The Impact of Object Immutability on Some Class Cohesion Metrics

Yegor Bugayenko
Huawei Technologies Co., Ltd.
yegor.bugayenko@huawei.com

Sergey Zykov
Higher School of Economics
szykov@hse.ru

ABSTRACT

Class cohesion is a degree that demonstrates how tightly class inner elements, like methods and attributes, are bound or related to one another. Higher cohesion is a trait of good design that leads to better maintainability. An immutable object is an object whose state cannot be modified after it is created. The goal of this research was to analyze whether immutable objects are more cohesive and, because of that, more maintainable.

1 INTRODUCTION

Class cohesion, defined by Yourdon and Constantine [28] as a degree of how tightly bound or related its internal elements are to one another, is considered by Kabaili et al. [21] to be one of the most important object-oriented software attributes. A class with low cohesion has disparate and non-related members; cohesion can be used to identify the poorly designed classes [1, 3, 10].

An immutable object, as defined by Goetz et al. [15], is an object whose state cannot be modified after it is created. Immutable objects have a number of well-known advantages. First, according to Bloch [4], they are thread-safe by definition, which means that they can be safely shared between concurrent threads without any fear of collisions. Second, according to Hakonen et al. [18], their usage is side-effect free, which means that they can be passed to other methods and functions with full confidence that they will remain intact, no matter what will happen there. Third, according to Nayebi [25], the "identity mutability" problem disappears when objects are immutable, which surfaces when the object's identity is a derivative of its state. According to Bugayenko [6, pp.74–93] and Nayebi [25], there are many other advantages.

This is a perfect example of a mutable Java class:

```
class Book {
  private int id;
  int setId(int i) {
    this.id = i;
  }
}
```

An object of this class is capable of modifying its state `id` after creation:

```
Book book = new Book();
// Here the state is NULL
book.setId(123);
// Here the state equals to 123
```

To the contrary, the following class is immutable since its state `id` can't be modified after being encapsulated through the constructor, anyhow:

```
class Book {
  private final int id;
  Book(int i) {
    this.id = i;
  }
}
```

According to Li and Henry [23], "it seems logical that the more cohesive a class is, the easier the class is to maintain." Highly cohesive classes, according to the very definition of cohesiveness, are more maintainable because their design is more focused and easier to understand. In their research, Dallal [12] demonstrated that "classes with higher cohesion values are more maintainable than those with lower cohesion values." Better maintainability is an obvious objective of any software development. Thus, high cohesiveness is what classes in object-oriented programming must aim for.

The question is whether making classes immutable helps them become more cohesive. An empirical analysis of a large group of classes from open source Java artifacts was performed to find an answer to this question.

2 RELATED WORK

Cohesion itself, as an indicator of object-oriented design, was introduced by Yourdon and Constantine [28] in 1978 without specifying the exact algorithm of calculating the metric. Since then, many different algorithms have been suggested Izadkhah and Hooshyar [20]. Each of them pays attention to certain parameter of a class or a combination of them, such as the number of methods, attributes, method arguments, attributes usage frequencies, and so on. Some studies on existing object-oriented cohesion metrics even found inconsistencies among some of them [16, 27].

Even though high cohesion is known as a virtue of object-oriented design [21], very little work was done to find out what design principles affect the cohesion. The study of Patidar et al. [26] concluded that implementation inheritance negatively affects class cohesion. The research of Guyomarc'h and Guéhéneuc [17] attempted to identify how the usage of AOP aspects affects cohesion. Recently published research

by Bugayenko [7] discovered the relationship between cohesion and object constructors and suggested that their presence in cohesion calculating formulas makes a significant impact on the metric. The research of Bugayenko and Zykov [8] demonstrated the relevance betwee class size and immutability.

So far, to our knowledge, no research has been attempted to demonstrate the relationship between object immutability and class cohesion.

3 FIVE COHESION METRICS

According to a summary published by Izadkhah and Hooshyar [20], there are over thirty different metrics to measure class cohesion. A few of them were selected for the experiment:

The Normalized Hamming Distance (**NHD**) class cohesion metric, introduced by Counsell et al. [11], measures the similarity in all methods of a class in terms of the types of their arguments. 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**), introduced by Fernández and Peña [14], is a ratio of the summation 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:

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

The Method-Method through Attributes Cohesion (**MMAC**) metric, introduced by Dallal and Briand [13], is the average cohesion of all pairs of methods. 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**), introduced by Bansiya et al. [2], measures the extent of intersections of individual method parameter type lists with the parameter type list of all methods in the class. 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 the 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 Henderson-Sellers et al. [19], 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. 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. Second, some metrics were suggested by their authors without 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.

