The Journal of Systems and Software 137 (2018) 163–183

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A Metrics Suite for code annotation assessment

Phyllipe Lima[a,*], Eduardo Guerra[a], Paulo Meirelles[b,c], Lucas Kanashiro[b], Hélio Silva[a], Fábio Fagundes Silveira[d]

[a] National Institute For Space Research – INPE, Brazil
[b] University of São Paulo – IME-USP, Brazil
[c] University of Brasília – FGA-UnB, Brazil
[d] Federal University of São Paulo – ICT-UNIFESP, Brazil

ARTICLE INFO

Article history:
Received 2 February 2017
Revised 9 October 2017
Accepted 9 November 2017
Available online 2 December 2017

Keywords:
Code annotation
Software metrics
Thresholds

ABSTRACT

Code annotation is a language feature that enables the introduction of custom metadata on programming elements. In Java, this feature was introduced on version 5, and today it is widely used by main enterprise application frameworks and APIs. Although this language feature potentially simplifies metadata configuration, its abuse and misuse can reduce source code readability and complicate its maintenance. The goal of this paper is to propose software metrics regarding annotations in the source code and analyze their distribution in real-world projects. We have defined a suite of metrics to assess characteristics of the usage of source code annotations in a code base. Our study collected data from 24947 classes extracted from open source projects to analyze the distribution of the proposed metrics. We developed a tool to automatically extract the metrics and provide a full report on annotations usage. Based on the analysis of the distribution, we defined an appropriate approach for the calculation of thresholds to interpret the metric values. The results allow the assessment of annotated code characteristics. Using the thresholds values, we proposed a way to interpret the use of annotations, which can reveal potential problems in the source code.

1. Introduction

Code annotations were introduced in version 5 of the Java language as a feature that adds custom metadata on programming elements, such as methods and classes. This metadata can be consumed by tools or frameworks to gather additional information about the software, allowing the execution of more specific behavior. The code annotations proximity to the source code makes it a simple and fast alternative for metadata configuration.

The relevance of code annotations as a programming language feature can be seen by its usage in Java APIs for enterprise applications (JSR, 2007). For instance, EJB API uses annotations to configure transactions and security constraints, and JPA API applies annotations for object-oriented mapping. A study performed in 2011 (Rocha and Valente, 2011) verified that from 106 projects of the Qualitas Corpus project database (Tempero et al., 2010), 65 projects used annotations, showing that it is a widely adopted feature in Java source code.

Despite that annotations had a great potential for many interesting applications, its misuse can harm the maintenance of a software and prevent its evolution. For instance, an excessive amount of annotations can reduce code readability, and annotations duplicated through the project might be hard to refactor. Even the coupling of a class with an annotation schema can prevent its usage outside the application context.

Metrics are a way to retrieve information from software to assess its characteristics. For instance, well-known techniques use metrics associated with rules to enable the detection of bad smells on the source code (Lanza and Marinescu, 2006; Van Rompaey et al., 2007). However, traditional code metrics does not recognize the existence of annotations on programming elements, which can lead to an incomplete code assessment (Guerra et al., 2009). For example, a domain class can be considered simple using current complexity metrics. However, it can contain complex annotations for object-relational mapping. Another example is that the usage of a set of annotations couples the application to a framework that can interpret them. Current coupling metrics does not explicitly handle this situation.

The goal of this paper is to define metrics to assess the use of annotations in the source code. Additionally, this work also investigates how these metrics behave in open source projects by

* Corresponding author.
E-mail addresses: eduardo.guerra@inpe.br (E. Guerra), paulormm@ime.usp.br (P. Meirelles), lkd@ime.usp.br (L. Kanashiro), fsilveira@unifesp.br (F.F. Silveira).

https://doi.org/10.1016/j.jss.2017.11.024
0164-1212/

analyzing their distributions. Based on the frequency distribution for each one (Meirelles, 2013), we proposed an approach to define thresholds that can be used to separate values as very frequent, frequent, or less frequent. These thresholds can be used to identify uncommon annotation usage scenarios and reveal potential problems in the software implementation. These might prevent its evolution and maintenance. Thresholds values can also be used, for example, to create bad smells detection strategies (Lanza and Marinescu, 2006).

In short, there is no rule of thumb saying what type of distribution source code metrics it belongs to. In this paper, such information is relevant to determine the statistical value (average, median, or a percentile) that should be taken into consideration to monitor the proposed metrics. As shown throughout this research, there is no consensus in the distribution of source code metrics for object-oriented programming. Therefore, we wish to conceive our annotation metrics with a complete statistical analysis of their behavior by using real-world projects. For that, we use the approach proposed by Lanza and Marinescu (2006) as well as our approach based on percentile rank empirical analysis, adapted from Meirelles (2013). This percentile rank analysis is a tool to aid in interpreting the annotation metric presented in this paper.

An Eclipse IDE plugin named Annotation Sniffer was developed to extract the proposed annotation metrics from Java code. The metrics values are presented to the user as a complete report in an XML document, which is rendered to a web page by using an XSLT file. A total of 24947 Java classes obtained from open source projects that used annotations were selected and evaluated applying this plugin. The XML files were processed by scripts that generated the values distribution used to find suitable thresholds for each metric. Based on the findings, a discussion is conducted about how these metrics can be used to assess characteristics that can reveal relevant information about the annotations usage in a Java project.

The remainder of this paper is organized as follows. Section 2 introduces the concept of code annotations. Section 3 shows the related studies discussing software metrics analysis and arguing that previous works could provide a more complete assessment based on annotation metrics proposed in this paper. Section 4 presents the research design of this study discussing our research questions and methods as well as explaining our approach to collect and analyze the proposed code annotation metrics. Section 5 introduces the proposed techniques to assess the characteristics of an annotated code, in short, a candidate metrics suite for code annotation assessment. In the sequence, Section 6 summarizes a statistical analysis and a discussion of code annotation metrics distribution, as well as it presents our proposal approach for thresholds calculation for the candidate metrics suite. Finally, Section 7 concludes the paper, highlighting its main contributions and pointing paths to future works.

2. Metadata Configuration using Code Annotations

"Metadata" is an overloaded term in computer science and can be interpreted differently according to the context. Considering object-oriented programming, metadata is information about the program itself, such as classes, methods, and attributes. A class metadata, for example, is composed of its name, its superclass, its interfaces, its methods, and its attributes.

Some tools or frameworks can consume metadata and execute routines based on class structure. For instance, it can be used for source code generation (Damyanov and Holmes, 2004), compile-time verifications (Ernst, 2008; Quinonez et al., 2008), class transformation (Lombok, 2016), and framework adaptation (Guerra et al., 2010c). Sometimes, only the class structure is not enough to provide substantial information to be utilized elsewhere. It is necessary to configure additional custom metadata to parametrize how each programming element should be interpreted.

One option to define custom metadata is to use external storage, such as an XML file or a database (Fernandes et al., 2010). The drawback of this approach is the distance between the metadata and the referenced element, which adds some verbosity since a complete path to the referenced element must be provided. Another alternative, which is used by some frameworks, like Ruby on Rails (Ruby et al., 2009), is to define additional information using code conventions (Chen, 2006). Although this choice can be very productive in some contexts, code conventions have a limited expressiveness and cannot be used to define more complex metadata. For instance, a code convention could be used to define a method as a test method as in JUnit 3. However, it could not be used to define a valid range of a numeric property as in Bean Validation API.

Some programming languages provide features that allow custom metadata to be defined and included directly on programming elements. This feature is supported in languages such as Java, through the use of annotations (JSR, 2004), and in C#, by attributes (Miller and Ragsdale, 2004). A benefit of this alternative is that metadata definition is closer to the programming element and its definition is less verbose than the external one. The usage of code annotations is a technique called by some sources as attribute-oriented programming (Schwarz, 2004), which is defined as a programming technique used to mark software elements with annotations to indicate application-specific or domain-specific semantics (Wada and Suzuki, 2005).

In Java, this technique started with XDoclet (Walls and Richards, 2003), a tool that retrieved metadata defined in the source code as special JavaDoc tags. Those tags were applied to generate source code and XML files, which, in most cases, are metadata descriptors. This tool was widely employed in the development of J2EE applications by applying the EJB standard until version 2.1 (JSR, 2003) to generate extensive and mandatory XML descriptors (Walnes et al., 2003).

Annotations became an official language feature in Java 1.5 (JSR, 2004) spreading, even more, the use of this technique by the development community. Some base APIs in Java EE 6, like EJB 3.0 and JPA (JSR, 2007), use metadata in the form of annotations extensively. This native support to annotations encourages many Java framework developers to adopt the metadata-based approach in their solutions.

A single annotation defines only a small piece of information. An annotation-based API usually uses a group of related annotations that represent the set of metadata necessary for its usage. An annotation schema can be defined as a set of related annotations that belongs to the same API.

3. Related Works

Like any other language feature, annotations can bring benefits to the application if appropriately used (Guerra and Fernandes, 2013), but it can also be misused. Therefore, it is important to extract some metrics to help analyze how software is using this resource. By evaluating the metrics, the developer might conclude some potentially negative consequences of annotations.

Source code metrics help summarize particular aspects of software elements, detecting outliers in large amounts of code. They are valuable in software engineering since they enable developers to keep control of complexity, making them aware of abnormal growth of certain characteristics of the system. Metrics aid developers in keeping control of the quality of the code. However, to effectively take advantage of metrics, they should provide meaninful information and not just numerical values.

In this context, some studies on analysis of source code metrics have a limitation of not assessing whether the average metrics values are statistically representative for such analysis. Some related works on software metrics, including source code metrics, have shown that their data follow the power law distribution (Clauset et al., 2007), in particular, object-oriented metrics. It means that the source code metrics follow statistical distributions that have, in general, long tail and scale-free graph, which indicates that average value is not meaningful (Clauset et al., 2007).

For example, Wheeldon and Counsell (2003) evaluated 4 Java projects: JDK (Java Development Kit), Apache HTTP Server, Ant, and Tomcat. They were able to identify power law distribution for metrics that measured the number of attributes and methods.

Potanin et al. (2005) analyzed 35 Java programs and noticed that the relationship between the objects is a scale-free graph, which is different to a graph with randomly distributed edges. In such graph, given that the vertex edges follow a power law distribution, the average value is not representative.

Baxter et al. (2006) collected metrics from 56 Java free software projects. They showed that not all metrics follow a power law distribution, such as fan-out, number of attributes and number of public attributes.

Louridas et al. (2008) partially evaluated 11 projects (10 open sources and 1 restrict) written in Java, C, Ruby, and Perl. They investigated the values of fan-in and fan-out of modules and classes. Their results show that these metrics have distributions belonging to a power law family, regardless of the programming paradigm.

Based on these studies, we can argue that source code metrics does not necessarily follow either an exponential distribution (such as a power law) or a normal distribution. However, Lanza and Marinescu (2006) defined three threshold values for source code metrics such as LOC (Lines of Code), NOM (Number of Methods), and CYCLO (Cyclomatic complexity) based on statistical measurements. They used these three threshold values on 37 projects developed in C++ and on 45 developed in Java, some of them open source. They generalized the analysis and considered that the metrics value follows a normal distribution. Using the average value, and the standard deviation, they created reference regions and used thresholds to be a delimeter.

As seen, the statistical distribution of source code metrics has been extensively studied. On the one hand, metrics for object-oriented programs, written in Java, seems to follow an exponential distribution(Wheeldon and Counsell, 2003; Potanin et al., 2005; Concas et al., 2007; Ferreira et al., 2009; Yao et al., 2009). On the other hand, there is evidence that not necessarily some metrics will follow such distributions (Baxter et al., 2006; Lanza and Marinescu, 2006; Herraiz et al., 2011; 2012).

Furthermore, there are several works in the literature describing the use of annotations to solve problems and to implement solutions for a diverse range of domains. In software engineering, some of them apply annotations to support the implementation of design patterns (Meffert, 2006) or to enable architectural refactoring (Krahn and Rumpe, 2006). However, only a few works evaluated the use of annotations itself, focusing on design practices for metadata modeling or even performing studies to assess how annotations are currently being used by developers.

The first empirical study about the use of annotations found in the literature was conducted in an application of fractal-based development (Rouvoy and Merle, 2006). The evaluation was based on a comparison between applications that used a metadata based-framework called Fraclet and others that do not. As a result, there was a reduction of about 50 percent of the hand-written code without losing application semantics (Rouvoy et al., 2006). Despite this study revealing some potential in the use of metadata-based solutions, the result was restricted to the fractal-based development domain.

Guerra (2010) performed a more general experimental study about the use of metadata-based frameworks. It considered the different architectural scenarios where a metadata-based framework can be applied (Guerra et al., 2010b). The experiment was conducted using undergraduate students, which developed 3 distinct implementations of 4 different software components. Each implementation applied a different approach: without frameworks; with an object-oriented framework; and with a metadata-based framework. Unit tests were used to ensure that all implementations fulfilled the same requirements. This study performed a comparison between the implementations using object-oriented metrics (Lanza and Marinescu, 2006), questionnaires, observations, development time and a manual code analysis. The strongest conclusion of this study indicates that metadata-based frameworks reduce the coupling between the framework and the application. It also found evidence that a metadata-based framework has the potential to reduce development time, comparing to traditional frameworks. However, the experiment also revealed that when a problem happened, it is harder to debug when a metadata-based framework is used, since the behavior is defined indirectly through metadata.

