Revista de Sistemas de Informação da FSMA
n. 30 (2022) pp. 28-37

SISTEMAS DE INFORMAÇÃO
http://www.fsma.edu.br/si/sistemas.html

TRILHA PRINCIPAL

Annotation Visualizer Plugin: An IDE-Integrated Tool for Code Annotations Visualization

Sergio Abilio, *Federal University of ABC, São Paulo, Brazil*
Phyllipe Lima, *Federal University of Itajubá, Brazil*
Everaldo Gomes, *Federal University of ABC, São Paulo, Brazil*
Eduardo Guerra, *Free University of Bolzano, Italy*
Paulo Meirelles, *Federal University of ABC, São Paulo, Brazil*

***Abstract*—Code annotations are used mainly in modern Java software, and their primary purpose is to configure metadata into programmable elements such as methods and classes. The Annotation Sniffer is a stable tool that analyses Java source code and generates a JSON report with code annotations metrics values. In other words, it measures metadata configuration in the Java source file. The Annotation Visualizer is a tool built as a web application to render the visualization using this report as input. In this work, we developed a tool that integrates the Annotation Visualizer as a plugin for the IntelliJ IDEA, a popular Java IDE. This integration allows developers to visualize their current project opened in the IDE without switching to an external environment. With an IDE plugin, developers can potentially benefit from the visualization and the metrics values it can display. Furthermore, IDEs have a marketplace that eases obtaining the tool and increases the probability that the developers might use it.**

***Index Terms*—Annotation Code, Annotation Metrics, Software Visualization, Visualization Tool**

I. Introduction

CODE annotations were introduced in version 5 of the Java programming language. It is extensively used in many Java enterprise software systems through frameworks and APIs such as Spring Boot, EJB, JPA, and JUnit. They add custom metadata into programmable elements such as methods, fields, and classes.

Our previous work [1] proposed and validated a suite containing seven source code metrics dedicated to code annotations to improve and further enable the study and analysis of code annotations. By mining open-source software hosted on GitHub, we identified that, on average, 76% of their classes have at least one annotation. We developed the ASniffer (Annotation Sniffer) tool to extract these metrics values [2].

Combining software metrics with software visualization, we proposed, in another previous work [3], the CADV (Code Annotations Distribution Visualization), a circle packing approach to visualize code annotations distribution in software systems. As a reference implementation, we developed the AVisualizer (Annotation Visualizer), a web application that reads the metrics values extracted with the ASniffer and draws the CADV for a target project under analysis.

During the evaluations conducted in [3], developers were interested in using the AVisualizer and thought the visualization approach could be helpful in the software comprehension process. However, they were not interested in leaving their development environment (usually an IDE) and switching to a browser to analyze the system. Furthermore, in the current solution, the developers must manually run the ASniffer to collect the metrics values and serve the AVisualizer.

This paper presents the AVisualizer Plugin[1], an open-source plugin for IntelliJ IDEA IDE, which integrates the ASniffer and AVisualizer in a single tool to overcome that constraint. It runs integrated into IntelliJ, a popular development environment for Java programmers. The plugin uses, as a dependency, the ASniffer to collect the project metrics in the current working directory. Afterward, it sends the extracted data to a service integrated with a web application to store reports. Finally, it renders the CADV via the AVisualizer running in an embedded browser provided by IntelliJ. The plugin can be obtained directly from the official Jetbrains Marketplace and installed on the latest version of the IntelliJ IDEA[2].

The Jetbrains Marketplace provides some insights about the usage of the available plugins. The AVisualizer plugin already had hundreds of downloads and was visited by users in several countries. Moreover, we conducted a preliminary validation and invited developers from the study in [3] to use the plugin and answer some code comprehension tasks. Two participants used the plugin and gave feedback that it was easy to install, and they were more inclined to use it embedded in the IDE rather than switching to another environment.

The remainder of this paper is organized as follows: Section II describes the software development engineering methodologies used in this work and how we structured the development cycles. Section III contains an analysis of

[1]https://github.com/metaisbeta/intelliJ-avisualizer-plugin
[2]https://plugins.jetbrains.com/plugin/18237-annotation-visualizer

how annotations are measured and visualized, introducing the related works used to serve as a basis for our work. In Section IV, we discuss how the technological system works and which improvements were implemented in the first version of both ASniffer and AVisualizer. In Section V, we present a validation on the plugin with users familiar with the ASniffer and AVisualizer. Finally, Section VI presents our final thoughts and points to be discussed in future works.

II. Development approach

OUR development approach was based on the collaboration flow for Free/Libre/Open Source Software (FLOSS) projects and the principles applied in the Agile Software Development Manifesto, in particular from XP (*Extreme Programming*) methodology [4]. XP is an agile software development framework aiming to produce high-quality software and higher quality of life for the development team. We based on the principles of iteration planning, acceptance of testing, listening to feedback, and constant small releases.

In our approach, the team responsible for this project meets biweekly to carry out the software development ceremonies. Each meeting was interpreted as a sprint, a fixed-length event that enables predictability by ensuring inspection and adaptation of progress toward a Product Goal at least every calendar month. At the end of each sprint, each member reviews the work done in the period and shares thoughts involving the development of the Annotation Sniffer ecosystem. Each meeting had a Planning and a Demo lasting for approximately one hour.

The "Planning" is a ceremony that kicks off the sprint. Sprint planning aims to define what can be delivered during the sprint and how it can be achieved. The team prioritized the features and decided which would be delivered in each sprint to reach the first version of our Minimum Valuable Product (MVP). After delivering the MVP, the order of priority changed to improve the user experience based on the team's feedback and insights generated during product development. The "Demo" event demonstrated the work carried out in the last sprint. As for the development of this work, the developed features were shown in each demo. This way, everyone on the team could share feedback and help define the next steps.

During the software development cycle, we used the GitFlow Workflow[3] to deliver our features to the code repository. We created a *feature branch* to implement each feature selected during the sprint planning ceremony. These branches carry a source code version containing the implementation of the new feature. This code version was demonstrated at the Demo ceremony. If the maintainers approved the change, pull requests were opened to merge these branches into the main one, and the maintainers had another opportunity to review the code. If the maintainers approved, these branches were merged, producing a new release. If the maintainer requested changes, they would be implemented and verified in another demo ceremony.

We created a fork to contribute to another previously developed tool from the Annotation Sniffer ecosystem. Forking is a git clone operation executed on a server copy of a project's repository. This method is commonly used in public FLOSS projects. During the forking workflow, a project version is created on the contributor's behalf (fork), and the changes are carried out in this copy. When the work is done, a pull request can be made to the origin repository. The maintainers can review it, and the fork feature can be added to the source code if approved.

