Rewriting Java Module in CloudBU in Eolang
Stage III

HSE Team
hsalekh@hse.ru
HSE
Moscow,Russia

Abstract

Eolang programming language is a research and development project that remains in an undeveloped state. In the effort to bring it above this state, we consider rewriting Java module in CloudBU in Eolang while we are able to find and fix bugs along the way. Our main objective is to develop the language with more reliable programming methods. In this work, we explore the Jpeek Java project, cohesion metrics calculations, and then rewrite this Jpeek metrics calculations in Eolang as the proposed CloudBU module.

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

1 Introduction

Software developers aim for systems with high cohesion and low coupling [3]. Cohesion metrics measure how well the methods of a class are related to each other. Metrics estimate the quality of different aspects of software. In particular, cohesion indicates how well the parts of a system hold together. Cohesion is one of software attributes representing the degree to which the components are functionally connected within a software module [10–12]. A metric to evaluate class cohesion is important in object-oriented programming because it gives an indication of a good design of classes. There are several metrics to measure class cohesion. A cohesive class performs one function. A non-cohesive class performs two or more unrelated functions. A non-cohesive class may need to be restructured 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. 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 [9].

A cohesive class performs one function. Lack of cohesion means that a class performs more than one function. This is not desirable. If a class performs several unrelated functions, it should be split up.

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 some standard metrics. This report focuses on re-implementing these metrics in the EO programming language.

2 Proposed Solution for EO-JPeek

The proposed solution is to integrate the metric(s) implemented in EO into the overall structure of the application (Module in EO built into jPeek Repo).

2.1 Description:

The proposed code Cloud BU module is written in EO and implements some of the metrics of JPeek, a static code analysis tool. Functionally, the module performs the tasks of calculating specific JPeek metrics utilizing the EO programming language and the capabilities of its standard object library. Moreover, the model has the infrastructure of Java classes that integrates EO metrics into the general architecture of JPeek. This document focuses on the general architecture of the proposed module, which includes the following elements:

1. A subroutine of parsing and reconstructing 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 of embedding the EO programming language transpiler into the building process of JPeek.

2.2 Integration of EO metrics into the pipeline of JPeek

To isolate EO metrics from XML, the following techniques are applied:

1. First, 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 [26]. 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 the following section.
2. Second, the "EOCalculus" [14] 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 metrics, the "EOCalculus" class utilizes a basic late binding technique made possible through specific embedding of the EO transpiler into the building process described in the following section.

The output of the metric is put into the XML file in the standard way to embed the results to the general pipeline of the JPeek application.

2.3 Embedding the EO transpiler into the building process

Since the EO to Java transpiler [13] enhanced by the Team is distributed as a Maven artifact it was embedded into the building process through the following steps:

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 is instructed to transpile all the EO metrics before building Java classes of the rest of the JPeek project structure. This makes possible to implement late binding of Java2EO references in Java code and, hence, to introduce a basic variant of Java2EO interoperability.

3 Metrics

Fig. 1 shows the graph of metrics results tested on Eclipse Deeplearning4J Examples. Fig. 2 shows the graph of metrics tested on Json Iterator. It includes the results for both EO implementation and that of the original Jpeek.

3.1 LCOM

LCOM was introduced in the Chidamber and Kemerer metrics suite [9]. It's 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.

$$LCOM1 = P - Q, if\, p > Q$$

$$LCOM1 = 0\, otherwise$$

LCOM1 = 0 indicates a cohesive class. LCOM1 > 0 indicates that the class needs or can be split into

Overall score[1] is **5.55** out of 10. Here is the matrix.

