Elegant Objects: Developing a Programming Language for Large-Scale Applications

Hadi Saleh
Professor, Ph.D.
National Research University Higher School of Economics, 101000 Moscow, Russia
Vladimir State University, 600000, Vladimir, Russia
hsalekh@hse.ru

Ivan Spirin
MSc.
Information System and Software Engineering
Moscow, Russia
wensen@list.ru

Sergey Zykov
Professor, Ph.D.
National Research University Higher School of Economics, 101000 Moscow, Russia
National Research Nuclear University Moscow Engineering Physics Institute, 115409, Moscow, Russia
szykov@hse.ru

Alexander Legalov
Professor, Ph.D.
National Research University Higher School of Economics
101000 Moscow, Russia
alegalov@hse.ru

Abstract: The EO programming language is a research and development project that is still in progress. It is a language based on the Elegant Object philosophy which advocates the vision of pure Object-Oriented Programming by having all components of the program as objects. In the effort to improve the language and prove that fully Object-Oriented Programming is possible in real programs aimed at solving practical problems, we propose rewriting jPeek application in EO programming language and in process, find and fix bugs in the compiler and the EO standard object library. The main objective is to improve the language reliability. In this paper, we explore the cohesion metrics calculations in jPeek, a cohesion analyzer of Java classes/projects, then rewrite them in EO programming language and compare the performances of the implemented solution to the original jPeek implementation.

Keywords: Cohesion, jPeek, EO, OOP, code quality, software reliability, metrics.

1 Introduction

The Software developers aim for systems with high cohesion and low coupling [1]. Cohesion metrics indicate how well the methods of a class are related to each other. Metrics estimate the quality of different aspects of software. In particular, cohesion metrics indicate how well the parts of a system operate together. Cohesion is one of software attributes representing the degree to which the components are functionally connected within a software module [2]–[4]. Class cohesion are important in object-oriented programming because it indicates the quality of class design. There are several metrics to measure class cohesion. typically, a cohesive class performs a single function. A non-cohesive class performs two or more unrelated functions. A non-cohesive class may need to be split into two or more smaller classes. The assumption behind these cohesion metrics is that methods are related if they work on the same class-level variables. Methods are unrelated if they work on different variables altogether. Thus, in a cohesive class, methods work with the same set of variables. In a non-cohesive class, there are some methods that work on different data [5].

A cohesive class performs exactly one function. Lack of cohesion means that a class performs more than one function. This is not desirable; if a class performs any unrelated functions, it should be split up. In summary:

1. High cohesion is desirable since it promotes encapsulation. As a drawback, a highly cohesive class has high coupling between the methods of the class, which in turn indicates high testing effort for that class.
2. Low cohesion indicates inappropriate design and high complexity. It has also been found to indicate a high likelihood of errors. The class should probably be split into two or more smaller classes.

jPeek application is a tool that is designed to analyze Java code (.class files) quality by measuring cohesion based on certain metrics. This paper focuses on re-implementing these metrics in the EO programming language, a newly developed Object-Oriented Programming language based on the concept of Elegant Objects. As EO determines to prove that fully Object-Oriented Programming is possible not only in books and abstract examples but also in real programs and codes aimed at solving practical problems [6], the following cohesion metrics were considered for reimplementation:

- LCOM1
- LCOM2
- LCOM3
- LCOM4
- LCOM5
- OCC
- PCC
- LCC
- TCC
- CCM
- CAMC
- NHD
- SCOM

2 The EO-jPeek Solution

First The proposed solution is to integrate the metric(s) implemented in EO into the jPeek tool (i.e., a module in EO built into jPeek Repo).

2.1 Outline

The proposed solution is the implementation of some of the metrics of jPeek entirely in EO. This module calculates certain jPeek metrics utilizing the EO programming language and the capabilities of its standard object library. An infrastructure of Java classes integrates EO metrics into the general architecture of jPeek tool. This paper focuses on the general architecture of the proposed solution, which includes the following elements:

- A subroutine that parses and reconstructs the data of a class being assessed in the form of EO objects.
- A class that integrates EO metrics into the general pipeline of the application, providing input and formatting output for each metric run.
- A technique that embeds the EO programming language transpiler into the building process of jPeek.

