Towards Visualizing Code Annotations Distribution

Phyllipe Lima
National Institute of Telecommunications – INATEL
National Institute for Space Research – INPE
phyllipe@inatel.br

Eduardo Guerra
National Institute for Space Research – INPE
eduardo.guerra@inpe.br

Paulo Meirelles
Federal University of São Paulo – UNIFESP
paulo.meirelles@unifesp.br

ABSTRACT

Java developers make extensive use of code annotations since their introduction in version 5 of the language. They are inserted directly on the source code for custom metadata configuration, similar to C# attributes. The software engineering community has few works investigating their usage and impact on source code. Being able to visualize characteristics of code annotations might aid developers in detecting potential misuse, outliers as well as increase the comprehensibility and readability of the source code. In this paper, we present an approach to use software metrics to generate a 2D polymetric view targeting the visualization of code annotations in Java classes. We developed a prototype tool using the Unity Game Engine. It displays classes and packages as rectangles and annotations as circles. We demonstrated the tool with a small sample Java program.

KEYWORDS

Metadata, Visualization, Annotations, Java

1 INTRODUCTION

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 previous work [1] proposed a novel suite of software metrics dedicated to code annotations, and to obtain threshold values from a Percentile Rank Analysis approach [2]. A version for the C# attribute was also performed by [3].

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 [4, 5]. Using metrics values as mere numbers presented on tables might not be very useful in software comprehension. Also, having to inspect source code to understand these values requires effort. As some systems may have hundreds of thousands of lines that are poorly documented, there is a need for practical approaches to aid developers in software understanding and bug detection [6].

Software visualization is being extensively investigated and successfully adopted to deal with existing software and to improve software comprehensibility [7, 8]. Lanza and Ducasse [6] proposed the polymetric view concept, which is a lightweight visualization of software entities enriched with software metrics. It allows the quick viewing of multiple software metrics values in a single entity. For instance, the measures of a displayed rectangle could be defined in terms of the LOC and number of methods in a class.

In this paper, we propose a 2D polymetric view for the visualization of code annotations characteristics. The goal is to provide an intuitive way to visually extract information about annotations and metadata configuration in software projects. Packages and classes are represented as rectangles, while annotations as circles. Values such as area, radius, color, and position are defined by metrics values. We developed a prototype tool named Annotation Visualizer (AVisualizer), using the Unity Game Engine to implement our visualization design. To demonstrated it, we created a small sample project with seven classes and annotations from different frameworks.

This paper is organized as follows. Section 2 presents the concepts of metadata in the context of object-oriented programming and how annotations are used. Section 3 presents the related work. Section 4 describes the suite of software metrics dedicated to code annotations. Section 5 describes our proposal for code annotations metrics visualization and how the tool was developed. Section 6 concludes the paper and presents future works.

2 METADATA AND CODE ANNOTATIONS

The term “metadata” is used in a variety of contexts in the computer science field. In all of them, it means data referring to the data itself. When discussing 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 [9].

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. Afterward, a framework or tool consumes them and executes the desired behavior. For instance, metadata can be used to generate source code [10], compile-time verification [11], framework adaptation [12], perform object-relational mapping, object-XML mapping and so forth.

Custom metadata can be configured using external storage, such as a database or an XML file [13]. 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 [14], used by the Ruby on Rails [15] and the CakePHP framework[1]. 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 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 [16], and in C#, by attributes [17]. A benefit is that the metadata definition is closer to the programming element, and its definition is less verbose than external approaches. Also, the metadata is being explicitly defined in the source code as opposed to code convention approaches. Some authors call the usage of code annotations as attribute-oriented programming since it is used to mark software elements [18, 19].