`5.5511`

Metric	E/C[2]	μ[3]	σ[3]	Min	Max	Green[4]	Yellow[4]	Red[4]	Score[5]	Defects[6]	Graph
CAMC	30/123 0%	0.31	0.20	0.07	0.75	2 2%	10 8%	18 15%	**1.80** -68%	23%	
EO_CAMC	31/123 0%	0.30	0.20	0.07	0.75	2 2%	10 8%	19 15%	**1.76** -68%	26%	
EO_LCOM1	30/123 0%	17.33	37.89	1.00	117.00	30 24%	0 0%	0 0%	**10.00** 80%	13%	
EO_LCOM2	30/123 0%	0.69	0.17	0.33	0.95	19 15%	10 8%	1 1%	**7.18** 29%	50%	
EO_LCOM3	26/123 0%	0.92	0.13	0.62	1.20	26 21%	0 0%	0 0%	**10.00** 80%	27%	
EO_NHD	31/123 0%	0.51	0.31	0.00	0.86	15 12%	6 5%	10 8%	**5.48** -1%	39%	
EO_SCOM	20/123 0%	0.08	0.12	0.00	0.48	0 0%	1 1%	19 15%	**0.60** -89%	11%	
LCOM	30/123 0%	17.33	37.89	1.00	117.00	13 11%	7 6%	10 8%	**5.08** -8%	13%	
LCOM2	30/123 0%	0.69	0.17	0.33	0.95	19 15%	10 8%	1 1%	**7.18** 29%	50%	
LCOM3	26/123 0%	0.92	0.13	0.62	1.20	26 21%	0 0%	0 0%	**10.00** 80%	27%	
NHD	24/123 0%	0.66	0.16	0.33	0.86	15 12%	7 6%	2 2%	**7.02** 26%	33%	
SCOM	20/123 0%	0.12	0.10	0.01	0.37	0 0%	0 0%	20 16%	**0.50** -91%	20%	

The average mistake of individual scores: **54%**, average defects rate: **28%**.

Figure 1. Metrics results on Eclipse Deeplearning4J Examples [5]

Overall score[1] is **5.63** out of 10. Here is the matrix.

`5.6280`

Metric	E/C[2]	μ[3]	σ[3]	Min	Max	Green[4]	Yellow[4]	Red[4]	Score[5]	Defects[6]	Graph
CAMC	54/151 0%	0.33	0.24	0.04	0.75	7 5%	16 11%	31 21%	**2.32** -59%	30%	
EO_CAMC	55/151 0%	0.32	0.24	0.04	0.75	7 5%	16 11%	32 21%	**2.29** -59%	18%	
EO_LCOM1	56/151 0%	60.79	200.18	1.00	885.00	56 37%	0 0%	0 0%	**10.00** 78%	6%	
EO_LCOM2	41/151 0%	0.57	0.25	0.08	0.95	20 13%	12 8%	9 6%	**5.72** 2%	37%	
EO_LCOM3	44/151 0%	0.77	0.31	0.09	1.33	33 22%	6 4%	5 3%	**7.90** 40%	25%	
EO_NHD	55/151 0%	0.53	0.32	0.00	0.93	27 18%	10 7%	18 12%	**5.53** -2%	25%	
EO_SCOM	35/151 0%	0.34	0.44	0.00	2.17	8 5%	3 2%	24 16%	**2.84** -49%	17%	
LCOM	56/151 0%	60.79	200.18	1.00	885.00	34 23%	5 3%	17 11%	**6.45** 15%	6%	
LCOM2	41/151 0%	0.57	0.25	0.08	0.95	20 13%	12 8%	9 6%	**5.72** 2%	37%	
LCOM3	44/151 0%	0.77	0.31	0.09	1.33	33 22%	6 4%	5 3%	**7.90** 40%	25%	
NHD	44/151 0%	0.66	0.19	0.33	0.93	27 18%	10 7%	7 5%	**6.78** 21%	23%	
SCOM	35/151 0%	0.56	0.60	0.01	1.82	13 9%	1 1%	21 14%	**4.09** -27%	20%	

The average mistake of individual scores: **33%**, average defects rate: **22%**.

Figure 2. Metrics results on JSON Iterator [8]

two or more classes, since its variables belong in 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 are attempting

to achieve many different objectives, and consequently are likely to behave in less predictable ways than classes that have lower LCOM1 values. Such classes could be more error prone and more difficult to test and could possibly be disaggregated into two or more classes that are more well defined in their 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 [19, 20].

This implementation has been tested on:

- Eclipse Deeplearning4J Examples

   Double click icon to open Graph
- Json-Iterator

   Double click icon to open Graph

3.2 LCOM2 and LCOM3

To overcome the problems of LCOM1, two additional metrics have been proposed: LCOM2 and LCOM3.

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 reusability. 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.