2.2 Integrating EO metrics into jPeek pipeline

To isolate EO metrics from XML, we apply the following techniques:

1. Firstly, the "*Skeleton.xml*" document containing information on the structure details of the class being assessed is parsed and reconstructed in the form of EO objects describing the structure of the class. This stage is done by the

EOSkeleton class [7]. To call and use EO objects, the *EOSkeleton* class utilizes a basic late binding technique made possible through specific embedding of the EO transpiler into the building process described in section 2.3.

2. Secondly, the *EOCalculus* [8] class extracts the “class” EO object from the formed *EOSkeleton* instance and loads it into the metric written in EO. To call EO objects of the metrics, the *EOCalculus* class utilizes a basic late binding technique through specific embeddings of the EO transpiler into the building process described in section 2.3. The output of the metric is written to the XML to embed the results into the pipeline of the jPeek tool.

As an example, consider the following Java class:

```java
package objects;

public class Car {
    int manufacturerId;

    public void m1(int x){
        this.manufacturerId = x;
    }
}
```

The above code is represented in xml as:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<skeleton date="2021-07-09T05:03:40.353105500Z"
          schema="xsd/skeleton.xsd"
          version="1.0-SNAPSHOT">
   <app id="D:\Projects\eo-jpeek\objects">
      <package id="objects">
         <class id="Car"><!--Package: objects; name: objects.Car; file: objects.Car-->
            <attributes>
               <attribute final="false" public="false" static="false" type="I">manufacturerId</attribute>
            </attributes>
            <methods>

