Attribute Sniffer: Collecting Attribute Metrics for C# Code

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

[1]National Institute of Telecomunications (INATEL), Santa Rita do Sapucaí – MG, Brazil

[2]National Institute for Space Research (INPE), São José dos Campos – SP, Brazil

[3]Federal University of São Paulo (UNIFESP), São Paulo – SP, Brazil

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

Abstract. *The C# attributes have been available since the release of this language as a means to configure custom metadata on code elements. Being very similar to Java annotations, they are both introduced directly in the source code. Several C# frameworks such as NUnit, Unity Game Engine, and JSON.net use attributes to allow applications to configure custom metadata, as to execute different behavior and validation. This paper presents the Attribute Sniffer, a tool to collect attributes metrics on the C# source code, also validating the metrics proposed initially for Java annotations.*
Video link: `https://youtu.be/6EypnkAz3i0`

1. Introduction

Since the C# programming language was released, it allows custom metadata configuration on programming elements through *attributes* [ECMA 2001]. This feature is similar to *annotations* on the Java programming language. Both attributes and annotations are inserted directly in the source code and, therefore, do not require external files to access the configured metadata. When using traditional source code metrics, such as the popular CK Suite [Chidamber and Kemerer 1991], they do not work well with attributes or annotations and may output incomplete or inadequate measurements [Guerra et al. 2009]. Established techniques associate metrics with rules to detect bad smells on the source code [Lanza and Marinescu 2006].

Our research group proposed, on a previous work [Lima et al. 2018b], a novel suite of metrics dedicated to code annotations. To obtain threshold values, we used the Percentile Rank Analysis approach [Meirelles 2013]. These metrics were defined considering code annotations, and therefore, we validated them on real-world open-source Java projects. To automate the process of extracting their values, we developed an open source tool called Annotation Sniffer [Lima et al. 2018a].

Given that attributes are very similar to annotations, it makes sense to extend our research and also encompass the C# programming language. Software metrics becomes more useful if more languages are also able to be analyzed and measured. Our analysis begins validating the suite of metrics in the context of C# attributes, without exploring specific characteristics at this point.

In this paper, we present the Attribute Sniffer, an open source tool developed to scan C# projects and extract metrics values regarding attributes usage. To evaluate the tool we selected the top 10 (ranked by the number of stars) real-world open source projects from the GitHub repository[1] to collect these metrics. On average, 44.5% of these classes contain at least one attribute. The project with the highest number has 54.61% of its classes with at least one attribute, while the lowest has 9.9%. This result showed that C# developers are implementing solutions with attributes and, therefore, further investigation and experiments might bring light to the usage of attributes.

2. Metadata and Attributes in C#

It is necessary to make a distinction in the terminology of some object-oriented elements. Some developers refer to class members as *attributes*. However, in the C# language, the term *attribute* is used to designate the feature that allows developers to configure custom metadata. To avoid confusion, in this paper, we refer to class members as *fields*.

The term "Metadata" refers to data about the data itself. In the object-oriented context, the data are instances, and the metadata is their description, i.e., the class. Considering the class, the programming elements, contain their metadata as well. For example, the access modifiers and the type are the metadata for a class field. Often, software developers are required to configure additional metadata on these code elements to allow a more specific behavior on their application[Guerra et al. 2013].

Attributes are a feature of the C# language that allows developers to configure metadata on programming elements. Frameworks or tools may consume them and execute a specific behavior. In Figure 1, we have a simple C# class with an attribute declared in line 1. It describes a `Player` video game character, with some fields such as name and health points. The attribute `[RequireComponent()]` is giving extra information, i.e., adding metadata, to this `Player` class. It is informing that the class must contain a rigid body and should be subject to physics laws. The Unity Game Engine defined this attribute, and it can consume the metadata and execute specific behavior.

```
1  [RequireComponent(typeof(Rigidbody2D))] //This is an attribute
2  public class Player : MonoBehaviour{
3     public string Name { get; set; }
4     public float Health { get; set; }
5     //Other parts omitted
6  }
```

Figure 1. Example class using a C# attribute

Attributes, by themselves, do not change the behavior of the code element they were added. If no other code can consume it, then nothing happens, and the metadata has no meaning other than documentation. We chose this example because Unity is one of the most popular game engines, and C# is the official programming language [Polančec and Mekterović 2017].