Once the code is reviewed and the pull request is approved, the automatic tests arranged by the GitHub platform's Continuous Integration (CI) server are executed. These tests automatically validate that the software operation is not compromised. After validation, the code is placed in the main branch and deployed to the specified cloud provider.

Finally, to verify how developers perceived the developed solution, we invited professional developers with previous experience using the AVisualizer web applications to participate in a survey.

III. Measuring and Visualizing Code Annotations

OUR research group proposed a novel suite of metrics for code annotations [1] and developed the ASniffer tool [2] to extract these metrics values. Moreover, we proposed a software visualization approach named CADV and the AVisualizer reference implementation, developed as a web application, to visualize code annotations [3]. Considering the need to evolve this ecosystem into an embedded solution for an IDE, we developed the AVisualizer plugin. In the following subsections, we discuss how we measure annotations and how we visualize these measurements.

A. *Visualizing Software Through Metrics*

Software systems are getting more complex with time. It can become enormous. For instance, the well-known Hibernate[4], a metadata-based framework for object-relational mapping, has more than 40 thousand classes and more than 550 thousand lines of code. This vast system is an example of an amount of information challenging to manage and comprehend.

Software comprehension is essential to keep software engineers productive [5]. Understanding and maintaining the software enables the team to share knowledge internally and empowers newcomers to follow up more quickly on the current tasks and team objectives. Software comprehension is also related to software evolution, recognized as the most costly and challenging activity in the software development life cycle. For software systems to maintain

[3]https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[4]https://github.com/hibernate/hibernate-orm

relevance, they might require constant changes, and software engineers should understand how this software works before changing it and adding new features [6].

Modern IDEs use graphical Interfaces to communicate with the user and share information about the code and its development pipeline. IntelliJ IDEA, for example, has a visual system to manage branches and version control in git[5]. This visualization feature makes code versioning easier for new developers. It aids in understanding the merging of branches and how they should keep up with the git-flow standards.

Even though some tools help the developer with graphical interfaces, the software is complicated to visualize because it is not inherently embedded in space, which means it has no ready geometric representation [7]. Therefore even though most information is perceived visually, software systems have no natural shape, complicating its visualization. For this reason, researchers keep investigating how software can be displayed so that humans can visualize and understand the underlying structure.

The visualization approach that this work is based on is the CADV (Code Annotations Distribution Visualization) [3]. The CADV is based on the circle packing and, therefore, displays everything as a circle. In other words, packages are circles, and classes are circles. Code elements and code annotations are also circles. Different colors, outlines, and some elements are only visible in specific views to differentiate them. The area of each leaf circle in a circle-packing diagram is proportional to a given number. For CADV, this is a code annotation metrics value, depending on what the users wish to see. Figure 1 presents a basic circle packing [8].

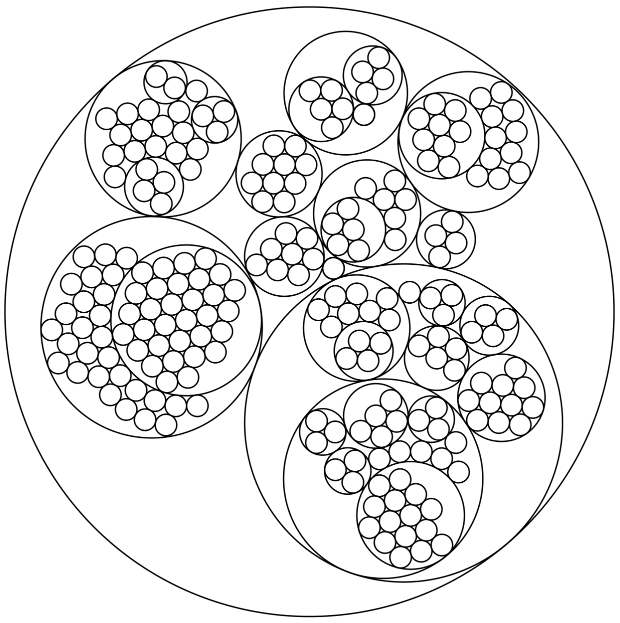


Fig. 1. Basic Circle Packing

B. Code Annotation Metrics

The visualization approach, CADV, requires five code annotations metrics from Lima et al. [1], briefly described in the following list.

Annotations in Class (AC): This metric counts the number of annotations declared on all code elements in a class, including nested annotations.

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 or if it is coupled to several different frameworks. The ASniffer measures this value by tracking the imports used for the annotations, and the higher the number of imports, the more schemas are required by the class.

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. Each annotation in the class generates an AA value.

Annotations in Element Declaration (AED): Code elements may contain several annotations. The AED metric counts how many annotations are configuring a given code element, including nested annotations.

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, but it counts the number of lines used in an annotation declaration.

C. Annotation Sniffer

The ASniffer's primary goal is to extract the code annotation metrics values from Java source code. It receives a Java source code as input, i.e., a ".java" file, extracts the metrics values, and outputs a JSON report.

It is a standalone tool executed through a command line. It also has an Application Programming Interface (API) hosted at Maven Central[6], which makes it easy to use the ASniffer as a library in any Java software. Therefore, the ASniffer can be used as a standalone tool or a dependency. The plugin developed for this work used the ASniffer as a dependency.

Potential ASniffer users are software engineers or researchers interested in static code analysis and mining software repositories. Additionally, since it is an extensible tool, other developers can implement their metrics and integrate them into the extraction process.

The Annotation Sniffer tool uses the Java Parser API[7] to build the Abstract Syntax Tree (AST) from a text file containing the source code. The ASniffer then traverses this AST, visiting the nodes of interest and gathering information about the code elements, specifically code annotations usage and the elements they are configuring.

[5]https://www.jetbrains.com/help/idea/settings-version-control.html

[6]https://mvnrepository.com/

[7]https://javaparser.org

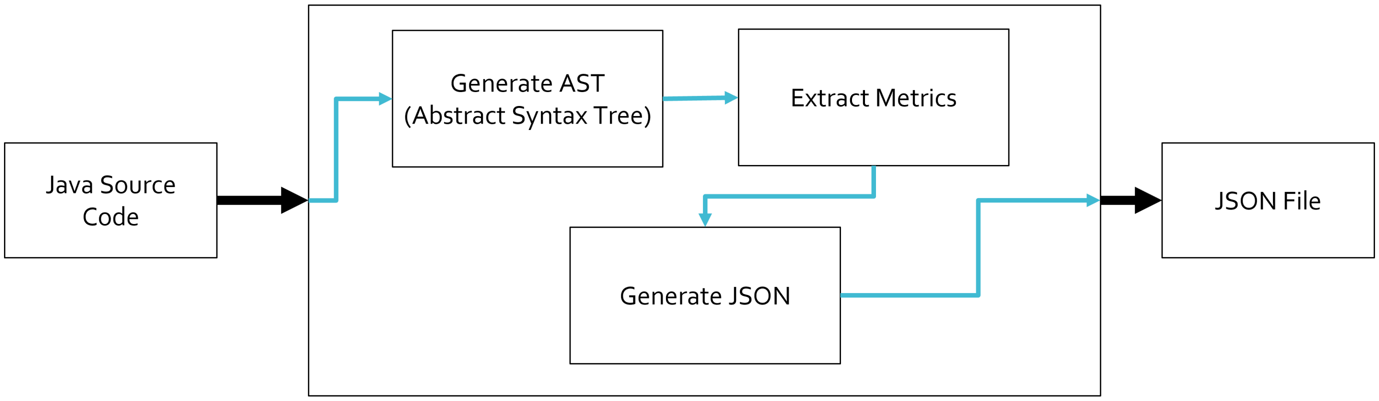


Fig. 2. Annotation Sniffer Diagram.