               <method abstract="false"
                       bridge="false"
                       ctor="false"
                       desc="(I)V"
                       name="m1"
                       static="false"
                       visibility="public">
                  <args>
                     <arg type="I">?</arg>
                  </args>
                  <return>V</return>
                  <ops>
                     <op code="put">manufacturerId</op>
                  </ops>
               </method>
            </methods>
         </class>
      </package>
   </app>
</skeleton>
```

This xml, containing all the necessary information of the class is parsed and reconstructed in the form of EO objects and subsequently used in the calculation of the various metrics in EO programming language.

2.3 Embedding the EO transpiler into the building process

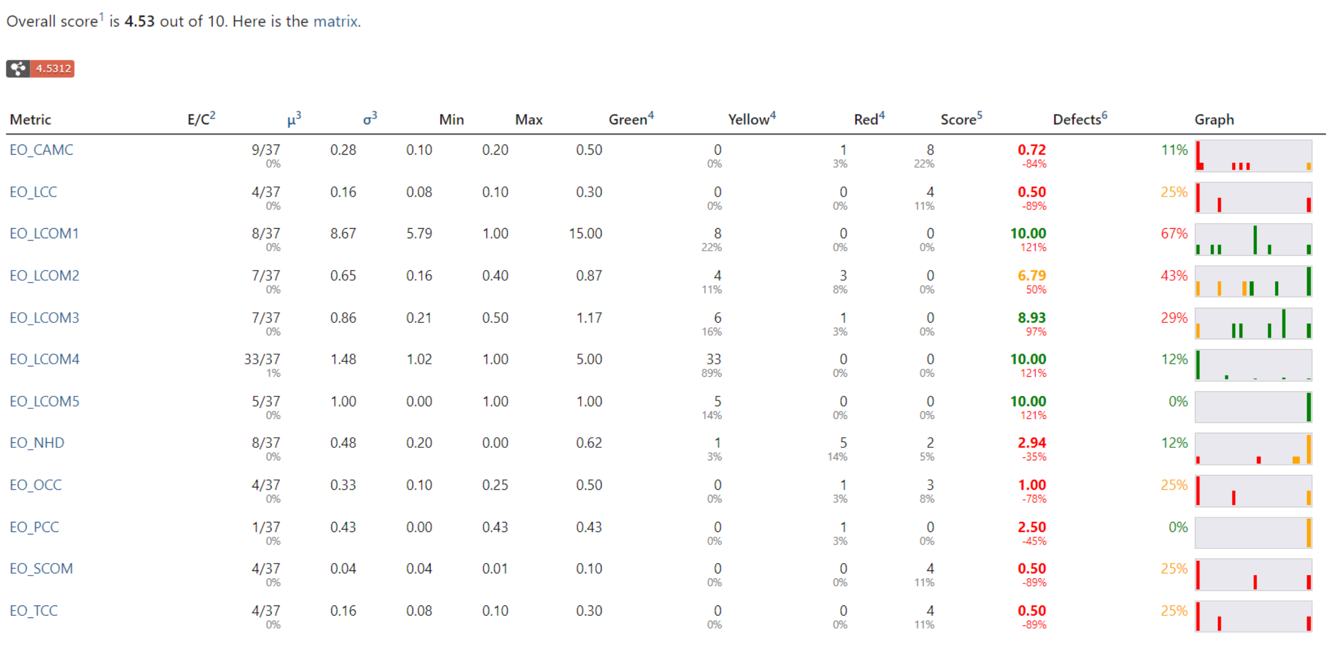
Since the EO to Java transpiler [9], enhanced by the Team, is distributed as a Maven artifact, it was embedded into the building process as follows:

- The artifact *org.eolang.eo-runtime* was added as a runtime dependency to the *pom.xml* of the jPeek project. It was needed to access the standard object library of EO at runtime.
- The artifact *org.eolang.eo-maven-plugin* was added as a plugin as the first step of the building process. The plugin transpiles the EO metrics before building Java classes of the rest of the jPeek project structure. This allows late binding of Java2EO references in Java code and, hence, provides basic variant of Java2EO interoperability.

Section 3 describes the metrics implemented in the proposed solution.

3 Metrics

Fig. 1 shows the results of the metrics tested using Node Packages library which contains some Java classes and is available on GitHub at [10]. Fig. 2 shows further details of EO_LCOM2. Similar further details exist for each of the metrics displayed in fig. 1. All these metrics implemented in EO have their test outputs presented here [11]. The implementation of these metrics can be found here [12] on GitHub.



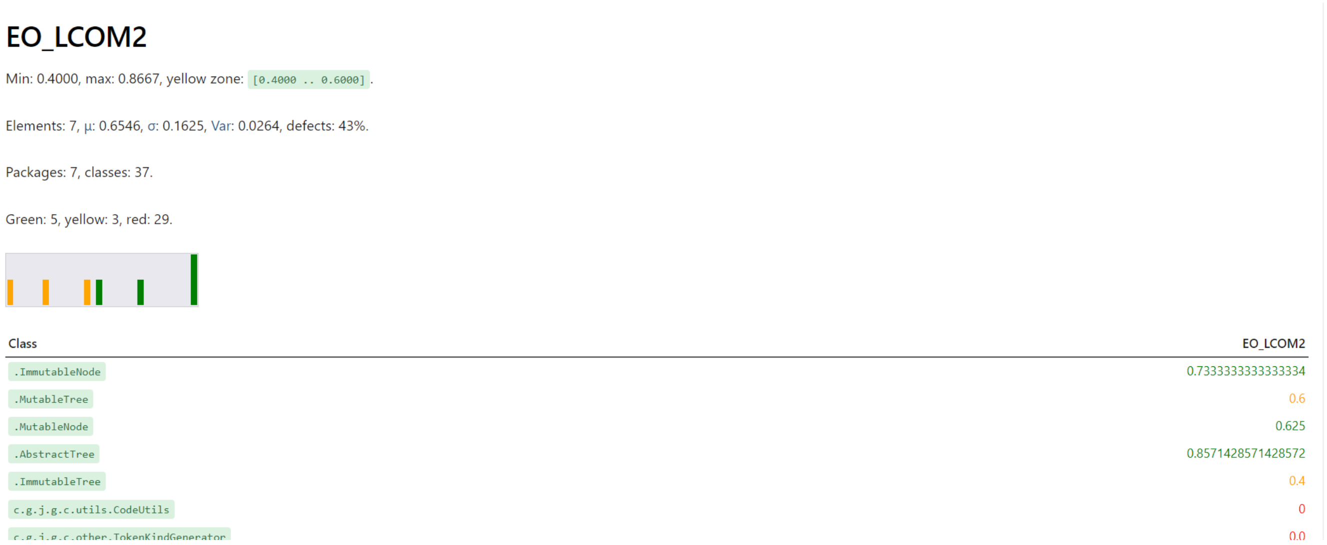
Overall score[1] is **4.53** out of 10. Here is the matrix.

4.5312

Metric	E/C[2]	µ[3]	σ[3]	Min	Max	Green[4]	Yellow[4]	Red[4]	Score[5]	Defects[6]	Graph
EO_CAMC	9/37 0%	0.28	0.10	0.20	0.50	0 0%	1 3%	8 22%	**0.72** -84%	11%	
EO_LCC	4/37 0%	0.16	0.08	0.10	0.30	0 0%	0 0%	4 11%	**0.50** -89%	25%	
EO_LCOM1	8/37 0%	8.67	5.79	1.00	15.00	8 22%	0 0%	0 0%	**10.00** 121%	67%	
EO_LCOM2	7/37 0%	0.65	0.16	0.40	0.87	4 11%	3 8%	0 0%	**6.79** 50%	43%	
EO_LCOM3	7/37 0%	0.86	0.21	0.50	1.17	6 16%	1 3%	0 0%	**8.93** 97%	29%	
EO_LCOM4	33/37 1%	1.48	1.02	1.00	5.00	33 89%	0 0%	0 0%	**10.00** 121%	12%	
EO_LCOM5	5/37 0%	1.00	0.00	1.00	1.00	5 14%	0 0%	0 0%	**10.00** 121%	0%	
EO_NHD	8/37 0%	0.48	0.20	0.00	0.62	1 3%	5 14%	2 5%	**2.94** -35%	12%	
EO_OCC	4/37 0%	0.33	0.10	0.25	0.50	0 0%	1 3%	3 8%	**1.00** -78%	25%	
EO_PCC	1/37 0%	0.43	0.00	0.43	0.43	0 0%	1 3%	0 0%	**2.50** -45%	0%	
EO_SCOM	4/37 0%	0.04	0.04	0.01	0.10	0 0%	0 0%	4 11%	**0.50** -89%	25%	
EO_TCC	4/37 0%	0.16	0.08	0.10	0.30	0 0%	0 0%	4 11%	**0.50** -89%	25%	

The average mistake of individual scores: 85%, average defects rate: 23%.

Figure 1 – Metrics results on Node Packages.



EO_LCOM2

Min: 0.4000, max: 0.8667, yellow zone: `[0.4000 .. 0.6000]`.

Elements: 7, µ: 0.6546, σ: 0.1625, Var: 0.0264, defects: 43%.

Packages: 7, classes: 37.

Green: 5, yellow: 3, red: 29.

Class	EO_LCOM2
`.ImmutableNode`	0.7333333333333334
`.MutableTree`	0.6
`.MutableNode`	0.625
`.AbstractTree`	0.8571428571428572
`.ImmutableTree`	0.4
`c.g.j.g.c.utils.CodeUtils`	0
`c.g.j.g.c.other.TokenKindGenerator`	0.0

Figure 2. LCOM2

3.1 LCOM

