National Institute for Space Research - INPE
Graduate Program on Applied Computing

Historical Analysis of Code Annotations

Author: Phyllipe Lima - Level: Doctorate

Advisor: PhD. Eduardo Guerra

Co-Advisor: PhD. Paulo Meirelles

Program Entry: September 2016
Estimate Conclusion: September 2020
Related CBSoft Events: SBES, SBCARS

phyllipe_slf@yahoo.com.br,guerraem@gmail.com,paulo.meirelles@unifesp.br

Abstract. *Code annotation is a JAVA feature that enables the introduction of custom metadata on programming elements. It was introduced on version 5, and it is widely used by main enterprise application, frameworks and APIs. Although popular, the software engineering community lack research dedicated to the usage of code annotations. As such, this paper presents the current state of the ongoing research and next steps into assessing code annotations.*

Keywords: Code annotations, repositories, metric, evolution

Historical Analysis of Code Annotations

Phyllipe Lima[1], Eduardo Guerra[1], Paulo Meirelles[2]

[1]National Institute of Space Research (INPE)
Av. dos Astronautas, 1758, Jardim da Granja, 12227-010 – São José dos Campos – SP

[2]Department of Health Informatics – Federal University of São Paulo (UNIFESP)
Rua Botucatu, 862 – Vila Clementino, 04023-062 – São Paulo – SP

phyllipe_slf@yahoo.com.br,guerraem@gmail.com,paulo.meirelles@unifesp.br

Abstract. *Code annotation is a JAVA feature that enables the introduction of custom metadata on programming elements. It was introduced on version 5, and it is widely used by main enterprise application, frameworks and APIs. Although popular, the software engineering community lack research dedicated to the usage of code annotations. As such, this paper presents the current state of the ongoing research and next steps into assessing code annotations.*

1. Introduction

Code annotations, or simply annotations, are a feature available on the JAVA programming language. It allow developers to introduce custom metadata directly on programming elements. These are usually consumed by tools or frameworks, so they can execute a specific behavior. Since annotations are inserted directly on the source code, they are a convenient and quick alternative to configure metadata.

Main enterprise JAVA APIs make extensive use of code annotations, making them a relevant feature used on a daily basis by developers. Examples of APIs are the EJB[1] (Enterprise Java Beans) used to configure transactions and security restrictions, and the JPA[2] (Java Persistence API) used to perform object-relational mapping. Observing the 30-top rated JAVA projects on GitHub, the one with the least amount of annotations has 22% of annotated classes. While the most annotated project has 97% of annotated classes. On average, 76% of classes are annotated.

Although popular, the software engineering community lack work dedicated to study and analyze code annotations. For this reason, we have begun a research on this topic and the first outcome was published on [Lima et al. 2018]. As a first result of this PhD thesis, we proposed a novel suite of source code metrics dedicated to annotations, as there were none in the literature. It also uses a Percentile Rank Analysis approach [Meirelles 2013] to obtain threshold values.

With the suite of metrics available, this research can take a step further and study the evolution of annotations during software lifespan, in other words, we will perform a historical analysis through MSR (Mining Software Repository). This is also not explored on the literature, and software evolution has recently become the center of attention of developers [Rajlich 2014].

[1]www.oracle.com/technetwork/java/index-jsp-140203.html
[2]www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

This PhD thesis will use the following research questions to guide the next steps:

- #RQ1: How annotation metrics behave during software development?
- #RQ2: What is the life cycle of an annotation during software development?
- #RQ3: What is the relationship between annotations and refactorings?

2. Metadata

The word metadata is used in a variate of contexts in the computer science field. In all of them, it means data referring to the data itself. When talking about databases, the data are the ones persisted, and the metadata is their description, or in other words, the tables structure. In the object oriented world, the data are the instances, and the metadata are, as expected, their description. As such, attributes, methods, super-classes and interfaces are all metadata of a class instance. A class attribute, in turn, has its type, access modifiers and name as its metadata [Guerra 2014].

The class structure might not be enough to allow a specific behavior or routine to be executed, and therefore additional metadata can be configured on the programming elements. Afterwards, a framework or tool consumes them and executes the desired behavior. For instance, metadata can be used to generate source code [Damyanov and Holmes 2004], compile time verification [Ernst 2008], framework adaptation [Guerra et al. 2010], perform object-relational mapping, object-XML mapping and more.

Custom metadata can be configured using external storage, such as a database or an XML file [Fernandes et al. 2010]. This approach adds verbosity to the system, since it is necessary to inform a complete path between the referenced element and its metadata. Another alternative is to define code conventions [Chen 2006], used by the Ruby on Rails [Ruby et al. 2009] and the CakePHP framework[3]. Developing with this method can be productive, however it is limited when it comes to configuring more complex metadata. For this reason, some programming languages offer features for custom metadata configuration. JAVA offers annotations, and in C# there is the attributes [Miller and Ragsdale 2004].