4 RESEARCH METHOD AND RESULTS

The most popular place for publishing open-source Java artifacts is the Maven Central Repository [24]. 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 only use a subset of the entire Maven Central. 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 on the list. This size-based selection criterion was selected due to the following assumptions: 1) artifacts with fewer classes may represent abnormally better design due to

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

the attention their developers were able to pay to each 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 an open-source Java command line utility and a web system. It parses Java bytecode `.class` files via Javassist [9], analyzes its internals with the help of ASM [5], 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 AspectJ aspect-oriented framework were filtered out because none of them represent actual 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 in the output XML file as such:

```
<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 [22] were applied to the XML file in each artifact 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.

Next, the minimum and the maximum of all `values` 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 that demonstrated values equal to either the minimum or maximum were filtered out. This was done to minimize the influence of trivial classes that most of the Java artifacts contain. For example, most metrics would consider this class to be highly cohesive or even "perfect," despite its lack of usefulness:

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

Since most Java artifacts contain classes of a similar kind, it was considered reasonable to filter out the 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 number of static methods and no attributes.

Then, cohesiveness C_i was calculated for each class i in each artifact as an average of all five metric values.

Then, the size S_i of each class i was calculated as a summary of the number of methods in the class and the number of attributes. This is a very rough and primitive metric, but it was required only to help visualize the entire set of classes on a graph.

Then, all 49,854 classes found were classified into two groups: mutable and immutable. A class was considered mutable if it had at least one non-final attribute. This definition of immutability is very far from being strict, but for this experiment, it was considered to be sufficient enough.

The visual presentation of 40704 mutable classes is in Figure 1.

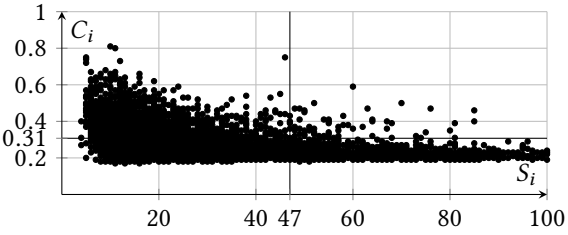

Figure 1: Mutable classes

The visual presentation of 9150 immutable classes is in Figure 2.

It is visually obvious that immutable classes are smaller and more cohesive. The average cohesiveness of mutable classes is 0.31, while immutable ones demonstrate a slightly bigger number of 0.34. The average size of mutable classes

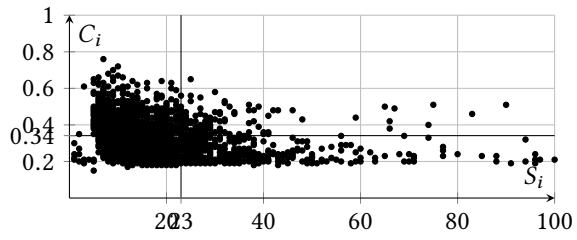

Figure 2: Immutable classes

is 67, while immutable ones have an average size of 41. As was mentioned above, this size metric won't make much sense for a single class or a small group of them, but on a bigger scale when dealing with thousands of classes, it does make sense. The numbers confirm the initial assumption that immutable classes are smaller and more cohesive, which is an obvious virtue for any object-oriented software. Higher cohesiveness and smaller size lead to higher maintainability of software modules, lowering the amount of time and effort a programmer must invest in them when modifications are required.

It was observed that the cohesiveness of all classes rarely dropped below 0.2. It was expected that they would start from zero and go up to 1 in a more or less normal distribution. However, the probability p_i of a cohesiveness C_i was distributed as Figure 3 demonstrates. It's difficult to say why 0.2 became a lower threshold for all five metrics being used in the analysis. This may be a good subject for further analysis.

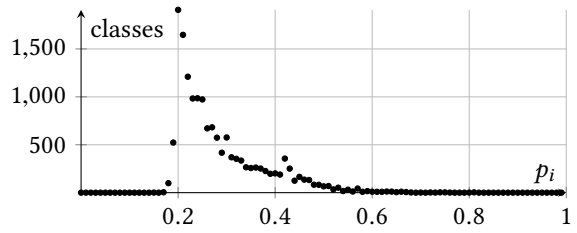

Figure 3: Distribution of cohesion probabilities

It is also visually obvious that the larger the class, the lower its cohesion. Even though it was intuitively understandable, the empirical analysis of a large number of Java classes proved that the size of a class negatively affects its cohesion and maintainability. It seems to be a more complex task to keep a class highly cohesive when its size grows.

5 DISCUSSION

Here is a practical comparison example of two Java libraries for sending emails. The first one is `commons-email` (version 1.5) by Apache with a large mutable class `SimpleEmail` at the core.[2] The second one is `jcabi-email` (version 1.10) with a set of immutable classes.[3]

Here is how Java source code may look if it sends an email using `commons-email`:

