The Journal of Open Source Software

# Annotation Sniffer: A tool to Extract Code Annotations Metrics

**Phyllipe Lima**[1,2], **Eduardo Guerra**[2], **and Paulo Meirelles**[3]

**1** CDG, National Institute of Telecommunications - INATEL, Brazil **2** LAC, National Institute for Space Research - INPE, Brazil **3** DHI, Federal University of São Paulo - UNIFESP, Brazil

**DOI:** 10.21105/joss.01960

**Software**

- Review
- Repository
- Archive

**Editor:** George K. Thiruvathukal

**Reviewers:**

- @arcuri82
- @danieledipompeo

**Submitted:** 29 November 2019
**Published:** 17 December 2019



## Summary

Enterprise Java frameworks and APIs such as JPA (Java Persistence API), Spring, EJB (Enterprise Java Bean), and JUnit make extensive use of code annotations as means to allow applications to configure custom metadata and execute specific behavior. Observing the top 30 ranked Java projects on GitHub, they have, on average, 76% of classes with at least one annotation. Some projects may have more than 90% of its classes annotated. To measure code annotations usage and analyze their distribution, our work in (P. Lima et al., 2018) proposed a novel suite of software metrics dedicated to code annotations. We used a Percentile Rank Analysis approach (Meirelles, 2013) to obtain threshold values.

Source code metrics retrieve information from software to assess its characteristics. Well-known techniques use metrics associated with rules to detect bad smells on the source code (Lanza & Marinescu, 2006). However, traditional code metrics do not recognize code annotations on programming elements, which can lead to an incomplete code assessment (Guerra, Silveira, & Fernandes, 2009). For instance, 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 a framework that can interpret them and current coupling metrics does not explicitly handle this.

To automate the process of extracting the novel suite of software metrics for code annotation in (P. Lima et al., 2018), we developed an open-source tool called Annotation Sniffer (`ASniffer`). It is a command-line tool that reads java source code, extracts the metrics values, and outputs an XML report. Potential `ASniffer` users are software engineers or researchers interested in static code analysis and mining software repositories. Additionally, given that it is an extensible tool, other developers can implement their metrics and integrate them in the extraction process. Figure 1 presents an overview diagram of the ASniffer tool.

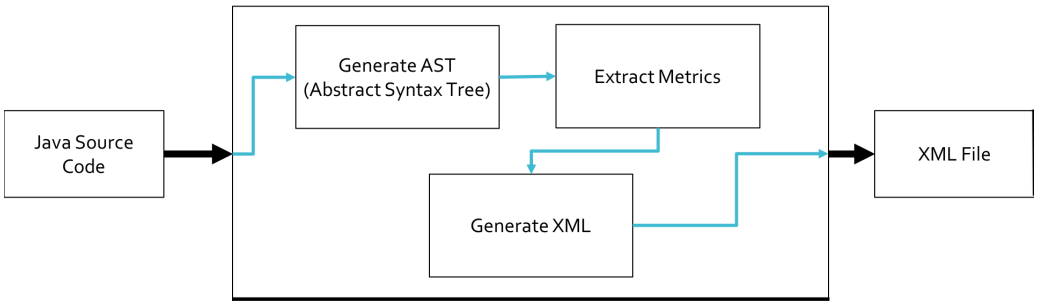


Figure 1: ASniffer overview diagram

We previously presented the first version of this tool and published it on a workshop (P. S. Lima et al., 2018). The current version has an improved extensibility mechanism as well as a more compact and complete report, to support our ongoing research about code annotations and metadata in object-oriented programming.

## Metadata and Code Annotations

A variety of contexts in the computer science field uses the term "metadata". In all of them, it means data referring to the data itself. In databases, the data are the ones persisted, and the metadata is their description, i.e., the structure of the table. In the object-oriented context, the data are the instances, and the metadata is their description, i.e., information that describes the class. As such, fields, methods, super-classes, and interfaces are all metadata of a class instance. A class field, in turn, has its type, access modifiers, and name as its metadata (Guerra, 2014).

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 and in C#, by attributes. A benefit is that the metadata definition is closer to the programming element, and its definition is less verbose than external approaches. Annotations are a feature of the Java language, which became official on version 1.5. The code on Listing 1 presents a simple `Player` class using code annotation to perform the object-relational mapping.

```
@Entity
@Table(name="Players")
public class Player {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(name = "health")
    private float health;

    @Column(name = "name")
    private String name;

    //getters and setters omitted
}
```

Listing 1: Example of code annotations

To map this `Player` class to a table in a database, to store the player's information, we need to pass in some `extra information` about these code elements. In other words, we need to define an object-relational mapping, and we need to configure which elements should be mapped to a column, table, and so forth. Using code annotations provided by the JPA API, this mapping is easily achieved. When this code gets executed, the framework consuming the annotations knows how to perform the expected behavior.

## Annotation Metrics

Our work in (P. Lima et al., 2018) proposed a novel suite of software metrics dedicated to code annotations. In this section, we briefly describe them and demonstrate how they are calculated. We have three categories of metrics:

- Class Metric: Outputs one value per class.
- Code Element Metric: Outputs one value per code element (fields, methods, etc.).
- Annotation Metric: Outputs one value per code annotation.