After completing the process, it generates a JSON report as an output. Figure 2 presents an overview diagram of the ASniffer tool [2].

The ASniffer has become a basis tool for our research on code annotations. For this reason, it is constantly improved and updated to meet the demands we require. Furthermore, being FLOSS and its source code available on Github[8], the software engineering community can contribute to the evolution of the tool.

D. Annotation Visualizer

The AVisualizer is a web application that displays code annotations using the Code Annotations Distribution Visualization (CADV) approach. As mentioned, the CADV uses a circle packing structure, which helps display the hierarchical structure of the analyzed source code.

When executed, the AVisualizer presents the user with three areas, as shown in Figure 3. These three areas are The **Header**, The **View**, and The **Schema Table**. Following, we detail each of these areas.

The **Header** contains four parts, described below [3]:

- **Project Under Analysis**: It displays the name of the analyzed software. For our studies, we used the *Spring-retry* software.
- **Visualization**: Since the CADV is composed of three views, the user must know which one is currently rendered. System View, Package View, and Class View are the three possible options. Figure 3 is displaying the System View.
- **Annotation**: We need to inform the user what metric is being used to generate the size of the leaf circles. These are the colored ones and may represent individual code annotations or annotation schemas. As an example, the System View uses the metric *Number of Annotations*, which means the colored circle size is calculated based on the number of code annotations that belong to a particular annotation schema. Further, every annotation schema has a different color.
- **Package**: Informs the user what package or class it is currently inspecting. For instance, the package inspected in Figure 3 is the `org.springframework.retry`, which is the root package of the *Spring Retry* library.

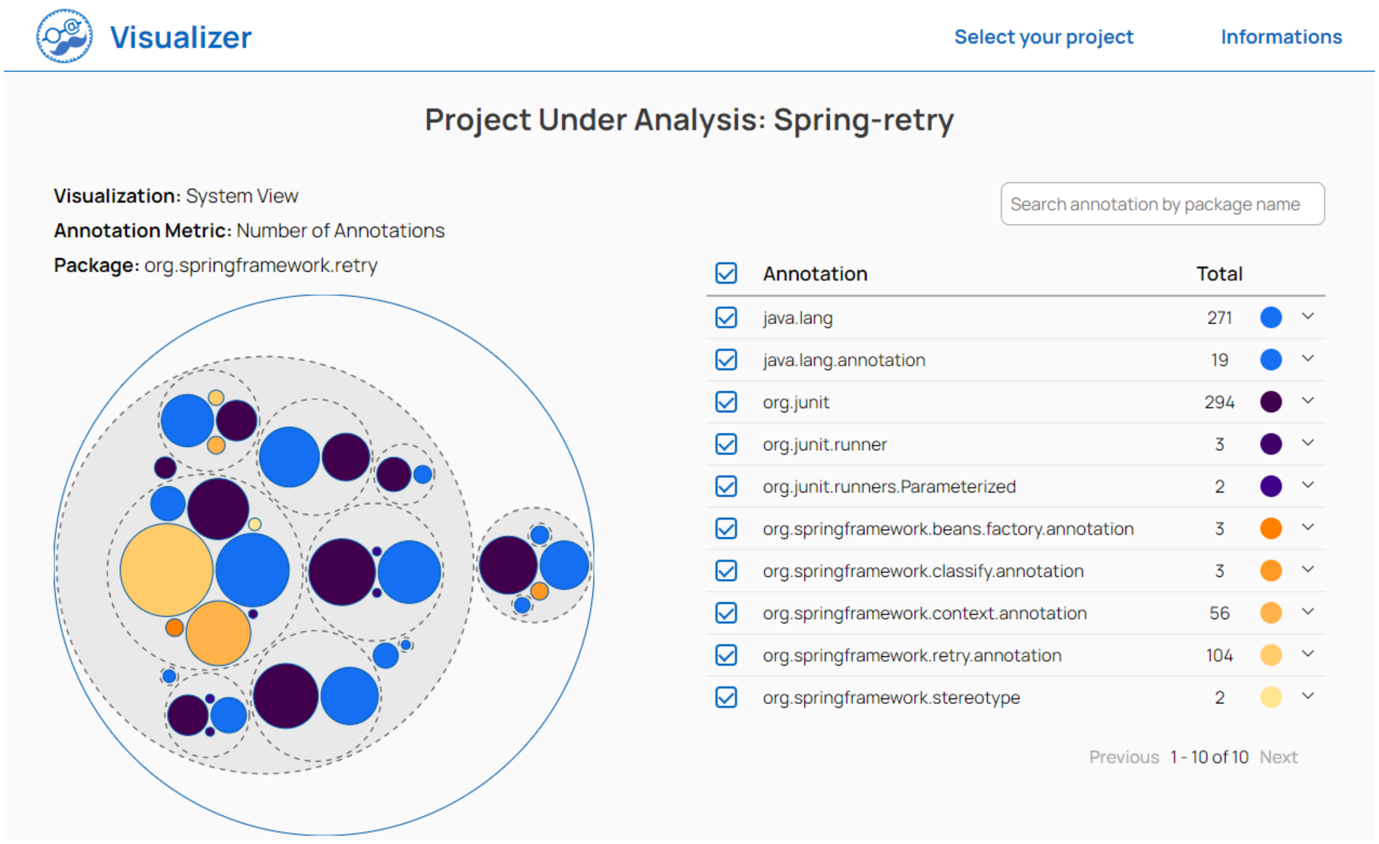
[8]https://github.com/metaisbeta/asniffer

The View is the area that displays the actual visualization. There are three different views: The System View, The Package View, and the Class View. For instance, Figure 3 is displaying the System View for the *Spring Retry*. The **View** is constantly changing and modifying to display one of the proposed views. Only one of the three is visible at a time. The **Header** also changes to inform what current view is displayed, and the **Schema Table** is mostly fixed. These last two are available for all three different views all the time.