```
Email email = new SimpleEmail();
email.setHostName("smtp.googlemail.com");
email.setSmtpPort(465);
email.setAuthenticator(new DefaultAuthenticator("user",
↪  "pwd"));
email.setFrom("yegor256@gmail.com", "Yegor Bugayenko");
email.addTo("friend@jcabi.com");
email.setSubject("How are you?");
email.setMsg("Hi, how are you?");
email.send();
```

Here is how the same email sending scenario would be implemented with `jcabi-email`:

```
Postman postman = new Postman.Default(
  new SMTP("smtp.googlemail.com", 465, "user", "pwd")
);
Envelope envelope = new Envelope.MIME(
  new Array<Stamp>(
    new StSender("Yegor Bugayenko <yegor256@gmail.com>"),
    new StRecipient("friend@jcabi.com"),
    new StSubject("How are you?")
  ),
  new Array<Enclosure>(
    new EnPlain("Hi, how are you?")
  )
);
postman.send(envelope);
```

In the first example, it is a monster `SimpleEmail` class that can do many things, including sending MIME messages via SMTP, creating the message, configuring its parameters, adding MIME parts to it, and so on. There are 33 private properties, over a hundred methods, and about two thousand lines of code.

In the second example, there are seven objects instantiated via seven `new` calls. `Postman` is responsible for packaging a MIME message; `SMTP` is responsible for sending it via SMTP; stamps (`StSender`, `StRecipient`, and `StSubject`) are responsible for configuring the MIME message before delivery; and enclosure `EnPlain` is responsible for creating a MIME part for the message, which is going to be sent. These seven objects are constructed, encapsulated one into another, and then the postman is asked to `send()` the envelope over the wire.

From a user perspective, there is almost nothing wrong. `SimpleEmail` is a powerful class with multiple controls; just hit the right one and the job gets done. However, from a developer perspective, the `SimpleEmail` class is very difficult to maintain, mostly because the class is very big. Multiple getters and setters, which are the control points of the class,

[2] http://commons.apache.org/proper/commons-email/
[3] http://www.jcabi.com

modify object attributes, configuring its behavior. When new functionality is required, a developer has to add new attributes to the class and a new pair of setters and getters. Of course, such a modification decreases the cohesion of the class since there is very little or no interconnection between newly added attributes and previously existing ones. Every new method added to such a big class turns into an isolated island of functionality with its own set of attributes.

Cohesion metrics are designed to spot such isolated parts inside classes and raise red flags. A class can't be cohesive it if consists of a few logical blocks without a strong interconnection between them.

The immutability of a class makes it difficult to make a class larger without spending a substantial amount of effort on refactoring. If the `SimpleEmail` class was immutable in the beginning, it wouldn't be possible to add so many methods into it and encapsulate so many properties because an immutable object only accepts a state through its constructors. It's difficult to imagine a 33-argument constructor. When a class is immutable from the start, its developers are forced to keep it cohesive and small because they can't encapsulate too much or modify what's encapsulated. Just two or three arguments of a constructor and the reasonable limit is reached; everything on top of that will look strange and clumsy.

The immutable design of `jcabi-email` implements the exact same email sending functionality, but employs seven classes for that, instead of one. Of course, the cohesiveness of each of them is much higher than the one of `SimpleEmail`. The length of each of them is below 300 lines of code, which by itself is a perfect indicator of high readability and maintainability.

Moreover, to extend the functionality of the library, existing classes don't need to be modified. Each new feature must be added through the creation of new classes and implementing existing interfaces.

6 CONCLUSION

It was demonstrated by the empirical analysis of almost a thousand Java open source libraries and 40 thousand Java classes that immutable classes are smaller and more cohesive, which makes them more maintainable. Hence, it is advised to make classes immutable to increase the overall quality of the software.

