

# A metrics-based comparative study on object-oriented programming languages

Di Wu

Lin Chen

Yuming Zhou

Baowen Xu \*

State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing, China

nju.wudi@gmail.com

lchen@nju.edu.cn

zhouyuming@nju.edu.cn

bwxu@nju.edu.cn

**Abstract**—There has been a long debate on which programming language can help write better object-oriented programs. However, to date little response is given to this issue with empirical evidence. In this paper, we perform a comparative study on C++, C#, and Java programs by using object-oriented metrics, which comprise measures for class size, complexity, coupling, cohesion, inheritance, encapsulation, polymorphism, and reusability. Our experiment is conducted on 78 tasks in Rosetta Code, a code repository providing solutions to the same programming tasks in different languages. The experimental results show that: (1) C++ classes are significantly larger than C# and Java classes in size, but their complexity does not differ significantly; (2) C# classes are significantly more likely to be coupled than C++ and Java classes through inter-class method invocations instead of direct data access; (3) C# and Java classes tend to be more cohesive than C++ classes; (4) C# and Java significantly outperform C++ in building deep inheritance trees; and (5) programs written in C++, C#, and Java do not show a significant difference in class encapsulation, polymorphism, and reusability. These findings could help practitioners choose suitable languages to develop object-oriented systems.

**Keywords**- *Programming Language, Comparative Study, Object-oriented Metrics*

## I. INTRODUCTION

Which programming language can help write better object-oriented programs? This question is often asked but it is hard to reach a consensus on the answer. From the practical perspective, it would be reliable to answer this question by empirically comparing real object-oriented programs written in different languages. The findings based on empirical evidence should be valuable in helping practitioners choose suitable languages to develop object-oriented systems.

To evaluate the quality of object-oriented programs, many metrics have been proposed, which are related to various language features like class size, coupling, cohesion, inheritance, encapsulation, and polymorphism [4-9]. In previous studies, these metrics are generally applied to fault prediction [10], class testability prediction [11], code refactoring [12], and code size estimation [13]. However, few researchers use the object-oriented metrics as indicators to compare programs written in different languages.

In this paper, we perform a preliminary comparative study on programming languages by employing 23 commonly-used

object-oriented metrics. More specifically, we use the standard statistical inference techniques to perform a differential analysis on the metric values for real programs written in C++, C#, and Java. The subject programs used in this study are provided by Rosetta Code [21], a code repository of solutions to common programming tasks implemented with various languages. By investigating 78 tasks in Rosetta Code, we attempt to answer the following issues: (1) Which language can help write classes of small size and low complexity? (2) Which language can help write classes of low coupling and high cohesion? (3) Which language can help create good type hierarchies? and (4) Which language can help write classes of good encapsulation, polymorphism, and reusability? These issues are of highly practical value, as they determine which programming language can help write better object-oriented programs. However, little is currently known on this subject with empirical evidence. Our study attempts to fill this gap by this comparative study.

Our experimental results based on object-oriented metrics show the following findings:

- C++ classes are significantly larger than C# and Java classes in size, but their complexity does not differ significantly;
- C# classes are significantly more likely to be coupled than C++ and Java classes through inter-class method invocations instead of direct data access;
- C# and Java classes tend to be more cohesive than C++ classes;
- C# and Java significantly outperform C++ in building deep inheritance trees;
- Programs written in C++, C#, and Java do not show a significant difference in class encapsulation, polymorphism, and reusability.

The rest of the paper is organized as follows. Section II introduces the object-oriented metrics in a nutshell. Section III describes the studied subjects, data collection procedure, and data analysis method. Section IV reports the experimental results. Section V presents the threats to validity. Section VI discusses related work. Section VII concludes the paper and outlines the direction for future work.

