Annotation Sniffer:
Open Source Tool for Annotated Code Elements

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

[1]National Institute for 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. *Since the introduction of code annotations in the Java language, this feature has been widely adopted by software developers across the globe. Main enterprise frameworks make extensive use of code annotations as a replacement for XML based solutions. Code annotations are inserted directly on code elements, providing a simple way to introduce custom metadata. To aid in our research that aims understanding how developers use this feature, we developed the Annotation Sniffer. An open source tool that extracts a suite of metrics dedicated to code annotations.*
Video link: *https://youtu.be/f9ER5zaJkXc*

1. Introduction

Source code metrics can 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 and Marinescu 2006, Van Rompaey et al. 2007]. 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 a framework that can interpret them and current coupling metrics does not explicitly handle this.

In the object-oriented world, metadata is data about the programming elements, or code elements. For example, the access modifiers and the type are the metadata from a class attribute. Frequently, software developers need to configure extra metadata on these code elements as a means to allow a more specific behavior on their application [Guerra et al. 2013].

The Java language provides code annotation as a means to introduce custom metadata on programming elements. The annotation is declared directly on the source code, as opposed to an XML approach. The latter requires an external file with the metadata, and a path to this XML file must be provided to the software consuming the metadata.

Even though code annotations are very popular, there were no software metrics in the literature focused on them. For this reason, we have published a paper [Lima et al. 2018] proposing an unprecedented suite of metrics exclusively for code annotations. It also discusses a percentile rank analysis to obtain threshold values [Meirelles 2013]. This work was the first outcome of an ongoing research regarding code annotations.

To automate the process of extracting the code annotation metrics, we developed an open source tool called Annotation Sniffier (ASniffer). It obtains the metrics values and outputs them in an XML report. The goal of this paper is to present this novel tool, since no other similar was found.

2. Annotation Metrics

This section briefly presents the annotation metrics collected by the ASniffer. For a more thorough definition and analysis refer to [Lima et al. 2018]. The Java source code in Figure 3 clarifies the usage 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.

2.1. 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.

2.2. 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 annotations on line 21, 25 and 28 are calculated only once, since they are equal.

2.3. 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` is a schema provided by the JPA, and the import `javax.ejb` is provided by EJB.

2.4. 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”. Therefore the AA is 1. But `@AssociationOverride`, on line 10, contains two attributes, “`name`” and “`joinColumns`”, so the AA value is 2.

2.5. 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`

2.6. 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 `@AssociationOverrides`, line 9, adds another nesting level, hence the value ANL = 2.