The **System View** is the default view displayed to the user. It presents the whole project in a single view, allowing users to quickly grasp the project under analysis. The System View displays only packages and annotation schemas rendered as circles. The following list presents the characteristics of these circles [3]:

- Packages: Every circle representing a package has a dashed outline. The outermost dashed circle represents the root package of the project. In the source code, this package contains every other package. We present this hierarchical information by displaying packages inside packages as “dashed outline circles contained in other dashed outline circles”. The circle's size depends on the number of code annotations used inside the package, regardless of the number of classes. Therefore, we are counting code annotations in all classes, but we are not counting the classes. The background used in these circles is gray.
- Annotation Schemas: These are colored filled circles rendered inside “dashed outline circles”. They represent annotation schemas being used inside a specific package. Each annotation schema has its unique colors, which are reflected in the filled circles. The size of these circles is proportional to the number of code annotations of a particular schema. The colors white and gray are not used to represent schemas since they already have other meanings in the CADV approach.

The **Package View** can display classes and individual code annotations inside a given observed package. Dif-



Project Under Analysis: Spring-retry

Visualization: System View
Annotation Metric: Number of Annotations
Package: org.springframework.retry

✓	Annotation	Total	
✓	java.lang	271	●
✓	java.lang.annotation	19	●
✓	org.junit	294	●
✓	org.junit.runner	3	●
✓	org.junit.runners.Parameterized	2	●
✓	org.springframework.beans.factory.annotation	3	●
✓	org.springframework.classify.annotation	3	●
✓	org.springframework.context.annotation	56	●
✓	org.springframework.retry.annotation	104	●
✓	org.springframework.stereotype	2	●

1 - 10 of 10

Fig. 3. Annotation Visualizer interface.

ferently from the **System View** designed to visualize the whole system, in the **Package View**, we are interested in a specific package. The circles are rendered with the following characteristics [3]:

- Package: The same characteristics from the **System View**, i.e., a circle with a dashed line and gray-colored background. Usually, in the **Package View**, this circle is the outermost one, working as a frame for the **View** area. Every other inner circle represents elements inside this package.
- Classes: Rendered as white-filled circles. Their size depends on the number of code annotations used inside the class and the metric used to draw them. The default metric used is the LOCAD. If a white circle appears larger than others, it represents a class with more code annotations.
- Code Annotations: These are colored (any color besides white and gray) filled circles rendered on top of white circles. They represent code annotations used inside a specific class. Their color matches the color of their schema, present on the **Schema Table**. The size of these circles is proportional to their LOCAD value,i.e., the default metric used.

The **Class View** can display classes and individual code annotations inside the observed class. It allows users to visualize how code annotations are configuring a specific programming element, such as a method, class member, or the class itself. The circles are rendered according to the following rules [3].

- Classes: Just as in the **Package View**, they are rendered as a white circle. Since we are analyzing a specific class, there will be only one white circle.
- Annotations: Colored circles representing individual annotations. The color is related to their schema present on the **Schema Table**. The size of the circle is obtained by some code annotation metrics such as AA, ANL, or LOCAD. The default metric is the AA - Arguments in Annotations.
- Gray-Circles: This color is also used to represent packages, but in the **Class View** they represent code elements, such as method and class members. The code annotations (colored circles) are rendered on top of these gray circles. Colored circles rendered directly on top of a white circle represent code annotations configuring the class itself. The number of colored circles rendered on top of the same gray circle is the number of code annotations configuring that code element.

In our previous work [3], we conducted two studies to investigate how the CADV, implemented as the AVisualizer web application, can aid in software comprehension. One study was conducted as an interview with six developers of *SpaceWeatherTSI*, a web application developed for INPE[9] (National Institute for Space Research) as part of the

[9]https://www.gov.br/inpe/pt-br

EMBRACE ("Brazilian Studies and Monitoring of Space Weather") program. During the interview, the participants used the AVisualizer to visualize the *SpaceWeatherTSI* software system and answer questions related to code comprehension. Overall, the interview was carried out informally, and the participants were free to explore the AVisualizer tool as much as possible. Based on the feedback from the interviews, we proposed to develop a new solution that will integrate the AVisualizer into a popular Java IDE. This resulted in the development of the AVisualizer Plugin for the IntelliJ IDE. According to StackOverflow 2019 annual survey [10], the IntelliJ is the most popular Java IDE in the community.

IV. Annotation Visualizer Plugin

THe AVisualizer Plugin is an evolution of the AVisualizer web application. It uses the ASniffer and runs integrated as a plugin for IntelliJ IDE. It also stores the collected metrics from ASniffer in a cloud database for more accessible retrieval. In short, we aimed to run, visualize, and share analyses through an IntelliJ IDE plugin. Generating the metrics to serve the visualization is transparent to the user.

The AVisualizer plugin allows us to visualize the system currently opened in the IntelliJ IDE without switching to another environment. The visualization is rendered in the embedded browser available in IntelliJ, which eases compatibility and lowers the development efforts to maintain both the AVisualizer plugin and the AVisualizer web application. From a user perspective, it runs as a native plugin for IntelliJ. As mentioned, in the plugin's back end, we have the ASniffer generating the metrics to serve the visualization. This report can also be stored in a cloud database. The AVisualizer plugin integrates these solutions and makes the functionality transparent for the final user.

A. Evolution of Annotation Sniffer

During the development of the proposed plugin, several improvements were made to the ASniffer. One improvement was having the ASniffer run on the Windows operating system correctly since it was previously only executed in UNIX-like environments. It had a bug in which the file path separator to save the JSON report was hard-coded in the project. It was using the reverse backlash as a path composer, which works for UNIX systems, but it did not work as intended in Windows. We fixed this issue with the *java.nio.File.Paths* API that identifies the operating system and composes the file path accordingly. This fix was made by opening an issue in GitHub and contributing a pull request to the project repository via Git Forking Flow. New unit tests were added to ensure that this change would not break the code[11]. Another improvement was the availability of ASniffer with the IntelliJ JDK. During our testing, we noticed that these technologies had some incompatibilities. A wrong `ClassPath` use drove the bug in the ASniffer. An issue[12] was opened on GitHub, discussed at our weekly meetings, and the project maintainer helped us with a solution.

We also created the Annotation Sniffer Web Application. It is very usual for the team to share tool views in software development, such as monitoring logs and dashboards. During the development of our solution, we came across the need to have reports generated by ASniffer saved and accessed by the AVisualizer. We decided to develop a web API with a web REST API and a MongoDB database. Its primary function is to save, update and consult projects generated by ASniffer.