LCOM was introduced in the Chidamber and Kemerer metrics suite [5]. It is also called LCOM1 or LOCOM, and it's calculated as follows:
Take each pair of methods in the class. If they access disjoint sets of instance variables, increase P by one. If they share at least one variable access, increase Q by one.

$$LCOM1 = \begin{Bmatrix} P - Q, & if\ P > Q \\ 0, & otherwise \end{Bmatrix} \tag{1}$$

LCOM1 = 0 indicates a cohesive class.

LCOM1 > 0 indicates that the class should be split into two or more classes, since its variables belong to disjoint sets.

Classes with a high LCOM1 have been found to be fault prone. A high LCOM1 value indicates disparateness in the functionality provided by the class. This metric can be used to identify classes that intend to achieve many different objectives, and consequently are likely to behave in less predictable ways than the classes that have lower LCOM1 values. Such classes could be more error prone to test and split into two or more classes with a better behaviour. The LCOM1 metric can be used by senior designers and project managers as a relatively simple way to track whether the cohesion principle is adhered to in the design of an application and advise changes.

3.2 LCOM2 and LCOM3

To overcome the problems of LCOM1, LCOM2 and LCOM3 were proposed. A low value of LCOM2 or LCOM3 indicates high cohesion and a well-designed class. It is likely that the system has good class subdivision implying simplicity and high re-usability. A cohesive class will tend to provide a high degree of encapsulation. A higher value of LCOM2 or LCOM3 indicates decreased encapsulation and increased complexity, thereby increasing the likelihood of errors. The choice of LCOM 2 or LCOM3 is often a matter of taste as these metrics are similar. LCOM3 varies in the range [0,1] while LCOM2 is in the range [0,2]. *LCOM2>=1* indicates a very problematic class. LCOM3 has no single threshold value. It is a good idea to remove any dead, variables before interpreting the values of LCOM2 or LCOM3. Dead variables can lead to high values of LCOM2 and LCOM3, thus leading to wrong interpretations of what should be done.

In a typical class whose methods access the class's own variables, LCOM3 varies between 0 (high cohesion) and 1 (no cohesion). When LCOM3=0, each method accesses all variables. This indicates the highest possible cohesion. LCOM3=1 indicates extreme lack of cohesion. In this case, the class should be split. When there are variables that are not accessed by any of the class's methods, $1 < LCOM3 <= 2$. If a class has one method or no methods, LCOM3 is undefined. If there are no variables in a class, LCOM3 is undefined. An undefined LCOM3 is displayed as zero [5], [13].

3.2.1 Parameters of LCOM2 and LCOM3

m = number of procedures (methods) in a class
a = number of variables (attributes) in a class
mA = number of methods that access a variable (attribute)
sum(mA) = sum of mA over the attributes of a class

3.2.2 Implementation details

Parameter *m* is equal to *WMC (Weighted Method per Class)*. Parameter *a* contains all variables whether these are Shared or not. Each access to a variable is counted.

$$LCOM2 = 1 - sum(mA) / (m \times a), \qquad (2)$$

LCOM2 equals the percentage of methods that do not access a specified attribute, averaged over all attributes in the class. If the number of methods or attributes is zero, LCOM2 is undefined and displayed as zero.

$$LCOM3 = (m - \frac{sum(mA)}{a}) / (m - 1), \qquad (3)$$

LCOM3 varies between 0 and 2. Values 1 through 2 inclusive are considered alarming.

3.3 LCOM4

LCOM4 measures the number of "connected components" in a class. A connected component is a set of related methods (and class-level variables). There should be only one such component in each class. If there are two or more components, the class should be split into so many smaller classes. [14]. The implementation of these metrics in EO is available at [15].