## II. OBJECT-ORIENTED METRICS

In the past decades, many object-oriented metrics have been proposed. The most well-known metrics are CK metrics [4] and MOOD metrics [6], which are applied to assess the quality

---

\* Corresponding author: Baowen Xu; Email: bwxu@nju.edu.cn

TABLE I. OBJECT-ORIENTED METRICS

Category	Metric name	Metric definition	Level	Expected value	Source
Size and Complexity	NOM (Number of methods)	The number of methods defined in a class	Class	Low	[7]
	NOA (Number of attributes)	The number of attributes defined in a class	Class	Low	[7]
	WMC (Weighted method complexity)	The sum of complexity for all methods in a class	Class	Low	[4]
	CC (Class complexity)	The sum of complexity for all methods in a class based on the information flow	Class	Low	[9]
Coupling	RFC (Response for a class)	The number of methods that can be potentially executed in response to a message received by an object of a class	Class	Low	[4]
	CBO (Coupling between objects)	The number of other classes to which a class object is coupled	Class	Low	[4]
	DAC (Data abstract coupling)	The number of ADT (Abstract Data Type) instances defined in a class	Class	Low	[7]
	MPC (Message passing coupling)	The number of send statements defined in a class	Class	Low	[7]
	CF (Coupling factor)	$CF = \frac{\sum_{i=1}^{TC} \sum_{j=1}^{TC} is\_client(C_i, C_j)}{TC^2 - TC}$ , where TC is total number of classes and $is\_client(C_i, C_j) = \begin{cases} 1, & \text{if } C_i \Rightarrow C_j \text{ and } C_i \neq C_j \\ 0, & \text{otherwise} \end{cases}$	System	Low	[6]
Cohesion	LCOM (Lack of cohesion in methods)	LCOM = (Number of pair of methods that have no common attributes) - (Number of pair of methods that have common attributes)	Class	Low	[4]
	TCC (Tight class cohesion)	TCC = (Number of pairs of directly connected public methods using common attributes) / (Number of pairs of public methods)	Class	High	[5]
	LCC (Loose class cohesion)	LCC = (Number of pairs of directly and indirectly connected public methods using common attributes) / (Number of pairs of public methods)	Class	High	[5]
	ICH (Information based cohesion)	The number of invocations to other member functions/methods	Class	High	[8]
Inheritance	NOC (Number of children)	The number of immediate subclasses of a class in a type hierarchy	Class	High	[4]
	DIT (Depth of inheritance tree)	The maximum length from the node to the root of the tree	Class	High	[4]
	MIF (Method inheritance factor)	MIF = (Number of methods inherited in all classes) / (Number of methods defined and inherited in all classes)	System	High	[6]
	AIF (Attribute inheritance factor)	AIF = (Number of attributes inherited in all classes) / (Number of attributes defined and inherited in all classes)	System	High	[6]
Encapsulation	MHF (Method hiding factor)	Let V (M) = number of classes where the method M is visible, then $MHF = 1 - \frac{\sum V(M)}{\text{total number of classes} - 1}$ $\text{Number of methods in all classes}$	System	High	[6]
	AHF (Attribute hiding factor)	Let V (A) = number of classes where the attribute A is visible, then $AHF = 1 - \frac{\sum V(A)}{\text{total number of classes} - 1}$ $\text{Number of attributes in all classes}$	System	High	[6]
Polymorphism	NMO (Number of methods overridden by a subclass)	The number of methods in a subclass overridden from its base class	Class	High	[7]
	PF (Polymorphism factor)	$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}$ , where TC is the total number of classes and $M_n(C_i)$ = Number of new methods of the class $C_i$ , $M_o(C_i)$ = Number of overriding methods of the class $C_i$ , $DC(C_i)$ = Number of descendants of the class $C_i$	System	High	[6]
Reusability	RR (Reuse ratio)	RR = (Total number of super classes) / (Total number of classes)	System	High	[7]
	SR (Specialization ratio)	SR = (Total number of sub-classes) / (Total number of super classes)	System	High	[7]

of object-oriented programs at different levels. To be specific, CK metrics are mainly used to evaluate single classes, while MOOD metrics are applied to assess the whole object-oriented systems. Table I gives a detailed description of the 23 commonly-used object-oriented metrics. According to this table, all the metrics can be divided into 7 categories [3], which cover the following object-oriented features:

- Size and complexity. NOM and NOA are used to measure the size of a class in terms of the number of methods and the number of attributes, respectively. WMC and CC are applied to measure the complexity of a class through

calculating the total complexity of its member functions/methods in different ways. Since classes are suggested to be designed as concise as possible, these metrics are expected to be low in their values.

- Coupling. Five metrics are used to evaluate class coupling from different perspectives. To be specific, the CF metric is used to evaluate the coupling of all classes at the system level. By comparison, the other four metrics measure coupling at class level. Among these metrics, RFC and MPC are used to assess method coupling, DAC embodies data coupling between classes, and CBO shows coupling

between class instances. Since highly coupled classes are less object-oriented, low metric values are preferable.

- Cohesion. Cohesion is measured with four class-level metrics, which are calculated in different ways to reflect the interactions between member functions/methods. Among these metrics, a low LCOM value is expected, while high TCC, LCC, and ICH values are desired.
- Inheritance. NOC and DIT are class-level metrics, which express class inheritance through the number of descendants and the depth of type inheritance, respectively. By comparison, MIF and AIF are system-level metrics, which refer to method inheritance and attribute inheritance, respectively. Since it is suggested to build hierarchical type trees in the object-oriented systems, the high inheritance metric values are expected.
- Encapsulation. MHF and AHF are indicators to show how well methods and attributes are hidden inside classes. These metrics are measured at system level and high metric values are preferable.
- Polymorphism. NMO and PF are polymorphism metrics at different levels. To be specific, NMO is a class-level metric, which refers to the number of methods overridden by a single subclass, while PF is a system-level metric, which measures the degree of method overriding in the whole type tree. Their metric values are desired to be high.
- Reusability. RR and SR are both system-level reusability metrics. They are calculated as the ratios of subclasses to all classes and to super classes, respectively. Since classes are expected to be highly reused, large reusability metric values are desirable.

### III. RESEARCH METHOD

In this section, we first introduce the subject programs used in our study. Then, we describe the data collection procedure. Finally, we show the data analysis method.

#### A. Studied Subjects

In order to conduct the comparative experiment, we need to investigate the programs that give solutions to the common goals and are written in C++, C#, and Java, respectively. For this reason, we employ the open-source programs in Rosetta Code [21], a code repository providing solutions to the same tasks in various languages. Currently, it contains 766 programming tasks implemented in 567 different languages. These tasks belong to 59 categories, including mathematics, games, and networking, etc. Due to its abundant resources, Rosetta Code has been effectively used to compare languages' concise, performance, and failure-proneness [2].

In the Rosetta Code repository, we totally find 381 tasks that have solutions in all the three investigated languages. By manually checking these solutions, we choose 78 tasks as our studied subjects, because they are all implemented in the object-oriented manner. In other words, the remaining 203 tasks are deleted from our concern either because they are implemented in the procedural manner in their C++ solutions or because that they are lack of entire implementation code. The detailed information of these tasks can be found at <http://ise.nju.edu.cn/wudi/Lang.Comp.Study>.

#### B. Data Collection

We collected the metric values by using ‘‘Understand’’ [20], a program analysis and measurement tool. Specifically, the data was collected by the following steps. At the first step, we built an Understand database for each solution implemented in C++, C#, and Java. At the second step, we collected the metric values for each solution by processing its database. Some simple metrics such as NOM, NOA, and WMC were directly reported by Understand, while other metrics including CC, MHF, and AHF were collected by running our own Perl scripts, which utilize the analysis-based information of programs through calling Understand APIs. At the third step, we calculated for each class-level metric its average metric value of all classes in each solution. At the fourth step, we selected for each task its optimal solution written in the same language. Of the 78 studied subjects, 20 tasks have more than one solution written in the same language. In order to pick out the best solutions for the 20 tasks, we compare for each task its solutions written in the same language according to the metric values and select the optimal one. At the last step, we gathered the metric values of all selected solutions to the 78 tasks and stored the data in a csv file, which was used for data analysis.