We developed it in Java 11 with the Spring Boot framework and Spring Boot Web to handle HTTPS requests. Spring Boot is one of the most popular frameworks for Java, extensively used in the industry. We used tools from the Spring ecosystem to develop this application: Spring Data is used for managing the connection with the MongoDB database. This application is responsible for saving and consulting reports generated by ASniffer in a database and returning its result in an HTTPS request. Each project has an ID that identifies and is unique across all saved documents.

The data is stored in MongoDB, a non-relational database. The decision to use a non-relational database comes from the need to store the documents generated by ASniffer, generated in JavaScript Object Notation (JSON). MongoDB is a document-based database and uses a Binary Javascript Object Notation (BSON) schema, which is a textual object notation widely used to transmit and store data across web-based applications. JSON is easier to understand as it is human-readable, but compared to BSON, it supports fewer data types. BSON encodes type and length information, making it easier for machines to parse[13]. Since this project is cloud-based, we choose *Mongo Atlas* as our host for the MongoDB database. *Mongo Atlas* is a multi-cloud database service built by the same corporation that developed MongoDB.

B. Evolution of Annotation Visualizer

The Annotation Visualizer is also a project of the plugin proposed in this work. We added the possibility of recognizing a query parameter in the project URL to achieve our goal. The query parameter of this project must be the same ID generated by the Annotation Sniffer Web APP. Using this information, the AVisualizer makes a REST request to get the ASniffer reports corresponding to this ID in Annotation Sniffer Web APP and renders the project on the screen. This enables a dynamic visualization integrated with our ecosystem. An example can be found in Figure 4 using the ID *spring-retry-16*. During the plugin's development, the AVisualzier web application switched from Angular to React, so we had to modify our features to this new technology.

[10]https://insights.stackoverflow.com/survey/2019/
[11]https://github.com/metaisbeta/asniffer/pull/4
[12]https://github.com/metaisbeta/asniffer/issues/5
[13]https://www.mongodb.com/basics/bson

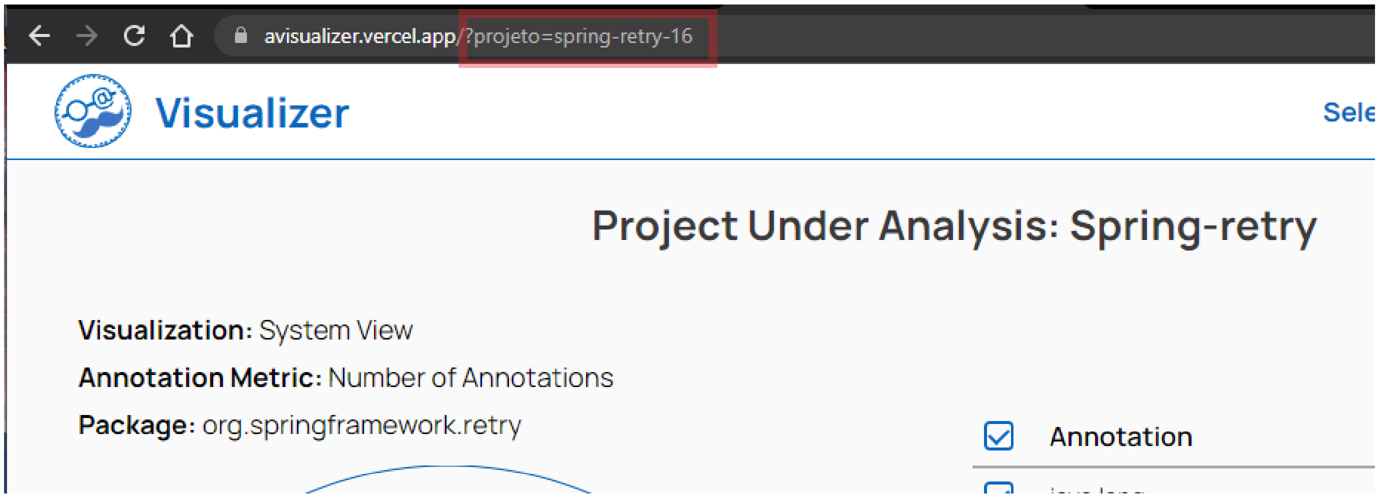

avisualizer.vercel.app/?projeto=spring-retry-16
Visualizer
Sele
Project Under Analysis: Spring-retry
Visualization: System View
Annotation Metric: Number of Annotations
Package: org.springframework.retry
Annotation

Fig. 4. Query parameter in Annotation Visualizer indicating the project

The interaction with the plugin can occur in two different ways. The first is a button on the toolbar to start the user's current project analysis. The second is a modified browser integrated with the plugin.

The browser integrated into this plugin is an adaptation of the GIdea Browser[14], which is a FLOSS implementation of a Chromium-based browser for IntelliJ. The browser engine used by this project is the Java Chromium Embedded Framework (JCEF), an implementation of Chromium in the Java language. The plugin is responsible for loading custom URLs via events. In this way, we show the user the final analysis of the project. The Plugin overview can be found in Figure 5.

C. System Architecture and Integration

For developing this system, we used the MVC (Model-View-Controller) architectural pattern composed of three layers. The model is the application object, the view is its screen presentation, and the controller defines how the user interface reacts to user input. Thus, MVC decouples them to increase flexibility and reuse. The AVisualizer presents the view, and the controller and model are represented in the Annotation Sniffer Web APP. Since both projects are constantly developing, this pattern was chosen because we want to decouple the plugin from both ASniffer and the AVisualizer. MVC also enables an expansion of this software by using the Annotation Sniffer Web APP to render projects in other views.

