The Impact of Constructors on the Validity of Class Cohesion Metrics

Yegor Bugayenko
Huawei Technologies Co., Ltd.
Russian Research Institute,
Moscow, Russia
yegor.bugayenko@huawei.com

***Abstract*—Class cohesion is a measure of the degree to which a class's inner elements, like methods and attributes, are bound or related to one another. There have been over thirty different formulas proposed in order to calculate the metric. None of them are explicitly designed to deal with constructors in any different way than with regular methods—they simply treat them as identical entities. However, as many object-oriented theorists say, constructors play a very specific role in object life-cycle. In the scope of this empirical research, five different formulas were implemented in two ways: including constructors and excluding them. Then both set of formulas were applied to the same set of 1000 mid-size open source Java projects. The results obtained demonstrated how much of a distraction constructors were bringing into metric calculations.**

I. Introduction

Class cohesion, as a degree of "how tightly bound or related its internal elements are to one another," [31] is considered as one of the most important object-oriented software attributes [2], [23], [31]. A class with low cohesion has disparate and non-related members; cohesion can be used to identify the poorly designed classes [2], [4], [10].

There are over thirty different metrics introduced so far to measure functional [14], [32] cohesion of a class [13], [22]. All of them in one way or the other analyze the amount and the structure of class attributes and methods in order to calculate the metric. Almost none of them, however, explicitly mention in their descriptions whether constructors should be treated as methods or whether they have to have a special status.

From a theoretical standpoint, constructors and methods play very different roles in the life-cycle of an object [30]. This fundamental difference between them must affect class cohesion in some way. The assumption is that excluding constructors from formulas will make these formulas produce more meaningful and reasonable results.

II. Constructors' Mission

Most object-oriented publications, both scientific and practical, pay a relatively small amount of attention to object constructors in comparison to methods, attributes, types, and other ingredients of object-oriented design. For example, *Clean Code* [25] and *Refactoring* [16], rather popular books about object-oriented programming, have no special sections or even paragraphs giving an explanation of what constructors are for and what is important in their design. The book *Object-Oriented Software Construction* [26] mentions the word "constructor" just nine times through almost 1300 pages of text, calling them "creation procedures of a class," and providing almost no additional information. *The C++ Programming Language* [29] says that constructors are "member functions... that define a way to initialize an object of its class," adding no recommendations or discussions for their design. *Growing Object-Oriented Software* [18] has a single side note about constructors, which says that "our experience is that busy constructors enforce assumptions that one day we will want to break, especially when testing, so we prefer to keep them very simple—just setting the fields," and that was the only thought dedicated to constructors in the entire book. *Object-Oriented Analysis and Design* [7] calls constructors at the same time "member functions," "operations," and "metaoperations," saying that their basic responsibility is "to create an instance of the class and populate it with a set of rules, which it then uses for evaluation," but does not develop the discussion in any further direction.

Most modern object-oriented programming languages, like Ruby, Python or PHP technically don't differentiate constructors and regular object methods—they just call certain methods by names, which are selected by convention, e.g. `initialize()` in Ruby or `__construct()` in PHP. Smalltalk doesn't have constructors at all [20].

It seems that constructors, information-wise, are treated as second class citizens; there is obviously a lack of attention to them in academic and professional publications. In most places, including in the formulas of cohesion metrics, which will be demonstrated further, there is no distinction being drawn between them and regular methods.

This doesn't seem right, however, since the nature of constructors is different from the nature of object methods. First, constructors are not intended to do any work, but must only initialize object attributes [8]. Second, they (by definition) must "touch" all attributes in order to initialize them, or call other constructors that do that job; some languages—like Kotlin or Scala [28]—make this separation of primary and secondary constructors explicit and mandatory. Third, they always "return" the same object type [5]. Forth, they are always present in any object, either explicitly or synthetically

978-1-7281-4659-1/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSA-C50368.2020.00021

generated by the compiler. At least because of these differences, applying the same design "best practices" to them as to regular methods seems to be a flawed idea. The empirical analysis performed below demonstrates that this concern has its grounds.