#### C. Data Analysis

We employ the standard statistical inference techniques to analyze the experimental data. More specifically, for each object-oriented feature, we perform a Wilcoxon’s signed rank analysis to compare the metric values of solutions implemented in different languages. In other words, C++, C#, and Java are compared in pair-wise to find out which language can help write best object-oriented programs. If the metric values of two languages show a difference at a significance level of 0.05 (p-value), we will conclude that the languages are significantly different. Also, we employ the Cliff’s  $\delta$  to examine whether the magnitude of difference is important [18]. By convention, the magnitude of the difference is considered either trivial ( $|\delta| < 0.147$ ), small (0.147-0.33), medium (0.33-0.474), or large ( $> 0.474$ ) [19]. Finally, we apply the signed ratio  $R$  to give an unstandardized measure of the difference between two medians [2]. The  $R$  value is calculated as:

$$R = \text{sgn}(M_x - M_y) \frac{\max(M_x, M_y)}{\min(M_x, M_y)} \quad (1)$$

where  $M_x$  and  $M_y$  denotes the median metric values of language X and language Y, respectively. A positive sign  $\text{sgn}(M_x - M_y)$  indicates that the median metric value of X is larger than the median metric value of Y, while a negative sign signifies a reverse result. Moreover, the absolute  $R$  value denotes how many times X’s median is larger/smaller than Y’s median under a specific metric.

### VI. EXPERIMENTAL RESULTS

In this section, we report in detail the experimental results. Table II shows the overall experimental results for language comparison. In this table, we present for each metric the significance of the Wilcoxon’s signed rank analysis (p-value), the magnitude of difference (Cliff’s  $\delta$ ), and the times between two median values ( $R$ ).

TABLE II. COMPARATIVE RESULTS FOR OBJECT-ORIENTED METRICS ON C++, C#, AND JAVA PROGRAMS

Metrics		C++ vs. C#			C++ vs. Java			C# vs. Java		
		$p$	$\delta$	$R$	$p$	$\delta$	$R$	$p$	$\delta$	$R$
Size and Complexity	NOM	< 0.001	0.478	2.222	< 0.001	0.484	1.667	1.000	-0.007	-1.333
	NOA	0.007	0.385	-	0.554	0.162	2.000	0.064	-0.151	-
	WMC	0.150	0.095	1.200	0.930	-0.024	1.000	0.006	-0.125	-1.200
	CC	0.111	-0.143	-2.773	0.460	-0.092	-1.340	1.000	0.049	2.069
Coupling	RFC	< 0.001	-0.635	-1.403	< 0.001	0.502	2.000	< 0.001	0.868	2.806
	CBO	< 0.001	-0.677	-3.000	< 0.001	0.463	-	< 0.001	0.916	-
	DAC	0.004	0.236	-	0.301	-0.027	-	< 0.001	-0.236	-
	MPC	0.954	-0.029	-	1.000	0.030	-	0.966	0.061	-
	CF	0.072	-0.827	-2.222	0.056	-0.753	-2.778	1.000	0.012	-1.250
Cohesion	LCOM	< 0.001	0.458	-	< 0.001	0.340	-	0.416	-0.070	-
	TCC	0.351	0.111	1.473	0.884	0.007	-1.018	0.966	-0.095	-1.500
	LCC	0.254	0.155	1.500	0.777	0.055	1.000	0.966	-0.098	-1.500
	ICH	0.014	-0.272	-	0.010	-0.268	-	1.000	-0.001	1.136
Inheritance	NOC	0.078	0.133	1.000	0.004	0.291	1.000	0.065	0.155	1.000
	DIT	< 0.001	-0.934	-	< 0.001	-0.939	-	0.210	-0.068	1.000
	MIF	0.150	0.094	-	0.760	0.052	-	0.378	-0.041	-
	AIF	1.000	0.033	-	1.000	0.034	-	1.000	0.001	-
Encapsulation	MHF	0.218	-0.219	-2.826	0.230	0.375	-	0.118	0.625	-
	AHF	0.608	0.184	2.000	0.385	0.306	-	0.497	0.388	-
Polymorphism	NMO	0.317	-0.028	-	0.569	0.026	-	0.178	0.053	-
	PF	1.000	0.333	-	1.000	0.000	1.000	1.000	-0.111	-
Reusability	RR	0.159	0.088	-	0.056	0.077	-	1.000	-0.012	-
	SR	0.432	-0.625	-1.833	1.000	0.250	1.500	0.378	0.500	2.750

