Rewriting Java Module in CloudBU in EO Stage III

HSE Team
hsalekh@hse.ru
HSE
Moscow,Russia

Abstract

EO programming language is a research and development project that is still in progress. In the effort to move it forward, we suggest rewriting Java CloudBU module in EO while finding and fixing bugs along the way. The main objective is to improve the language reliability. In this paper, we explore the Jpeek project on cohesion metrics calculations, and then rewrite them in EO as a CloudBU module.

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

1 Introduction

Software developers aim for systems with high cohesion and low coupling [4]. 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 [12–14]. Class cohesion is one of the important quality factors 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 [11].

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 code 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.

2 The EO-jPeek Solution

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 CloudBU module 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 module, which includes the following elements:

1. A subroutine that parses and reconstructs the data of a class being assessed in the form of EO objects.
2. A class that integrates EO metrics into the general pipeline of the application, providing input and formatting output for each metric run.
3. 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 [30]. 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* [16] 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 to the pipeline of the JPeek tool. This is done in the *EOCalculus* class where we use *org.xembly.Directives*, *org.xembly.Xembler*, *org.jpeek.Header*, *org.cactoos.collection* and the *com.jcabi.xml* to help build the xml and return the result to the calling method (in *org.jpeek.App* class) of JPeek to include it in the JPeek tool pipeline.

2.3 Embedding the EO transpiler into the building process

Since the EO to Java transpiler [15] enhanced by the Team is distributed as a Maven artifact, it was embedded into the building process as follows:

1. 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.
2. 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.

3 Metrics

Fig. 1 shows the metrics tested using a github library call Node Packages [7]. Go to https://hse-eolang.github.io/ to browse through the results.

3.1 LCOM

LCOM was introduced in the Chidamber and Kemerer metrics suite [11]. It's also called LCOM1 or LOCOM, and it's calculated as follows: Let m be the number of methods, a be the number of attributes and μ_j be the amount of methods, which use attribute j, then

$$LCOM1 = \frac{1}{(1-m)}(\frac{1}{a}\sum_{j=1}^{a}\mu_j) - m$$

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 disparities 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

Rewriting Java Module in CloudBU in EO
Stage III

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 [7]

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 behavior. 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. The implementation of this metric is available at [21, 22].

This implementation has been tested on:

- Node Packages
  See graph here EO-LCOM1

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. This happens if the variables are dead or they are only accessed outside the class. Both cases represent a design flaw. The class is a candidate for rewriting as a module. Alternatively, the class variables should be encapsulated with accessor methods or properties. There may also be some dead variables to remove. 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.

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

Implementation details

Parameter *m* is equal to *WMC*. Parameter *a* contains all variables whether these are Shared or not. Each access to a variable is counted.

$$LCOM2 = 1 - sum(mA)/(m * a)$$

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.

$$LCOM3 = (m - sum(mA)/a)/(m - 1)$$

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

The implementation of these metrics is available at [23, 24].

This implementation has been tested on:

- Node Packages
  LCOM2: See graph here EO-LCOM2
  LCOM3: See graph here EO-LCOM3

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. In some cases, a value that exceeds 1 does not make sense to split the class (e.g., a class implements a form or a web page) as would affect the user interface. The explanation is that they store information in the underlying object that may be not directly using in the class itself. This implementation has been tested on:

- Node Packages
  See graph here EO-LCOM4

3.4 LCOM5

'LCOM5' is a 1996 revision by B. Henderson-Sellers, L. L. Constantine, and I. M. Graham [10] 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 [1] are:

1. 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").
2. 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' [10]

The implementation of this metric is available at [25] and has been tested on:

- Node Packages
  see graph here EO-LCOM5

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. This information is available when all method's prototypes have been defined, well before a class' methods are completely implemented [3].

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 [3]. 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

$$CAMC = \frac{1}{kl}\Sigma\Sigma O_{ij} = \frac{\sigma}{kl}$$

The implementation of this metric is available at [17].

This implementation has been tested on:

- Node Packages
  See graph here EO-CAMC

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 M1 is indirectly connected with a method Mn if there is a sequence of methods M_2 , M_3 , . . ., M_{n-1} such that M_1 δ M_2 ; . . .; M_{n-1} δ M_n where M_i δ M_j represents a direct connection.

The implementation of these metrics is available at [20, 31].

3.6.2 Parameter of TCC and LCC. 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:

$$NP(C) = N * (N - 1)/2,$$

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:

$$TCC(C) = NDC(C)/NP(C).$$

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

$$LCC(C) = (NDC(C) + NIC(C)) = NP(C).$$

This implementation of TCC has been tested on:

- Node Packages
  TCC: See graph here EO-TCC
  LCC: See graph here EO-LCC

3.7 NHD

The hamming distance (HD) metric was introduced by Counsell et al [6]. 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 [6], 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.

The implementation of this metric is available at [26] and has been tested on:

- Node Packages
  See graph here EO-NHD

3.8 SCOM

The Sensitive Class Cohesion Metrics (SCOM) is a ration of the sum of connection intensities $C_{(i,j)}$ of all pairs (i, j) of m methods to the total number of pairs of methods. Connection intensity must be given more weight $\alpha_{(i,j)}$ when such a pair involves more attributes. SCOM is normalized to produce values in the range [0..1], thus yielding meaningful values [5, 8]:

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

$$SCOM = \frac{2}{m(m-1)} \sum_{i=1}^{(m-1)} \sum_{j=i+1}^{m} (C_{i,j}\alpha_{i,j})$$

The implementation of this metric is available at [29].