2.7. LOC in Annotation Declaration - LOCAD

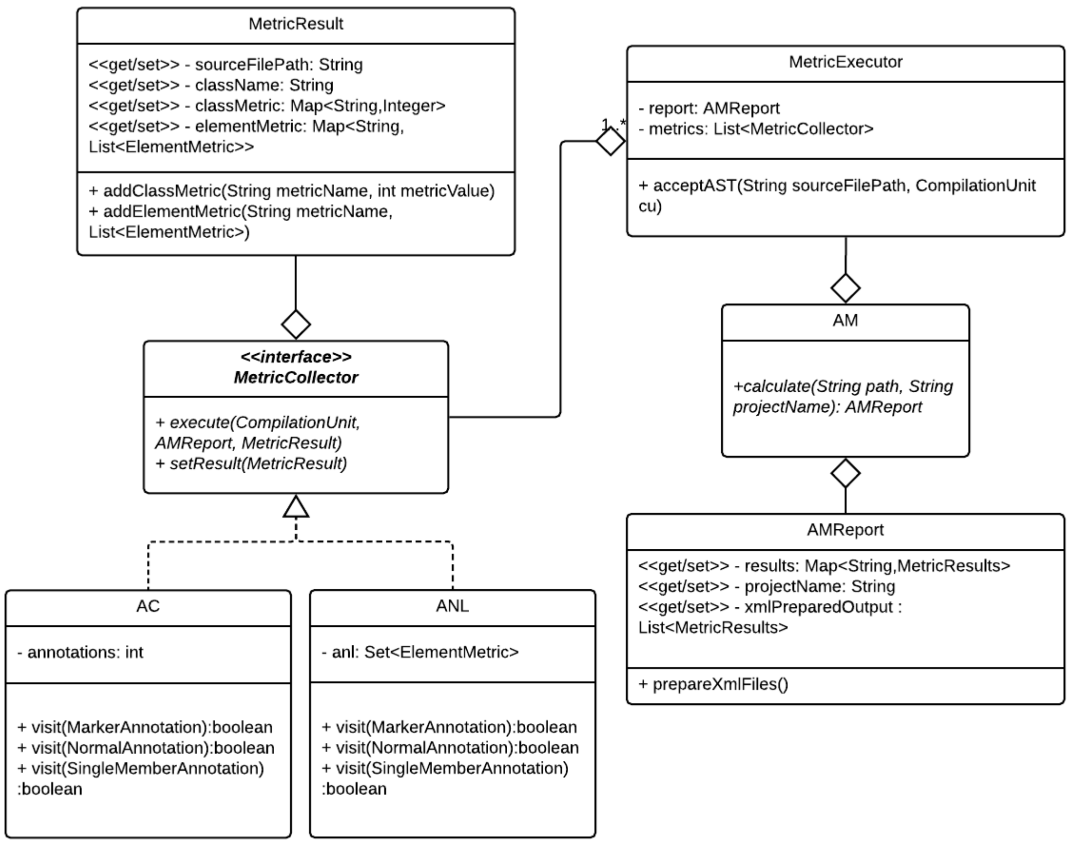
LOC (Line of Code), is a well-known metric that counts the number of code lines [Lanza and Marinescu 2006]. The LOCAD is proposed as a variant of LOC, but it counts the number of lines used in an annotation declaration. `@AssociationOverrides` on line 9 has the valued LOCAD of 5, while `@NamedQuery`,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

3. Annotation Sniffer

This section describes the internal structure of the tool.

3.1. ASniffer Architecture

The ASniffer tool uses the JDT[1] (Java Development Tools) API. This API exposes the `ASTParser` class, which allows the creation of compilation units from a text file. Once these are obtained, the ASniffer extracts the metrics from them. Figure 2 presents the core UML diagram.



MetricResult
<<get/set>> - sourceFilePath: String
<<get/set>> - className: String
<<get/set>> - classMetric: Map<String,Integer>
<<get/set>> - elementMetric: Map<String, List<ElementMetric>>
+ addClassMetric(String metricName, int metricValue)
+ addElementMetric(String metricName, List<ElementMetric>)
MetricExecutor
- report: AMReport
- metrics: List<MetricCollector>
+ acceptAST(String sourceFilePath, CompilationUnit cu)
1..*
<<interface>> MetricCollector
+ execute(CompilationUnit, AMReport, MetricResult)
+ setResult(MetricResult)
AM
+calculate(String path, String projectName): AMReport
AMReport
<<get/set>> - results: Map<String,MetricResults>
<<get/set>> - projectName: String
<<get/set>> - xmlPreparedOutput : List<MetricResults>
+ prepareXmlFiles()
AC
- annotations: int
+ visit(MarkerAnnotation):boolean
+ visit(NormalAnnotation):boolean
+ visit(SingleMemberAnnotation) :boolean
ANL
- anl: Set<ElementMetric>
+ visit(MarkerAnnotation):boolean
+ visit(NormalAnnotation):boolean
+ visit(SingleMemberAnnotation) :boolean

Figure 2. ASniffer Class Diagram.

3.2. Creating the AST

The first step is to create the AST (Abstract Syntax Tree), which is done in class `AM`. Since ASniffer may extract metrics from several java source code, it is more efficient to create the ASTs all at once, using the method `ASTParser.createASTs`. This method receives an array of strings containing the file path of each and every source code. 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.