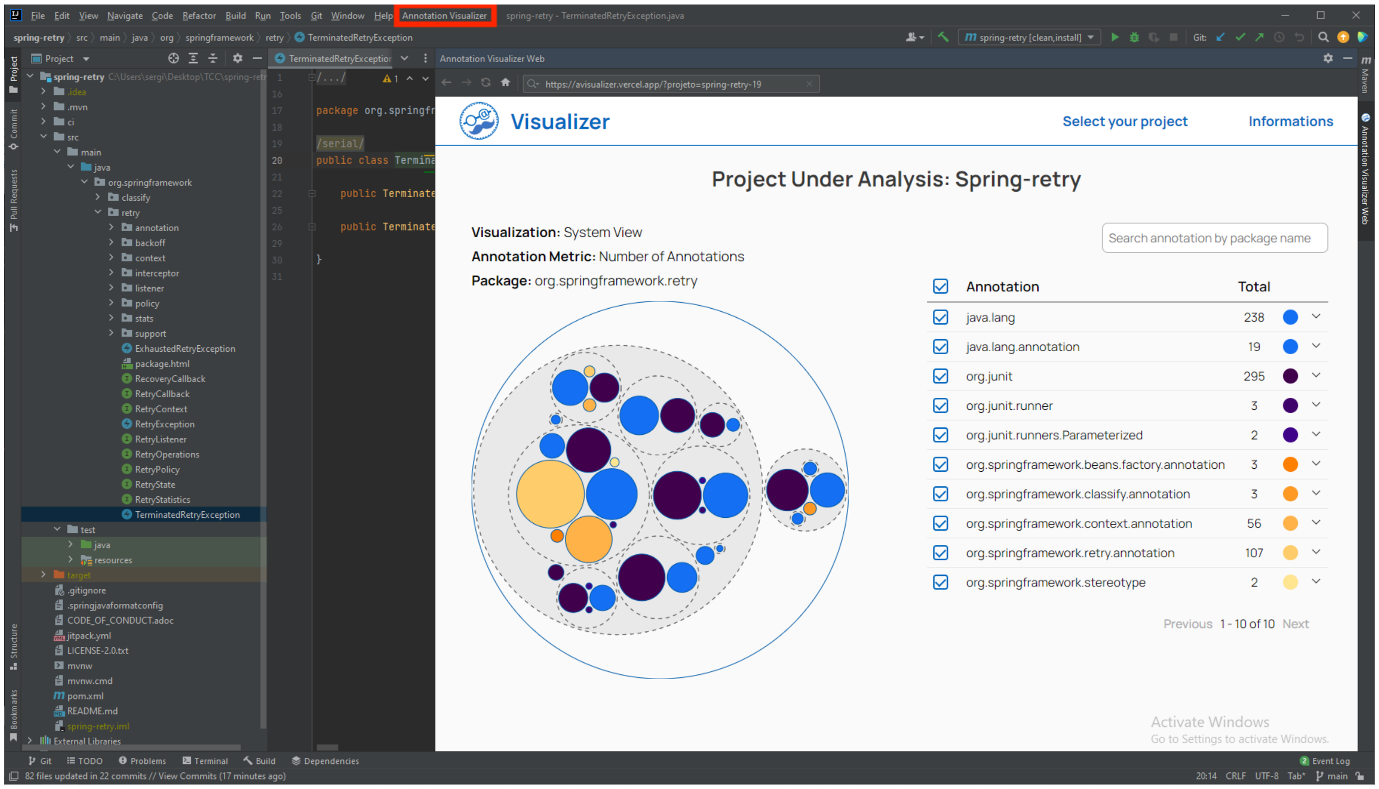
A plugin for IntelliJ IDEA is an application that runs on the JetBrains Runtime. This software was built using IntelliJ SDK, which contains the IntelliJ OpenAPI package that integrates the IDE. The language used for developing the plugin is Kotlin with the Gradle framework for dependency management and building. The main dependency of this project is ASniffer, hosted at Maven Central.

The plugin for IntelliJ is event-oriented. An event-driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications. An event is a state change or an update, like a button click. Every user action generates an event that the JetBrains Runtime computes. The button and the sidebar generate events that the plugin interprets and execute corresponding actions. The button triggers the event to start the Annotation Sniffer analysis. The sidebar button triggers the event to open the embedded browser. The communication between the Jetbrains Runtime and the embedded browser is event-based. When the user clicks the button, the Runtime generates an event for the browser to load the landing page. If everything runs correclty — the event to load the project triggers. Otherwise, the error page event is triggered.

To understand our system solution, we provide Figures 6 and 7 as references. In step 1 from Figure 6, the user interacts with the plugin button and requests the analysis of the project. In step 2, the ASniffer is activated and analyzes the project locally. Step 3 sends the files generated by the analysis to the Annotation Sniffer Web APP. Step 4 is to receive the project ID within the Annotation Sniffer APP. Step 5 consists of modifying the browser's URL to receive the ID collected by the previous step as a parameter. Step 6 symbolizes the project AVisualizer sending the page HTML to be displayed by the browser. Step 7 is the page displayed by the embedded browser, closing our software loop.

Annotation Sniffer Plugin option is shown in the navigation bar and sidebar, as shown in Figure 5. By clicking on the button on the navigation bar, the Plugin shows two options. The first is "Run Annotation Analysis", and the second is "Run Annotation Analysis (Not Persist)", as seen in Figure 8. The main difference is that the second does not persist the data in the database. After the button clicks, the IntelliJ Runtime executes an event that triggers the analysis.

[14]https://github.com/Jonatha1983/GIdeaBrowser


File Edit View Navigate Code Refactor Build Run Tools Git Window Help Annotation Visualizer
spring-retry - TerminatedRetryException.java
Annotation Visualizer Web
https://avisualizer.vercel.app/?projeto=spring-retry-19
Visualizer
Select your project
Informations
Search annotation by package name
Activate Windows
Go to Settings to activate Windows.

Project Under Analysis: Spring-retry

Visualization: System View
Annotation Metric: Number of Annotations
Package: org.springframework.retry

Annotation	Total
java.lang	238
java.lang.annotation	19
org.junit	295
org.junit.runner	3
org.junit.runners.Parameterized	2
org.springframework.beans.factory.annotation	3
org.springframework.classify.annotation	3
org.springframework.context.annotation	56
org.springframework.retry.annotation	107
org.springframework.stereotype	2