This implementation has been tested on:

- Node Packages
  See graph here EO-SCOM

3.9 CCM

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

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 [9]. The implementation of this metric is available at [18] and has been tested on:

- Node Packages
  See graph here EO-CCM

3.10 OCC and 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 [2]. The weak-connection graph represents attribute-sharing. Furthermore, accesses to attributes include data-reading and data-writing. 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 [2].

The implementation of OCC and PCC metrics are available at [27, 28] and PCC been tested on:

- Node Packages
  PCC: See graph here EO-PCC
  OCC: See graph here EO-OCC

.

3.11 Comparison of Jpeek-Java vs Jpeek-EO

The comparison of JPeek-EO with JPeek-Java,in terms of runtime performance,is available in this attached file:

 Double Click icon to open.
or go to https://github.com/HSE-Eolang/hse-eolang.github.io/blob/main/performance_assesment.xlsx to download the excel workbook. Overall, the assessments shows that the JPeek-EO runs slower than JPeek-Java.

4 Conclusion

The proposed CloudBU module updated certain cohesion metrics originally implemented in JPeek now in EO. The module calculates specific JPeek metrics by the EO programming language and its library. This module was built into the framework of the JPeek tool.

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 it's standard library were fixed, which improved the compiler performance. The EO-JPeek application is available at [19]. Testing this module proved that for certain EO use cases, reliability is comparable to that of Java.

References

[1] Mohammed T. Abo Alroos. 20XX. Software Metrics. http://site.iugaza.edu.ps/mroos/files/Software-Metrics1.pdf

[2] Hirohisa Aman, Kenji Yamasaki, Hiroyuki Yamada, and Matu-Tarow Noda. 2004. A Proposal of Class Cohesion Metrics Using Sizes of Cohesive Parts. (April 2004).

[3] J. Bansiya, L. Etzkorn, C. Davis, and W. Li. 1999. A Class Cohesion Metric For Object-Oriented Designs. *undefined* (1999). /paper/A-Class-Cohesion-Metric-For-Object-Oriented-Designs-Bansiya-Etzkorn/27091005bacefaee0242cf2643ba5efa20fa7c47

[4] James M. Bieman and Byung-Kyoo Kang. 1995. Cohesion and Reuse in an Object-Oriented System. *SIGSOFT Softw. Eng. Notes* 20, SI (Aug. 1995), 259–262. https://doi.org/10.1145/223427.211856

[5] Yegor Bugayenko. 2020. The Impact of Constructors on the Validity of Class Cohesion Metrics. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 67–70. https://doi.org/10.1109/ICSA-C50368.2020.00021

[6] Steve Counsell, Stephen Swift, and Jason Crampton. 2006. The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design. *ACM Trans. Softw. Eng. Methodol.* 15, 2 (April 2006), 123–149. https://doi.org/10.1145/1131421.1131422

[7] Eclipse. [n.d.]. Node Packages. Retrieved 2021-07-08 from https://github.com/aventador3000/Node_package

[8] Luis Fernández and Rosalía Peña. [n.d.]. A Sensitive Metric of Class Cohesion. *Information Theories and Applications* 13 ([n. d.]), 2006.

[9] Maryam Hooshyar Habib Izadkhah. 2017. Class Cohesion Metrics for Software Engineering: A Critical Review. http://www.math.md/files/csjm/v25-n1/v25-n1-(pp44-74).pdf

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

[11] Chidamber & Kemerer. [n.d.]. Project Metrics Help - Chidamber & Kemerer object-oriented metrics suite. Retrieved 2021-06-14 from https://www.aivosto.com/project/help/pm-oo-ck.html

[12] G. Myers. 1976. Software reliability - principles and practices. *undefined* (1976). /paper/Software-reliability-principles-and-practices-Myers/facfb2e637168e463942977e69d9004bac50b487

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

[14] W. P. Stevens, G. J. Myers, and L. L. Constantine. 1974. Structured design. *IBM Systems Journal* 13, 2 (June 1974), 115–139. https://doi.org/10.1147/sj.132.0115

[15] HSE Team. 2021. EO. Retrieved June 14, 2021 from https://github.com/HSE-Eolang/eo

[16] HSE Team. 2021. EO Calculus. Retrieved June 14, 2021 from https://github.com/jizzel/jpeek/tree/master/src/main/java/org/jpeek/calculus/eo/EOCalculus.java

[17] HSE Team. 2021. EO CAMC. Retrieved June 14, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/camc.eo

[18] HSE Team. 2021. EO CCM. Retrieved June 21, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/CCM.eo

[19] HSE Team. 2021. EO JPeek. Retrieved June 14, 2021 from https://github.com/HSE-Eolang/jpeek

[20] HSE Team. 2021. EO LCC. Retrieved June 14, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcc.eo

[21] HSE Team. 2021. EO LCOM1. Retrieved June 14, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcom1.eo

[22] HSE Team. 2021. EO LCOM1 1. Retrieved June 14, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcom1_1.eo

[23] HSE Team. 2021. EO LCOM2. Retrieved June 14, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcom2.eo

[24] HSE Team. 2021. EO LCOM3. Retrieved June 14, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcom3.eo

[25] HSE Team. 2021. EO LCOM5. Retrieved June 28, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcom5.eo

[26] HSE Team. 2021. EO NHD. Retrieved June 14, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/nhd.eo

[27] HSE Team. 2021. EO OCC. Retrieved June 21, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/occ.eo

[28] HSE Team. 2021. EO PCC. Retrieved July 05, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/pcc.eo

[29] HSE Team. 2021. EO SCOM. Retrieved June 14, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/scom.eo

[30] HSE Team. 2021. EO Skeleton. Retrieved June 14, 2021 from https://github.com/jizzel/jpeek/tree/master/src/main/java/org/jpeek/skeleton/eo/EOSkeleton.java
[31] HSE Team. 2021. EO TCC. Retrieved June 14, 2021 from https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/tcc.eo