With respect to annotation definition, Rocha and Valente (2011) investigated how annotations are used in open source Java systems. In this study, authors analyzed 106 open source projects from Qualitas Corpus project database (Tempero et al., 2010), from which 65 projects used annotations. The only information considered was the number of annotations and its type. In some of the evaluated systems, a high density of annotations was detected, indicating a possible misuse. Some other data extracted from this study also revealed that more than 90% of the annotations are in methods, and framework annotations are the most used ones.

Alba (2011) conducted a survey about legibility on annotated code. The author used a questionnaire to present 2 similar codes that represented different approaches expressing the same semantics, where developers should choose the most legible one. The questionnaire was answered by more than a hundred developers and had 27 questions focusing on the usage of annotations and the implementation of annotation idioms for different domains (Guerra et al., 2010a). The study pointed out that annotated code is perceived as more legible than the unannotated one. Besides that, the usage of annotation idioms can improve annotation readability and context where the annotation was used has an influence on the perception of legibility.

In short, these related works could provide a more complete assessment by considering the use of annotation metrics based on annotation itself and annotated elements. In this context, this paper aims to propose a suite for annotation metrics and shows how to evaluate code annotation in a set of real-world projects.

4. Research Design

In this paper, we argue that previous works studied annotated code from a different perspective due to the lack of approaches to assess the characteristics of code annotations. Thus, this section presents how this research was designed, describing the steps we adopted to reach our main goal: *assess the usage of annotated code from a software developer point of view*.

4.1. Research Questions

Based on such goal, we focus our study in the following research questions:

#RQ1 - What measurements could be performed in the source code to assess the characteristics of its code annotations usage?

Code annotations and annotated code elements have several characteristics that can be measured. For instance, an annotation has attributes values that can be defined. Code elements, such as classes and methods, can have several annotations. To answer this question, we identify essential characteristics and propose a suite of candidate metrics that enables the assessment of them. Moreover, to provide a means to generate and interpret the proposed metrics, we will use the Goal-Question-Metric (GQM) approach (Basili, 1992; Basili et al., 1994).

#RQ2 - For each metric, is it possible to define reference thresholds that can be used to classify its values?
Annotation metrics might behave as object-oriented metrics, that in general might not have a meaningful average value. Our candidate metrics will be extracted from the source code of a set of selected projects, and their distribution values will be statistically analysed. An analysis based on percentile rank (our proposed approach) will be carried out to provide a first step in interpreting the values. For comparison purpose, we also perform empirical analyses based on Lanza's method (based on average and standard deviation values). Afterwards, we determine which approach is the most suitable for thresholds values calculation.

#RQ3 - What is the common profile of a class that uses code annotations in Java? How large are the metrics outliers found?
The values obtained for each metric, their distribution and the code inspection of outliers can help to reach conclusions about how annotations are used by the projects, and how the candidate metrics can be used to detect potential design problems. It is important to detect if there are very large outliers because they can become a problem in code maintenance. Detecting common profiles in annotated classes may help developers keep track of code quality.

#RQ4 - May the usage of annotations create problems that can compromise code maintenance?
Currently, the software engineering community lacks studies dedicated to understanding how code annotations can impact software maintenance. Having threshold values available as a tool to analyze annotations, and being able to point out the presence of outlier values will indicate if code annotations can indeed become troublesome in maintaing the code. This question will leave a lot of ground to be researched in future studies.

4.2. Research Method

This section describes the steps performed to answer the research questions.

Step 1 - Propose the candidate metrics:
Based on important characteristics of annotations and annotated elements, as well as, using the GQM paradigm (Basili, 1992; Basili et al., 1994), we propose a metrics suite composed by the candidate metrics presented in Section 5.

Step 2 - Create a tool for the candidate metrics extraction:
To enable the candidate metrics extraction in projects, we developed a tool named Annotation Sniffer (ASniffer) that analyze Java source code and collect information about annotations. The tool generates an XML report containing the metrics for all project classes, annotations, and code elements[1].

Step 3 - Select projects to be analyzed:
To enable the analysis of metric values in real-world projects, we selected a set of open source projects. We chose projects that have a different number of annotated classes. For instance, some projects contain 80% of their classes annotated, while others contain only 10% of annotated classes. The selected projects are listed in Section 4.2.1.

Step 4 - Data extraction and processing:
The metrics values were extracted from the selected 25 projects, providing 24947 annotated Java classes as our actual sample data. The collected values were submitted to a Python script[2] to prepare the data. Afterwards, an R script[3] performed the statistical calculations on the prepared data. A distribution graph was generated for every metric of each project.

Step 5 - Statistical and qualitative analysis:
We analyzed the distribution of the candidate metrics values among all real-world projects and verified if the average value was meaningful. When the normal distribution was not able to fit the data, a percentile rank analysis (empirical) was also performed. In short, we compared normal distribution (Lanza's approach) and empirical percentile rank (the proposed approach). Based on the data and code inspection on outliers, a qualitative analysis allowed to identify how the metrics can characterize a common usage of annotations and how design problems can be detected.

4.2.1. Data Collection

The collected data came from a total of 24947 annotated Java classes extracted from 25 real-world projects selected to participate in this analysis. Given that annotations are an optional language feature, projects were elected considering the presence of annotations and the projects domains.

To conduct our analysis, we needed a large set of annotated classes as our sample. Selecting these 25 projects were not straightforward. Using random selection was not an option, since we needed projects that contained enough annotated classes to provide meaningful data to us. For instance, selecting the top-rated projects on GitHub[4] did not provide the sample we expected, since not all of these top projects had enough annotated classes. Projects such as Hibernate, JUnit, and Apache TomCat are not listed in the top GitHub Java projects[5], but they are well known Java projects and are recognized for their high annotation usage. Therefore, they were fundamental in our analysis, in particular to discuss the existing outliers. The selected projects are shown in Table 1.

Based on Nagappan et al. (2013), we want to determine the similarity and diversity among the selected projects. For this analysis, we propose three dimensions: Type, Percentage of Annotated Classes (PAC), and LOC. Type can assume two values: "framework" or "application". This dimension is important because it allows capturing two different approaches of annotation usage. From an overall perspective, a framework has its own annotations, and applications use annotations provided by frameworks. PAC, the second dimension, exposes how many classes contain annotations. This allows us to fetch projects ranging from a low annotation usage to heavily based ones. Since this analysis aims to show similarity and diversity, capturing projects with different sizes aids in reaching this goal. Therefore, we include the well-known LOC

[1] https://gitlab.com/annotationmetrics/annotationsniffer.

[2] https://gitlab.com/annotationmetrics/annotationmetrics/blob/master/data/output_FromXMLtoCSV/glueMetrics.py.

[3] https://gitlab.com/annotationmetrics/annotationmetrics/tree/master/data/R.

[4] By top GitHub projects, we mean the projects rated with top stars.

[5] https://gitlab.com/annotationmetrics/annotationmetrics/blob/master/topStarsJavaProjects.txt.

Table 1
Selected projects.

Project	Repository	Version
Agilefant	github.com/Agilefant/agilefant	3.5.4
ANTLR	github.com/antlr/antlr4	4.5.3
Apache Derby	github.com/apache/derby	10.12.1.1
Apache Isis	github.com/apache/isis	1.13
Apache Tapestry	github.com/apache/tapestry-5	5.4.1
Apache Tomcat	github.com/apache/tomcat	9.0.0
ArgoUML	argouml.tigris.org/source/browse/argouml/trunk/src	0.34
Eclipse CheckStyle	http://github.com/acanda/eclipse-cs	6.2.0
Dependometer	github.com/dheraclio/dependometer	1.2.9
ElasticSearch	github.com/elastic/elasticsearch	5.0.0-rc1
Hibernate Commons	github.com/hibernate/hibernate-commons-annotations	4.0.5
Hibernate Core	github.com/hibernate/hibernate-orm	5.2.0
JChemPaint	github.com/JChemPaint/jchempaint	3.3–1210
Jenkins	github.com/jenkinsci/jenkins	2.25
JGit	github.com/eclipse/jgit	4.5.0
JMock	github.com/jmock-developers/jmock-library	2.8.2
JUnit	github.com/junit-team/junit5	5.0.0-M2
Lombok	github.com/rzwitserloot/lombok	1.16.10
Megamek	github.com/MegaMek/megamek	0.41.24
MetricMiner	github.com/mauricioaniche/metricminer2	2.6
OpenCMS	github.com/alkacon/opencms-core	10.0.1
OVal	github.com/sebthom/oval	1.86
Spring Integration	http://github.com/spring-projects/spring-integration	4.3.4
VRaptor	github.com/caelum/vraptor4	4.2.0-RC4
VoltDB	github.com/VoltDB/voltdb	6.5.1

Table 2
Project dimensions.

Project	Type	PAC (%)	PAC Category	LOC	LOC Category
Agilefant	Application	10.50	Low Usage	43,539	Low
Dependometer	Application	35.00	Low Usage	28,123	Low
ANTLR	Application	10.60	Low Usage	101,600	High
ArgoUML	Application	31.70	Low Usage	195,670	High
Apache Derby	Application	18.80	Low Usage	689,869	High
Eclipse Checkstyle	Application	44.70	Medium Usage	20,453	Low
JChemPaint	Application	63.70	Medium Usage	27,371	Low
Jenkins	Application	64.00	Medium Usage	124,576	High
Elastic Search	Application	48.20	Medium Usage	615,637	High
Megamek	Application	70.60	High Usage	306,210	High
OpenCMS	Application	72.60	High Usage	476,074	High
VoltDB	Application	80.80	High Usage	542,030	High
Apache Isis	Framework	21.80	Low Usage	163,665	High
Apache Tapestry	Framework	25.60	Low Usage	156,450	High
Apache Tomcat	Framework	31.00	Low Usage	300,819	High
Hibernate Commons	Framework	54.40	Medium Usage	2812	Low
JGit	Framework	64.80	Medium Usage	173,681	High
Hibernate Core	Framework	54.70	Medium Usage	593,854	High
JMock	Framework	66.40	High Usage	9580	Low
Metric Miner	Framework	71.00	High Usage	23,602	Low
OVal	Framework	75.00	High Usage	17,381	Low
JUnit	Framework	68.00	High Usage	25,935	Low
Lombok	Framework	69.40	High Usage	50,324	Low
Spring Integration	Framework	76.70	High Usage	208,750	High
VRaptor	Framework	85.00	High Usage	26,660	Low

metric to measure the project's size. With this dimension, this paper can discuss how annotations behave in projects with different sizes. Table 2 associates each project with the proposed dimensions.

For the dimension PAC, we defined three categories: below 35% - "low usage", between 35% and 65% - "medium usage", and greater than 65% - "high usage". For LOC, we defined two categories: below 100 thousand lines - "low" and above - "high". Among the projects, there are 12 frameworks and 13 applications, roughly 50% for each dimension. Considering projects with high annotations usage, we have 7 projects classified as "framework" and 3 projects classified as "application". This observation highlights the fact that framework projects are usually more annotation based. From an annotation perspective, the projects have a PAC ranging from 10% up to 85%, which shows the diversity. For LOC, there are 11 projects considered "low" and 14 considered "high".

Combining the proposed dimensions in pairs, there are a total of 16 possibilities, considering the categories that each dimension can assume. The combinations are Type with PAC, Type with LOC, and PAC with LOC. For the pair Type-PAC, we have "application" combined for each of the 3 PAC categories, as well as "framework" for the same three categories. This pattern follows on to the total of 16 combinations. For each pair combination, we count the number of projects that fulfill both dimension involved in it. Table 3 presents the pair combination between dimensions, with the obtained values. There is an average of 4.7 projects per combination, with a minimum value of 2 ("Low PAC - Low LOC") and a maximum of 8 ("Application - High LOC"). With this analysis we

Table 3
Pair combination of dimensions.

Combinations	Number of Projects
Application - Low Annotation	5
Application - Medium Annotation	4
Application - High Annotation	3
Application - Low LOC	4
Application - High LOC	8
Framework - Low Annotation	3
Framework - Medium Annotation	3
Framework - High Annotation	7
Framework - Low LOC	7
Framework - High LOC	6
Low Annotation - Low LOC	2
Low Annotation - High LOC	6
Medium Annotation - Low LOC	3
Medium Annotation - High LOC	4
High Annotation - Low LOC	6
High Annotation - High LOC	4
Average	**4.69**

show that, according to our proposed dimensions, we have similar projects and also diverse projects among the chosen 25.

4.2.2. Data Analysis

We are interested in analyzing the metrics distribution values among all projects as well as verify if the average value is representative. As already mentioned, some object-oriented metrics follow an exponential distribution graph, which means that the average value does not contribute to understanding the behavior. This paper investigates if annotation metrics also falls into that same category.

