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/© 2017 Elsevier Inc. All rights reserved.

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 meaningful information and not just numerical values.