Annotations are a feature of the Java language, which became official on version 1.5, spreading, even more, the use of this technique in the development community. Some base APIs, starting in Java EE 5, like EJB 3.0 and JPA, use metadata in the form of annotations extensively. This native support to annotations encourages many Java frameworks and API developers to adopt the metadata-based approach in their solutions. They were also a response to the tendency of keeping the metadata files inside the source code itself, instead of using separate files [20].

```java
@Entity
@Table(name="Players")
@Component //Used by Spring Framework
public class Player {

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

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

    @Column(name = "birthdate", nullable = false)
    private Date birthDate;
    //getters and setters omitted
}
```

Figure 1: Example Java Class with Annotations

Consider the code on Figure 1. It is a simple Java class representing a player from a video game code. To map this 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.

[1]cakephp.org

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, which occurs as described below:

- The class `Player` is mapped as an Entity and to a table named `Player`
- The private member `id` is mapped into a primary key on the table.
- The members `health, name, and birthDate` are all mapped to columns.
- The `@Component` makes the `Player` class a bean managed by the Spring Framework. It is not related to the object-relational mapping.

Another important definition is that of an annotation schema, defined as a set of associated annotations that belong to the same API. The annotations used in the example code are part of the JPA schema, with the exception of `@Component`, which belongs to the Spring framework. An annotation-based API usually uses a group of related annotations that represent the set of metadata necessary for its usage [1].

3 RELATED WORK

Wettel and Lanza [21] proposed a 3D visualization based on the city metaphor, i.e., an object-oriented software is represented as a city that can be visited and interacted with. The authors opted for this metaphor since it is by far the most adopted one and provides a suitable means to transmit a good sense of habitability and locality. Areas such as downtown and suburb are a familiar concept to potential users of this tool.

To demonstrate and exemplify their visualization design, they developed a tool named *CodeCity* that allows the user to interact and visit "cities" that represent software. Classes are rendered as buildings and packages as districts. The authors used three large software systems, such as ArgoUML, Azureus, and VisualWorks Smalltalk, to assess their proposal. The tool was able to scale to accommodate these software projects and allowed interaction features such as selection, filtering, and visual tagging of the displayed elements.

The city metaphor also gained a Virtual Reality (VR) version [8], and a new tool Code2City was developed to support VR. The authors conducted experiments comparing three different approaches to software visualization. They use of Code2City (i), and the VR version (ii) displayed on a regular computer screen. They also used a plugin for the Eclipse IDE named *Metrics and Smells* that collects metrics and detect bad smells (iii). As other research results, the authors presented that the city metaphor increases software comprehension, and users using the VR version concluded the experiment tasks more quickly and were more satisfied.

The Forest Metaphor [22] depicts software as a forest of trees. Each tree is rendered as a class. Trunks, branches, leaves, and their color indicate characteristics of classes (for instance, a method is a branch of the tree that, in turn, represents a class). They created a prototype tool named CodeTree to implement this metaphor.

Francese et al. [7] proposed a polymetric visualization approach to object-oriented software. The authors implemented a prototype tool, executing as an Eclipse Rich Client Platform (RCP) application. The tool was named *MetricAttitude*. It provides a large-scale understanding of a software system, visualizing all classes together, and handles class relationships.

The work of [23] proposes a SOM (Self-Organizing Map) to classify Java classes of open source projects based on code annotations usage. Annotation metrics [1] values were used to generate the map. The authors were able to identify three different groups of classes based on code annotations.

The work that we are proposing in this paper is not based on any metaphor, but rather a simple 2D polymetric view. We are investigating how to visualize code annotations metrics values from Java projects. To the best of our knowledge, no prior work investigates visualizing such information.