3.4 LCOM5

'LCOM5' is a 1996 revision by B. Henderson-Sellers, L. L. Constantine, and I. M. Graham [13] of the initial LCOM metric proposed by MIT researchers. The values for LCOM5 are defined in the real interval [0, 1] where '0' means "perfect cohesion" and '1' means "no cohesion". Two problems with the original definition are:

- LCOM5 produces values across the full range and no specific value has a higher probability than any other (the original LCOM has a preference towards the value "0").
- Following on from the previous point, the values can be uniquely interpreted in terms of cohesion, suggesting that they are percentages of the "no cohesion" score '1' [13].

3.5 CAMC

In the CAMC metric, the cohesion in the methods of a class is determined by the types of objects (parameter access pattern of methods) these methods take as input parameters. The metric determines the overlap in the object types of the methods' parameter lists. The amount of overlap in object types used by the methods of a class can be used to predict the cohesion of the class [16].

The CAMC metric measures the extent of intersections of individual method parameter type lists with the parameter type list of all methods in the class. To compute the CAMC metric value, an overall union (T) of all object types in the parameters of the methods of a class is determined. A set M_i of parameter object types for each method is also determined. An intersection (set P_i) of M_i with the union set T is computed for all methods in the class. A ratio 7 of the size of the intersection (P_i) set to the size of the union set (T) is computed for all methods. The sum of all intersection sets P_i is divided by product of the number of methods and the size of the union set T, to give a value for the CAMC metric [16].

For a class with 'n' methods, if M_i is the set of parameters of method *i*, then
T = Union of M_i, $i = 1$ to n, iff P_i is the intersection of set M_i with T_i i.e.,

$$P = M_i \cap T_i$$

, then

$$CAMC = \frac{1}{kl}\sum\sum o_{ij} = \frac{\sigma}{kl} \qquad (4)$$

3.6 TCC and LCC

3.6.1 Connectivity between methods (CC)

The direct connectivity between methods is determined by the class abstraction. If there exists one or more common instance variables between two method abstractions, then the two corresponding methods are directly connected. Methods that are connected through other directly connected methods are indirectly connected. The indirect connection is the transitive closure of direct connection relationship. Thus, a method M_1 is indirectly connected with a method M_n if there is a sequence of methods $M_2, M_3, \ldots, M_{n-1}$ such that $M_1 \sigma M_2; \ldots; M_{n-1} \sigma M_n$ where $M_i \sigma M_j$ represents a direct connection [1].

3.6.2 Connectivity between methods (CC)

let NP(C) be the total number of pairs of methods. NP is the maximum possible number of direct or indirect connections in a class. If there are N methods in a class C:

$$NP(C) = N \times (N-1)2, \qquad (5)$$

let NDC(C) be the number of direct connections, and NIC(C) be the number of indirect connections.

Tight class cohesion (TCC) is the relative number of directly connected methods:

$$TCC(C) = NDC(C)/NP(C). \qquad (6)$$

Loose class cohesion (LCC) is the relative number of directly or indirectly connected methods:

$$LCC(C) = (NDC(C) + NIC(C)) = NP(C). \qquad (7)$$

3.7 NHD