In exploratory data analysis (Section 6), based on an approach proposed by Meirelles (2013), it is verified if the normal distribution is able to fit the data and make a percentile rank empirical analysis to find where the data seems to be meaningful. We use Lanza's approach (Lanza and Marinescu, 2006) to calculate possible thresholds values based on average and obtained the percentile rank (5%, 10%, 25%, 50%, 75%, 90%, 95%, 99%) of each metric to find the threshold where the average is not representative. Afterwards, we confront each method to conclude whether calculating the average value correctly provides useful information.

On the one hand, Lanza's approach is based on a calculation using the average and standard deviation value that was used to define what they called reference regions (Low, Medium, and High). In this paper, we use the following nomenclature:

- Lanza-Low = average − standard deviation
- Lanza-Medium = average
- Lanza-High = average + standard deviation

The value of a metric is considered an outlier when it is 50% greater than the maximum Lanza-High threshold. In summary, Lanza and Marinescu (2006) assume that the average value is meaningful.

On the other hand, adapted from Meirelles (2013), our percentile ranking analysis considers a more realistic approach and yielded in better threshold values. Percentile analysis can be a more flexible way of obtaining the thresholds. As discussed in Section 6, we use the percentile 90 as a reference point. For instance, Meirelles (2013) showed that for object-oriented metrics, the percentile reference is 75. In a normal distribution, we can observe that the median is the reference point because its value is close to the average one. The percentile analyzed in this paper is divided into 3 boundaries, explained below.

- Very Frequent = until percentile 90
- Frequent = between percentile 90 and 95
- Less Frequent = between 95 and 99

Using the average and standard deviation (Lanza's approach) is a strict rule, and so it supposes that every metric will behave the same way, which is not true, as we present in Section 6. Some metrics distributions have an abrupt growth in the final percentiles, in general from the percentile 90. Hence, the percentiles analysis provides a complete visualization of what is happening.

5. The Candidate Metrics for Code Annotations

This section presents the main proposal of this work, which is to evaluate code annotations. To achieve this goal, a metrics suite is defined to assess the quality and complexity of annotated codes, measuring how code annotations are used in the implementation of software. The metrics are classified in this work according to the type of code element used as a basis, which can be the annotation itself, an element (such as a class, method or attribute declaration) or from an overall class perspective. All of the proposed metrics are extracted directly from the source code using the Annotation Sniffer tool. For this reason, the suite proposed can be considered a set of primary metrics and may allow future metrics to be derived from them (R. B. Grady, 1987).

The source code defined in Fig. 1 exemplifies how the metrics are extracted. The annotations of such code exist in real-world frameworks. However, their usage mixed in the same class are only for illustrative purposes.

5.1. Metrics Suite

We designed a GQM model to guide in the definition of the metrics suite. Our model consists of four questions, which aims to understand how we can assess annotated code. With these questions, we were able to propose seven metrics that, combined, provides enough information to answer the proposed questions from our GQM model presented in Table 4.

Alongside the GQM, we grouped the proposed metrics into 3 categories: i) Class metrics; ii) Annotation Metrics; and iii) Code Element Metrics. A class metric measures the annotation from a class perspective, an annotation metric measures the annotation declaration itself and a code element metric measures the annotation on a declared element (it can be a class declaration, a method, or an attribute).

The following subsections define the candidate metrics, explain how each one can help to answer the metrics elaborated in our GQM model, as well as, classifies them and uses Fig. 1 to illustrate a simple extraction of each of them.

5.1.1. Attributes in Annotation - AA

Attributes of an annotation is an important characteristic to be used to provide additional information. A high number of attributes in the same annotation can result in a messy and hard-to-read code. The candidate metric Attributes in Annotation (AA) measures the number of attributes inside an annotation in a code element.

It is possible to define an annotation or a list of annotations as the type of an attribute that goes inside an annotation. In the case of nested annotations, each one is reported separately with their respective attributes number. AA is classified as an Annotation Metric.

Considering the example of Fig. 1, the value of AA for the annotation `@DiscriminatorColumn` is 2, since it has the attributes `discriminatorType` and `name`. Another example is `@AssociationOverrides`, which contains some nested annotations. Its AA value is 1, since there is only 1 attribute (value), while both `@AssociationOverride` contains 2 attributes each (`value` and `joinColumns`); therefore, their AA value is 2.

```
1  import javax.persistence.AssociationOverrides;
2  import javax.persistence.AssociationOverride;
3  import javax.persistence.JoinColumn;
4  import javax.persistence.NamedQuery;
5  import javax.persistence.DiscriminatorColumn;
6  import javax.ejb.Stateless;
7  import javax.ejb.TransactionAttribute;
8
9  @AssociationOverrides(value = {
10       @AssociationOverride(name="ex",
11          joinColumns = @JoinColumn(name="EX_ID")),
12       @AssociationOverride(name="other",
13          joinColumns = @JoinColumn(name="O_ID"))})
14 @NamedQuery(name="findByName",
15       query="SELECT c " +
16             "FROM Country c " +
17             "WHERE c.name = :name")
18 @Stateless
19 public class Example {...
20
21    @TransactionAttribute(SUPPORTS)
22    @DiscriminatorColumn(name = "type",
23             discriminatorType = STRING)
24    public String exampleMethodA(){...}
25
26    @TransactionAttribute(SUPPORTS)
27    public String exampleMethodB(){...}
28
29    @TransactionAttribute(SUPPORTS)
30    public String exampleMethodC(){...}
31
32 }
```

Fig. 1. Code for candidate metrics examples.

Table 4
GQM approach applied for our annotation metrics proposal.

Goal	(Purpose)	*Assess*
	(Issue)	*the usage of*
	(Object)	*annotated code*
	(Viewpoint)	*from software developer viewpoint*
(Question)	Q1	**What is the amount of information defined in an annotation?**
(Metric 1)	**AA**	Attributes in Annotation
(Metric 2)	**LOCAD**	LOC in Annotation Declaration
(Question)	Q2	**How hard is to define and maintain an annotation definition?**
(Metric 1)	**AA**	Attributes in Annotation
(Metric 2)	**LOCAD**	LOC in Annotation Declaration
(Metric 3)	**ANL**	Annotation Nesting Level
(Question)	Q3	**What is the amount of metadata defined in a class by using annotations?**
(Metric 4)	**AED**	Annotation in Element Declaration
(Metric 5)	**AC**	Annotations in Class
(Metric 6)	**UAC**	Unique Annotation in Class
(Question)	Q4	**How a class is coupled with annotation schemas?**
(Metric 6)	**UAC**	Unique Annotation in Class
(Metric 7)	**ASC**	Annotation Schemas in Class

Based on the GQM from Table 4, AA contribute for the answers of Q1 and Q2. That is justified because the number of attributes is directly related to the amount of information present in the annotation, which also reflects the number of parameters that should defined, influencing the code maintenance. This metric can be formally defined by the pseudocode in Algorithm 1.

Algorithm 1 Attributes in Annotation pseudocode.

```
procedure AA(annotation)
    Initialize aa = 0
    for each attribute in annotation do
        aa ← aa + 1
    end for
end procedure
```

5.1.2. Lines of Code in Annotation Declaration - LOCAD

The types of annotation attributes are limited to primitive types, Strings, enums, instances of a Class, other annotations, or an array of any of these. Due to such limitation, sometimes the information of an attribute can be defined using a large String. Because of that, only the number of attributes defined in an annotation might not be enough to reflect the amount of information defined in an annotation.

The candidate metric LOC (Lines of Code) in Annotation Declaration (LOCAD) measures the number of lines of code used in the source file to define an annotation. LOCAD is classified as an Annotation Metric.

Looking at the code in Fig. 1, it is possible to verify that the LOCAD for the annotation @NamedQuery is 4, although the value of AA is only 2.

Similarly to AA, LOCAD aids in Q1 and Q2. Q1 is justified since there is a direct relationship between the number of lines of code and the amount of information in an annotation declaration. Also, the number of lines declared in an annotation influences its maintainability. It is similar to the LOC metric, the more lines a class(or any other code element) contains, the harder it gets to maintain. The pseudocode in Algorithm 2 presents the extraction procedure

Algorithm 2 LOC in Annotation Declaration.

```
procedure LOCAD(annotation)
    Initialize locad = 0
    for each line in annotation.toString do
        locad ← locad + 1
    end for
end procedure
```

for LOCAD. Notice the call `toString`, it returns the lines present in annotation declaration.

5.1.3. Annotation Nesting Level - ANL

The definition of annotations as attributes of other annotations can help to define a more organized data structure. However, its overuse can lead to annotation definitions that are hard to understand and maintain. The candidate metric Annotation Nesting Level (ANL) measures the maximum level of nesting reached inside an annotation. ANL is classified as an Annotation Metric.

As an example, the annotation @AssociationOverrides in Fig. 1 has an ANL value of 0. It is easy to figure this out since this annotation is the declaration. Therefore, it is not nested in any other annotation. However, the Annotation AssociationOverride has an ANL of 1, as it is defined inside @AssociationOverrides. Moreover, the Annotation @JoinColumn has ANL value of 2, considering it is defined inside @AssociationOverride which already is the first level of nesting. Notice this metric does not measure the number of attributes in an annotation declaration, while AA does. ANL measures the maximum nesting level of an annotation declaration, which in this example is @JoinColumn.

According to our GQM model, ANL contributes to Q2. The higher an annotation is nested, the more complexity is involved in its definition, complicating its maintenance. This is similar to a conditional loop (if-else). The deeper a conditional loop gets, the more complicated it gets to maintain it. The pseudocode in Algorithm 3 presents the ANL extraction procedure. The call `previousCodeElement` returns the type of parent that contains the calling annotation. If it the parent type if an annotation declaration, the call `isAnnotation` returns `true`.

Algorithm 3 Annotation Nesting Level.

```
procedure ANL(annotation)
    Initialize anl = 0
    if annotation.previousCodeElement.isAnnotation then
        anl ← 1+ ANL(annotation.previousCodeElement)
    else
        anl ← 0
    end if
end procedure
```

5.1.4. Annotations in Element Declaration - AED

An element in the source code, such as attributes, methods, and classes, may need to have several annotations to inform some metadata. However, an excessive number of annotations in the same element can also reduce code legibility, maintenance, and reusability. The candidate metric Annotations in Element Declaration (AED) focus on each code element individually and measures the number of annotations defined in its context. This metric also counts nested annotations. AED is a Code Element Metric since it focuses on a single element declaration and not the whole class.

As an example, the value of AED for the method exampleMethodA() in Fig. 1 is 2 since it has the annotations @TransactionAttribute and @DiscriminatorColumn. A more complicated example is the value of AED for the class Example. Counting the nested annotations the value for this element is 7.

This metric is not concerned with the annotation definition itself, but rather the number of annotations declared in a code element. With our GQM model it contributes to Q3, since it reports the number of metadata configured in a class or code element. The pseudocode in Algorithm 4 exposes the AED extraction procedure.

Algorithm 4 Annotation in Element Declaration.

```
procedure AED(code_element)
    Initialize aed = 0
    for each annotation in code_element do
        aed ← 1 + ANNOTATIONS_IN_ATTRIBUTES(annotation)
    end for
end procedure
procedure ANNOTATIONS_IN_ATTRIBUTES(annotation)
    Initialize aiA = 0                    ▷ Attributes in Annotation
    for each attribute in annotation do
        if attribute.value.isAnnotation then
            aiA ← 1 + ANNOTATIONS_IN_ATTRIBUTES(attribute.value)
        end if
    end for
end procedure
```

Algorithm 5 Annotation in Class.

```
procedure AC(class)
    Initialize ac = 0
    for each code_element in class do
        ac ← ac + AED(code_element)
    end for
end procedure
```

Notice there are two procedures involved. The first one counts how many annotations a code element contains in its declaration. For each annotation, nested annotations must be fetched and accounts for the total AED in a code element, hence the procedure ANNOTATIONS_IN_ATTRIBUTES.

5.1.5. Annotations in Class - AC

The total number of annotations of a given class can also be important information for the assessment of how annotations are used in its context. The candidate metric Annotations in Class (AC) counts the number of annotations in all elements of a given class including nested annotations. AC is a Class Metric since it being measuring from a class perspective and not from a single element.

Although it can be directly obtained by counting all class annotations, it can also be calculated from the value of AED from all the class elements. The following equation can be used to calculate this metric:

$$AC = \sum_{\text{each class element}} AED \tag{1}$$

Algorithm 6 Unique Annotations in Class.

```
define ANNOTATIONS_LIST
procedure UAC(class)
    Initialize uac = 0
    for each code_element in class do
        for each annotation in code_element do
            if !ANNOTATIONS_LIST.contains(annotation) then
                uac ← 1 + UNIQUE_ANNOTATIONS_COUNT(annotation)
                ANNOTATIONS_LIST.add(annotation)
            end if
        end for
    end for
end procedure
procedure UNIQUE_ANNOTATIONS_COUNT(annotation)
    Initialize uaC = 0                    ▷ Unique Annotations Counter
    for each attribute in annotation do
        if attribute.value.isAnnotation AND
!ANNOTATIONS_LIST.contains(attribute.value) then
            uaC ← 1 + UNIQUE_ANNOTATIONS_COUNT(annotation)
        end if
    end for
end procedure
```

Algorithm 7 Annotation Schemas in Class.

```
define SCHEMAS_LIST
procedure ASC(class)
    Initialize asc = 0
    for each code_element in class do
        for each annotation in code_element do
            if !SCHEMAS_LIST.contains(annotation.schema) then
                asc ← asc + 1
                SCHEMAS_LIST.add(annotation.schema)
            end if
        end for
    end for
end procedure
```

As an example, the class Example in Fig. 1 has the value of 11 for the AC metric. All annotations, including the nested ones, are counted. As seen on our GQM model, Q3 is concerned at measuring the number of metadata configured in a class, and the AC metric provides a notion for this.