4 ANNOTATION METRICS

This section describes the suite of annotation metrics proposed by [1]. Our proposed polymetric view uses these metrics values as input to generate our visualization. The Java source code in Figure 2 is used to clarify the usage of the metrics further.

```
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 2: Code for candidate metrics examples.

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 11.

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, the annotation `@AssociationOverride` on line 10 is different from the one on line 12, 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. Notice that the annotations on lines 21, 25, and 28 are calculated only once for they are equal.

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 since different schemas on a class imply the class is using different frameworks. 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.

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, on line nine the `@AssociationOverrides` has only one argument "`value`", so the AA value is equal one. But `@AssociationOverride`, on line 10, contains two arguments, `name` and `joinColumns`, so the AA value is two.

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 two, it has the `@TransactionAttribute` and `@DiscriminatorColumn`

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` on line 18 has ANL value of zero, while `@JoinColumn` on line 11 has ANL equals two. This data is because it has `@AssociationOverride`, line 10, as a first level, and then the `@AssociationOverrides`, line nine, adds another nesting level, hence the value ANL is two.

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` on line nine has a LOCAD value of five, while `@NamedQuery`, line 14, has LOCAD equals four.

Number of Elements (NEC). This metric measures the number of elements that can be annotated in a class, i.e., the number of programming elements that can potentially be configured with code annotations. In the example class we have three methods, `exampleMethodA`, `exampleMethodB` and `exampleMethodC`, and the class declaration `Example`. Hence we have an NEC value of four.

5 VISUALIZING CODE ANNOTATIONS

Being able to visualize software can aid in detecting bad smells, improve code readability, comprehension, monitor code complexity, and identify misuse [7, 8]. Specifically for code annotations, developers are usually using more than one metadata-based framework, i.e., dealing with multiple annotations schemas. As such, poorly designed Java classes might be overloaded with annotations from several different schemas, bringing impact to the readability, evolution, and maintenance of such classes. Furthermore, there are indications of classes with a high number of annotations (AC metric), a high number of schemas (ASC metric), and annotations with several lines used in their declaration (LOCAD value). These findings suggest the existence of outliers and bad smells related to code annotations [1].

We argue that being able to visualize code annotation metrics values might aid developers in identifying potential misuse, outliers, and metadata-configuration issues in their code. To overcome this, we present in this section our proposed visualization.

5.1 The Visualization Design

Our proposed design uses five source code metrics values - LOC, NEC, ASC, AA, and AED - to generate the visualization of the packages, classes, and annotations. The latter is represented as a circle, while the other two are rectangles. The circles (annotations) will be rendered inside the rectangles (classes), overlapping them, since it is 2D. The class-rectangles will overlap the package-rectangle. This approach also reinforces that annotations are written inside the class, and we want to emphasize that on our visualization. The measurement unit being used is centimeter (cm). To exemplify our visualization, we will use the class in Figure 1.