The hamming distance (HD) metric was introduced by Counsell et al [17]. Informally, it provides a measure of disagreement between rows in a binary matrix. The definition of HD leads naturally to the Normalised Hamming Distance (NHD) metric [17], which measures agreement between rows in a binary matrix. Clearly, this means that the NHD metric is an alternative measure of the cohesion to CAMC metric. The parameter agreement between methods m_i and m_j is the number of places in which the parameter occurrence vectors of the two methods are equal. The parameter agreement matrix A is a lower triangular square matrix of dimension $k - 1$, where a_{ij} is defined as the parameter agreement between the methods i and j for $1 < j < i < k$, and 0 otherwise.

3.8 SCOM

The Sensitive Class Cohesion Metrics (SCOM) is a ration of the sum of connection intensities C_{ij} of all pairs *(i, j)* of m methods to the total number of pairs of methods. Connection intensity must be given more weight α_{ij} when such a pair involves more attributes. SCOM is normalized to produce values in the range [0..1], thus yielding meaningful values [18], [19]:

- Value zero means no cohesion at all. Thus, every method deals with an independent set of attributes.
- Value one means full cohesion. Thus, every method uses all the attributes of the class.

$$SCOM = \frac{2}{m(m-1)} \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} c_{ij} \times \alpha_{ij} \tag{8}$$

3.9 CCM

$$CCM = \frac{NC(C)}{NMP(C).NCC(C)} \tag{9}$$

where NC(C) is the number of actual connections among the methods of class, *NMP(C)* is the number of the maximum possible connections among the methods of the class C, *NCC(C)* is the number of connected components of the connection graph G_c that represents each method as a node and two methods A and B are connected in the connection graph if A and B access one or more attributes in common, method A invokes method B or vice versa, or methods A and B invoke one or more methods in common [20]. The implementation of this metric is available at [21].

3.10 OCC & PCC

If two or more methods have access to one attribute directly or indirectly, those methods seem to share the attribute. Such sharing is a kind of connections among methods. OCC quantifies the maximum extent of such connections within a class. This would be the maximum size of cohesive part of the class [22]. The weak-connection graph represents attribute-sharing. Furthermore, accesses to attributes include data-readings and data-writings. If a method writes data into an attribute, and another method reads data from the attribute, then these methods are dependent. Such relationship is a strong-connection graph.

When methods have access to attributes, those accesses include data-readings and data-writings. By focusing on such differences in accesses, we can consider dependent relationships among methods, which would be strong connections among methods. PCC quantifies the maximum extent of such dependent relationships within a class. This would be the maximum size of the highly cohesive part of the class [22].

$$OCC(C) = \begin{Bmatrix} max_{i=0\ldots n}\left[\frac{|R_w(m_i)|}{n-1}\right], & (n > 1) \\ 0, & (n = 1) \end{Bmatrix} \tag{10}$$

$$PCC(C) = \begin{Bmatrix} max_{i=0\ldots n}\left[\frac{|R_s(m_i)|}{n-1}\right], & (n > 1) \\ 0, & (n = 1) \end{Bmatrix} \tag{11}$$

4 Comparison of jPeek-Java vs jPeek-EO

Further tests were performed using other libraries to compare the original jPeek with EO-jPeek in terms of execution time. We observed that, while the output test results were correct, EO-jPeek performed a little slower than the original jPeek. Tables 1 through 5 show the summary of the comparison of the performances of the two solutions using the rio library [23], google guava, google guice, google Java format and the google gson libraries.

Table 1 – Performance Comparison (CQFN/Rio)