5.1.6. Unique Annotations in Class - UAC

This metric is similar to AC, but it measures the distinct number of annotations in a class. For an annotation to be considered similar to another, they should have the same annotation type and the same value for all of its attributes, which means that at least two code elements are configured with the same metadata. For instance, if two different code elements contain the annotation such as @Annotation1(atr1 = "This is an example"), then they will be counted only once. They contain the same name (@Annotation1) and their attributes have the same value (atr1 = "This is an example"). If, for example, the attribute atr1 had different values, they would not be a unique annotation since they are configuring different metadata. In short, the goal of this metric is to register the number of distinct metadata configurations that used annotations in the class. Therefore, UAC is a Class Metric.

Based on the metric values of AC and UAC, it is possible to obtain the number of annotations that are similar to other ones in the same class. A high number of similar annotations can reveal repetition in configurations that might be hard for software maintenance and a high number of distinct annotations means that several particular metadata configurations are performed in the given class.

The UAC value for the class Example (Fig. 1) is 9. The annotations @TransactionAttribute are considered similar as they have the same attribute values, but it is counted only once. However, annotations @AssociationOverride and @JoinColumn are not similar because they have different values for their attributes. Thus, each one is counted separately.

5.1.7. Annotation Schemas in Class - ASC

As stated before, annotation schema can be defined as a set of related annotations that belongs to the same API. For example, the JPA API has an annotation schema with a set of annotations that can be used by applications to define object-relational mapping. When using annotations from different schemas, the code may become tightly coupled with their domains raising reusability problems. Also, it may prevent software evolution as well as compromise its readability and maintenance. The candidate metric ASC (Annotation Schemas in Class) measures the number of annotation schemas that a class is currently using. For the implementation of this metric, we considered that annotations from the same annotation schema belongs to the same package. Therefore, the extraction is simply done by counting the number of distinct annotation packages imported. ASC is a Class Metric. On our code example illustrated in Fig. 1 the ASC is 2. The class is using the EJB schema as well as the JPA schema. Both of these were obtained by directly observing the imported package, and identifying the packages javax.ejb and javax.persistence.

5.2. Measurements Classification

The intent of the proposed metrics is to measure certain characteristics about code annotations usage. Briand et al. (1996) proposed a generic mathematical framework that defines several concepts, such as size, length, complexity, cohesion, and coupling, used to classify the metrics. The goal of this section is to classify the proposed metrics according to the concepts proposed by Briand et al. (1996), reasoning how they fulfill their required properties.

AA and LOCAD metrics reflect the **size** properties of an annotation definition: its number of attributes and its lines of code, respectively. These two metrics can be considered similar to measurements that exist for other code elements in regular object-oriented metrics. For instance, it is also possible to measure the number of lines of code and the number of fields (similar to attributes) from a class. AED and AC metrics use the same measurement regarding the number of annotations for a single code element and for the class as a whole, respectively. These four metrics fulfil all the requirements of size metrics, since they cannot be negative (Non-negativity), are null or zero in the absence of annotations (Null value), and can be summed to reflect the size of two elements (Module Additivity).

ANL reflects a **length** property of an annotation. This classification is justified by the fact that the nesting level of a group of annotations is the greatest nesting level among them, which relates to the Disjoint Modules property. Annotation definitions are static and can always be considered disjoint to each other. The fact that is not possible to add relationships between them make this metric fulfil the properties of Non-Decreasing Monotonicity for Non-Connected Components and Non-Increasing Monotonicity for Connected Components.

UAC performs a measurement that counts the number of unique annotations. It captures the number of distinct metadata definitions and according to Briand et al. (1996) properties can be considered a **complexity** metric. The unique annotations from a group of classes cannot be less than the UAC from a single class

Table 5
Metrics summary.

Name	Acronym	Reference	Type	Summary
Attributes in Annotation	AA	Annotation	Size	Measures the number of attributes in an annotation definition
LOC in Annotation Declaration	LOCAD	Annotation	Size	Measures the number of lines in an annotation declaration
Annotation Nesting Level	ANL	Annotation	Length	Measures the nesting level of an annotation
Annotation in Element Declaration	AED	Code Element	Size	Measures the number of annotations declared on a code element
Annotations in Class	AC	Class	Size	Measures the total number of annotations in a class
Unique Annotation in Class	UAC	Class	Complexity	Measures the number of distinct annotations in a class
Annotation Schemas in Class	ASC	Class	Coupling	Measures the number of different annotations schemas in a class

and cannot be greater than the sum of their values, fulfilling the properties of Module Monotonicity and Disjoint Module Additivity.

Finally, ASC reflects a **coupling** property that characterizes the relationship of a class with annotation schemas. Making an analogy with Afferent Coupling, which count the number of external classes accessed, ASC considers the annotations configured instead of method invocation. It fulfills the properties of coupling metrics since adding new annotations will never decrease its number (Monotonicity), merging two annotation schemas or two classes can only decrease its value (Merging of Modules) and merging unrelated classes will result in a sum of their ASC (Disjoint Module Additivity).

5.3. Metrics Summary

We summarize the proposed metrics suite in Table 5, which has the metric name, its acronym, its reference for measurement, its type based on (Briand et al., 1996) concepts as well as a brief summary of the metrics definition.

The metrics in Table 5 refers to 3 different types of code element. The first type, called Annotation, measures information regarding the annotation definition itself, such as the number of attributes. The second type, Code Element, deals with measuring annotations on specific code elements. And finally, there is the Class type, which measures the definition of annotations regarding a whole class, for example, the total number of annotations in a class.

These metrics will guide this study that aims to understand how annotations are used among projects. It is desired to understand how the metrics distribution will behave, hoping that it will be possible to define some threshold values that might enable the identification of scenarios when annotations are being misused.

6. Exploratory Data Analysis

This section presents an analysis of the metric values extracted from the projects listed in Section 4.2.1. We worked with a hypothesis that the majority of the candidate annotation metrics do not follow a normal distribution, possibly indicating that the average value is not meaningful for its analysis. The average value can be considered the best approach to analyze data when no other information is provided, and thus it is considered that the data being analyzed follows a normal distribution.

For this paper, all metrics values were fully collected from 24947 Java classes from real-world projects, which provides enough data for a more complete assessment. With the metrics values collected, the distribution can be analyzed to conclude further if the average value is indeed meaningful for the candidate metrics. We also believe that the median value is not meaningful, due to annotation candidate code metrics presenting an exponential distribution graph where the majority of the data are focused on small portions of the graph.

Values for all metrics were individually analyzed providing a complete report. Based on that, it was defined the most suitable approach to find thresholds values for each metric. Having thresholds values, it is feasible to detect outliers from these metrics, possibly finding misused annotations on projects. It can also help developers to monitor their source code regarding annotations usage. For instance, a developer might notice that the number of annotations in a class is way beyond a typical value found in most projects. Thresholds values are not absolute truth that developers should blindly follow. However, evaluating the metrics based on threshold values helps to detect outliers, which can reveal a possible design problem. Our analysis considered 2 approaches to determine the thresholds, as discussed in Section 4.

Next subsections focus on the analysis of specific metrics. For each metric, there is a table containing the percentile rank values, ranging from percentile 5 up to 99, for each project. This table also presents the average and standard deviation values obtained from the metrics values per project. We also present a specific threshold table, containing threshold values obtained from Lanza's approach and our approach.

From the percentile rank values (our approach), we provide 3 threshold values: very frequent, frequent, and less frequent. The "very frequent" value is the average obtained from the values in the column "percentile90" for all projects. The "frequent" value refers to the average obtained from percentile 95 column. Lastly, the "less frequent" value comprises the average obtained from percentile 99 column. The way these values were calculated imply the main difference between the original approach proposed by Meirelles (2013) and our approach introduced in this paper.

Finally, we also present the percentile rank charts for some selected projects. To present all projects percentiles rank charts in a single diagram, for each metric, would make the picture unclear and unreadable, since the curves are somewhat close to each other. The result is a lot of overlapping curves. Therefore, we chose to select a single project and display its percentile rank. The criteria used to select a project is the one that represents the largest outlier value for that specific metric. For this reason, each metric may have a different project representing the outlier value. Also, a special attention is being given to the outlier project because it will guide us in answering our fourth research question, which deals with detecting how code annotations can impact code maintenance. A supplemental material is available online containing the remaining percentile ranks for all metrics.

6.1. Attributes in Annotations (AA)

The Attributes in Annotations (AA) metric measures how many attributes are present in a specific annotation declaration. Table 7 shows the AA percentile values for all projects analyzed.

Analyzing Table 7, we notice that below the percentile 90, for most projects, the metric has a value of 0. From the percentile 90, it starts to manifest and may reach the value 4. This result shows that only 10% of the data is really useful for analysis, but the values are still low. So, the average value may bring information to characterize the data. Table 6 shows our threshold values compared to the thresholds obtained by using Lanza's approach.

Table 6
Threshold values for AA metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
AA	90	1.00	1.00	2.00	9.00	−0.21	0.21	0.63

Table 7
Percentiles from AA metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	4.00	0.31	0.53
ANTLR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	4.00	0.03	0.20
Apache_Derby	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.04	0.20
Apache_Isis	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	8.00	0.15	0.45
Apache_Tapestry	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	4.00	0.28	0.53
Apache_Tomcat	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	9.00	0.04	0.23
ArgoUML	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.03	0.18
Checkstyle	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	0.27	0.45
Dependometer	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	3.00	0.42	0.65
ElasticSearch	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	5.00	0.06	0.31
Hibernate_commons	0.00	0.00	0.00	0.00	1.00	1.00	1.20	2.00	2.00	0.49	0.61
Hibernate_core	0.00	0.00	0.00	0.00	1.00	1.00	2.00	3.00	6.00	0.35	0.65
JChemPaint	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	0.06	0.23
Jenkins	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	4.00	0.21	0.48
JGit	0.00	0.00	0.00	0.00	0.00	0.00	1.00	3.00	5.00	0.16	0.59
JMock	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	0.25	0.43
Junit	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	0.22	0.42
Lombok	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	0.22	0.42
Megamek	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	0.08	0.27
Metric_Miner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.06
OpenCMS	0.00	0.00	0.00	0.00	0.00	1.00	1.00	3.00	4.00	0.20	0.51
Oval	0.00	0.00	0.00	1.00	1.00	2.00	3.00	4.00	4.00	0.84	0.97
Spring	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	9.00	0.18	0.44
VoltDB	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	2.00	0.16	0.40
VRaptor	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	4.00	0.10	0.31

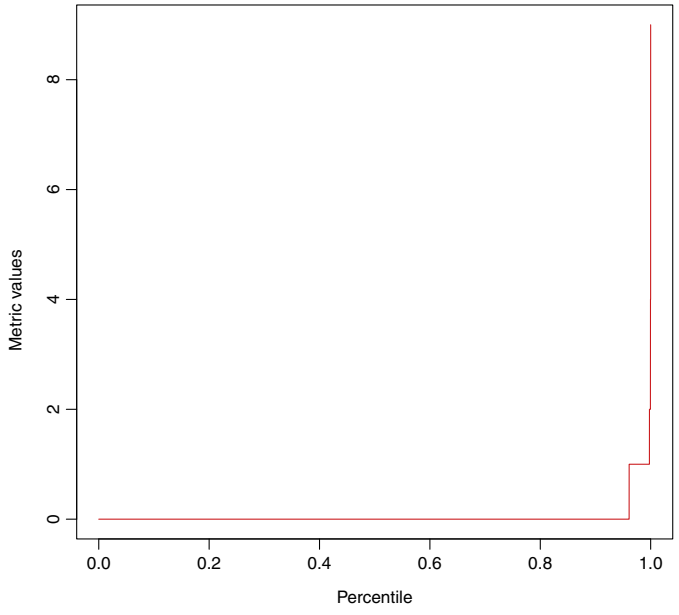

Fig. 2. Percentile of AA metric: Apache Tomcat.

Our analysis shows that having 1 attribute declared in an annotation can be considered a reliable average value, and not 0 (as obtained by Lanza's approach). Annotations containing more than 2 attributes are "less frequent", and values greater than this might reveal uncommon scenarios.

Fig. 2 presents an example of an outlier and illustrates an AA metric percentile rank chart. From our sample, the greatest value found was 9 on Apache Tomcat project. The ParamServlet class has the annotation @WebServlet with all possible attributes configured. The annotation is even more complicated because 1 of its attributes contains 2 other nested annotations.