\* Note: (1) All p-values have been adjusted using the Benjamini-Hochberg method; (2) Cells marked with “-” denote the denominator of formula (1) is zero; (3) Cells in gray background denote the significant results (p-values < 0.05).

### A. Size and Complexity

We employ the result from NOM and NOA metrics to compare the size of classes written in C++, C#, and Java. According to Table II, we find that C++’s NOM value is significantly different from C#’s NOM value ( $p < 0.001$ ), and the magnitude of difference is large in terms of Cliff’s  $\delta$  (0.478). Moreover, the median NOM value of C++ is over 2 times larger than the median NOM value of C# ( $R = 2.222$ ). Besides, the comparison between C++’s NOM value and Java’s NOM value shows a similar result. This indicates that C++ classes significantly have more member functions/methods than both C# and Java classes. As for NOA, we find C++ classes significantly have more attributes than C# classes ( $p = 0.007$ ) and the magnitude of difference is medium ( $\delta = 0.385$ ). However, the comparison between C++ and Java does not show a significant difference.

In terms of class complexity (WMC and CC metrics), we do not observe a significant difference between C++ and C#/Java. This indicates that C++ classes are not significantly more complex than C# and Java classes. However, C#’s WMC value is significantly different from Java’s WMC value ( $p < 0.001$ ), but the effect size is trivial according to Cliff’s  $\delta$  (-0.125). This signifies that C# methods tend to be less complex than Java methods, but the difference is not obvious.

To summarize, the core observation from the size and complexity metrics is that **C++ classes are significantly larger than C# and Java classes in size, but their complexity does not differ significantly.**

**Interpretation.** One possible explanation for this result is that C++ has two different paradigms, namely the procedural programming and the object-oriented programming. When

programmers implement C++ classes, they are likely to think in the procedural manner, thus resulting in a large number of member functions to be created inside a class.

### B. Coupling

We employ the result from RFC, CBO, DAC, MPC, and CF metrics to compare the coupling of classes written in C++, C#, and Java. As for CF, the system-level metric, we do not find any significant difference among the three languages (all p-values > 0.05). This indicates that C++, C#, and Java do not differ in class coupling from the system-level perspective. As for the class-level metrics, however, these three languages show significant differences. To be specific, for RFC, which evaluates class coupling based on method invocations, C# has a significantly larger RFC value than C++ and Java (both p-values < 0.001). Moreover, the effect sizes are large in terms of Cliff’s  $\delta$  ( $0.635 \leq |\delta| \leq 0.868$ ). Besides, the  $R$  values also show a difference between the medians ( $1.403 \leq |R| \leq 2$ ). This evinces that C# classes are more likely to interact with each other through inter-class method invocations. Also, CBO, a metric reflecting coupling between class objects, shows a similar result. However, DAC, a metric evaluating class coupling through data access, indicates a contrary result. More specifically, it shows that C#’s DAC value is significantly smaller than C++ and Java’s DAC values (both p-values < 0.005). Moreover, the effect sizes are small in terms of Cliff’s  $\delta$  (both  $|\delta| = 0.236$ ). This result signifies that C# classes are less likely to interact with each other through data interaction. As for MPC, another coupling metric based on member functions/methods, does not show a significant result. To summarize, the core observation from the coupling metrics is that **C# classes are significantly more likely to be coupled**

than C++ and Java classes through inter-class method invocations instead of direct data access.