Metric	**Average Execution time (Original jPeek - CQFN)**	**Average Execution time (EO-jPeek-HSE)**	**CQFN/HSE**

LCOM	2.933101416	3.245975018	1.106669889
LCOM2	2.930070376	3.269466186	1.115831965
LCOM3	2.958978605	3.281973624	1.109157605
TCC	2.986430311	3.301571417	1.105524346
LCC	3.00814209	3.636467814	1.208875015
CAMC	2.892634487	3.212618709	1.110620344
NHD	3.042537141	3.290088701	1.081363529
		AVG (CQFN/HSE)	1.119720385

11.97% slower

Table 2 – Performance Comparison (Google Guava)

Metric	Average Execution time (Original jPeek – CQFN)	Average Execution time (EO-jPeek-HSE)	CQFN/HSE
LCOM	19.72788124	33.64671805	1.705541393
LCOM2	19.41652284	26.34026833	1.356590392
LCOM3	19.43815939	26.46744926	1.361623224
TCC	20.12063608	30.70516448	1.526053369
LCC	19.78492129	-1	-1
CAMC	19.22448852	23.29836535	1.211910805
NHD	19.13157122	23.56187158	1.231570126
		AVG (CQFN/HSE)	1.398881551

39.89% slower

Table 3 – Performance Comparison (Google Guice)

Metric	Average Execution time (Original jPeek - CQFN)	Average Execution time (EO-jPeek-HSE)	CQFN/HSE
LCOM	15.65215547	24.38325188	1.557820706
LCOM2	15.44143045	19.81895785	1.28349235
LCOM3	15.32358763	19.68824632	1.28483269
TCC	15.81485426	22.30919578	1.410648205
LCC	16.17761297	-1	-1
CAMC	15.14845912	18.34164848	1.210793014
NHD	15.00212016	18.56659935	1.237598362
		AVG (CQFN/HSE)	1.330864221

33.09% slower

Table 4 – Performance Comparison (Google Java Format)

Metric	Average Execution time (Original jPeek - CQFN)	Average Execution time (EO-jPeek-HSE)	CQFN/HSE
LCOM	6.8989501	12.26070285	1.777183872
LCOM2	6.548145032	9.12882328	1.394108291

LCOM3	6.588100481	9.186386037	1.394390699
TCC	6.91207695	10.53059473	1.523506582
LCC	6.942795968	-1	-1
CAMC	6.4429775	8.751528692	1.358305022
NHD	6.487771201	8.648385978	1.333028818
		AVG (CQFN/HSE)	1.463420547

46.34% slower

Table 5 – Performance Comparison (Google GSON)

Metric	**Average Execution time (Original jPeek - CQFN)**	**Average Execution time (EO-jPeek-HSE)**	**CQFN/HSE**
LCOM	9.588523412	20.0149703	2.087388166
LCOM2	9.018808293	12.15699949	1.347960739
LCOM3	9.294581914	12.19988844	1.312580658
TCC	9.577659488	16.01909444	1.672547919
LCC	9.305973649	-1	-1
CAMC	9.279319882	11.15777552	1.202434625
NHD	9.071305108	10.90333118	1.201958379
		AVG (CQFN/HSE)	1.470811748

47.08% slower

In the above comparisons, we see that Eolang generally computes the results accordingly but suffers in terms of computational speed. It runs slower than that of the original Jpeek implementation. Improving the computational speed will be considered in future works.

5 Conclusion

We have reimplemented several cohesion metrics originally implemented in Java now in EO. This proposed implementation was embedded into the framework of the jPeek tool as a module. The module calculates specific JPeek metrics by the EO programming language and its library. Certain metrics in the original JPeek had different results from that of EO-JPeek. These include LCOM4, CCM, SCOM, TCC, and LCC. During the development of these metrics in the EO programming language, our team ensured correctness by doing some manual checks, which showed that the corresponding EO-JPeek metrics were correct. Also, while developing the module, a number of bugs in the EO compiler and its standard library were fixed, which improved the compiler performance. The EO-JPeek tool is available at [12]. Testing this module proved that for certain EO use cases, reliability is comparable to that of Java.

Acknowledgements

This paper and the research behind it would not have been possible without the funding by Huawei Co. Ltd (TC202012080007) and the exceptional support of Yegor Bugayenko, Ivan Spirin, Eugene Popov and Vitaliy Korzun.

References