3. Attribute Metrics

This section presents the modified version of the metrics defined for annotations on [Lima et al. 2018b]. In this first part, we wanted to validate these metrics for the C# language, and even though attributes contain some features not available in Java annotations, we did not create new metrics to measure this.

[1]The list of projects is available at the tool repository `https://github.com/LydiaBraga/AttributeSniffer`

Attributes in Class - AC. It counts the number of attributes declared on all code elements in a class. This metrics comes directly from the Java version.

Unique Attributes in Class - UAC. The AC metric counts all attributes, even repeated ones. The UAC metric, however, counts only distinct attributes. Two attributes are equal only if they have the same name, and the passed arguments are all identical. For instance, the attribute `[Attribute(1,2)]` is different from `[Attribute(2,1)].` As the AC, this metric derives directly from the Java version.

Arguments in Attributes - AA. Attributes may contain arguments. They may be primitive values, the type `System.Type` or the type object. The AA metric counts the number of arguments contained in the attribute declaration. This metric outputs a value for each attribute encountered in the class. Behaves the same way in the Java version.

Attributes in Element Declaration - AED. It counts how many attributes are declared in each code element. It has an equivalent metric for Java and outputs a value for each code element that has at least 1 (one) attribute.

LOC in Attribute Declaration - LOCAD. LOC (Lines of Code) is the metric that counts the number of code lines. The LOCAD is a variant of LOC, but it counts the number of lines used in an attribute declaration. It reports a value for each attribute in the class. It works the same way for Java annotations.

Overall, five out of seven metrics proposed to annotations on the Java code was straightforward to adapt to C# attributes. The exception was ANL (Annotation Nesting Level), and the ASC (Annotation Schemas per Class) metric. The first one measures annotations inside annotations, and C# has a different way to achieve this through inheritance. The latter measures the group of annotations from the same package, and C# has the concept of *namespace*, which is different.

4. Attribute Sniffer

This section describes the Attribute Sniffer tool. We first present the Roslyn compiler and how it is used to create the AST (Abstract Syntax Tree) of a C# code. The tool runs on the .Net Core instead of the .Net Framework to enable cross-platform execution. Afterward, we present the internal structure of the tool and how to run it.

4.1. Creating ASTs with the Roslyn Compiler

The .NET Compiler Platform, commonly known as *Roslyn*[2] is an open source, cross-platform suite of compilers and code analysis APIs for .NET languages such as C# and Visual Basic [Microsoft 2019]. These APIs expose the `CSharpSyntaxTree` class, which allows the parse of source code into syntax trees, the primary structure to the lexical and syntactic analysis of the compiler API.

Syntax trees hold all the source information in full fidelity; in other words, it contains every grammatical construct and lexical token, besides white spaces, comments, and preprocessors directives. Each node of this tree represents declarations, statements, clauses, and expressions present in the source code, and they can be visited to analyze or refactor information.

[2]`https://github.com/dotnet/roslyn`

Although it is enough to describe the source code, this data structure is unable to identify what is being referenced. For this, the semantic model can be used. In short, the semantic model encapsulates the language rules and gives a simple way to match identifiers with the program elements. It can be obtained through the creation of a `CSharpCompilation` from the syntax tree. The Attribute Sniffer tool uses both models to collect the metrics.