Previous 1 - 10 of 10 Next

Fig. 5. IntelliJ IDEA with the Plugin

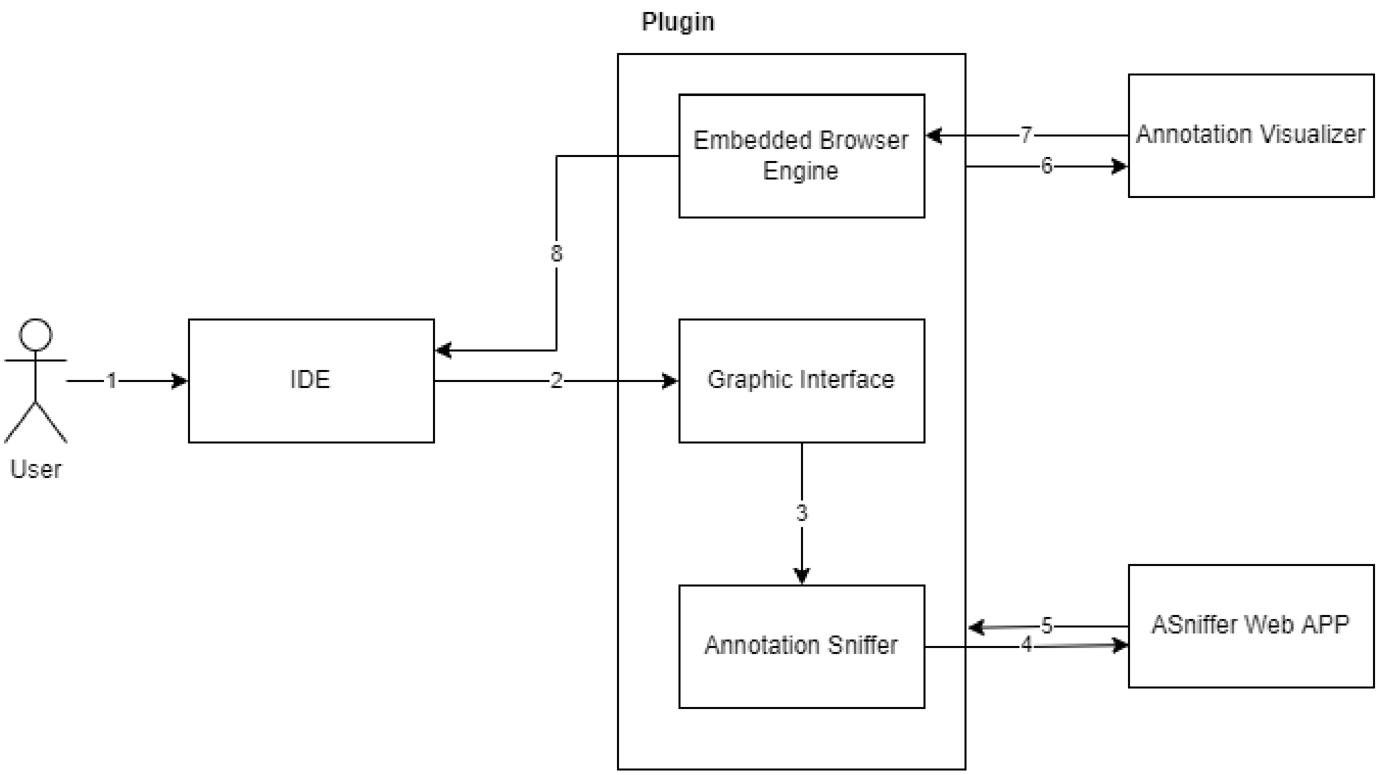


Fig. 6. Architecture of the plugin

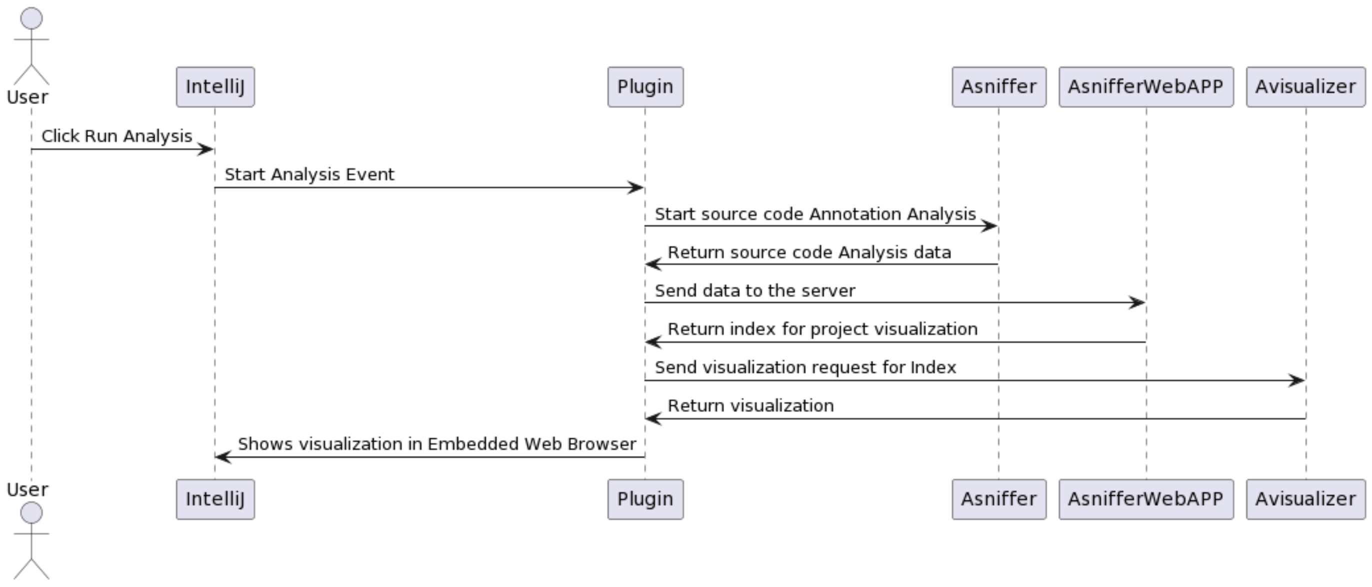

User
IntelliJ
Plugin
Asniffer
AsnifferWebAPP
Avisualizer
Click Run Analysis
Start Analysis Event
Start source code Annotation Analysis
Return source code Analysis data
Send data to the server
Return index for project visualization
Send visualization request for Index
Return visualization
Shows visualization in Embedded Web Browser

Fig. 7. Sequence diagram for requesting and showing the visualization

V. Evaluation

We used Spring Retry[15] to evaluate and verify the developed plugin. Spring Retry is a popular spring boot library that provides declarative retry support for Spring applications. It is used in Spring Batch, Spring Integration, and other software from the Spring family.

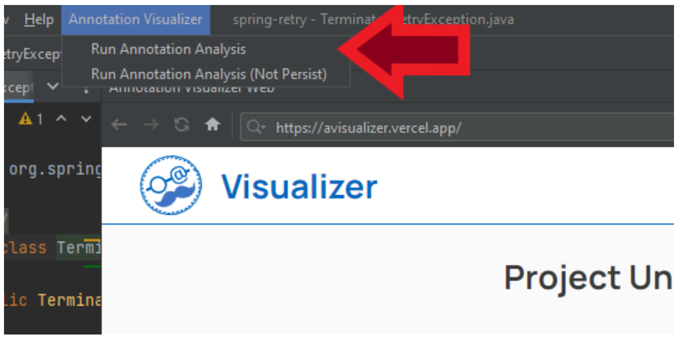

Help
Annotation Visualizer
Run Annotation Analysis
Run Annotation Analysis (Not Persist)
https://avisualizer.vercel.app/
Visualizer
Project Un

Fig. 8. Menu Button Options

The first step in the evaluation procedure was downloading IntelliJ IDEA. Then the plugin, properly speaking, is found in the IntelliJ Marketplace in the “Settings” options browsing the “Plugins” sidebar. After downloading and installing the plugin, two new changes occur in the IDE UI. There is a button in the header, as seen in Figure 8, and a button in the right sidebar, which opens the embedded browser. The default project is *SpaceWeatherTSI*, and it is just a placeholder. In the Menu button, there are two options to choose from. The first *Run Annotation Analysis* runs the Annotation Sniffer analysis, send it to the Annotation Sniffer Web App, and stores this project in the database. There is a second option *Run Annotation Analysis (Not Persist)* which will delete the data from our database after the first consult. This second option enables the user to use this tool without having this analysis saved in our database.