Annotations were introduced in the version 5 of the JDK(Java Development Kit) as a response to the tendency of keeping the metadata files inside the source code itself, instead of using separate files [Córdoba-Sánchez and de Lara 2016]. Despite their potential for many interesting applications, its misuse can prevent software maintenance and evolution. For instance, an excessive amount of annotations can reduce code readability, and duplicated annotations through out the source code might pose a challenge during refactoring. Even the coupling of a class with an annotation schema (i.e, a set of related annotations) can prevent its usage outside the application context [Lima et al. 2018].

3. Annotation Metrics

Source code metrics are used to retrieve information from software and assess its characteristics. LOC (Lines of Code) and CYCLO (Cyclomatic Complexity) are

[3] `cakephp.org`

examples of established source code metrics. Well known techniques use metrics combined with rules to detect bad smells on the source code [Lanza and Marinescu 2006]. However, traditional code metrics does not recognize code 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-XML mapping. Also, using a set of annotations couples the application to the framework that can interpret them and current coupling metrics does not explicitly handle this. For this reason, the first step of this PhD thesis was to propose a novel suite of metrics dedicated to code annotations.

This section presents the annotation metrics published on [Lima et al. 2018]. The Java source code in Figure 1 is used to further clarify the usage and definition of the metrics.

```
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", discriminatorType = STRING)
23    public String exampleMethodA(){...}
24
25    @TransactionAttribute(SUPPORTS)
26    public String exampleMethodB(){...}
27
28    @TransactionAttribute(SUPPORTS)
29    public String exampleMethodC(){...}
30 }
```

Figure 1. Code for candidate metrics examples.

3.1. Annotations in Class - AC

This metric counts the number of annotations declared on all code elements in a class, including nested annotations. In our example code, the value of AC is 11.

3.2. Unique Annotations in Class - UAC

While AC counts all annotations, even repeated ones, UAC counts only distinct annotations. Two annotations are equal if they have the same name and all attributes match. For instance, the annotation @AssociationOverride on line 10 is different

from the one on line 12, for they have a nested annotation @JoinColumn that have different attributes. The first is "EX_ID" while the latter is "O_ID". Hence they are distinct annotations and will be computed separately. The UAC value for the example class is 9. Notice that the annotation on line 21, 25 and 28 are calculated only once for they are equal.

3.3. Annotations Schemas in Class - ASC

An annotation schema represents a set of related annotations provided by a framework or tool. This metric measures how coupled a class is to a specific framework since different schemas on a class imply different frameworks are being used by the application. The ASniffer measures this value by tracking the imports used for the annotations. On the example code, the ASC value is 2. The import `javax.persistence` which is a schema provided by the JPA, and the import `javax.ejb` provided by EJB.

3.4. Attributes in Annotations - AA

Annotations may contain attributes. They can be a string, integer or even another annotation. The AA metric counts the number of attributes contained in the annotation. For each annotation in the class, an AA value will be generated. For example, on line 9 the `@AssociationOverrides` has only one attribute "value", so the AA is 1. But `@AssociationOverride`, on line 10, contains two attributes, "`name`" and "`joinColumns`", so the AA value is 2.

3.5. Annotations in Element Declaration - AED

The AED metric counts how many annotations are declared in each code element, including nested annotations. In the example code, line 23, the method `exampleMethodA` has an AED value of 2, it has the `@TransactionAttribute` and `@DiscriminatorColumn`

3.6. Annotation Nesting Level - ANL

Annotations can have other annotations as attributes, which translates into nested annotations. ANL measures how deep an annotation is nested. The root level, is considered value 0. So `@Stateless` on line 18 has ANL = 0, while `@JoinColumn` on line 11 has ANL = 2. This is because it has `@AssociationOverride`, line 10, as a first level, and then the `@AssociationOverrides`, line 9, adds another nesting level, hence the value ANL = 2.

3.7. LOC in Annotation Declaration - LOCAD

LOC (Line of Code), is a well-known metric that counts the number of code lines [Lanza and Marinescu 2006]. The LOCAD is proposed as a variant of LOC that counts the number of lines used in an annotation declaration. `@AssociationOverrides` on line 9 has the valued LOCAD of 5, while `@NamedQuery`, line 14, has LOCAD = 4.

4. Historical Analysis of Code Annotation Metrics

According to [Rajlich 2014] software evolution is the constant changes that occurs during development. During evolution, developers add new features, correct previous mistakes, adapt to new requirements and new technologies. Software changes are the basic building blocks of software evolution and they introduce a new feature or a new property into software. On the other hand, software maintenance is where programmers no longer executes major changes in the software, but only small repairs to keep the software usable. Software in this stage has been called "legacy software", "aging software", or "software in maintenance".