**Interpretation.** According to Table II, we find RFC and MPC, the metrics for inter-class method coupling, show completely different results. This is due to the different ways in calculating their metric values. To be specific, the get/set accessors in C# classes are regarded as ordinary methods when we compute the RFC values. But they are removed during calculating the MPC values. For this reason, we conjecture that the tight coupling among C# classes is generally caused by frequent inter-class get/set method invocations.

### C. Cohesion

We employ the result from LCOM, LCC, TCC, and ICH metrics to compare the cohesion of classes written in C++, C#, and Java. As for LCOM, we find that C++ has a significantly larger LCOM value than C# and Java (both p-values < 0.001). Moreover, the magnitudes of difference are medium in terms of Cliff's  $\delta$  ( $0.340 \leq |\delta| \leq 0.458$ ). Since a low LCOM value is preferable, this result indicates that C++ classes are less cohesive than C# and Java classes. ICH, another cohesion metric, shows a consistent result. To be specific, C++'s ICH value is significantly smaller than both C#'s ICH value ( $p = 0.014$ ) and Java's ICH value ( $p = 0.010$ ). Furthermore, the effect sizes are small in terms of Cliff's  $\delta$  ( $0.268 \leq |\delta| \leq 0.272$ ). However, the other two cohesion metrics, namely LCC and TCC do not show significant results. To summarize, the core observation from the cohesion metrics is that among the four metrics, two of them significantly evince that C# and Java outperform C++ in creating higher cohesive classes. From this reasoning, we conclude that **C# and Java classes tend to be more cohesive than C++ classes.**

**Interpretation.** One possible explanation for this result is that many member functions in C++ classes are still implemented in the procedural manner. As a result, the member functions do not interact well through sharing the common attributes and thus result in a low cohesion for the whole class.

### D. Inheritance

We employ the result from NOC, DIT, MIF, and AIF metrics to compare the inheritance of classes written in C++, C#, and Java. As for NOC, we find C++'s NOC value is significantly larger than Java's NOC value ( $p = 0.004$ ). Moreover, the effect size is small in terms of Cliff's  $\delta$  (0.291). The comparison between C+ and C#, however, does not show a significant result ( $p = 0.078$ ). As for DIT, we observe that C++'s DIT value is significantly smaller than both C# and Java's DIT values (both p-values < 0.001). Furthermore, the effect sizes are relatively large in terms of Cliff's  $\delta$  ( $0.934 \leq |\delta| \leq 0.939$ ). As for other two metrics, namely MIF and AIF, no significant result is revealed. To summarize, the core observation from the inheritance metrics is that **C# and Java significantly outperform C++ in building deep inheritance trees (DIT).**

**Interpretation.** Both C# and Java have the root type "Object", which is a super type for all classes. Therefore, the depth of inheritance trees in C# and Java systems is not surprisingly larger than the C++ systems.