The width (horizontal) of the rectangle is the class NEC value. The example class in Figure 1 has an NEC value of five, so the rectangle will have a width of five centimeters. The height (vertical) is obtained dividing the class LOC value by a normalization factor called CHN (Class Height Normalization). Equation 1 shows how the CHN factor is calculated for the project. The MAX_AED is the highest AED value, considering the whole project, while MAX_LOC is the LOC value of the class that contains this element. In case of a tie for the number of annotations, i.e., the AED value, we will use the lowest LOC value. Using this technique, we guarantee that classes with high AED values will still have their annotations rendered correctly. In other words, the circles will be drawn with adequate proportions.

$$CHN = \frac{MAX\_LOC}{MAX\_AED} \tag{1}$$

Consider that the class in Figure 1 belongs to a project where it has the highest number of annotations configuring a single element, i.e., the AED metric value is the highest. The class declaration, `Player`, has three annotations - `@Entity`, `@Table` and `@Component` - so MAX_AED is three. The LOC value for the class is 18 (for this demonstration, consider the total absolute number of lines). Hence CHN is equal to 18/3 = 6. For this project, every rectangle height will be the LOC value of the class being rendered divided by six, which is the CHN factor for the current project. For the example class, the height of the rectangle will be 18 (LOC) / 6 (CHN) = 3 (cm).

As mentioned, the annotations are rendered as circles overlapping the rectangles (classes). The diameter of these circles depends on the AA value of the annotation, and the MAX_AA value of the project normalizes it. In other words, the width of the circle depends on the number of arguments declared inside the annotation (AA), and can be calculated as shown in Equation 2.

$$Diameter\left(cm\right) = \frac{AA\_VALUE + 1}{MAX\_AA + 1} * 0.9 \tag{2}$$

Annotations with an AA value equal to the MAX_AA value will be rendered with a diameter of 0.9 cm, which represents the largest possible circle. We multiply the diameter value by 0.9 to leave some space between circles (padding value). As the AA value decreases, so will the diameter, rendering smaller circles. Developers using our proposed visualization will be able to quickly differentiate annotations with different AA values, simply looking at the circles. In the example class on Figure 1 the MAX_AA value is two, given that the annotation `@Column` on line 15 has two arguments. This annotation will be rendered as a circle with a diameter of 0.9 cm, as well as every other annotation with an AA value of two. Annotations with smaller values of AA will have the diameter decreased following the Equation 2.

Another characteristic of the circles is their color, which varies according to the schema they belong to. For this prototype, we are adopting the color red for JPA annotations and blue for Spring annotations. The example class in Figure 1 contains only one Spring annotation, so it will be rendered blue, and all other circles will be red. Using this method, developers can quickly visualize different annotation schemas in the class.

The circles are rendered in a grid fashion. The width of the rectangle determines the number of columns (NEC value), and the height determines the number of rows (LOC value normalized by CHN). All annotations configuring the same code element will be drawn in the same column. Figure 3 presents the visualization for the class on Figure 1. The largest circle has an AA value of two since the others are smaller; we can easily assure that they represent annotations with less than two arguments. Where the AED value indicates the number two, means this column represents a code element with two annotations configured.