3.3. Extracting the Metric

To understand the extraction process, we will use the code from the Annotations in Class metric, known as AC. The code is presented on Figure 3. Every class that

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

represents a metric must extend the `ASTVisitor` class, which provides the methods to visit the compilation unit. For instance, the method on line 5 is called for every Marker Annotation[2] - an annotation with no attributes - on the compilation unit. Since the AC metric counts all annotations, we increment our counter. For other metrics, we might need to perform more complex operations.

These metric classes must also implement our custom interface `MetricsCollector` and both methods `execute` and `setResult`. The first one is to begin the execution process. In this example, it simply accepts the compilation unit. The latter is where the result is stored. Notice in Figure 3 in line 22, we store the value "annotations" and also the name of the metric "AC". The class `MetricResult` is a container for all metrics values for a single compilation unit. It stores information such as fully qualified name, package and the source file path.

The class `AMReport` is the container for every `MetricResult`. Hence, it contains the result of a whole project. The XML file is generated using the AMReport.

```
 1  @AnnotationMetric
 2  public class AC extends ASTVisitor implements MetricCollector{
 3    private int annotations = 0;
 4    @Override
 5    public boolean visit(MarkerAnnotation node) {
 6      annotations++;
 7    }
 8    @Override
 9    public boolean visit(NormalAnnotation node) {
10      annotations++;
11    }
12    @Override
13    public boolean visit(SingleMemberAnnotation node) {
14      annotations++;
15      }
16    @Override
17    public void execute(CompilationUnit cu, MetricResult result, AMReport report) {
18      cu.accept(this);
19    }
20    @Override
21    public void setResult(MetricResult result) {
22      result.addClassMetric("AC",annotations);
23    }
24  }
```

Figure 3. Code for the AC metric.

3.4. Usage instruction

Potential ASniffer users are software engineers or researchers interested in static code analysis and mining software repositories. Moreover, since its an extensible tool, other developers can implement their own metrics and integrate them in the extraction process. Following is the command line to run ASniffer:

```
java -jar asniffer.jar <path to project> <path to xml report> <single/multi>
```

The first parameter is a path to a folder containing the source code. It may carry a single or several projects. For a single project, the tool assumes that the

[2]`help.eclipse.org/oxygen/index.jsp`

folder's name is the name of the project. If, on the other hand, this path contains several projects, each one should be placed in a sub-directory. The tool will use each sub-directory's name as the project's name. The second parameter is the path to store the generated XML report. An XML file is generated for each project, in the case of multiple projects. The XML file's name matches the project's name. The last parameter informs ASniffer if the provided path contains only one or multiple projects.

4. Conclusion

Before [Lima et al. 2018], the literature lacked studies that effectively performed an analysis of code annotations. The first step was to propose a novel suite of metrics and perform an analysis. Since it is impractical to collect these metrics manually, the ASniffer was developed. An open source tool to extract code annotation metrics.

As a future work, the tool will extend its capability and perform historical collection. The goal is to extract the metrics per commit, and generate a report to observe how the metrics behave during the development process. This will help understand how code annotations can impact software development and maintenance. Also, the development of a GUI is being carried out to further ease using ASniffer.

During the development of ASniffer, no similar tool was found to perform a comparison. Therefore, all validation was done manually, which may compromise its accuracy. To help overcome these limitations, the tool is available as open source at the GitHub (`https://github.com/phillima/asniffer`) under the GNU Lesser General Public License (LGPL V3).

References

[Guerra et al. 2013] Guerra, E., Alves, F., Kulesza, U., and Fernandes, C. (2013). A reference architecture for organizing the internal structure of metadata-based frameworks. *Journal of Systems and Software*, 86(5):1239 – 1256.

[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.

[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. 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.

[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].

[Van Rompaey et al. 2007] Van Rompaey, B., Du Bois, B., Demeyer, S., and Rieger, M. (2007). On the detection of test smells: A metrics-based approach for general fixture and eager test. *Software Engineering, IEEE Transactions on*, 33(12):800–817.