4.2. Attribute Sniffer Internal Structure

The AttributeSniffer tool uses two structures provided by the Roslyn compiler, `SyntaxTree` and `SemanticModel`, to build the AST of the input source code and visit its nodes, collecting the metrics values. Both of these structures are created in the `MetricsCollector` class for each input source code read by the `Sniffer` class. The method `CSharpSyntaxTreeParseText` takes the source code as a parameter and creates the AST, returning the `CSharpCompilation`. From this, it is possible to extract the `SemanticModel`. Figure 2 presents the UML diagram of the core of the tool.

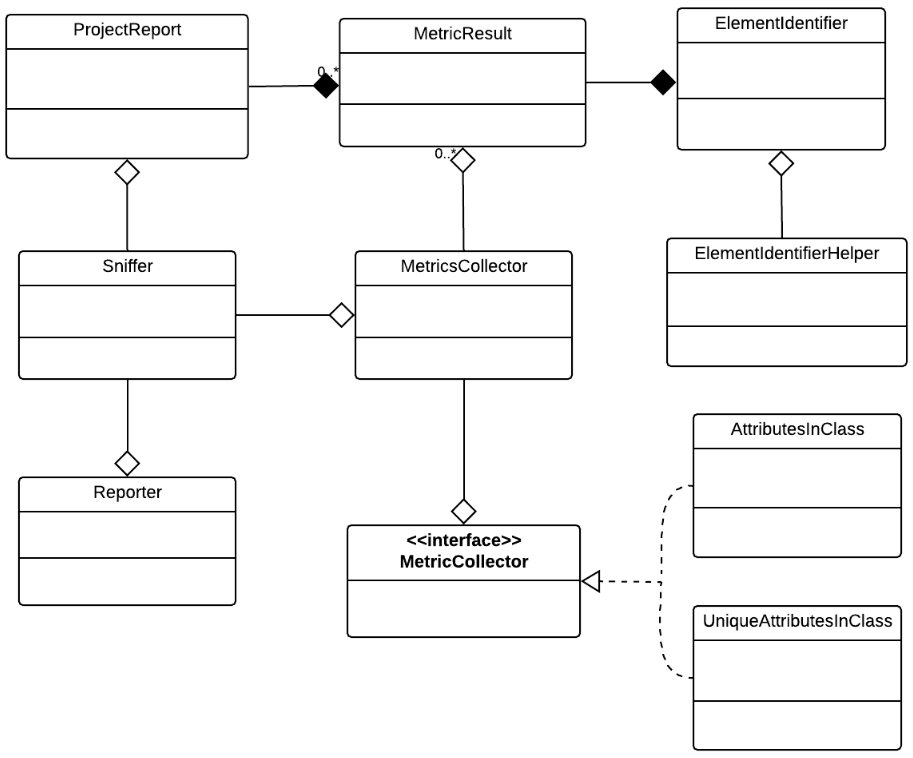

Figure 2. Attribute Sniffer UML Diagram

We exemplify the extraction process using the code for the AC metric, shown in Figure 3. Each class that represents a metric must extend the `CSharpSyntaxWalker` class, provided by Roslyn, and override the available methods that visit the AST nodes of interest. This modeling allows processing of any declaration present on the AST without previously knowing its structure. To extract the AC metric, the method used was `CSharpSyntaxWalker.VisitAttribute()` present on line 10. This method is invoked each time an attribute is found on the AST.