### E. Encapsulation, Polymorphism, and Reusability

We employ the result from MHF and AHF metrics for encapsulation, NMO and PF metrics for polymorphism, and RR and SR for reusability to compare the classes written in C++, C#, and Java. According Table II, there is no significant result revealed by all these metrics. For this reason, we conclude that **programs written in C++, C#, and Java do not show a significant difference in class encapsulation, polymorphism, and reusability.**

**Interpretation.** Regarding the language features for class encapsulation, C++, C#, and Java all provide private, protected, and public keywords to control the accessibility to the methods and attributes inside a class. As a result, there is no difference in information hiding of classes written in different languages. Moreover, all these three languages support static binding and dynamic dispatching in similar ways. Therefore, the polymorphism of classes does not differ. Finally, class reusability is generally independent from language support. Instead, it is determined by the ways programmers define new classes. For this reason, there is not a significant difference in the reusability of classes written in C++, C#, and Java.

## VII. THREATS TO VALIDITY

The threat to the construct validity is the correctness of the metric values collected from "Understand" databases. Since many historical studies have produced reliable empirical results by using "Understand" [20], the data in our study can also be considered as acceptable. The threat to internal validity is the object-oriented metrics used in this study. We totally use 23 metrics to investigate class size, complexity, coupling, cohesion, inheritance, encapsulation, polymorphism, and reusability. Even though these metrics do not include all object-oriented metrics proposed in historical studies, they are generally regarded as the most representative ones. Since these metrics are also used in other empirical studies [1, 3], they are also applicable in this study. The threat to the external validity is that we only use the programs provided by Rosetta Code to conduct the experiment. Since many solutions to our studied tasks have a small number of classes, our empirical results need to be further examined on more complex object-oriented systems in the future.

## VIII. RELATED WORK

The most related study to our work was undertaken by Kumari and Bhasin [1], who used the object-oriented metrics to compare C++ and Java programs. They investigated 15 object-oriented applications and found that Java is more object-oriented than C++ as per intuition. The metrics applied in our study and in [1] are basically the same. However, we use more strict research method and thus obtain more reliable findings. To be specific, our study has the following advantages. First, we studied three languages, namely C++, C#,

and Java, while in [1], only C++ and Java were analyzed. Second, we used 78 open-source programming tasks to do the experiment, while Kumari and Bhasin only employed 15 tasks. Moreover, the detailed information of their data set was not given, making their experiment not replicable. Third, we used the standard statistical inference techniques (such as the Wilcoxon's signed rank analysis) to analyze the experimental data, while Kumari and Bhasin only got the comparison result based on bar graphs for the raw metric values. For this reason, our empirical results are more reliable. In terms of the conclusions, the main difference between the two papers lies in the languages' support to build deep inheritance trees. In [1], the authors showed that the DIT metric value of C++ programs was larger than Java programs. However, we get an opposite result. Due that the result in this study is drawn using the standard statistical analysis, our conclusion is more acceptable.

Another related study was conducted by Nanz and Furia [2], who used the Rosetta Code repository to compare the conciseness, performance, and failure-proneness of programs written in different languages. By using the standard statistical inference methods, the authors had the following findings: (1) functional and scripting languages are more concise than procedural and object-oriented languages; (2) C is hard to beat when it comes to raw speed on large inputs; and (3) compiled strongly-typed languages are less prone to runtime failures than interpreted or weakly-typed languages. Compared with [2], our study focuses on a different research question, namely languages' support to write good object-oriented programs. For this reason, the conclusions of the two studies are not comparable. However, the two studies still share some similarities. First, they both use Rosetta Code as the experimental subject. Second, they both use the standard statistical inference techniques to analyze the experimental data. Third, the conclusions of both studies are drawn on the result of statistical analysis. Other related empirical programming language research include the comparison on languages' difference in running time, memory consumption, and productivity [14], the survey of developers' behaviors in using object-oriented concepts [15], the study on languages' support for code quality [16], and the investigation on languages' adoption [17], etc.