6.2. LOC in Annotation Declaration (LOCAD)

The LOCAD metric measures how many lines are used to fully declare the annotation. Annotations that take too many lines to be declared can compromise its readability, maintenance, and evolution. Usually, annotations with high LOCAD also have a high number of attributes (measured by AA). However, it is not a one-to-one relation since a single line of annotation can have multiple attributes. On the other hand, 1 attribute that receives a long string or a list of values might be defined using several lines of code.

Table 8 shows that even though LOCAD gains some values higher than 1 in the percentile 99, the majority of the values are still 1, so the average value is meaningful for this metric. Table 9 presents the comparison between thresholds values calculated using percentiles and the values based on average and standard deviation values. It is clear they are not very different from each other. We conclude that values greater than 2 are a "less frequent" value, as opposed to Lanza-High value of 1.

Despite this metric has low values in average, it is possible to find some cases of annotations with a high number of lines of code. As an example, the highest number found for the LOCAD metric was 58 in a class from Hibernate Core as presented in Fig. 3. This annotation is used for query definition, and have both a high number of nested annotations and attribute values containing large strings that spread through several lines.

6.3. Annotation Nesting Level (ANL)

Annotation Nesting Level measures how deep an annotation is nested. As expected, nesting levels are usually very low, since there are a few annotation types that use other annotations as attributes.

Table 8
Percentiles from LOCAD metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	6.00	1.01	0.20
ANTLR	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	5.00	1.00	0.07
Apache_Derby	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Apache_Isis	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	13.00	1.08	0.52
Apache_Tapestry	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.00	9.00	1.02	0.21
Apache_Tomcat	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	8.00	1.00	0.12
ArgoUML	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Checkstyle	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Dependometer	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	4.00	1.08	0.50
ElasticSearch	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	44.00	1.00	0.34
Hibernate_commons	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Hibernate_core	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	58.00	1.08	0.91
JChemPaint	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	29.00	1.20	2.36
Jenkins	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	1.00	0.06
JGit	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	6.00	1.00	0.08
JMock	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Junit	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.00	1.00	0.04
Lombok	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.00	1.00	0.04
Megamek	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Metric_Miner	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
OpenCMS	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	1.00	0.07
Oval	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.05
Spring	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	8.00	1.01	0.14
VoltDB	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	7.00	1.05	0.37
VRaptor	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.03

Table 9
Threshold values for LOCAD metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
LOCAD	90	1.00	1.00	2.00	58.00	0.78	1.02	1.27

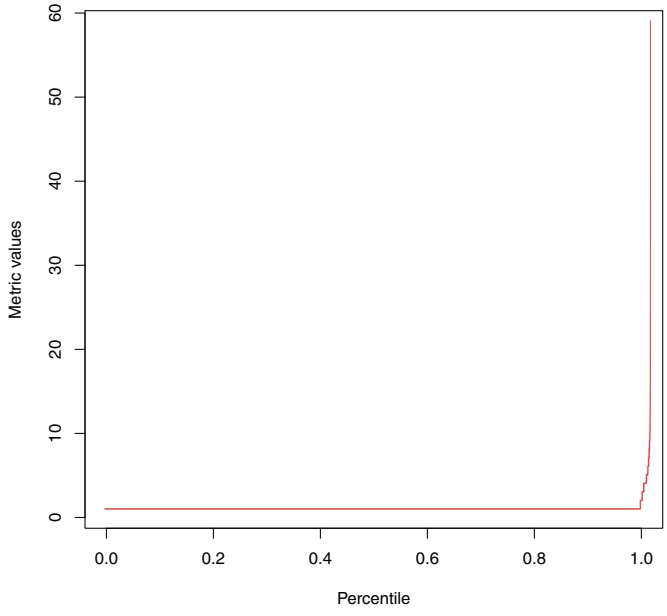

Fig. 3. Percentile of LOCAD metric: Hibernate Core.

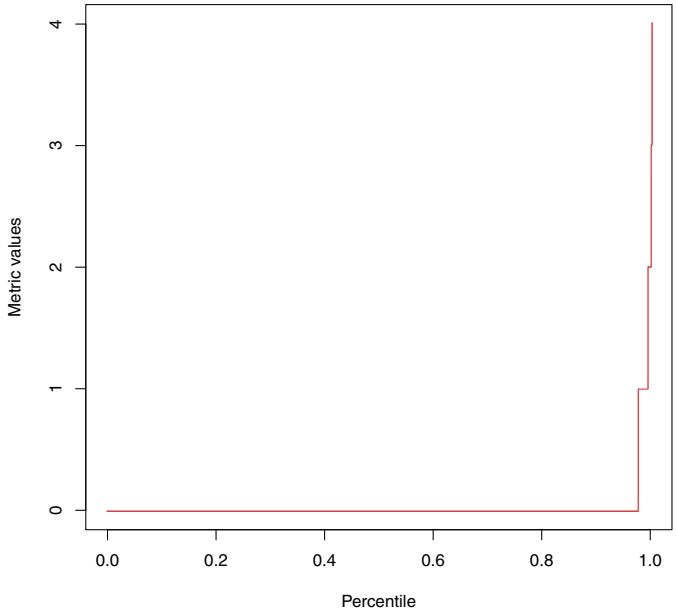

Fig. 4. Percentile of ANL metric: Hibernate core.

The values only slightly increase from the percentile 99, and the majority of the values are zero. Hence, the average value is a good approximation when comparing to our approach based on the percentile rank.

Table 10 presents the percentile values. Table 11 shows the thresholds values. As showed in Fig. 4, the outlier value found was 4 for Hibernate Core which can be considered an extremely high value, since it is common for the ANL to be 0. Using the percentile rank, we obtained the value 0.08 to be a "less frequent" boundary, considering that the Lanza's approach was 0.04.

6.4. Annotations in Element Declaration (AED)

Source code elements such as methods, attributes, and classes can be annotated. AED measures how many annotations are declared for a specific element, counting also nested annotations. A high number of annotations in the same element might reveal a

Table 10
Percentiles from ANL metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	2.00	0.02	0.16
ANTLR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Apache_Derby	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Apache_Isis	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.03
Apache_Tapestry	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.02
Apache_Tomcat	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.00	0.00	0.05
ArgoUML	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Checkstyle	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Dependometer	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ElasticSearch	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Hibernate_commons	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Hibernate_core	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	4.00	0.03	0.23
JChemPaint	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Jenkins	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
JGit	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
JMock	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Junit	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.01	0.08
Lombok	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.01	0.08
Megamek	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Metric_Miner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
OpenCMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.03
Oval	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.01	0.09
Spring	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.05
VoltDB	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
VRaptor	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 11
Threshold values for ANL metric.

Metric	Percentile reference	Very frequent	Frequent	Less frequent	Outlier value	Lanza-Low	Lanza-Medium	Lanza-High
ANL	90	0.00	0.00	0.08	4.00	-0.03	0.00	0.04

Table 12
Percentiles from AED metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	0.00	0.00	0.00	0.00	1.00	2.00	2.00	4.00	8.00	0.59	0.85
ANTLR	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	3.00	0.54	0.50
Apache_Derby	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	0.12	0.33
Apache_Isis	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	7.00	0.45	0.65
Apache_Tapestry	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	5.00	0.44	0.65
Apache_Tomcat	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	5.00	0.38	0.50
ArgoUML	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	0.18	0.39
Checkstyle	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	0.12	0.32
Dependometer	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	0.19	0.42
ElasticSearch	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	18.00	0.27	0.46
Hibernate_commons	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	2.00	0.18	0.45
Hibernate_core	0.00	0.00	0.00	0.00	1.00	1.00	2.00	3.00	27.00	0.54	0.79
JChemPaint	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	0.24	0.43
Jenkins	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	4.00	0.43	0.61
JGit	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	3.00	0.38	0.50
JMock	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	0.21	0.43
Junit	0.00	0.00	0.00	1.00	1.00	1.00	1.00	3.00	6.00	0.58	0.69
Lombok	0.00	0.00	0.00	1.00	1.00	1.00	1.00	3.00	6.00	0.58	0.69
Megamek	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	0.15	0.36
Metric_Miner	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	2.00	0.41	0.50
OpenCMS	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	0.17	0.41
Oval	0.00	0.00	0.00	0.00	0.00	1.00	2.75	4.00	4.00	0.39	0.91
Spring	0.00	0.00	0.00	1.00	1.00	1.00	2.00	3.00	7.00	0.59	0.68
VoltDB	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	24.00	0.30	0.51
VRaptor	0.00	0.00	0.00	1.00	1.00	1.00	1.00	3.00	5.00	0.59	0.64

code that is hard to maintain. An element that has an excessive amount of annotations might prevent the code to evolve without breaking other parts.

Table 12 shows the values of percentiles for AED. Before percentile 90, several projects present the value 0, meaning that there is no annotation declared on these elements. As annotations are not mandatory, this value is perfectly acceptable.

From percentile 90, the value stabilizes at 1 until percentile 99, when the value reaches 2. We can then create a region with the number 1 being the delimiter of “Very Frequent” values. Between 1 and 2 we have a “Frequent” value, and above 2 the value can be considered “Less Frequent”.

Using Lanza’s approach, we obtain an average value of 0 and high margin of 1. This can be seen as an AED value greater than 1 being considered high. In our analysis, we conclude that this num-

Table 13
Threshold values for AED metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
AED	90	1.00	1.00	2.00	27.00	−0.19	0.36	0.91

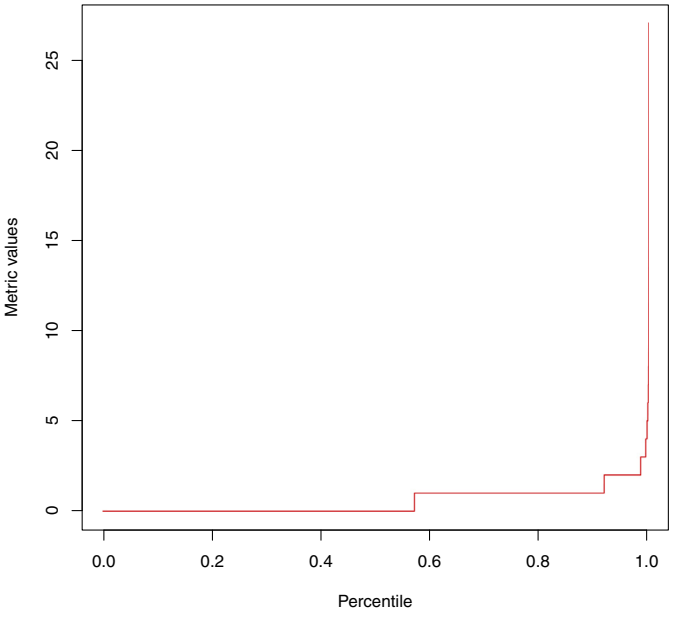

Fig. 5. Percentile of AED metric: Hibernate Core.

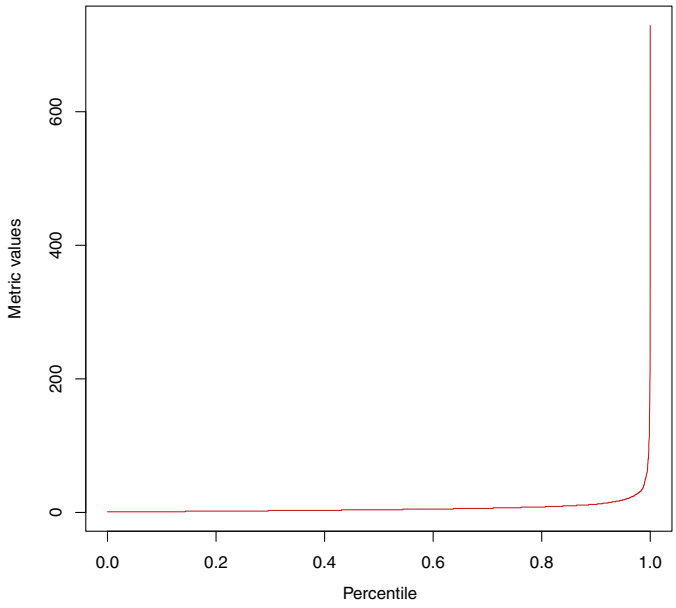

Fig. 6. Percentile of AC metric: Hibernate Core.

ber can be pushed up to 2. The comparison between the 2 kinds of thresholds is presented on Table 13.

Although the statistical threshold shows that values of AED higher than 2, which can be considered high, it is common to find elements with 3 or 4 annotations that are not overloaded with metadata. Therefore, this metric should be interpreted combined with others, such as AA and LOCAD, to really find annotation declaration that can cause maintenance problems.

Fig. 5 presents the distribution graph from Hibernate Core that have the highest value for LOCAD. The highest number found for AED was 27 in the same class. In this case, most of the annotations are nested inside a single one.

6.5. Annotations in Class (AC)

AC measures how many annotations a class contains. Since annotations can be considered an optional feature, not all classes contain it. However, we notice that some classes may contain a high number of annotations, reaching more than 500. Table 15 contains the thresholds values determined for the AC metrics.