We use the code presented in Listing 2 as an example.

```
import javax.persistence.AssociationOverrides;
import javax.persistence.AssociationOverride;
import javax.persistence.JoinColumn;
import javax.persistence.NamedQuery;
import javax.persistence.DiscriminatorColumn;
import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;

@AssociationOverrides(value = {
      @AssociationOverride(name="ex",
         joinColumns = @JoinColumn(name="EX_ID")),
      @AssociationOverride(name="other",
         joinColumns = @JoinColumn(name="O_ID"))})
@NamedQuery(name="findByName",
      query="SELECT c " +
            "FROM Country c " +
            "WHERE c.name = :name")
@Stateless
public class Example {...

   @TransactionAttribute(SUPPORTS)
   @DiscriminatorColumn(name = "type", discriminatorType = STRING)
   public String exampleMethodA(){...}

   @TransactionAttribute(SUPPORTS)
   public String exampleMethodB(){...}

}
```

Listing 2: Example of code to extract annotation metrics.

- 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 equal to 10. It is a `Class Metric`.
- 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 arguments match. For instance, both annotations `@AssociationOverride` are different, for they have a nested annotation `@JoinColumn` that have different arguments. 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 nine. Note that the annotation `@TransactionAttribute()` is counted only once. It is a `Class Metric`.
- Annotations Schemas in Class (ASC): An annotation schema represents a set of related annotations provided by a framework or tool. This measures how coupled a class is to a framework. This value is obtained by tracking the imports used for the annotations. On the example code, the ASC value is two. The import `javax.persistence` is a schema provided by the JPA, and the import `javax.ejb` is provided by EJB. It is a `Class Metric`.
- Arguments in Annotations (AA): Annotations may contain arguments. They can be a string, integer, or even another annotation. The AA metric counts the number of

arguments contained in the annotation. For each annotation in the class, an AA value will be generated. For example, the `@AssociationOverrides` has only one argument named `value`, so the AA value is equal one. But `@AssociationOverride`, contains two arguments, `name` and `joinColumns`, so the AA value is two. It is an `Annotation Metric`.

- 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, the method `exampleMethodA` has an AED value of two, it has the `@TransactionAttribute` and `@DiscriminatorColumn`. It is a `Code Element Metric`.
- Annotation Nesting Level (ANL): Annotations can have other annotations as arguments, which translates into nested annotations. ANL measures how deep an annotation is nested. The root level is considered value zero. The annotations `@Stateless` has ANL value of zero, while `@JoinColumn` has ANL equals two. This data is because it has `@AssociationOverride` as a first level, and then the `@AssociationOverrides` adds another nesting level, hence the value ANL is two. It is an `Annotation Metric`.
- LOC in Annotation Declaration (LOCAD): LOC (Line of Code), is a well-known metric that counts the number of code lines. The LOCAD is proposed as a variant of LOC that counts the number of lines used in an annotation declaration. `@AssociationOverrides` has a LOCAD value of five, while `@NamedQuery` has LOCAD equals four. It is an `Annotation Metric`.

## Annotation Sniffer

The ASniffer tool uses the JDT[1](Java Development Tools) API to build the Abstract Syntax Tree (AST) from a text file containing the source code. The ASniffer traverses this AST, visiting the nodes and gathering information about the code elements. After the processing is done, an XML is generated as output.

To create the AST (Abstract Syntax Tree), we use the method `ASTParser.createASTs`. This method is exposed by the JDT and receives an array of strings containing the file path of each source code that we wish to analyze. Another parameter for the method is a class that will handle the compilation units. Our class is the `MetricsExecutor` and this class must extend the `FileASTRequestor`. From inside `MetricsExecutor` we call every metric class and pass in the compilation unit (generated by the `ASTParser`).

To understand the extraction process, we will use a snippet from the code that collects the `Annotations in Class` metric, presented in Listing 3. Since this is a `Class Metric`, i.e., outputs one value per class, it must extend the `ASTVisitor` class and implement our custom interface `IClassMetricCollector`. The superclass provides methods that are used to visit the nodes from the Compilation Unit. For instance, for the AC metric, we visit every annotation encountered, and increment the value for `annotations`. Our custom interface provides two methods, the first one, (`execute()`), initializes the extraction process, while the second one, (`setResult()`), is where the result is stored.

```
@ClassMetric
public class AC extends ASTVisitor implements IClassMetricCollector{
  //We also visist MarkerAnnotation and SingleMemberAnnotation
    private int annotations = 0;
    @Override
    public boolean visit(NormalAnnotation node) {
```

[1] https://www.eclipse.org/jdt/

```
            annotations++;
            return super.visit(node);
        }
        @Override
        public void execute(CompilationUnit cu, MetricResult result,
                            AMReport report) {
            cu.accept(this);
        }
        @Override
        public void setResult(MetricResult result) {
            result.addClassMetric("AC",annotations);
        }
}
```

Listing 3: Snippet from the code that implements the Annotations in Class metric

## License

Annotation Sniffer is licensed under the GNU Lesser General Public License v3.0

## Acknowledgements

This work is supported by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo), grant 2014/16236-6 and CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior)

## References

Guerra, E. (2014). *Componentes reutilizáveis em java com reflexão e anotações* (1st ed.). Casa do Código.

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* (Vol. 26, pp. 35–39).

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. doi:10.1007/3-540-39538-5

Lima, P., Guerra, E., Meirelles, P., Kanashiro, L., Silva, H., & Silveira, F. (2018). A metrics suite for code annotation assessment. *Journal of Systems and Software*, *137*, 163–183. doi:https://doi.org/10.1016/j.jss.2017.11.024

Lima, P. S., Guerra, E. M., & Meirelles, P. R. (2018). Annotation sniffer: Open source tool for annotated code elements. In *CBSoft 2018 - tools session ()*.

Meirelles, P. R. M. (2013). *Monitoring source code metrics in free software projects* (PhD thesis). Department of Computer Science – Institute of Mathematics; Statistics of University of São Paulo. Retrieved from http://www.teses.usp.br/teses/disponiveis/45/45134/tde-27082013-090242/pt-br.php