Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2020) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

24th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems

The Impact of Object Immutability on the Java Class Size

Yegor Bugayenko*[a], Sergey Zykov[b]

[a]*Huawei Technologies Co., Ltd., Moscow, Russia*
[b]*Higher School of Economics, Moscow, Russia*

Abstract

According to the subjective opinions of many industry experts, object immutability is a virtue in object-oriented programming, since it leads to side-effect-free design, cleaner code, better concurrency, and many other factors. However, it has never been empirically demonstrated exactly how immutability affects quality metrics of object-oriented programs. In the following research, we analyzed 97508 classes from 240 public Java repositories to find out how immutability affects the size of the code.

Keywords: Object-Oriented Programming; Immutability; NCSS

1. Introduction

On one hand, one of the most important factors negatively affecting the quality of object-oriented software is the *size* of classes, as was demonstrated by Li and Henry [9], Al Dallal [1]: classes with lower size have better maintainability. One of the simplest way to calculate the size of a Java class is via the NCSS metric, which stands for Non-Commenting Source Statements. Larger classes with bigger amount of statements demonstrate higher values of NCSS.

On the other hand, *immutability* is a property of a class in OOP. A class is immutable if it's impossible to modify its attributes after instantiation. In Java this technically means that all attributes have `final` modifier attached to them. Immutable classes greatly simplify programming, program maintenance, and reasoning about programs. Immutable classes can be freely shared, even between concurrent threads and with untrusted code, without the need to worry about modifications, even temporary ones, that could result in inconsistent states or broken invariants. Immutability is a recommended coding practice for Java [2].

This research empirically validates the relationship between immutability and class size.

E-mail address: yegor256@huawei.com

The paper is organized as follows. Section 2 defines various terms used in the paper. Section 3 covers related work in the areas of immutability and class size. Section 4 covers our empirical case study of 240 Java source code repositories. Section 5 covers limitations of both the metric and the study. Finally, we summarize our study in Section 6.

2. Background

Immutability in Java is a compile-time restriction of an object, making it impossible for other objects to modify any of its attributes. For example, objects of this class are immutable, thanks to the `final` modifier attached to all of its attributes:

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

The only way to modify `final` attributes is through the constructor of the class, which in Java by convention has the same name as the class (`Book` in the example above).

A class of any size may be immutable. A class with no attributes is immutable. A utility class, which is a class with no attributes, many static methods, and a private constructor, is immutable.

3. Related Work

Object immutability has been the subject of multiple researches since the begining of object-oriented programming [6, 12, 7, 15]. Even though, according to Potanin et al. [13], "the notion of immutability is not as straightforward as it might seem, and many different definitions of immutability exist," most industry experts consider immutability being a virtue of classes and objects in Java and other object-oriented programming languages [2]. There a few important concerns of immutability addresses so far by object-oriented researchers.

First, even though Java encourages programmers to explicitly make objects immutable by attaching the `final` modifier to its attributes, very often, as was demonstrated by Unkel and Lam [17], the modifier is not used even when attributes are not modified after object instantiation or, as demonstrated by Nelson et al. [10], after initialization. Such a delayed initialization of attributes, as explained by Fahndrich and Xia [5], leads to the prevalence of `null` and the ability to initialize circular data structures, while both of these practices have negative impact on software quality.

Second, despite the existence of the `final` modifier, Reflection API enables modification of any attributes in Java. Some mechanisms were suggested to overcome this technical vulnerability, for example via "freezer" objects [8], a new type system [16], or immutability assertion framework [11]. However, without any modifications to Java [14], thanks to the existence of Reflection API, it's not possible to say whether a particular Java object is modified after instantiation or not—any object can be modified. As was noticed by Hakonen et al. [6] in relation to immutability, "none of the current OO languages can prevent the programmer from writing a piece of code which violates the integrity of an object"

Third, aside from the `final` modifier, which is an explicit way of declaring read-only status of an attribute, there are also methods of code analysis, enabling the detection of implicit immutability. For example, a static flow-sensitive analysis algorithm was introduced by Porat et al. [12] to classify fields and classes as either mutable or immutable.

Forth, even if programmers do not violate encapsulation via reflection, the modified `final` only guarantees shallow immutability, as explained by Hakonen et al. [6]. Deep immutability, on the other hand, exists only if the object pointed by the attributes are deeply immutable (recursively).

However, to our knowledge, the impact of immutability on the Java class size has not been analyzed yet.

4. Empirical Results

A list of Java repositories were retrieved from GitHub via their public API. The first 240 repositories were taken, which satisfied the selection criteria: 1) more than 1,000 GitHub stars, 2) more than 200 Kb of data, 3) not archived, and 4) public. The list included popular Java open source products, such as Spring, RxJava, Guava, MyBatis, Clojure, JUnit, Lombok, Graal, Selenium, Spark, Mockito, Neo4j, Jenkins, Netty, and others.

Files with `.java` extension were taken from all repositories. There were 97508 files found. Classes without a single non-`static` attribute were excluded (such as utility classes, interfaces, or enums), despite the fact that some OOP experts consider such classes valid and immutable, saying that "the simplest immutable objects have no internal fields at all" [7]. We decided to not take these classes into account because they do not instantiate objects and because of that do not belong to object-oriented paradigm, as explained by [18, 3].