As shown on Table 14, the metric values start to become meaningful after percentile 90, showing that only 10% of the analyzed classes has relevant information regarding its number of annotations. The obtained Lanza-Low value is negative, which indicates a high standard deviation value. The AC distribution graph has an abrupt growth after percentile 90, which concludes that the average value of this metric is not meaningful.

As an example, projects like MetricMiner and Hibernate Core have classes with really high values, but most of the classes have low values. Thus, the average of 29 for Metric Miner is not representing the data, since more than 90% of the classes have values lower than 12.

Using our percentile analysis for the AC metric, considering all 24947 Java classes, the value chosen as a starting point was percentile 90. The “very frequent” value obtained was 11, “frequent” was 20, and 62 as a “less frequent” value. For annotations, it cannot be stated that a class has a very low number of annotations since it is an optional feature. However, it makes sense to assume that a class has a high number of annotations. Based on this analysis, the frequent region was determined to be between 11 and 20 annotations per class, while values greater than 62 denote outliers.

The higher number found for AC was the CoreHibernateLogger class from Hibernate with 729 annotations, as presented in Fig. 6. These annotations configure logging information for all interface methods. For each method, it is configured attributes for the logging level and its respective message, which is different for the majority of the annotations definition. In this case, the high number of methods is probably more problematic than their annotations.

6.6. Unique Annotations in Class (UAC)

Unlike the AC metric, which counts equivalent annotations, UAC metric is focused on the number of distinct annotations in a class. An annotation is considered equivalent to another when it has the same type and the same attribute values. Subtracting UAC from AC it yields the number of repeated annotations. By definition, the UAC value is never greater than AC value for a class.

Table 17 presents the obtained threshold values and Table 16 shows the percentile values for all projects. In our analysis, the percentile 75 could have been used as reference point to determine the threshold values. However, this would lead to a smaller “Very Frequent” region, and since the goal is to make the threshold values as flexible as possible, the reference point was pushed to the percentile 90, to allow a broader “Very Frequent” region. Moreover,

Table 14
Percentiles from AC metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	1.00	1.00	3.00	5.00	11.00	22.10	34.05	62.41	128.00	9.67	13.47
ANTLR	1.00	1.00	2.00	4.00	12.00	26.00	33.00	97.68	98.00	10.15	15.25
Apache_Derby	1.00	1.00	1.00	2.00	5.00	10.00	12.50	28.90	53.00	4.27	5.91
Apache_Isis	1.00	1.00	1.00	2.00	5.00	9.00	15.00	33.00	151.00	4.70	7.82
Apache_Tapestry	1.00	1.00	1.00	2.00	4.00	9.00	15.00	35.26	161.00	4.43	7.87
Apache_Tomcat	1.00	1.00	1.00	3.00	8.00	20.00	34.00	82.70	265.00	8.72	19.45
ArgoUML	1.00	1.00	1.00	2.00	4.00	7.00	13.00	19.00	51.00	3.52	4.33
Checkstyle	1.00	1.00	1.00	1.00	1.00	2.00	2.45	3.00	3.00	1.21	0.54
Dependometer	1.00	1.00	1.00	1.00	2.00	2.80	3.00	7.74	9.00	1.65	1.48
ElasticSearch	1.00	1.00	2.00	3.00	5.00	10.00	14.00	30.00	269.00	4.70	9.00
Hibernate_commons	1.00	1.00	1.00	1.50	2.00	3.20	5.00	5.00	5.00	1.85	1.23
Hibernate_core	1.00	1.00	2.00	4.00	7.00	12.00	19.00	46.60	729.00	6.67	14.37
JChemPaint	1.00	1.00	1.00	1.00	2.00	4.40	9.80	19.72	27.00	2.47	4.25
Jenkins	1.00	1.00	2.00	4.00	7.00	14.00	22.00	53.52	131.00	6.85	10.21
JGit	1.00	1.00	2.00	4.00	8.00	16.00	21.00	53.58	169.00	7.28	12.23
JMock	1.00	1.00	1.00	2.00	3.00	4.20	7.20	14.00	14.00	2.52	2.60
Junit	1.00	1.00	2.00	3.00	5.00	12.00	21.00	50.68	228.00	6.11	15.47
Lombok	1.00	1.00	2.00	3.00	5.00	12.00	21.00	50.68	228.00	6.11	15.47
Megamek	1.00	1.00	1.00	1.00	3.00	8.00	14.25	64.20	121.00	4.39	10.41
Metric_Miner	1.00	1.00	1.00	1.00	3.00	11.80	85.60	596.64	723.00	28.94	119.68
OpenCMS	1.00	1.00	1.00	2.00	4.00	8.00	11.00	27.88	63.00	3.82	5.37
Oval	1.00	1.00	2.00	3.00	8.00	8.00	8.00	11.02	22.00	4.44	3.12
Spring	1.00	1.00	2.00	4.00	10.00	17.00	24.75	47.00	281.00	7.66	11.19
VoltDB	1.00	1.00	1.00	3.00	8.00	18.00	27.00	59.92	316.00	7.90	18.78
VRaptor	1.00	2.00	3.00	4.00	8.00	13.00	17.80	40.24	76.00	6.46	7.80

Table 15
Threshold values for AC metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
AC	90	11.00	20.00	62.00	729.00	−7.24	6.26	19.76

Table 16
Percentiles from UAC metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	1.00	1.00	2.00	3.00	5.00	7.00	9.05	29.87	40.00	4.22	4.70
ANTLR	1.00	1.00	1.00	1.00	2.00	2.00	2.00	4.00	14.00	1.38	0.93
Apache_Derby	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.00	3.00	1.05	0.25
Apache_Isis	1.00	1.00	1.00	1.00	2.00	4.00	5.00	9.00	29.00	1.92	1.95
Apache_Tapestry	1.00	1.00	1.00	1.00	3.00	4.00	6.00	11.26	36.00	2.19	2.35
Apache_Tomcat	1.00	1.00	1.00	1.00	1.00	2.00	3.00	5.00	15.00	1.40	0.97
ArgoUML	1.00	1.00	1.00	1.00	1.00	2.00	2.00	3.00	4.00	1.14	0.44
Checkstyle	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.49	2.00	1.02	0.14
Dependometer	1.00	1.00	1.00	1.00	1.00	2.00	2.90	4.74	6.00	1.30	0.89
ElasticSearch	1.00	1.00	1.00	1.00	2.00	2.00	3.00	5.00	13.00	1.36	0.78
Hibernate_commons	1.00	1.00	1.00	1.00	1.25	2.00	2.15	4.43	5.00	1.40	0.94
Hibernate_core	1.00	1.00	1.00	2.00	4.00	7.00	10.00	16.00	375.00	3.41	5.87
JChemPaint	1.00	1.00	1.00	1.00	1.00	1.00	1.20	2.00	2.00	1.05	0.23
Jenkins	1.00	1.00	1.00	3.00	4.00	6.00	7.60	14.52	27.00	3.15	2.55
JGit	1.00	1.00	1.00	1.00	2.00	3.00	4.90	11.58	22.00	1.88	2.04
JMock	1.00	1.00	1.00	1.00	2.00	3.00	3.10	5.00	5.00	1.61	0.97
Junit	1.00	1.00	1.00	2.00	2.00	4.00	5.00	12.28	36.00	2.24	2.58
Lombok	1.00	1.00	1.00	2.00	2.00	4.00	5.00	12.28	36.00	2.24	2.58
Megamek	1.00	1.00	1.00	1.00	1.00	1.00	2.00	5.00	80.00	1.31	3.59
Metric_Miner	1.00	1.00	1.00	1.00	2.00	2.80	3.00	3.48	4.00	1.43	0.75
OpenCMS	1.00	1.00	1.00	1.00	1.00	2.00	3.00	12.00	59.00	1.58	3.22
Oval	1.00	1.00	1.00	3.00	5.00	5.70	6.00	9.17	13.00	3.13	2.17
Spring	1.00	1.00	1.00	2.00	4.00	7.00	8.00	15.15	102.00	3.06	3.95
VoltDB	1.00	1.00	1.00	1.00	2.00	4.00	7.00	14.00	25.00	2.08	2.51
VRaptor	1.00	1.00	2.00	3.00	4.00	5.00	6.00	8.56	29.00	3.08	2.00

Table 17
Threshold values for UAC metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
UAC	90	3.00	4.00	9.00	375.00	0.01	1.98	3.95

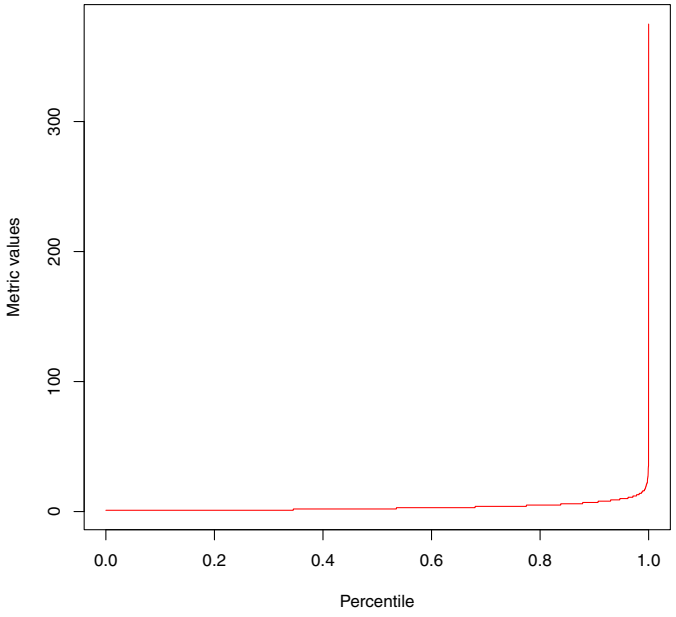

Fig. 7. Percentile of UAC metric: Hibernate Core.

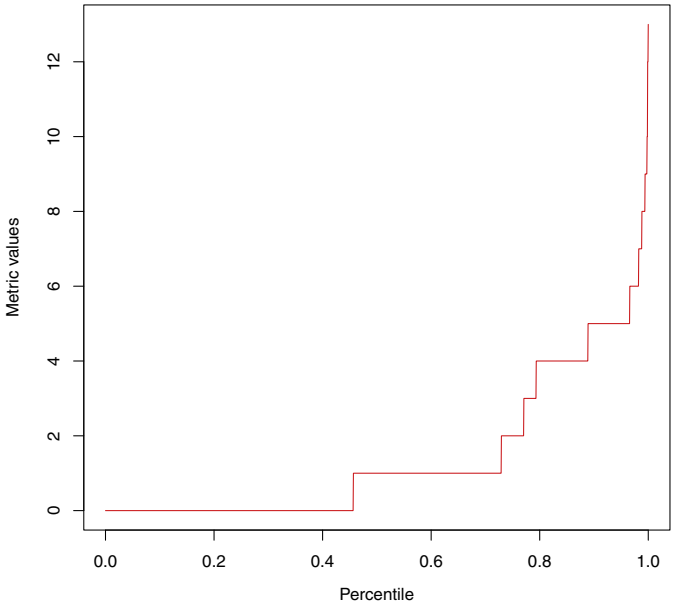

Fig. 8. Percentile of ASC metric: Spring.

the UAC metric never assumes values below 1, since our analyzed classes contain at least 1 annotation.

Therefore, we obtained 3 as the "Very Frequent" value, 4 as "Frequent" and 9 as "Less Frequent". In Lanza's approach, the "Lanza-Medium" is 2, the "Lanza-Low" is 0 and "Lanza-High" is 4. Hence, our analysis covers a wider range of values, and the obtained results share some proportional growth with the AC percentile rank. Accordingly, the obtained values are coherent.

Some Lanza-low values are negative for the same reasons explained in the AC section. Since the UAC value is never greater than the AC value, as expected, the standard deviation is not as high as for the AC. Therefore, the average value can still be useful. However, using our analysis based on percentiles, a more precise threshold value can be obtained.

The highest value found for UAC was 729 in the same class on Hibernate Core that has the highest value of AC (CoreHibernateLogger), as seen in Fig. 7. Although it has 729 annotations, only 375 are unique. This class has several annotations of the type @Message, but their attribute values are different.

6.7. Annotations Schemas in Class (ASC)

When software applications use annotations in their elements, they belong to a set that represents metadata for a given domain, which is called schema. For instance, when an application uses a metadata-based framework, it usually uses a group of annotations from this framework to represent the necessary metadata. An application may use as many schemas as needed. The ASC metric represents the number of schemas, which can be identified by the annotation package.

Table 18 shows the percentile values for this metric and we observe that before percentile 90 the majority of the projects have at maximum 1 ASC per class. Based on that, we conclude that a very frequent value for ASC is 1, which means that when annotations are used in a class, most of the time all of them belongs to a single domain. From percentile 90 to 99, we have 2 as a "frequent" value for the ASC metric. Beyond that, the value can be considered high. From Lanza's point of view, the higher value is 1, while the average is 0. Once again, our approach has proved to be flexible, yielding a wider range of values for the thresholds, which better accommodates real-world projects.