In this PhD thesis we are concerned about annotations usage during both of these stages, since we will analyze from a historical point of view. In short, we will perform a study on changes that already occurred and how annotations were used in those changes. For this analysis, repositories play an essential role. These keep a set of data that represents changes, and through them it becomes possible to perform a historical analysis on certain characteristics [Sun et al. 2015].

However, mining software repositories and performing numerical analysis on our metrics is not enough to really comprehend the impact of code annotations on software maintenance and evolution. As we'll present additional discussion regarding our research questions, it becomes clear that they are not trivial. Hence, we need further investigation. For this purpose, we will also perform a qualitative analysis on selected projects, to better tackle all proposed research questions.

4.1. Annotation Metrics Extraction

To automate the process of extracting annotation metrics, we developed an open source and extensible tool called Annotation Sniffer (ASniffer)[4]. By extensible we mean that other developers are able to implement their own metrics and integrate them in the extraction process.

For the historical analysis, we need to collect these metrics values on a per commit basis. That is, for every commit we will extract the metrics values. Currently, the tool ASniffer cannot perform the extraction process with this granularity, so a new tool called ARTHAS (Automatic Recursive Tool for Historical Annotations Sniffing) is being developed to support this part of the research. ARTHAS is a tool that combines the core of the ASniffer with the Repodriller[5] capabilities to mine software repository. After the extraction process, a CSV file is generated for every project being analyzed. With these values available, the research proceeds on to further investigate how annotations behave from a historical perspective.

4.2. Selected Projects

To perform the analysis, a set of open source projects is required. Since it is unpractical to use all available projects, a sample will be used. The criteria used to select these will be based on the work of [Nagappan et al. 2013]. This paper introduces a systematic way of obtaining a set of projects using the concepts of similarity and

[4] `github.com/phillima/asniffer`
[5] `github.com/mauricioaniche/repodriller`

diversity. It first needs to define a group of dimensions, which represents relevant characteristics of software for the specific problem. Example of dimensions are: Number of annotated classes, Total Number of LOC (Lines of Code) per class and so forth. Following this criteria our goal is to update the set of projects selected to the first part of this PhD thesis [Lima et al. 2018] that already are both diverse and representative.

4.3. Metrics Evolution

To answer our first RQ lets consider the AC metric for example. For every commit, ARTHAS outputs an AC value per class. We are interested in observing how the AC value evolves from the whole project point of view, not just a single class.

It is expected that this value will grow during the software development life cycle. We want to be able to identify if it stabilizes at some point, if there is a fallback or if it just keeps on growing for as long as the software is being developed. We also want to detect if there is a stage during the development where more annotations are being added/created. For instance, are annotations primarily added during the initial, middle, or close to a release version of the software?

One reason for this investigation is that, if annotation metrics values just keeps on growing, then we can conclude that indeed annotations are highly impacting software maintenance. If, on the other hand, it stabilizes quickly, then it might not be a top priority in terms of software maintenance.

4.4. Annotation Life Cycle

Our second RQ aims at investigating the annotation life cycle. The goal is to track what happens to a specific annotation as soon as it is created and used. It is important to understand the difference between these two. In the first case we want to investigate if, after an annotation is created, it goes through a refactoring process and how that affects the code elements using it. In the second case we want to track what happens to annotations already in use. Do they get deleted? Do they persist through out the software life-cycle? If they get deleted, what triggered that?

4.5. Refactorings associated to Annotations

The third RQ seeks to understand how code annotations are associated to software refactoring. This process occurs for several reasons, such as to make code cleaner, extensible, maintainable, decoupled and so forth. Furthermore, we will pay attention if annotations was used in the correction of bugs. The following list presents open questions that we aim to answer with this research.

- To make the code cleaner was annotations added or removed?
- How annotations was used to make the code extensible? Did adding annotations ease the process of making the code extensible? Or was it the opposite?
- During a refactoring process, were more annotations added or removed? Or were they irrelevant?
- What is the association between code bugs and annotations? Did an annotation introduced a bug? Was annotations used in a solution to correct a bug?

5. Related Work

Since no previous annotation metrics were available, it was hard to perform a comparison study. Most works on the literature uses annotations to solve problems, and not analyze their structure. Some works apply annotations to support the implementation of design patterns [Meffert 2006] or to enable architectural refactoring [Krahn and Rumpe 2006]. A general experimental study about the use of metadata-based frameworks was done by [Guerra et al. 2010]. This study indicates that metadata-based frameworks reduce the coupling between the framework and the application.