## IX. CONCLUSIONS AND FUTURE WORK

In this paper, we perform a comparative study on C++, C#, and Java programs to investigate which language can help write better object-oriented programs. By analyzing 23 object-oriented metrics on the solutions to 78 real programming tasks, we find that C# and Java outperform C++ in creating concise and cohesive classes. Also, the empirical result shows that C# and Java can help build deeper inheritance trees than C++. Moreover, we find that C# classes are significantly more likely to be coupled than C++ and Java classes through inter-class method invocations instead of direct data access. Finally, the statistical result reveals that the programs written in C++, C#, and Java do not show a significant difference in class encapsulation, polymorphism and reusability. Our empirical evidence should be valuable in helping practitioners choose

suitable languages to develop object-oriented systems. In the future work, we will investigate more object-oriented languages and replicate the study on more applications.

## ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (61170071, 61432001, 91418202, 61472175, 61472178), the National Natural Science Foundation of Jiangsu Province (BK20130014), and the program B for Outstanding PhD candidate of Nanjing University.

## REFERENCES

- [1] U. Kumari, S. Bhasin. Application of object-oriented metrics to C++ and Java: A comparative study. *ACM SIGSOFT Software Engineering Notes*, 36(2), 2011: 1-10.
- [2] S. Nanz, C. A. Furia. A comparative study of programming languages in Rosetta Code. *ICSE*, 2015.
- [3] K. K. Aggarwal, Y. Singh, A. Kaur, R. Malhotra. Empirical study of object-oriented metrics. *Journal of Object Technology*, 5(8), 2006: 149-173.
- [4] S. R. Chidamber, C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Trans. Software Eng.*, 20(6), 1994: 476-493.
- [5] L. C. Briand, J. W. Daly, J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1): 1998: 65-117.
- [6] R. Harrison, S. J. Counsell, R. V. Nithi. An evaluation of MOOD set of object oriented software metrics. *IEEE Trans. Software Eng.*, 24(6), 1998: 491-496.
- [7] B. Henderson-Sellers. Object-oriented metrics: measures of complexity. *Prentice Hall*, 1995.
- [8] Y. S. Lee, B. S. Liang, S. F. Wu, F. J. Wang. Measuring the coupling and cohesion of an object-oriented program based on information flow. *QSIC*, 1995.
- [9] Y. S. Lee, B. S. Liang, F. J. Wang. Some complexity metrics for OO programs based on information flow: A study of C++ programs. *Journal of Information Science and Engineering*, 10(1), 1994: 21-50.
- [10] T. Gyimothy, R. Ferenc, I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng.*, 31(10), 2005: 897-910.
- [11] M. Bruntink, A. van Deursen. Predicting class testability using object-oriented metrics. *SCAM*, 2004: 136-145.
- [12] K. O. Elish, M. Alshayeb. Using software quality attributes to classify refactoring to patterns. *Journal of Software*, 7(2), 2012: 408-419.
- [13] Y. Zhou, Y. Yang, B. Xu, H. Leung, X. Zhou. Source code size estimation approaches for object-oriented systems from UML class diagrams: A comparative study. *Information & Software Technology*, 56(2), 2014: 220-237.
- [14] L. Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10), 2000: 23-29.
- [15] T. Gorschek, E. Tempero, L. Angelis. A large-scale empirical study of practitioners' use of object-oriented concepts. *ICSE*, 2010: 115-124.
- [16] B. Ray, D. Posnett, V. Filkov, P. T. Devanbu. A large scale study of programming languages and code quality in Github. *FSE*, 2014: 155-165.
- [17] L. A. Meyerovich, A. Rabkin. Empirical analysis of programming language adoption. *OOPSLA*, 2013: 1-18.
- [18] E. Arisholm, L. Briand, B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1), 2010: 2-17.
- [19] J. Romano, J. Kromrey, J. Coraggio, J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's *d* for evaluating group differences on the NSSE and other surveys? *Annual Meeting of the Florida Association of Institutional Research*, 2006: 1-3.
- [20] SciTools Understand. <https://scitools.com/>.
- [21] The Rosetta Code Repository. [http://rosettacode.org/wiki/Rosetta\\_Code](http://rosettacode.org/wiki/Rosetta_Code).