Fig. 8 presents the Spring project distribution graphs as an example of an outlier. This project has a class, `StompIntegrationTest`, with the value of 13 for ASC. Despite the fact that most of the classes have low values, we found classes coupled with several annotations schemas. Table 19 presents the Thresholds value obtained for ASC.

6.8. Annotation Metrics Results Summary

This section presented the analysis of the metrics values collected from 24947 Java classes. We proposed a new approach using percentile rank, to analyze the distribution of those values. We believed that the distribution would follow an exponential graph. Therefore, the average value would not represent the data.

Of the 7 analyzed metrics, AA, LOCAD, and ANL have low values even after percentile 90, except the outliers. Since the values are not very sparse, the average value can be used as a good first guess to represent the data. The other 4 metrics using the percentile analysis yields better results in representing the data. When the distribution graph has an abrupt growth, such as the AC metric, the Lanza-Low margin yields in a negative result because the standard deviation is really high. In practical usage, this negative number should be considered 0, but it shows that this calculation is not indicated for every metrics, and its distribution should be analyzed first.

Our percentile analysis considered a more realistic approach and yielded in better threshold values. For instance, the metric AED has a Lanza-High value of 1, while our analysis concluded the number to be 2. Having 2 annotations declared on an element can be considered a "frequent" value. Above this, we consider it a high value or a "less frequent" one. Percentile rank analysis can be a more flexible way of obtaining the thresholds.

To conclude our analysis, the research questions on Section 4 are fully answered in the next paragraphs. Each of them represents a goal that our research aims to reach. Through these questions, we formulated what steps were necessary to take, and by the end, we managed to obtain the needed results and observations.

#RQ1 - What measurements could be performed in the source code to assess the characteristics of its code annotations usage?

Table 18
Percentiles from ASC metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	0.00	0.00	1.00	2.00	3.00	5.00	5.00	5.41	6.00	2.29	1.62
ANTLR	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	0.38	0.49
Apache_Derby	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Apache_Isis	0.00	0.00	0.00	0.00	1.00	1.00	2.00	3.00	4.00	0.49	0.73
Apache_Tapestry	0.00	0.00	0.00	1.00	1.00	2.00	2.00	3.00	5.00	0.84	0.77
Apache_Tomcat	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	3.00	0.32	0.52
ArgoUML	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.04	0.20
Checkstyle	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Dependometer	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	0.19	0.39
ElasticSearch	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	3.00	0.26	0.48
Hibernate_commons	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	0.15	0.37
Hibernate_core	0.00	0.00	0.00	1.00	2.00	2.00	3.00	3.00	5.00	0.95	0.93
JChemPaint	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.44	2.00	0.58	0.53
Jenkins	0.00	0.00	1.00	1.00	2.00	3.00	4.00	5.52	10.00	1.51	1.30
JGit	0.00	0.00	0.00	1.00	1.00	1.00	1.00	3.00	3.00	0.57	0.60
JMock	0.00	0.00	0.00	0.00	1.00	2.00	2.00	2.00	2.00	0.51	0.75
Junit	0.00	0.00	1.00	1.00	1.00	2.00	2.00	2.00	3.00	1.00	0.62
Lombok	0.00	0.00	1.00	1.00	1.00	2.00	2.00	2.00	3.00	1.00	0.62
Megamek	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	2.00	0.04	0.22
Metric_Miner	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	0.30	0.46
OpenCMS	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	0.10	0.31
Oval	0.00	0.00	0.00	1.00	2.00	2.00	2.00	3.00	4.00	1.00	0.91
Spring	0.00	0.00	0.00	1.00	2.00	5.00	5.00	8.00	13.00	1.44	1.98
VoltDB	0.00	0.00	0.00	0.00	1.00	1.00	2.00	2.92	4.00	0.52	0.70
VRaptor	1.00	1.00	1.00	2.00	2.00	3.00	4.00	4.00	5.00	1.74	0.94

Table 19
Threshold values for ASC metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
ASC	90	1.50	1.80	2.40	13.00	−0.01	0.65	1.30

To assess the characteristics of annotated code elements, we proposed a suite of candidate metrics. A total of 7 metrics was proposed, each of them measuring a particular characteristic of annotations in the source code. We extracted information such as the number, the attributes, and the nesting level of annotations. With that suite of metrics, we were able to gather all needed data from 24947 Java classes to perform our empirical analysis. The extracted values, as well as the qualitative and quantitative analysis, demonstrated the usefulness of this suite. We consider the proposed metrics to be an important contribution to the Software Engineering community.

#RQ2 - For each metric, is it possible to define reference thresholds that can be used to classify its values?

We were expecting that the average value would not be meaningful since it is known that object-oriented metrics follow an exponential distribution graph. It is no different for annotation metrics; we showed that all 7 metrics behave exponentially. The values extracted for the metrics AA, LOCAD, and ANL are low, therefore, the average could be considered a medium point. However, it fails to find an appropriate high margin threshold, while the percentile rank analysis yields a more realistic value. For the remaining 4 metrics, the average value does not bring much information due to the abrupt growth after percentile 90. Overall, using the proposed percentile rank analysis yields better results due to its flexible nature. Therefore, by knowing the distributions behavior of the candidate metrics we are able to propose a more realistic threshold value for these metrics.

Threshold values determine boundaries to classify data that falls within a specific region. If thresholds values are obtained, it can aid developers to maintain control and quality of their source code. For our proposed candidate metrics, the obtained threshold values are intended to be a guide for developers. However, every project has its own design criteria, and therefore it is up to the developer to check whether the threshold values makes sense for that specific project.

Table 20
Percentile Rank Threshold.

Metric	Very Frequent	Frequent	Less frequent
AA	1.00	1.00	2.00
LOCAD	1.00	1.00	2.00
ANL	0.00	0.00	0.08
AED	1.00	1.00	2.00
AC	11.00	20.00	62.00
UAC	3.00	4.00	9.00
ASC	1.50	1.80	2.40

Table 21
Lanza's threshold values: approximated and without negative numbers.

Metric	Lanza-Low	Lanza-Medium	Lanza-High
AA	0.00	0.00	1.00
LOCAD	1.00	1.00	1.00
ANL	0.00	0.00	0.04
AED	0.00	0.00	1.00
AC	0.00	6.00	20.0
UAC	0.00	2.00	4.00
ASC	0.00	0.60	1.30

From the metrics values extracted from 25 real-world projects and with the empirical percentile rank analysis, we were able to reach some thresholds values presented on Table 20. Table 21 displays threshold values collected using Lanza's approach. These values were approximated to reflect feasible numbers, as no Java class can contain 1.2 annotations. These tables summarize the individual thresholds tables obtained for each metric, using both our and Lanza's

approach. As discussed in the previous subsections, we concluded that the threshold values presented in Table 20 are more realistic and might better represent code annotations on real-world projects.

#RQ3 - What is the common profile of a class that uses code annotations in Java? How large are the metrics outliers found? Since annotations are an optional information added on classes, in general, annotation metrics present low values. Usually, an annotation has 0 or 1 attribute, it is defined in 1 line of code. Also, it is not nested on other annotations and is the only annotation in the element where it is defined. A class with around 20 annotations defined can be considered normal. However, usually, there is a high number of repetitive definitions, since the common number of unique annotations is 4. Considering the number of different annotations schemas present, a class usually has configurations of 1 or 2. Despite most of the cases have lower values, in our analysis, we found some outliers with really high values for the candidate metrics. We found classes with more than 700 annotations and with 13 different annotation schemas. We also found a single annotation containing 9 attributes and another defined through 58 lines of code. There was an element with 27 annotations, and the nesting level reached the value of 4 (i.e. an annotation in the fourth level of nesting). Having found very large outlier, opens room for additional investigation on future works.

#RQ4 - May the usage of annotations create problems that can compromise code maintenance?

Code elements that are outliers for a code metric are usually related to problems in code maintenance. Some examples are documented code smells, such as Brain Method and Good Class (Fowler, 1999). The detection strategies for these code smells has rules that consider threshold values for a group of metrics (Lanza and Marinescu, 2006).

Analyzing the impact of annotations in code maintenance, the contribution of this study is to reveal the existence of these outliers. These extreme cases illustrate well that abuses in the usage of annotations can happen in the development of real-world projects. For instance, an annotation with 58 lines of code, classes with more than 700 annotations, and a class with 13 different annotation schemas are a clear abuse of annotation usage. As a large class and a large method can compromise code maintenance, the same applies to an annotated element or class with a high number of configurations. Based on that, this finding is an evidence that it is important to monitor and evaluate the usage of annotations to avoid such situations.

Each metric captures a particular characteristic from a class and, despite we should pay attention to outliers, it is important to combine the metrics for a more precise analysis to verify what is the problem on how annotations are used. For instance, the Java8Parser class from the MetricMiner project has more than 700 annotations, but the UAC value is only 3, and the number of annotations on each element is only 1 or 2. Observing the class, we notice that the problem is actually a high number of methods with few annotations on them. As a consequence, due to the repetition of annotations, a general change in the annotations on these methods would be hard to perform. However, the classification and detection of design problems in annotations is out of the scope of this work.

Even though the presence of these outliers reveal possible maintenance problems, we consider this result a first step in this direction. A further study should investigate the relation of issues and bugs with code changes in annotations. In this sense, the proposed metrics and their respective thresholds can help to characterize annotations in frequently changed code.

7. Conclusion

Code annotations is a Java language feature used by several frameworks and tools. Recently, its usage has become very popular, especially in web and enterprise applications. Despite this, there are only superficial studies in the literature about design problems and the assessment of annotated code.

In this paper, we conducted our research according to four research questions. Based on them, we proposed five steps to reach our main goal: *assess the usage of annotated code from a software developer point of view*. Initially, a suite of candidate metrics was proposed, based on a GQM approach. These metrics provide values that measure how annotations are present in the source code, regarding their declaration, attributes, and overall presence in a Java class. They are a new group of metrics used exclusively for annotations in the source code.

The analyzed data was composed of 24947 Java classes extracted from 25 real-world projects. These projects contain a wide range of annotated classes to provide diversity in our analysis. We generated a percentile rank for all metric values to understand their behavior and to identify possible threshold values. It was pointed out that all metrics have an exponential distribution, which indicates that the average and standard deviation might not be good representation of the data. Some metrics, such as ANL, have small overall values and the average value could be considered a reference point, but metrics such as AC have an abrupt growth at percentile 90. Thus, the average value is not a good middle point.

To determine threshold values, we used our proposed percentile rank analysis based on Meirelles' findings (Meirelles, 2013) and compared it to Lanza's approach (Lanza and Marinescu, 2006), which defines three threshold values: low, medium, and high. We also defined three threshold values: very frequent, frequent, and less frequent. Lanza's thresholds are obtained through the average value, which directly yields the medium point. Our "very frequent" was obtained by analyzing the percentile rank, and for some metrics, the medium point and "very frequent" values are close to each other. However, when it comes to determining the "high value" (Lanza's) and "less frequent" value (our approach) the percentile rank analysis provides a more realistic value, which usually is greater than Lanza's "high value". That is to say, the percentile rank analysis considers a wider range of values, and they can still be considered common values. These values might help to indicate a potential misleading in annotations usage.

Our analysis showed that most of the classes have low values for all of the metrics. However, we have found some outliers with really high values for some candidate metrics. These extreme cases reinforce the need to evaluate and further improve the techniques to study code annotations. Prior to the proposed metrics suite, there was no suitable way to measure code annotations, therefore we would not have identified these extreme cases. Our suite of metrics captures several aspects of metadata information presented in the source code, which is possible because they measure annotations from three points of view. The first is from a class perspective, where we defined the AC (Annotations in Class), UAC (Unique Annotations in Class) and ASC (Annotations Schema per class). We also analyzed annotations defined on code elements, proposing the AED (Annotations in Element Declaration) metric. And for our third point of view, which is the annotation itself, we presented three metrics: AA (Attributes in Annotation), LOCAD (LOC in Annotation Declaration) and ANL (Annotation Nesting Level). Since the main goal of this paper was to measure code annotations, we defined a novel metrics suite for annotations us-

age. This suite of metrics brings an unpresented practical way to measure code annotations, advancing the state of art regarding the assessment of code annotations.

7.1. Threats to Validity

As a threat to the obtained results, the ASniffer accuracy was verified manually, checking if some obtained metric values were indeed correct since no other similar tool was found to compare the results. Therefore, this can potentially compromise the ASniffer accuracy. The tool will evolve not only to act as a Bad Smell detector but also its usability and results report will be upgraded.

The idea behind the selected 25 real-world Java projects was to combine different domains and different annotation usage. To validate our sample data, we used the concept of diversity and similarity, using dimensions. We defined three dimensions: Type, LOC, and Percentage of Annotated Classes (PAC). We combined these three dimensions in a pairwise fashion, and for each combination we had at least 2 projects and a maximum of 8. So, we guarantee diversity and similarity among our chosen projects. However, the list of projects is not easily reproduced, which can bring a variability on future research, regarding threshold values. One of the reasons for this, is the fact that the defined dimensions were used to validate the chosen projects. Instead, they could have been used to choose the projects from a wider universe.