To give a broader view of our visualization approach, Figure 4 presents a simple fictional Java project being drawn. It has two packages, each with three classes. Packages are

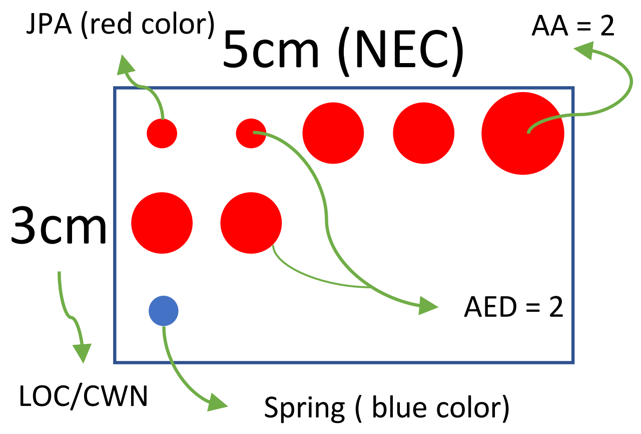

Figure 3: Example of the Proposed Visualization for a Single Class

drawn as rectangles with the names on top, left-aligned. On our initial proposal, each package renders a maximum of three classes horizontally, with a subsequent trio of classes being drawn on new lines. The package-rectangle height and width are calculated based on the class with the highest LOC and NEC values. This approach guarantees that the classes-rectangles will be rendered smoothly. Each class is represented as a rectangle overlapping the package-rectangle, with its name on top, left-aligned. This approach is intuitive since Java projects have the same behavior, i.e., classes reside inside packages.

Observing Figure 4, we can quickly see that in package *pkg1* `Class1` has the highest NEC and LOC value, it has annotations from two different schemas, one code element has three annotations, and so forth. Meanwhile, `Class6` has only two annotations; by the size, we can conclude that they have no arguments since as the number of arguments increase, so does the diameter of the circles. Both these annotations belong to the Spring schema (blue color). Analyzing package `pkg2`, we see that `Class3` has the highest AC value and that the majority of the annotations are concentrated on a single code element. On the other hand, `Class4` has the lowest AC value in the package, and the annotations have low AA values, i.e., a small number of arguments. Empty columns represent elements with no annotations configuring them.

5.2 Annotation Visualizer

The Annotation Visualizer (AVisualizer) is an open-source tool [2] developed using the Unity Game Engine[3]. Game Engines are frameworks dedicated to game development. They control the execution of the application by implementing the game loop and are also equipped with libraries extensively used by game developers such as physics, artificial intelligence, visual effects, user interface, graphics programming, and so forth. Unity is programmed using the C# language and is one of the most popular game engines, used across the globe [24]. Recently, researches and developers are also using Unity for non-game purposes, such as simulation of systems using UAV (unmanned aerial vehicle) for security [25] and simulation of future cities [26]. We chose the Unity Game Engine for this prototype version of the tool to also investigate if using a Game Engine suits the needs for 2D data visualization tools.

The AVisualizer requires, as input, the XML file provided by the ASniffer [27]. This XML file contains all annotation metrics values as well as the LOC value. The tool begins performing an XML-object parsing to an instance of the class `ProjectContainer`, which will hold all the information about the project, such as packages, classes, and annotations that will be drawn. After the parsing is completed, the `SpawnManager` class reads all data in the `ProjectContainer` and initiates the spawning process. Rectangles and circles will be rendered following the design presented in the previous section.

To demonstrate the prototype tool, we created a small example project. It contains seven classes and two annotation schemas, JPA and Spring. We collected the needed metrics values, and the AVisualizer used these values to generate the visualization presented in Figure 5. This figure is a screenshot taken directly from Unity, i.e., the prototype tool is running from inside Unity Game Engine.