Many thanks 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, and Mehmet Yildirim.

REFERENCES

[1] Linda Badri, Mourad Badri, and Alioune Badara Gueye. 2008. Revisiting Class Cohesion: An empirical investigation on several systems. *Journal of Object Technology* 7, 6 (2008), 55–75.

[2] Jagdish Bansiya, Letha Etzkorn, Carl G. Davis, and Wei Li. 1999. A Class Cohesion Metric For Object-Oriented Designs. *Journal of Object-Oriented Programming* 11, 8 (01 1999), 47–52.

[3] V. R. Basili, L. C. Briand, and W. L. Melo. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering* 22, 10 (1996), 751–761.

[4] Joshua Bloch. 2006. How to Design a Good API and Why It Matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. ACM, New York, NY, USA, 506–507.

[5] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*.

[6] Yegor Bugayenko. 2016. *Elegant Objects*. Vol. 1. Amazon.

[7] Yegor Bugayenko. 2020. The Impact of Constructors on the Validity of Class Cohesion Metrics. In *IEEE International Conference on Software Architecture*. San Paulo, Brazil.

[8] Yegor Bugayenko and Sergey Zykov. 2020. The Impact of Object Immutability on the Java Class Size. In *24th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems*.

[9] Shigeru Chiba. 1998. Javassist—a reflection-based programming wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vol. 174. Citeseer, 21.

[10] Istehad Chowdhury and Mohammad Zulkernine. 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture* 57, 3 (2011), 294–313.

[11] Steve Counsell, Stephen Swift, and Jason Crampton. 2006. The Interpretation and Utility of Three Cohesion Metrics for Object-oriented Design. *ACM Transactions on Software Engineering Methodologies* 15, 2 (2006), 123–149.

[12] Jehad Al Dallal. 2013. Object-oriented class maintainability prediction using internal quality attributes. *Information and Software Technology* 55, 11 (2013), 2028–2048.

[13] Jehad Al Dallal and Lionel C. Briand. 2010. An object-oriented high-level design-based class cohesion metric. *Information and Software Technology* 52, 12 (2010), 1346–1361.

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

[15] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. 2005. *Java Concurrency in Practice*. Addison-Wesley Professional.

[16] Bindu S Gupta. 1997. *A critique of cohesion measures in the object-oriented paradigm*. Master's thesis. Citeseer.

[17] Jean-Yves Guyomarc'h and Y Guéhéneuc. 2005. On the impact of aspect-oriented programming on object-oriented metrics. In *Proceedings of the 9th ECOOP Workshop on Quantitative Approaches to Object-Oriented Software Engineering (QAOOSE 2005), Glasgow, UK*.

[18] Harri Hakonen, Ville Leppänen, Timo Raita, Tapio Salakoski, and Jukka Teuhola. 1999. Improving object integrity and preventing side effects via deeply immutable references. In *In Proceedings of sixth Fenno-Ugric Symposium on Software Technology, FUSST'99*.

[19] Brian Henderson-Sellers, Larry L. Constantine, and Ian M. Graham. 1996. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems* 3 (1996), 143–158.

[20] Habib Izadkhah and Maryam Hooshyar. 2017. Class Cohesion Metrics for Software Engineering: A Critical Review. *The Computer Science*

Journal of Moldova 25 (2017), 44–74.

[21] H. Kabaili, R. K. Keller, and F. Lustman. 2001. Cohesion as changeability indicator in object-oriented systems. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. 39–46.

[22] Michael Kay. 2008. *XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer)* (4 ed.). Wrox Press Ltd., Birmingham, UK, UK.

[23] Wei Li and Sallie Henry. 1993. Object-oriented metrics that predict maintainability. *Journal of systems and software* 23, 2 (1993), 111–122.

[24] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. 2010. *Apache Maven*. Alpha Press.

[25] Fatih Nayebi. 2017. *Swift Functional Programming*. Packt Publishing Ltd.

[26] Kailash Patidar, R Gupta, and Gajendra Singh Chandel. 2013. Coupling and cohesion measures in object oriented programming. *International Journal of Advanced Research in Computer Science and Software Engineering* 3, 3 (2013).

[27] Ahmed M Salem and Abrar A Qureshi. 2011. Analysis of inconsistencies in object oriented metrics. *Journal of Software Engineering and Applications* 4, 02 (2011), 123.

[28] Edward Yourdon and Larry L. Constantine. 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (1st ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.