Which one to choose, LCOM2 or LCOM3? This is a matter of taste. LCOM2 and LCOM3 are similar measures with different formulae. 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.

Definitions used for LCOM2 and LCOM3
m = number of procedures (methods) in class
a = number of variables (attributes) in class
mA = number of methods that access a variable (attribute)
sum(mA) = sum of mA over attributes of a class
Implementation details
m is equal to WMC. a contains all variables whether Shared or not. All accesses to a variable are counted.

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

LCOM2 equals the percentage of methods that do not access a specific attribute averaged over all attributes in the class. If the number of methods or attributes is zero, LCOM2 is undefined and displayed as zero.
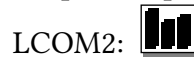
$$LCOM3 = (m - sum(mA)/a)/(m - 1)$$

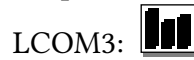LCOM3 varies between 0 and 2. Values 1..2 are considered alarming.

The implemetation of these metrics in available at [21, 22].

This implementation has been tested on:
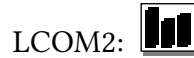
- Eclipse Deeplearning4J Examples

  LCOM2: Double click icon to open Graph

  LCOM3: Double click icon to open Graph
- Json-Iterator

  LCOM2: Double click icon to open Graph LCOM3: Double click icon to open Graph

3.2.1 Definitions for metric algorithm: m number of procedures (methods) in class a number of variables (attributes) in class mA number of methods that access a variable (attribute) sum(mA) sum of mA over attributes of a class

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

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

In a normal 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 there are no more than one method in a class, LCOM3 is undefined. If there are no variables in a class, LCOM3 is undefined. An undefined LCOM3 is displayed as zero.

3.3 LCOM4

LCOM4 measures the number of "connected components" in a class[12]. A connected component is a set of related methods (and class-level variables). There should be only one such a component in each class. If there are 2 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 if implementing a form or web page as it would affect the user interface of your program[13]. 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:

- Eclipse Deeplearning4J Examples

  Double click icon to open Graph
- Json Iterator

  Double click icon to open Graph

3.4 LCOM5

'LCOM5' is a 1996 revision by B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, of the initial LCOM metric proposed by MIT researchers. The values for LCOM5 are defined in the real interval [0, 1] where '0' describes "perfect cohesion" and '1' describes "no cohesion". Two problems with the original definition[15] are addressed: a) LCOM5 has the ability to give values across the full range and no specific value has a higher probability of attainment than any other (the original LCOM has

a preference towards[16] the value "0") b) Following on from the previous point, the values can be uniquely interpreted in terms of cohesion, suggesting that they be treated as percentages of the "no cohesion" score '1' [17].

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) that method's 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 [2].

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 Mi 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 summation 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 [2]. 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 If Pi is the intersection of set M_i with T i.e. P M T i i = then

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

The development of this metric is available at [15].

This implementation has been tested on:

- Eclipse Deeplearning4J Examples

  Double click icon to open Graph
- Json Iterator

  Double click icon to open Graph

3.6 MMAC

The MMAC metric calculates the average cohesion of all pairs of methods, where cohesion of pair of methods is the degree of similarity between them calculated as one minus normalized distance between a pair of rows, which represents the methods in the Direct Attribute-Type matrix.The MMAC is the average cohesion of all pairs of methods. In simple words this metric shows how many methods have the same parameters or return types. When class has some number of methods and most of them operate the same parameters it assumes better. It looks like class contains overloaded methods[18]. Preferably when class has only one method with parameters and/or return type and it assumes that class do only one thing. Value of MMAC metric is better for these one classes. Metric value is in interval [0, 1]. Value closer to 1 is better [19].

The Direct Attribute-Type (DAT) matrix that uses the types of the attributes themselves instead of using the types of the method parameters. The matrix is a binary k×l matrix, where k is the number of methods and l is the number of distinct attribute types in the class of interest. To construct the matrix, the names and return types of the methods and the types of the parameters and the attributes are extracted from the UML class diagram overviewed in Section 2.3. The DAT matrix has rows indexed by the methods and columns indexed by the distinct attribute types, and for $1 \leq i \leq k$, $1 \leq j \leq l$:

3.7 TCC and LCC

3.7.1 Connectivity between methods (CC). :
The direct connectivity between methods is determined from the class abstraction. If there exists one or more common instance variables between two method abstractions then the two corresponding methods are directly connected . Two methods that are connected through other directly connected methods are indirectly connected. The indirect connection relation is the transitive closure of direct connection relation. 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

$M_1 \, \delta \, M_2$; $\ldots$; $M_{n-1} \, \delta \, M_n$ where $M_i \, \delta \, M_j$ represents a direct connection.