III. Five Cohesion Metrics

There are over thirty different metrics existing to measure class cohesion [22]. A few of them were selected for the experiment:

The Normalized Hamming Distance (**NHD**) class cohesion metric measures the similarity in all methods of a class in terms of the types of their arguments [11]. Let l be the number of distinct parameter types, k be the number of methods, and c_j be the number of methods that have a parameter of type j, then,

$$NHD = 1 - \frac{2}{lk(k-1)} \sum_{j=1}^{l} c_j(k - c_j). \tag{1}$$

The Sensitive Class Cohesion Metric (**SCOM**) is a ratio 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 [15]:

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

The Method-Method through Attributes Cohesion (**MMAC**) metric is the average cohesion of all pairs of methods [12]. Let k be the number of methods, l be the number of distinct parameter types, and x_i be the number of methods that use type i, then

$$MMAC = \frac{1}{lk(k-1)} \sum_{i=1}^{l} x_i(x_i - 1). \tag{3}$$

The Cohesion Among Methods in Class (**CAMC**) measures the extent of intersections of individual method parameter type lists with the parameter type list of all methods in the class [3]. Let l be the number of distinct parameter types, k be the number of methods and p_i be the number of distinct parameter types used by method i, then

$$CAMC = \frac{1}{lk} \sum_{i=1}^{k} p_i. \tag{4}$$

The Lack of Cohesion of Methods (**LCOM**) is a correlation between the methods and the local instance variables of a class (we use the version suggested by [21], also known as LCOM5). 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

$$LCOM = \frac{1}{1-m} \left(\frac{1}{a} \sum_{j=1}^{a} \mu_j \right) - m. \tag{5}$$

For all metrics, except LCOM, greater values mean higher cohesion. The LCOM metric is reversed and demonstrates smaller values for higher cohesion. In order to make the discussion easier we use the inverted version of this metric, which is:

$$LCOM = 1 - LCOM. \tag{6}$$

The values of all metrics are in the $[0, 1]$ interval.

Even though it would be beneficial for this experiment to use all or most of the available metrics, this is not technically feasible for a number of reasons. First, the implementation of each metric takes a certain amount of time to understand the formula, implement the algorithm in Java, and make sure it works as intended (at least 80 work hours). Second, some metrics were suggested by their authors without a thorough testing with all possible Java code samples. In other words, they work in theory but can't be implemented "as is" in practice, while adapting their algorithms to the reality of Java code may break their integrity and compromise the original idea of their authors. The metrics used in this research are implemented exactly as they were suggested by their authors and they work correctly with all available Java classes.

IV. Research Method and Results

The most popular place for publishing open source Java artifacts is Maven Central Repository [27]. There were 35211 artifacts found,[1] which released at least one version in 2017. Even though such a large data set constitutes a perfect analysis corpus, it was decided to use only a subset of the entire Maven Central Repository. The main reason behind this decision was the amount of time and computing resources required for the analysis of a single Java artifact: 100-500 seconds (more than 100 days for the entire Maven Central).

Artifacts with less than one hundred or more than two thousand `.class` files were filtered out. 926 artifacts remained in the list. This size-based selection criteria was selected due to the following assumptions: 1) artifacts with fewer classes may represent abnormally better design due to the attention their developers were able to pay to each individual class, 2) artifacts with a lot of classes may represent the opposite situation, where developers didn't have enough time to pay attention to the quality of design. To exclude these abnormalities it was decided to exclude too big and too small artifacts and analyze only mid-sized ones.

Cohesion metrics were implemented in the scope of jPeek, an open source Java command line utility and a web system.[2] jPeek parses Java bytecode `.class` files via Javassist [9], analyzes its internals with the help of ASM [6], and then creates an XML representation of the entire artifact, where each class is presented by an XML element.

Interfaces, annotations, enums, anonymous classes, and classes generated by the AspectJ aspect-oriented framework were filtered out, because none of them represent actual

[1] https://github.com/yegor256/scrape-maven-central
[2] https://github.com/yegor256/jpeek

objects in terms of object-oriented design. They are either auto-generated or surrogates (like enums).

For example, take a simple Java class:

```
class Book {
  private int id;
  int getId() {
    return this.id;
  }
}
```

It would be represented by jPeek in the output XML file as follows:

```
<class id='Book'>
  <attributes>
   <attribute public='false' static='false'
    ↪  type='I'>id</attribute>
  </attributes>
  <methods>
    <method abstract='false' ctor='false' desc='()I'
    ↪  name='getId' public='true' static='false'>
      <return>I</return>
      <args/>
    </method>
  </methods>
</class>
```

Next, metric-specific XSL transformations [24] were applied to the XML file in each artifact, in order to generate measurements for each metric. For example, MMAC metric produced this XML file in the `org.mockito:mockito-all` artifact:

```
<metric>
  <title>MMAC</title>
  <app>
    <class id='InstantiatorProvider' value='1'/>
    <class id='InstantationException' value='0'/>
    <class id='AnswersValidator' value='0.0583'/>
    <class id='ClassNode' value='0.25'/>
    [... skipped ...]
  </app>
</metric>
```

Here, `AnswersValidator` and `ClassNode` are class names, while `0.0583` and `0.25` are their corresponding values of the MMAC formula, referred to below as $v(c, m, a)$, where c is the class, m is the metric, and a is the Java artifact.