In this figure, we see two packages drawn as white rectangles, seven classes, drawn as green rectangles, and annotations drawn as circles. If there were more packages, they would be drawn below, so the user has to scroll down or zoom out to analyze more packages. We quickly see that `Class1` and `Class3` has higher values for LOC and NEC. `Class3` also has a concentration of annotations on a single code element. Some of these annotations have a high number of arguments (this is seen by the diameter of the circles).

Observing `Class6` and `Class4`, they are small classes with only two annotations from the Spring framework (blue color). A developer analyzing this visualization, could potentially identify that these two classes should belong in the same package, for code organization purposes. Or, if there was a red circle (meaning a JPA annotation) in `Class4`, the developer could further analyze the source code to reassure that the configuration is indeed adequate.

The visualization shows that `Class7` and `Class2` are similar in terms of LOC, NEC and metadata-configuration. Further inspecting the code, the developer can potentially conclude that these classes could be merged, or they could share the same abstraction and refactor towards an adequate design pattern in the project.

6 CONCLUSION

In this paper, we proposed an approach to visualization design, in terms of a polymetric view for code annotation metrics, and implemented it through the AVisualizer, an open-source tool developed using the Unity Game Engine.

[2]gitlab.com/phillima/vem_2019_proto
[3]unity.com

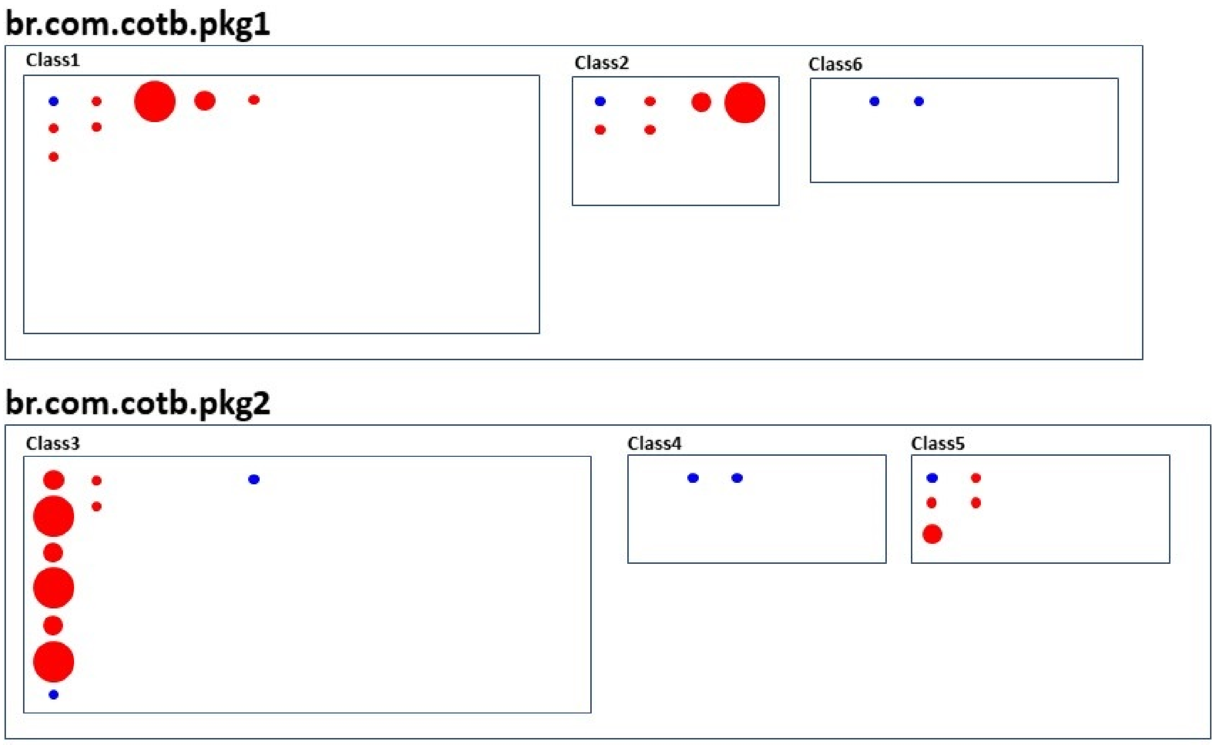

Figure 4: Example of the Proposed Visualization for a Sample Project with six classes

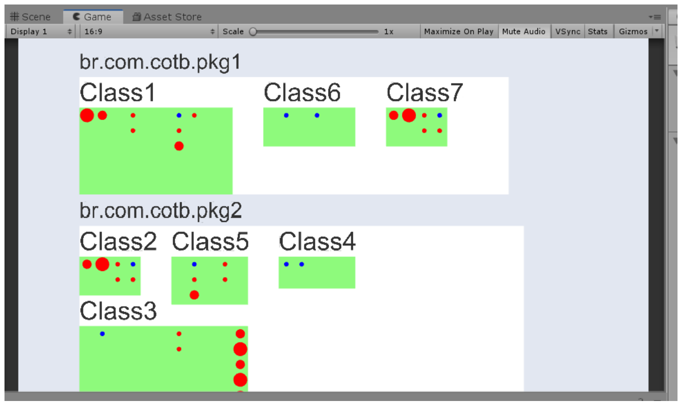

Figure 5: Output example of the Annotation Visualizer for a sample project with seven classes

The tool reads code annotation metrics values and renders classes and packages as rectangles and annotations as circles. The measures of the rectangle are based on the number of elements and the LOC value of the class, while the circles vary color, radius, and position according to annotation metrics values. We demonstrated the visualization running the tool on a small sample project, where it was possible to quickly visualize specific annotations characteristics, such as arguments, schemas, and the code elements it configures.

Regarding the Unity Game Engine, it was not trivial to use it as a development environment for a static 2D visualization tool. Much of the Unity's available functionality are suited for Game Development, i.e, software based on the Game Loop Pattern [28]. Furthermore we realized that using Unity to develop our tool made using the AVisualizer consume much more resources (memory and cpu) than a usual software engineering tool should. This could be a major draw back to have potential researches and developers adopting our tool to monitor their source code. The tool should be lightweight and have minimal requirements to run. As such, we will proceed the development of the AVisualizer using the D3[4].

This work is ongoing research, and we intend to cover more annotations metrics from the suite, as well as improve the tool. For instance, we have the metric ANL (Annotation Nesting Level), that measures how deep an annotation is nested inside another one. In our visualization design, we could represent this as the transparency of the circle, where the deeper an annotation is nested, the more transparent it becomes. Another metric to cover is the LOCAD (LOC in Annotation Declaration), and could be visualized as the color intensity, where the lighter the color, the more lines of code was used to declare the annotations. Consider a JPA annotation that is opaque and dark red. This would translate in ANL of zero and LOCAD of zero as well. Being able to quickly visualize all of these metrics values might aid developers to quickly detect problems related to code annotations, that could otherwise be harder to find by inspecting the source code. We also have other aspects to explore, such as:

Scalability. The tool will need to operate on very large software systems and provide visualization for them. The design will be improved to accommodate such systems. One preliminary option is to create three layers of visualization, with different granularity. For instance, a system, package, and class layer. The current version is implementing a package layer.

User Interaction. Users of AVisualizer will need to interact with shown elements to obtain more data from them. The tool also needs to offer mechanisms so users can navigate the visualization, interact with different layers, and highlight specific characteristics. For instance, the user might want to highlight a particular annotation and be able to see every other class with that same annotation quickly.

Tool Execution. Currently, to run the AVisualizer, the user requires the Unity Game Engine installed since we are still prototyping the tool. As it evolves, we will create executable standalone versions for Desktop and Web, using the Unity Game Engine, since it supports more than 20 platforms.

Evaluation. We intend to execute controlled experiments with professional software developers and also students. We will prepare questionnaires and tasks to verify the effectiveness of the tool. For instance, we will have two different groups performing the experiments, with one group using the AVisualizer and the other one merely inspecting the source code. We want to find out if the tool is indeed able to improve code readability, comprehension, and task completion, regarding code annotations. We will also execute the tool on real-world projects to help validate the tool.

REFERENCES

[1] Phyllipe Lima, Eduardo Guerra, Paulo Meirelles, Lucas Kanashiro, Hélio Silva, and Fábio Silveira. A metrics suite for code annotation assessment. *Journal of Systems and Software*, 137:163 – 183, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2017.11.024. URL http://www.sciencedirect.com/science/article/pii/S016412121730273X.

[2] Paulo Roberto Miranda Meirelles. *Monitoring Source Code Metrics in Free Software Projects.* PhD thesis, Department of Computer Science – Institute of Mathematics and Statistics of University of São Paulo, 2013. URL http://www.teses.usp.br/teses/disponiveis/45/45134/tde-27082013-090242/pt-br.php. [in portuguese].

[3] Lydia Braga, Phyllipe Lima, Eduardo Guerra, and Paulo Meirelles. Attribute sniffer: Collecting attribute metrics for c# code. In *CBSoft 2019 - Sessão de Ferramentas ()*, sep 2019. doi: https://doi.org/10.5753/cbsoft_estendido.2019.7664.

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

[5] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. 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, 2007. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4359471.

[6] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sep. 2003. ISSN 0098-5589. doi: 10.1109/TSE.2003.1232284.

[7] Rita Francese, Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. Proposing and assessing a software visualization approach based on polymetric views. *Journal of Visual Languages & Computing*, 34-35:11 – 24, 2016. ISSN 1045-926X. doi: https://doi.org/10.1016/j.jvlc.2016.05.001. URL http://www.sciencedirect.com/science/article/pii/S1045926X16300623.

[8] Simone Romano, Nicola Capece, Ugo Erra, Giuseppe Scanniello, and Michele Lanza. On the use of virtual reality in software visualization: The case of the city metaphor. *Information and Software Technology*, 114:92 – 106, 2019. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2019.06.007. URL http://www.sciencedirect.com/science/article/pii/S0950584919301405.