[1] J. M. Bieman and B.-K. Kang, ‘Cohesion and reuse in an object-oriented system’, in *Proceedings of the 1995 Symposium on Software reusability - SSR ’95*, Seattle, Washington, United States, 1995, pp. 259–262. doi: 10.1145/211782.211856.

[2] G. Myers, ‘Software reliability - principles and practices’, *undefined*, 1976, Accessed: May 28, 2021. [Online]. Available: /paper/Software-reliability-principles-and-practices-Myers/facfb2e637168e463942977e69d9004bac50b487

[3] M. Page-Jones, *The practical guide to structured systems design: 2nd edition*. USA: Yourdon Press, 1988.

[4] W. P. Stevens, G. J. Myers, and L. L. Constantine, ‘Structured design’, *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, Jun. 1974, doi: 10.1147/sj.132.0115.

[5] C. & Kemerer, ‘Project Metrics Help - Chidamber & Kemerer object-oriented metrics suite’. Accessed: Jun. 11, 2021. [Online]. Available: https://www.aivosto.com/project/help/pm-oo-ck.html

[6] H. Saleh, S. Zykov, and A. Legalov, ‘Eolang: Toward a New Java-Based Object-Oriented Programming Language’, in *Intelligent Decision Technologies*, Singapore, 2021, pp. 355–363. doi: 10.1007/978-981-16-2765-1_30.

[7] H. S. E. Team, ‘EO Skeleton’, *GitHub*. HSE, 2021. [Online]. Available: https://github.com/jizzel/jpeek/tree/master/src/main/java/org/jpeek/skeleton/eo/EOSkeleton.java

[8] H. S. E. Team, ‘EO Calculus’, *GitHub*. HSE, 2021. [Online]. Available: https://github.com/jizzel/jpeek/tree/master/src/main/java/org/jpeek/calculus/eo/EOCalculus.java

[9] H. S. E. Team, ‘EO’, *GitHub*. HSE, 2021. [Online]. Available: https://github.com/HSE-Eolang/eo

[10] Eclipse, ‘Eclipse Deeplearning4J Examples’. [Online]. Available: https://github.com/eclipse/deeplearning4j-examples/tree/master/dl4j-examples

[11] ‘jpeek’. https://hse-eolang.github.io/ (accessed Jul. 09, 2021).

[12] H. S. E. Team, ‘EO JPeek’, *GitHub*. HSE, 2021. [Online]. Available: https://github.com/HSE-Eolang/jpeek

[13] B. Henderson-Sellers, L. Constantine, and I. Graham, ‘Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)’, *Object Oriented Syst.*, vol. 3, pp. 143–158, 1996.

[14] M. Hitz and B. Montazeri, ‘Measuring coupling and cohesion in object-oriented systems’, in *Proceedings of International Symposium on Applied Corporate Computing*, 1995, pp. 25–27.

[15] H. S. E. Team, ‘EO LCOM4’, *GitHub*. HSE, 2021. [Online]. Available: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcom4.eo

[16] J. Bansiya, L. Etzkorn, C. Davis, and W. Li, ‘A Class Cohesion Metric For Object-Oriented Designs’, *undefined*, 1999, Accessed: Jun. 12, 2021. [Online]. Available: /paper/A-Class-Cohesion-Metric-For-Object-Oriented-Designs-Bansiya-Etzkorn/27091005bacefaee0242cf2643ba5efa20fa7c47

[17] S. Counsell, S. Swift, and J. Crampton, ‘The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design’, *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 2, pp. 123–149, Apr. 2006, doi: 10.1145/1131421.1131422.

[18] L. Fernando Capretz, ‘Bringing the Human Factor to Software Engineering’, *IEEE Software*, vol. 31, no. 2, pp. 104–104, Mar. 2014, doi: 10.1109/MS.2014.30.

[19] Y. Bugayenko, ‘The Impact of Constructors on the Validity of Class Cohesion Metrics’, in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, Mar. 2020, pp. 67–70. doi: 10.1109/ICSA-C50368.2020.00021.

[20] L. Fernández and R. Peña, ‘A Sensitive Metric of Class Cohesion’, *Information Theories and Applications*, vol. 13, p. 2006.

[21] H. S. E. Team, ‘EO CCM’, *GitHub*. HSE, 2021. [Online]. Available: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/CCM.eo

[22] H. Aman, K. Yamasaki, H. Yamada, and M.-T. Noda, ‘A Proposal of Class Cohesion Metrics Using Sizes of Cohesive Parts’, Apr. 2004.

[23] *cqfn/rio*. CQFN, 2021. Accessed: Jul. 09, 2021. [Online]. Available: https://github.com/cqfn/rio