After the button is clicked, the event starts, and the processing analysis runs asynchronously. This job should run asynchronously to avoid blocking the main thread from Jetbrains Runtime. During this job execution, a loading screen in the embedded browser gives some feedback to the user on what is going on.

Moreover, we evaluated the plugin with two users familiar with the ASniffer and AVisualizer. They participated in a previous interview study with AVisualizer [3]. Participants must install and use the plugin and answer some questions. The questions were divided into two parts: (a) answers about the information shown in the plugin that has a correct answer, and (b) answers about the user’s opinion regarding the plugin’s usability.

The first set of questions was all answered correctly. The overall answers about usability were positive. Participants agree that this plugin is easy to use and has a clean user interface. They also considered the documentation good.

One of the users needed some help installing the plugin, which was related to a need for more experience installing plugins in IntelliJ IDEA since it may not be apparent to users where the option can be found. Also, the usage of the embedded browser received an “average grade” in feedback. Having a browser and an IDE in the same space can be tight and ineffective for smaller monitors. Also, we received user feedback that when drilling down on classes, the table of annotations should be updated by applying the selected class or package filter. It would be interesting to choose the metric the user wants to visualize if there are other visualization possibilities for the chosen selection. Currently, the AVisualizer does not have this capability. The user should click on the package to see it.

In summary, both users answered all questions accu-

[15]https://github.com/spring-projects/spring-retry

rately from the *correctness* part. This result is expected since they used AVisualizer in another context. The usage feedback was good. We found room for improvement in the plugin UI and that the plugin fulfilled its role.

VI. Conclusion

THE AVisualizer plugin enabled better integration between ASniffer and AVisualizer tools. Having the plugin embedded in the IDE empowers the developer to access the data during the development and could help newcomers to a project gain better insight into how the annotations are used. This plugin lowers the barrier of entry to the Annotation Visualizer ecosystem because it is easy to use, intuitive, and embedded in an IDE. From the Jetbrains Marketplace, we identified that people from other countries unrelated to our project used the plugin and could have already benefited from the Annotation Visualizer.

The ASniffer and AVisualizer integration can be extended to other development processes. The technology developed in this work can run the ASniffer analysis and display the AVisualizer visualization during the Continuous Integration (CI) process. One possible application for this connection is to develop a CI workflow that analyzes the source code committed by the developer and displays a URL for the AVisualizer visualization of the project. This approach can be implemented via GitHub Actions, for example.

Finally, another improvement is to enable connecting directly to AVisualizer using the files generated by the ASniffer, stored in the local (.idea) root project directory. Moreover, performance improvements and new features based on user feedback will be beneficial recommendations to maintain the project's purpose.

Acknowledgments

We thank all the respondents who participated in our evaluation. This work is partially supported by FAPESP (grant #2019/12743-4).

References

[1] P. Lima, E. Guerra, P. Meirelles, L. Kanashiro, H. Silva, and F. F. Silveira, "A metrics suite for code annotation assessment," *Journal of Systems and Software*, vol. 137, 2018, ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.11.024.

[2] P. Lima, E. Guerra, and P. Meirelles, "Annotation sniffer: A tool to extract code annotations metrics," *Journal of Open Source Software*, vol. 5, no. 47, p. 1960, 2020. DOI: 10.21105/joss.01960.

[3] P. Lima, J. Melegati, E. Gomes, N. S. Pereira, E. Guerra, and P. Meirelles, "Cadv: A software visualization approach for code annotations distribution," *Information and Software Technology*, vol. 154, p. 107 089, 2023, ISSN: 0950-5849. DOI: doi.org/10.1016/j.infsof.2022.107089.

[4] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional, 2004, ISBN: 0321278658.

[5] W. Hasselbring, A. Krause, and C. Zirkelbach, "Explorviz: Research on software visualization, comprehension and collaboration," *Software Impacts*, vol. 6, p. 100 034, 2020, ISSN: 2665-9638. DOI: 10.1016/j.simpa.2020.100034.

[6] V. Rajlich, "Software evolution and maintenance," in *Future of Software Engineering Proceedings*, ser. FOSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 133–144, ISBN: 9781450328654. DOI: 10.1145/2593882.2593893.

[7] F. P. Brooks, "No silver bullet essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, 1987. DOI: 10.1109/MC.1987.1663532.

[8] K. Stephenson, J. Cannon, W. Floyd, and W. Parry, "Introduction to circle packing: The theory of discrete analytic functions," *The Mathematical Intelligencer*, vol. 29, pp. 63–66, Jan. 2007. DOI: 10.1007/BF02985693.

Sergio Abilio Graduated in Computer Science from the Federal University of ABC (UFABC). His interests are distributed software architecture and cloud computing.
E-mail: *sergio.abilio@aluno.ufabc.edu.br*

Phyllipe Lima is an adjunct professor of computer science and researcher at the Federal University of Itajubá (UNIFEI). His research interests include static source code analysis, mining software repositories, software visualization, game design and development. He received his Ph.D. in Applied Computing by the National Institute for Space Research (INPE). E-mail: *phyllipe@unifei.edu.br*

Everaldo Gomes is a scholarship student at the Federal University of ABC region (UFABC). His research interests include distributed systems, fault tolerance, model checking, and source code metrics. Gomes received a master's degree in Computer Engineering from the Federal University of Rio Grande, Brazil.
E-mail: *everaldogjr@gmail.com*

Eduardo Guerra is currently a researcher at the Free University of Bozen-Bolzano, worked in Brazil as a researcher in the National Institute for Space Research in Brazil and as a teacher at Aeronautics Institute of Technology. His research interests include agile methods, software patterns, framework development, software analytics, and dynamic architectures. Guerra received a Ph.D. in computer engineering from the Aeronautics Institute of Technology and has practical experience in architecture and framework design.
E-mail: *eduardo.guerra@unibz.it*

Paulo Meirelles is an adjunct professor of computer science at the Federal University of ABC region (UFABC) and a researcher at the Free/Libre/Open Source Software Competence Center at the University of São Paulo (USP). His research interests include free software development, agile methodologies, DevOps, and source code metrics. Meirelles received a PhD in computer science from the University of São Paulo.
E-mail: *paulo.meirelles@ufabc.edu.br*