Next, the minimum and the maximum of all v in each XML file were found. For example, in `MMAC.xml` they were 0 and 1 respectively, for the Mockito artifact mentioned above. Then, all classes, which demonstrated values equal to either minimum or maximum, were filtered out. This was done in order to minimize the influence of trivial classes, which most of the Java artifacts contain. For example, most metrics would consider this class as highly cohesive or even "perfect," despite its very low usefulness:

```
class Book {
  // The body of the class is empty,
  // no attributes, no methods.
}
```

Since most Java artifacts contain classes of a similarly trivial kind, it was considered reasonable to filter our highest and lowest values, to reduce noise. It was observed that some libraries have many classes with maximum or minimum metric values. The investigation of more than a hundred of them showed that all of them are either 1) empty classes with no methods and no attributes, or 2) utility classes with a large amount of static methods and no attributes.

Then, the arithmetic mean $\mu_{m,a}$ was calculated for a set of all N measurements $v(1, m, a), v(2, m, a), \ldots, v(N, m, a)$ as:

$$\mu_{m,a} = \frac{1}{N} \sum_{i=1}^{N} v(i, m, a). \tag{7}$$

Then, the standard deviation $\sigma_{m,a}$ was calculated as:

$$\sigma_{m,a} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (v(i, m, a) - \mu_{m,a})^2}. \tag{8}$$

It was observed that even though most artifacts demonstrated rather small values of σ (within one standard deviation of μ, or about 32%), in some of them the distribution of values was far from normal (normality checks were performed to test this fact). It was decided to filter out artifacts which had σ values larger than one standard deviation, in order to focus on artifacts where classes are designed with the highest uniformity. 100 artifacts remained in the list.

Thus, after the described calculations, the entire list of Java artifacts produced the matrix of $\mu_{m,a}$, where m values are columns and a values are rows, as shown in the Table I.

TABLE I
METRICS CALCULATED PER EACH ARTIFACT

	NHD	SCOM	MMAC	CAMC	LCOM
io.vavr:vavr	0.31	0.59	0.47	0.55	0.31
xerces:xercesImpl	0.78	0.51	0.30	0.33	0.14
org.takes:takes	0.21	0.44	0.38	0.87	0.32
com.h2database:h2	0.75	0.73	0.18	0.44	0.17
...					

Next, a ranking formula was introduced, in order to calculate the position of each particular artifact a in the entire set. Artifact rank r_a was calculated as an arithmetic average of all $\mu_{m,a}$ values in all M metrics:

$$r_a = \frac{1}{M} \sum_{j=1}^{M} \mu_{j,a}. \tag{9}$$

Thus, the entire list of artifacts was sorted by artifact ranks r_a. Each artifact got its own position p_a in the list.

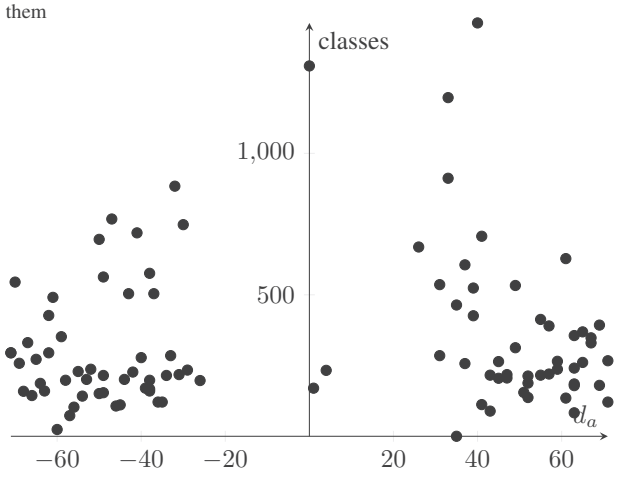
Then class cohesion formulas were modified, in order to exclude constructors from calculations, and all artifacts were ranked again and re-ordered. Ergo, two ordered lists of artifacts were created. The first one, where all formulas treated constructors as regular methods; and the second one, where constructors were excluded. The position of the artifact a in the first list is denoted as p_a, while the position of the same artifact in the second list is p'_a.

Then, the differences between p_a and p'_a were calculated, in order to understand how big the effect of constructor removal is and whether the effect has an obvious tendency:

$$d_a = p_a - p'_a. \tag{10}$$

The d_a numbers obtained are presented in the Figure 1, where horizontal axis is the value of d_a and the vertical axis is the amount of classes in the artifact.