Classes representing metrics, such as the one in Figure 3, must also implement our custom interface `MetricsCollector` and both its methods: `SetSemanticModel` and `SetResult`. The first one includes the `SemanticModel` instance in the extraction process, to allow identifying a code element with an attribute. The logic behind this process is being handled by the `ElementIdentifierHelper` class. The latter method stores the result of the extraction process in the `MetricResult` instance, which holds the name of the extracted metric, the value, and the element identifier – this includes the namespace, type, and line of declaration. The `ProjectReport` class encapsulates every extracted metric value, containing the result for the whole project. Finally, the `Reporter` class is capable of generating a report in both XML and JSON file.

```
//imports omitted
namespace AttributeSniffer.analyzer.metrics
{
    // Visit a compilation unit to extract the AC metric.
    class AttributesInClass : CSharpSyntaxWalker, MetricCollector{
        private SemanticModel SemanticModel { get; set; }
        private int NumberOfAttributes { get; set; } = 0;
        private AttributeSyntax VisitedAttribute;

        public override void VisitAttribute(AttributeSyntax node){
            this.VisitedAttribute = node;
            this.NumberOfAttributes++;
        }
        private MetricResult GetResult(){
            ElementIdentifier elementIdentifier = ElementIdentifierHelper
                .getElementIdentifierForClassMetrics(SemanticModel,
                    VisitedAttribute.AncestorsAndSelf());

            string metricName = Metric.ATTRIBUTES_IN_CLASS.GetIdentifier();
            return new MetricResult(elementIdentifier, metricName,
            NumberOfAttributes);
        }
        public void SetResult(List<MetricResult> metricResults){
            if (ElementIdentifier != null) {
                string metricName = Metric.ATTRIBUTES_IN_CLASS.GetIdentifier();
                string metricType = MetricType.CLASS_METRIC.GetIdentifier();
                metricResults.Add(new MetricResult(ElementIdentifier, metricType,
                    metricName, NumberOfAttributes));
            }
        }
    }
}
```

Figure 3. Code for the AC metric.

4.3. Usage instruction

Potential users of the Attribute Sniffer are software engineers, researches, and developers interested in static code analysis, specifically in metadata usage. To run the tool, use the command below.

dotnet run −−project AttributeSniffer.csproj <path-project> <path-report> <single/multi> <report-type>

The first parameter is the path to a directory that contains a C# project. The second parameter is the path to store the report. The third parameter defines if the analysis will be performed in a single project or multiple projects at the same time. In the case of multiple projects, the root directory should contain a sub-folder for each project, and a report will be generated for each one. The final parameter determines the type of the report file, which can be either XML or JSON. This parameter is optional, and the default value is a JSON. In case the user runs the tool without passing in any arguments, he will be prompted to insert these before execution.

5. Conclusion

Based on the metrics proposed by [Lima et al. 2018b] for Java annotations, this paper presented a tool that collected a subset of these metrics for C# attributes. The original suite contains seven metrics, and five of them applies directly to C# attributes. As such, we can generalize these metrics, as not only annotation metrics, but rather declarative metadata metrics. The Attribute Sniffer was developed based on the Annotation Sniffer [Lima et al. 2018a], a tool to collect Java annotation metrics. As such, it also has the same minor threat, i.e., no similar tool was found to perform a comparison. Hence, all validations were done manually, which may compromise the accuracy.

As future work, the tool will extend its capabilities and also extract metrics that capture characteristics of the C# attributes not present on Java annotations. For instance, C# attributes support inheritance, may contain behavior, may be applied to whole assemblies and modules. With these new enhancements, a more thorough analysis will be carried out. The tool is available as open source at the GitHub (`github.com/LydiaBraga/AttributeSniffer`) under the GNU Lesser General Public License (LGPL V3).

Acknowledgement

This work is supported by FAPESP (Fundação de Amparo à Pequisa do Estado de São Paulo), grant 2014/16236-6.

References

Chidamber, S. R. and Kemerer, C. F. (1991). Towards a metrics suite for object-oriented design. In *Proceedings of OOPSLA'91*, pages 197–211.

ECMA (2001). *C# Language Specification*. European Computer Manufacturers Association.

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, 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, 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, P., Guerra, E., and Meirelles, P. (2018a). Annotation sniffer: Open source tool for annotated code elements. In *CBSoft 2018 - Tools Session ()*.

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

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].

Microsoft (2019). The .net compiler platform sdk (roslyn apis). Accessed: 2019-03-29.

Polančec, D. and Mekterović, I. (2017). Developing moba games using the unity game engine. In *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1510–1515.