Finally, the proposed metrics suite was organized following the presented GQM model, and further validated with known mathematical properties from software engineering measurement. All properties are satisfied with the proposed metrics suite, with one exception being the cohesion. However, we argue that from a practical point of view, the metrics suite captures enough information in code annotations.

7.2. Future Works

With the annotation metrics suite and threshold values calculated, future studies can use this work as the foundation to enable assessment and evolution of annotation usage. For instance, frameworks can be improved by searching for bad smells related to its annotations in applications using them. The proposed techniques and the ASniffer tool can be used on real case studies, analyzing the impact of an improved annotation structure on the application maintenance. Since the ASniffer is an open source tool, other developers can improve it in several ways, such as: adding new metrics, implementing bad smell detection, adding visualization techniques, etc.

What the present study found is that classes with very high values for annotation metrics actually exist in real projects. We consider the detection of these outliers as an evidence that a deeper investigation of this issue is important. After this first step, future studies will aim to investigate how much updates actually occur in lines of code with annotations and if commits with these updates are related to bug corrections.

Having an extremely high value for only one metric, or falling in a single “less frequent” region is not enough to conclude whether a class contains or not a bad smell related to annotations usage. With this in mind, this research aims to take a step forward and work on bad smell detection, combining metrics and taking the calculated threshold values into account. With this approach, more knowledge can be extracted from annotations in the source code.

Additionally, other languages also contain metadata features. C# introduced this concept with “attributes”, and Python defined the “decorator”. Despite the fact that they can play the same role of Java annotations, their behavior is different from them. This research can be extended to comprise these languages as well, from the tool perspective to the percentile rank analysis.

Acknowledgment

This work is supported by FAPESP (grant 2014/16236-6).

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.jss.2017.11.024

References

Alba, A., 2011. Code Legibility Analysis by Means of Annotation Patterns. Technical Report. Aeronautical Institute of Technology, Brazil. [in portuguese]
Basili, V.R., 1992. Software Modeling and Measurement: The Goal/Question/Metric Paradigm. Technical Report.
Basili, V.R., Caldiera, G., Rombach, H.D., 1994. The goal question metric approach. Encyclopedia of Software Engineering. Wiley.
Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E., 2006. Understanding the shape of java software. In: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. ACM, New York, NY, USA, pp. 397–412. doi:10.1145/1167473.1167507.
Briand, L.C., Morasca, S., Basili, V.R., 1996. Property-based software engineering measurement. IEEE Trans. Softw. Eng. 22 (1), 68–86. doi:10.1109/32.481535.
Chen, N., 2006. Convention over configuration. URL: http://softwareengineering.vazexqi.com/files/pattern. html
Clauset, A., Shalizi, C.R., Newman, M.E.J., 2007. Power-law distributions in empirical data. SIAM Rev.. arXiv:0706.1062. preprint
Concas, G., Marchesi, M., Pinna, S., Serra, N., 2007. Power-laws in a large object-oriented software system. IEEE Trans. Softw. Eng. 33 (10), 687–708.
Damyanov, I., Holmes, N., 2004. Metadata driven code generation using .net framework. In: Proceedings of the 5th international conference on Computer systems and technologies. ACM, pp. 1–6.
Ernst, M. D., 2008. Type annotations specification (jsr 308).
Fernandes, C., Ribeiro, D., Guerra, E., Nakao, E., 2010. Xml, annotations and database: a comparative study of metadata definition strategies for frameworks. May 19–20, Vila do Conde, 115.
Ferreira, K.A.M., Bigonha, M.A.S., Bigonha, R.S., Mendes, L., Almeida, H.C., 2009. Reference values for object-oriented software metrics. In: XXIII Brazilian Symposium on Software Engineering, 1, pp. 62–72.
Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.
Guerra, E., 2010. A Conceptual Model for Metadata-based Frameworks. Aeronautical Institute of Technology, Brazil Ph.D. thesis.
Guerra, E., Cardoso, M., Silva, J., Fernandes, C., 2010. Idioms for code annotations in the java language. In: Proceedings of the 8th Latin American Conference on Pattern Languages of Programs. ACM, New York, NY, USA, pp. 1–14. doi:10.1145/2581507.2581514.
Guerra, E., Fernandes, C., 2013. A qualitative and quantitative analysis on metadata-based frameworks usage. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 375–390. doi:10.1007/978-3-642-39643-4_28.
Guerra, E., Fernandes, C., Silveira, F.F., 2010. Architectural patterns for metadata-based frameworks usage. In: Proceedings of the 17th Conference on Pattern Languages of Programs. ACM, New York, NY, USA, pp. 1–25. doi:10.1145/2493288.2493292.
Guerra, E.M., Silveira, F.F., Fernandes, C.T., 2009. Questioning traditional metrics for applications which uses metadata-based frameworks. In: Proceedings of the 3rd Workshop on Assessment of Contemporary Modularization Techniques (ACoM'09), October, 26, pp. 35–39.
Guerra, E.M., de Souza, J.T., Fernandes, C.T., 2010. A pattern language for metadata-based frameworks. In: Proceedings of the 16th Conference on Pattern Languages of Programs. ACM, New York, NY, USA, pp. 3:1–3:29. doi:10.1145/1943226.1943230.
Grady, R.B., Caswell, D.L., 1987. Software Metrics: Establishing A Company-Wide Program. Prentice-Hall.
Herraiz, I., GermÃ¡n, D.M., Hassan, A.E., 2011. On the distribution of source code file sizes.. In: Cuaresma, M.J.E., Shishkov, B., Cordeiro, J. (Eds.), ICSOFT (2). SciTePress, pp. 5–14.
Herraiz, I., Rodriguez, D., Harrison, R., 2012. On the statistical distribution of object-oriented system properties. In: Emerging Trends in Software Metrics (WETSoM), 2012 3rd International Workshop on, pp. 56–62. doi:10.1109/WETSoM.2012.6226994.
JSR, 2003. Jsr 153: Enterprise javabeans 2.1. URL: http://www.jcp.org/en/jsr/detail?id=153
JSR, 2004. Jsr 175: A metadata facility for the java programming language, URL: http://www.jcp.org/en/jsr/detail?id=175.
JSR, 2007. Jsr 220: Enterprise javabeans 3.0. URL: http://jcp.org/en/jsr/detail?id=220.
Krahn, H., Rumpe, B., 2006. Towards enabling architectural refactorings through source code annotations. Lect. Notes Inf. P-82, 203–212.

Lanza, M., Marinescu, R., 2006. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer.

Lombok, P., 2016. http://projectlombok.org/.

Louridas, P., Spinellis, D., Vlachos, V., 2008. Power laws in software. ACM Trans. Softw. Eng. Methodol. 18 (1), 2:1–2:26. doi:10.1145/1391984.1391986.

Meffert, K., 2006. Supporting design patterns with annotations. In: Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on, pp. 8pp.–445. doi:10.1109/ECBS.2006.67.

Meirelles, P.R.M., 2013. Monitoring Source Code Metrics in Free Software Projects. Department of Computer Science – Institute of Mathematics and Statistics of University of São Paulo Ph.D. thesis. [in portuguese], URL http://www.teses.usp.br/teses/disponiveis/45/45134/tde-27082013-090242/pt-br.php

Miller, J.S., Ragsdale, S., 2004. The Common Language Infrastructure Annotated Standard. Addison-Wesley Professional.

Nagappan, M., Zimmermann, T., Bird, C., 2013. Diversity in software engineering research. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 466–476. doi:10.1145/2491411.2491415.

Potanin, A., Noble, J., Frean, M., Biddle, R., 2005. Scale-free geometry in oo programs. Commun. ACM 48 (5), 99–103.

Quinonez, J., Tschantz, M., Ernst, M., 2008. Inference of reference immutability. In: ECOOP 2008–Object-Oriented Programming, pp. 616–641. URL: http://www.springerlink.com/index/6M5U5M330T81763T.pdf

Rocha, H., Valente, H., 2011. How annotations are used in java: an empirical study. In: 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 426–431.

Rouvoy, R., Merle, P., 2006. Leveraging component-oriented programming with attribute-oriented programming. In: 11th International ECOOP Workshop on Component-Oriented Programming (WCOP'06), pp. 10–18.

Rouvoy, R., Pessemier, N., Pawlak, R., Merle, P., 2006. Using attribute-oriented programming to leverage fractal-based developments. 5th International ECOOP Workshop on the Fractal Component Model (Fractal '06), Nantes, France.

Ruby, S., Thomas, D., Hansson, D., 2009. Agile Web Development with Rails, Third Edition, 3rd edition Pragmatic Bookshelf.

Schwarz, D., 2004. Peeking inside the box: Attribute-oriented programming with java 1.5, part. URL: http://archive.oreilly.com/pub/a/onjava/2004/06/30/insidebox1.html.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J., 2010. The qualitas corpus: a curated collection of java code for empirical studies. In: Software Engineering Conference (APSEC), 2010 17th Asia Pacific, pp. 336–345. doi:10.1109/APSEC.2010.46.

Van Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M., 2007. On the detection of test smells: A metrics-based approach for general fixture and eager test. Software Engineering, IEEE Trans. 33 (12), 800–817. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4359471.

Wada, H., Suzuki, J., 2005. Modeling turnpike frontend system: A model-driven development framework leveraging uml metamodeling and attribute-oriented programming. Model Driven Engineering Languages and Systems 584–600. URL: http://www.springerlink.com/index/l166363337837142.pdf.

Walls, C., Richards, N., 2003. XDoclet in action. Manning Publications. URL: http://www.lavoisier.fr/livre/notice.asp?id=R62W32A6KOAOWK.

Walnes, J., Abrahamian, A., Cannon-Brookes, M., 2003. Java Open Source programming: with XDoclet, JUnit, WebWork, Hibernate. Java Open Source library, Wiley. URL: http://books.google.com.br/books?id=0N79mLFHtdUC.

Wheeldon, R., Counsell, S., 2003. Power law distributions in class relationships. In: Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '03). Amsterdam, The Netherlands, pp. 45–54.

Yao, Y., Huang, S., Ren, Z.-p., Liu, X.-m., 2009. Scale-free property in large scale object-oriented software and its significance on software engineering. In: Proceedings of the 2009 Second International Conference on Information and Computing Science. IEEE Computer Society, Washington, DC, USA, pp. 401–404.

Phyllipe Lima PhD student in Applied Computing at the National Institute for Space Research (INPE) working on Empirical Software Engineering. Master in Computer Science by the Federal University of Itajubá (UNIFEI) and Telecommunications Engineer by the National Institute of Telecommunications (INATEL). His research interest areas are: Statical analysis, Source code metrics, Mining Software Repositories and Search Based Software Engineering.

Eduardo Guerra Received his M.Sc. and his Ph.D. in Computer Science from the Technological Institute of Aeronautics (ITA). He is currently an Associate Researcher at the National Institute of Space Research (INPE) in Brazil, doing research on test automation, software design and architecture, experimental Software Engineering, agile methods, mining software repository and framework development. He is currently in the board of the Hillside Group and was the program chair of PLoP 2012.

Paulo Meirelles Received his M.Sc. in Computer Science from the Federal University of Rio Grande do Sul and his Ph.D in Computer Science from the University of São Paulo (USP). He was a visiting researcher at the Southern Illinois University Carbondale (SIUC). Paulo is currently a full-time Professor at University of Brasilia (UnB). He is also posdoctorate researcher from the Institute of Mathematics and Statistics (IME) at the University of São Paulo (USP), working on behalf of the Free/Libre/Open Source Software Competence Center (CCSL). His research interest areas are: Free/Libre/Open Source Software development, Agile methods, Statical analysis, Source code metrics, Mining software repositories, and Healthcare software development.

Lucas Kanashiro Computer Science Master student from the Institute of Mathematics and Statistics (IME) at the University of São Paulo (USP), working on behalf of the Free/Libre/Open Source Software Competence Center (CCSL). He received his Bachelor's in 2015 from the University of Brasília (UnB) where he majored in Software Engineering. His research interest area are:Free/Libre/Open Source Software development, Source code metrics, Smart Cities, Distributed systems. Moreover, he is a Debian developer and a Free Software enthusiast.

Hélio Silva Software developer specialized on Object-Oriented Design and Enterprise application architectures. Has experience on financial market applying Domain-Driven Design to connect business rules and software creating an Ubiquitous Language throughout the company areas. Speaker in major conferences in Brazil, including The Developers Conference (TDC) and Ruby Conference Brazil in São Paulo. Also, writes on his blog (blog.hlegi.us) about software in general but mostly on how to spread Test-Driven Development practice on agile teams.

Fábio Silveira Received his M.Sc. from the Federal University of Rio Grande do Sul in 2001 and his Ph.D. in Computer Science from the Technological Institute of Aeronautics (ITA) in 2007. He was a visiting researcher at the Technischen Universität Berlin from June to August 2009. He is currently an Associate Professor at the Science and Technology Institute of the Federal University of São Paulo (UNIFESP), doing research on object-oriented and aspect-oriented software testing, experimental Software Engineering, agile methods, and metadata.