Fig. 1. Differences between artifact positions, with constructors and without them



The distribution of numbers is visibly wide, taking into account the amount of items in the list: one hundred. This means that for many of them the position p_a has been changed significantly—most of the numbers d_a are grouped around +50 and -50. This means that the exclusion of constructors changed the essence of the opinions the metrics were giving about the artifacts provided. Highly cohesive classes became very uncohesive and vice versa.

V. Conclusion

First, the observed effect of constructor removal from the formulas of the five class cohesion metrics confirms that constructors can't be treated similarly to regular methods. They play a different role in object design and must find their own place in existing formulas or some new formulas have to be introduced.

Second, a more detailed individual analysis of the effect constructor removal had on a few selected classes from a few Java open source artifacts, made it obvious that a larger number of constructors does not make a class less cohesive, despite the opinion all analyzed metrics showed.

VI. Acknowledgements

The author is thankful to the contributors who helped design and implement jPeek open source Java software, including, but not limited to (in alphabetic order): Mihai Andronache, George Aristy, Sergey Kapralov, Sergey Karazhenets, Paulo Lobo, Vladimir Motsak, Alonso A. Ortega, Rok Povšič, Vseslav Sekorin, Mehmet Yildirim.

References

[1] K. K. Aggarwal, *Empirical Study of Object-Oriented Metrics*, Journal of Object Technology, Volume 5, Number 8, 2006.
[2] L. Badri et al., *Revisiting Class Cohesion: An empirical investigation on several systems*, Journal of Object Technology, Volume 7, Number 6, 2008.
[3] J. Bansiya et al., *A class cohesion metric for object-oriented designs*, Journal of Object-Oriented Programming, Volume 11, Number 8, 1999.
[4] V. R. Basili et al., *A validation of object-oriented design metrics as quality indicators*, IEEE Transactions on Software Engineering, Volume 22, Issue 10, 1996.
[5] J. Bloch, *Effective Java*, Addison-Wesley Professional, 3rd edition, 2018.
[6] E. Bruneton et al., *ASM: a code manipulation tool to implement adaptable systems*, Adaptable and extensible component systems, 30(19), 2002.
[7] G. Booch et al., *Object-Oriented Analysis and Design with Applications*, Addison-Wesley Professional, 3rd edition, 2007.
[8] Y. Bugayenko, *Elegant Objects*, Volume 1, Create Space, 2016.
[9] S. Chiba, *Javassist—A Reflection-based Programming Wizard for Java*, Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java, Volume 174, 1998.
[10] I. Chowdhury et al., *Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities*, Journal of Systems Architecture, Volume 57, Issue 3, 2011.
[11] S. Counsell et al., *The interpretation and utility of three cohesion metrics for object-oriented design*, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 15, Issue 2, 2006.
[12] J. A. Dallal, *A Design-Based Cohesion Metric for Object-Oriented Classes*, International Journal of Computer and Information Engineering, Volume 1, Number 10, 2007.
[13] J. A. Dallal, *Mathematical Validation of Object-Oriented Class Cohesion Metrics*, International Journal of Computers, Issue 2, Volume 4, 2010.
[14] H. Dhama, *Quantitative models of cohesion and coupling in software*, Journal of Systems and Software, Volume 29, Issue 1, 1995.
[15] L. Fernández et al., *A Sensitive Metric of Class Cohesion*, International Journal "Information Theories & Applications", Volume 13, 2006,
[16] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 1999.
[17] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002.
[18] S. Freeman et al., *Growing Object-Oriented Software, Guided by Tests*, Addison-Wesley Professional, 2009.
[19] E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.
[20] A. Goldberg, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
[21] B. Henderson-Sellers et al., *Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)*, Object Oriented Systems, Volume 3, Number 3, 1996.
[22] H. Izadkhah et al., *Class Cohesion Metrics for Software Engineering: A Critical Review*, Computer Science Journal of Moldova, Volume 25, Number 1(73), 2017.
[23] H. Kabaili, *Cohesion as changeability indicator in object-oriented systems*, Proceedings Fifth European Conference on Software Maintenance and Reengineering, 2001.
[24] M. Kay, *XSLT Programmer's Reference*, Wrox Press, 2000.
[25] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.
[26] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 2nd edition, 1997.
[27] F. P. Miller et al., *Apache Maven*, Alpha Press, 2010
[28] M. Odersky, *The Scala Language Specification*, EPFL, 2014.
[29] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Professional, 4th edition, 2013.
[30] D. West, *Object Thinking*, Microsoft Press, 2004.
[31] E. Yourdon et al., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 2nd Edition, Yourdon Press, 1978.
[32] J. M. Bieman, *Cohesion and reuse in an object-oriented system*, Proceedings of the 1995 Symposium on Software, 1995.