NCSS metric and immutability were calculated for each Java class, using `javalang`, an open source Java-parsing library written in Python. A class was considered immutable if it didn't have any attributes without `final` modifier.

The Figure 1 demonstrates the results obtained. The x-axis is the value of NCSS of Java classes. The y-axis is the share of immutable classes among all classes with the given NCSS. The diameter of the plot on the graph is related to the amount of classes found for the specific NCSS. Classes with NCSS larger than 1000 were excluded from the graph. This decision was motivated by the confounding effect the size of a class has on the validity of object-oriented metrics, as was empirically shown by El Emam et al. [4]. The right side of the graph has mostly (over 90%) mutable classes. The largest value of NCSS observed was 34212.

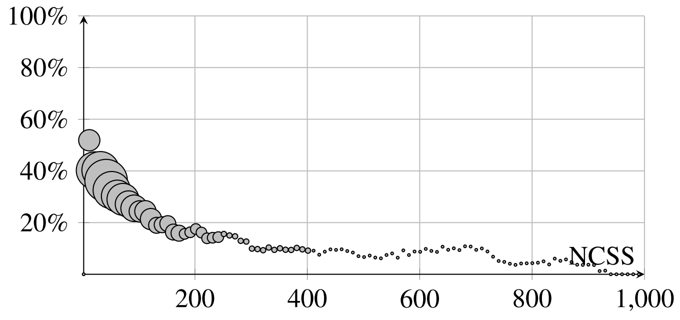

Fig. 1. Distribution of class immutability by NCSS

The tendency is visually obvious: larger classes are less often immutable. It was also observed that the majority of classes have NCSS values smaller than 100: 78304 classes out of 97508 total.

5. Discussion

The results obtained empirically confirm the hyposesis that immutability not only leads to better concurrency, side-effect free design, cleaner contracts between classes, but also to smaller classes, by the amount of methods and attributes inside.

This is a practical comparison example of two Java libraries designed by two different groups of programmers for the same purpose: sending emails from Java code. The first one is `commons-email` (version 1.5) by Apache with a large mutable class `SimpleEmail` at the core. The second one is `jcabi-email` (version 1.10) with a set of immutable classes. 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 message 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 thousands 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; enclosure `EnPlain` is responsible for creating a MIME part for the message, which is going to be send. These seven objects being constructed, encapsulating 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—a project needs to pick the right one and the job gets done. However, from a developer perspective `SimpleEmail` class is very difficult to maintain, mostly because the class is large. Multiple getters and setters, which are the control points of the class, modify object attributes, configuring its behavior. When a 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.

The immutability of a class makes it difficult to make a class larger without spending a big amount of efforts for refactoring. If the `SimpleEmail` class would be 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 in the first place, its developers are forced to keep it cohesive and small. Because they can't encapsulate too much and they can't 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 exactly the 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 creation of new classes and implementing existing interfaces.

Emprical results collected above confirm the logic explained. Larger classes tend to be mutable, which makes it easier for their authors to make them even larger when it's necessary.

6. Conclusion

It was empirically confirmed that larger Java classes tend to be immutable less frequently than smaller ones. At the same time it is possible to conclude that immutable classes tend to be smaller. Both conclusions justify the hypothesis that immutability in object-oriented programming leads to higher maintainability, since classes will be smaller.

References

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

[2] Joshua Bloch. 2016. *Effective Java*. Pearson Education India.

[3] Yegor Bugayenko. 2017. *Elegant Objects*. Amazon.

[4] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. 2001. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering* 27, 7 (2001), 630–650.

[5] Manuel Fahndrich and Songtao Xia. 2007. Establishing object invariants with delayed types. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. 337–350.

[6] 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*. Citeseer.

[7] Douglas Lea. 2000. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional.

[8] K Rustan M Leino, Peter Müller, and Angela Wallenburg. 2008. Flexible immutability with frozen objects. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 192–208.

[9] Wei Li and Sallie M Henry. 1993. Object-oriented metrics which predict maintainability. (1993).

[10] Stephen Nelson, David J Pearce, and James Noble. 2012. Profiling field initialisation in Java. In *International Conference on Runtime Verification*. Springer, 292–307.

[11] Igor Pechtchanski and Vivek Sarkar. 2005. Immutability specification and its applications. *Concurrency and Computation: Practice and Experience* 17, 5-6 (2005), 639–662.

[12] Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. 2000. Automatic detection of immutable fields in Java. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 10.

[13] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D Ernst. 2013. Immutability. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 233–269.

[14] Guy Steele, James Gosling, Gilad Bracha, and Bill Joy. 2005. The Java™ Language Specification.

[15] Antero Taivalsaari. 1993. On the notion of object. *Journal of Systems and Software* 21, 1 (1993), 3–16.

[16] Matthew S Tschantz and Michael D Ernst. 2005. Javari: Adding reference immutability to Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 211–230.

[17] Christopher Unkel and Monica S Lam. 2008. Automatic inference of stationary fields: a generalization of java's final fields. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 183–195.

[18] David West. 2004. *Object thinking*. Pearson Education.