[9] Eduardo Guerra. *Componentes Reutilizáveis em Java com Reflexão e Anotações.* Casa do Código, 1st edition, 2014. ISBN 978-85-66250-50-3. [in portuguese].

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

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

[12] Eduardo M. Guerra, Jerffeson T. de Souza, and Clovis T. Fernandes. 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, 2010. ACM. ISBN 978-1-60558-873-5. doi: 10.1145/1943226.1943230. URL http://doi.acm.org/10.1145/1943226.1943230.

[13] Clovis Fernandes, Douglas Ribeiro, Eduardo Guerra, and Emil Nakao. Xml, annotations and database: a comparative study of metadata definition strategies for frameworks. *May 19–20, Vila do Conde*, page 115, 2010.

[14] N. Chen. Convention over configuration, 2006. URL http://softwareengineering.vazexqi.com/files/pattern.html.

[15] Sam Ruby, Dave Thomas, and David Hansson. *Agile Web Development with Rails, Third Edition.* Pragmatic Bookshelf, 3rd edition, 2009. ISBN 1934356166, 9781934356166.

[16] JSR. Jsr 175: A metadata facility for the java programming language, August 2004. URL http://www.jcp.org/en/jsr/detail?id=175.

[17] ECMA. Ecma - 334: C# language specification, December 2017. URL https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

[18] Hiroshi Wada and Junichi Suzuki. Modeling turnpike frontend system: A model-driven development framework leveraging uml metamodeling and attribute-oriented programming. *Model Driven Engineering Languages and Systems*, pages 584–600, 2005. URL http://www.springerlink.com/index/l166363337837142.pdf.

[19] Don Schwarz. Peeking inside the box: Attribute-oriented programming with java 1.5, part, 2004. URL http://archive.oreilly.com/pub/a/onjava/2004/06/30/insidebox1.html.

[20] Irene Córdoba-Sánchez and Juan De Lara. Ann: A domain-specific language for the effective design and validation of java annotations. *Computer Languages, Systems & Structures*, 45:164 – 190, 2016. ISSN 1477-8424. doi: https://doi.org/10.1016/j.cl.2016.02.002. URL http://www.sciencedirect.com/science/article/pii/

[4]d3js.org

S14778424163003l8.

[21] Richard Wettel and Michele Lanza. Visualizing software systems as cities. pages 92–99, 06 2007. doi: 10.1109/VISSOF.2007.4290706.

[22] Ugo Erra and Giuseppe Scanniello. Towards the visualization of software systems as 3d forests: The codetrees environment. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 981–988, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0857-1. doi: 10.1145/2245276.2245467. URL http://doi.acm.org/10.1145/2245276.2245467.

[23] Phyllipe Lima, Eduardo Guerra, and Paulo Meirelles. 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*, july 2017. [in portuguese].

[24] D. Polančec and I. Mekterović. 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, May 2017. doi: 10.23919/MIPRO.2017.7973661.

[25] Shabnam Sadeghi Esfahlani. Mixed reality and remote sensing application of unmanned aerial vehicle in fire and smoke detection. *Journal of Industrial Information Integration*, 2019. ISSN 2452-414X. doi: https://doi.org/10.1016/j.jii.2019.04.006. URL http://www.sciencedirect.com/science/article/pii/S2452414X18300773.

[26] Pablo Garcia-Aunon, Juan Jesús Roldán, and Antonio Barrientos. Monitoring traffic in future cities with aerial swarms: Developing and optimizing a behavior-based surveillance algorithm. *Cognitive Systems Research*, 54:273 – 286, 2019. ISSN 1389-0417. doi: https://doi.org/10.1016/j.cogsys.2018.10.031. URL http://www.sciencedirect.com/science/article/pii/S1389041718303279.

[27] Phyllipe Lima, Eduardo Guerra, and Paulo Meirelles. Annotation sniffer: Open source tool for annotated code elements. In *CBSoft 2018 - Tools Session ()*, sep 2018.

[28] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 1st edition, 2014. ISBN 9780990582908. [in portuguese].