As such, the work published on [Lima et al. 2018] is novel, introducing a suite of annotation metrics and applying a statistical analysis proposed by [Meirelles 2013] to obtain thresholds values. Combining this with the next steps in this thesis, we intend to present to the community a deep analysis in code annotations usage.

6. Preliminary Results

Prior to [Lima et al. 2018], there were no suite of metrics dedicated to annotations and the software engineering community lacked studies that performed an analysis of code annotations. With the suite of metrics available to the community we hope other researches evaluate them as well. At this moment, the main contribution of this PhD has been the release of this suite of metrics.

We also began investigating if projects could be clustered based on annotations usage, and if that had any relationship with the software domain. For this, a Kohonen Self Organizing Maps was used and the input was the annotation metrics values. The results showed a possibility of 4 different clusters. This work was published on [Lima et al. 2017]

To support our research we developed the ASniffer, an open source extensible tool. It is available to other developers, and we encourage them to use and contribute to it. The ASniffer is evolving to a new tool, ARTHAS, that collects annotation metrics on a commit basis to allow historical analysis. In short, the ARTHAS tool will be essential to answer the questions presented and discussed in this paper.

References

[Chen 2006] Chen, N. (2006). Convention over configuration.

[Córdoba-Sánchez and de Lara 2016] Córdoba-Sánchez, I. and de Lara, J. (2016). Ann: A domain-specific language for the effective design and validation of java annotations. *Computer Languages, Systems Structures*, 45:164 – 190.

[Damyanov and Holmes 2004] Damyanov, I. and Holmes, N. (2004). Metadata driven code generation using .net framework. In *Proceedings of the 5th international conference on Computer systems and technologies*, pages 1–6. ACM.

[Ernst 2008] Ernst, M. D. (2008). Type annotations specification (jsr 308).

[Fernandes et al. 2010] Fernandes, C., Ribeiro, D., Guerra, E., and Nakao, E. (2010). Xml, annotations and database: a comparative study of metadata definition strategies for frameworks. *May 19–20, Vila do Conde*, page 115.

[Guerra 2014] Guerra, E. (2014). *Componentes Reutilizáveis em Java com Reflexão e Anotações*. Casa do Código, 1st edition. [in portuguese].

[Guerra et al. 2010] Guerra, E. M., de Souza, J. T., and Fernandes, C. T. (2010). A pattern language for metadata-based frameworks. In *Proceedings of the 16th Conference on Pattern Languages of Programs*, PLoP '09, pages 3:1–3:29, New York, NY, USA. ACM.

[Guerra et al. 2009] Guerra, E. M., Silveira, F. F., and 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*, volume 26, pages 35–39.

[Krahn and Rumpe 2006] Krahn, H. and Rumpe, B. (2006). Towards enabling architectural refactorings through source code annotations. *Lecture Notes in Informatics*, P-82:203–212.

[Lanza and Marinescu 2006] Lanza, M. and Marinescu, R. (2006). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer.

[Lima et al. 2017] Lima, P., Guerra, E., and Meirelles, P. (2017). Definição de clusters para classificação do uso de anotações em código java. In *5th Workshop on Software Visualization, Evolution and Maintenance, VEM at CBSoft'17*. [in portuguese].

[Lima et al. 2018] Lima, P., Guerra, E., Meirelles, P., Kanashiro, L., Silva, H., and Silveira, F. (2018). A metrics suite for code annotation assessment. *Journal of Systems and Software*, 137:163 – 183.

[Meffert 2006] 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*, pages 8 pp.–445.

[Meirelles 2013] Meirelles, P. R. M. (2013). *Monitoring Source Code Metrics in Free Software Projects*. PhD thesis, Department of Computer Science – Institute of Mathematics and Statistics of University of São Paulo. [in portuguese].

[Miller and Ragsdale 2004] Miller, J. S. and Ragsdale, S. (2004). *The Common Language Infrastructure Annotated Standard*. Addison-Wesley Professional.

[Nagappan et al. 2013] Nagappan, M., Zimmermann, T., and Bird, C. (2013). Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 466–476, New York, NY, USA. ACM.

[Rajlich 2014] Rajlich, V. (2014). Software evolution and maintenance. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 133–144, New York, NY, USA. ACM.

[Ruby et al. 2009] Ruby, S., Thomas, D., and Hansson, D. (2009). *Agile Web Development with Rails, Third Edition*. Pragmatic Bookshelf, 3rd edition.

[Sun et al. 2015] Sun, X., Li, B., Leung, H., Li, B., and Li, Y. (2015). Msr4sm: Using topic models to effectively mining software repositories for software maintenance tasks. *Information and Software Technology*, 66:1 – 12.