. Williams. Validating software metrics: A spectrum of philosophies. *ACM Trans. Softw. Eng. Methodol.*, 21(4), February 2013. doi:10.1145/2377656.2377661.
- 47 J.C. Munson and T.M. Khoshgoftaar. Measuring dynamic program complexity. *IEEE Soft.*, 9(6):48–55, 1992. doi:10.1109/52.168858.
- 48 Sirisha Murthy, Lohith Nagaraja, Karan Wale, and William Andreopoulos. Evaluating ai code generation quality with software metrics. *SSRN 6087362*, January 2026. doi:10.2139/ssrn.7076299.
- 49 Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot’s code suggestions. In *Conf. on Mining Soft. Repositories*, pages 1–5, 2022. doi:10.1145/3524842.3528470.
- 50 Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, et al. Empirical standards for software engineering research. *arXiv preprint*, 2020. arXiv:2010.03525.
- 51 Paul Ralph, Miikka Kuutila, Hera Arif, and Bimpe Ayoola. *Teaching Software Metrology: The Science of Measurement for Software Engineering*, pages 101–154. Springer, Cham, 2024. doi:10.1007/978-3-031-71769-7_5.
- 52 Paul Ralph and Ewan Tempero. Construct validity in software engineering research and software metrics. In *Proc. of EASE 2018*, EASE’18, pages 13–23. ACM, New York, NY, USA, 2018. doi:10.1145/3210459.3210461.
- 53 William Revelle. *psych: Procedures for Psychological, Psychometric, and Personality Research*. Northwestern University, Evanston, Illinois, 2026. R package version 2.6.4. URL: <https://CRAN.R-project.org/package=psych>.
- 54 Yves Rosseel. lavaan: An R package for structural equation modeling. *Journal of Statistical Software*, 48(2):1–36, 2012. doi:10.18637/jss.v048.i02.
- 55 Yves Rosseel, Terrence D. Jorgensen, and Luc De Wilde. *lavaan: Latent Variable Analysis*, 2025. R package version 0.6-20. doi:10.32614/CRAN.package.lavaan.
- 56 J. Saraiva, E. Barreiros, A. Almeida, F. Lima, A. Alencar, G. Lima, S. Soares, and F. Castor. Aspect-oriented software maintenance metrics: A systematic mapping study. In *Conf. on Evaluation & Assessment in Soft. Eng. (EASE 2012)*, pages 253–262. IET, 2012.
- 57 Tushar Sharma and Diomidis Spinellis. A survey on software smells. *J. of Sys. & Soft.*, 138:158–173, 2018. doi:10.1016/j.jss.2017.12.034.
- 58 Tushar Sharma and Diomidis Spinellis. Do we need improved code quality metrics?, 2020. arXiv:2012.12324.
- 59 Dag I. K. Sjøberg, Bente Anda, and Audris Mockus. Questioning software maintenance metrics: A comparative case study. In *Proc. of the Int. Symposium on Empirical Soft. Eng. & Measurement*, pages 107–110, 2012. doi:10.1145/2372251.2372269.
- 60 Dag I. K. Sjøberg and Gunnar Rye Bergersen. Construct validity in software engineering. *IEEE Trans. on Soft. Eng.*, 49(3):1374–1396, 2023. doi:10.1109/TSE.2022.3176725.
- 61 Srdjan Stevanetic and Uwe Zdun. Software metrics for measuring the understandability of architectural structures: a systematic mapping study. In *Proc. of EASE*, EASE ’15. ACM, New York, NY, USA, 2015. doi:10.1145/2745802.2745822.
- 62 Ewan Tempero and Paul Ralph. A framework for defining coupling metrics. *Science of Computer Programming*, 166:214–230, 2018. doi:10.1016/j.scico.2018.02.004.
- 63 Ewan Tempero and Paul Ralph. Making software metrics useful. *arXiv*, 2026. doi:10.48550/arXiv.2603.16012.
- 64 William Trochim, James P. Donnelly, and Arora Kanika. *The Research Methods Knowledge Base*. Wadsworth Publishing, 2015.
- 65 E.J. Weyuker. Evaluating software complexity measures. *IEEE Trans. on Soft. Eng.*, 14(9):1357–1365, 1988. doi:10.1109/32.6178.