The implementation of these metrics is available at [18, 27]

3.7.2 Definition of Measures. method pairs.
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:

- Eclipse Deeplearning4J Examples

    ▮▮ Double click icon to open Graph

3.8 NHD

The hamming distance (HD) metric was introduced by Counsell et al. [2001]. Informally, it provides a measure of disagreement between rows in a binary matrix. The definition of HD leads naturally to the NHD metric Counsell et al. [2002], which measures agreement between rows in a binary matrix. Clearly, this means that the NHD metric could be used as an alternative measure of the cohesion in the sense computed by the CAMC metric. The parameter agreement between methods mi and mj 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 to be the parameter agreement between methods i and j for 1 j < i k, and 0 otherwise. The parameter agreement matrix for the Alert class is shown in Figure 1c (the column totals have been shown in the parameter agreement matrix in order to facilitate the calculation of the NHD metric; 0s

have been omitted for convenience). For a class C, the NHD is defined as follows [Counsell et al. 2002]:

The implementation of this metric is available at [23].

This implementation has been tested on:

- Eclipse Deeplearning4J Examples

  📊 Double click icon to open Graph
- Json Iterator

  📊 Double click icon to open Graph

3.9 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 [4, 6]:

1. Value zero means no cohesion at all. Thus, every method deals with 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 [25].

This implementation has been tested on:

- Eclipse Deeplearning4J Examples

  📊 Double Click icon to open Graph
- Json Iterator

  📊 Double click icon to open Graph

3.10 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 or method A invokes method B or vice versa or methods A and B invoke one or more methods in common [7]. The implementation of this metric is available at [16] and has been tested on:

- Eclipse Deeplearning4J Examples

  📊 Double Click icon to open Graph
- Json Iterator

  📊 Double click icon to open Graph

3.11 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 [1]. The weak-connection graph represents attribute-sharings. Furthermore, accesses to attributes include data-readings and data-writings. If a method writes data onto an attribute, and another method reads data from the attribute, then a dependent relationship might be occurred between the methods. Such relationship is emphasized by the 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 highly cohesive part of the class [1].

The implementation of OCC metric is available at [24].

3.12 Comparison of Jpeek-Java vs Jpeek-EO

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

Double Click icon to open.

Overall, the assessments shows that the JPeek-EO runs slower than JPeek-Java.

4 Conclusion

As the proposed CloudBU module, we have implemented some cohesion metrics by re-writing some of these metrics originally implemented in JPeek now in EO. The module performs tasks of calculating specific JPeek metrics based on the EO programming language and its standard library. This module was built into the general architecture of the JPeek Java Application. The performance of JPeek-EO vs JPeek-Java has been assessed and provided in the attached file . The process of writing this module enabled us to fix bugs in the EO compiler and the standard library which in turn optimized the performance of the compiler by some margin. The EO-JPeek application is available at [17]. By writing this module, and assessing the result, we can conclude that EO programming is capable of being applied to similar use-cases as Java reliably.

References

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

[2] 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

[3] 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

[4] 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

[5] Eclipse. [n.d.]. Eclipse Deeplearning4J Examples. Retrieved 2021-06-17 from https://github.com/eclipse/deeplearning4j-examples/tree/master/dl4j-examples

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

[7] 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

[8] Jsoniter. [n.d.]. Json Iterator. Retrieved 2021-06-17 from https://github.com/json-iterator/java

[9] 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

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

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

[12] 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

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

[14] 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

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

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

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

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

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

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

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

[22] HSE Team. 2021. EO LCOM3. Retrieved June 14, 2021 from https://github.com/HSE-

     Eolang/jpeek/blob/master/src/eo/lcom3.eo

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

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

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

[26] 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

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